Enhancing the HPC-Lab Snow Simulator with More Realistic Terrains and Other Interactive Features

Andreas Nordahl

Master of Science in Computer Science
Submission date: June 2013
Supervisor: Anne Cathrine Elster, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

At NTNU's HPC-Lab, previous Master students have over several years developed a real-time GPU-based snow simulation. The goal of this work is to make the current simulation more realistic, while still being able to simulate several million particles in real-time. A major feature would be to add support for terrain interactions. Efforts could also include improving the rendering of the terrain itself through real-time rendering techniques such as triplanar texturing, slope and height based texture mapping. Artificial features that improve the visual impression such as distance fog and skybox could also be considered. Perlin noise may also be used to improve the visual result though texture generation and blending. A list of current features, new features and suggestions for future improvements, should also be included, along with an updated GUI and a better structured code base.

Abstract

Taking advantage of the raw processing power offered by today's graphic processing units (GPUs) has become a major research topic. The amount of repeated independent computation that goes into the simulation of physical systems such as wind and snow simulations makes them great candidates for GPU parallelization.

The HPC-Lab at the Norwegian University of Science and Technology (NTNU) has had several master students contributing to a real-time snow simulation. This thesis presents the work done to implement a number of real-time rendering techniques to improve the realism of this snow simulator. The main implementation of our snow simulator, which this thesis is based on, runs on the GPU using CUDA.The rendering is done in OpenGL, so the use of CUDA/OpenGL interoperability has been integral to achieve the performance needed to render the simulator output in real-time. The additional rendering techniques introduced by this thesis work are mesh texturing and lighting, triplanar texturing, scalar texture mixing, Perlin noise texture blending, shadow mapping, distance fog, skybox, billboarding and procedural texturing. Keeping the strict real-time constraint of the snow simulator has been the most important factor in choice and implementation of the rendering techniques.

Our results show that scalar mixing, shadow mapping, distance fog and skybox all give significant visual improvements to the snow simulator, at a relatively low cost, keeping the frame rate above 24 frames per second (fps) for terrains of resolution 1024x1024 vertices. Triplanar texturing, however, turns out to not be that well suited for the snow simulator because of the added computational cost of doing several texture samples for each fragment, and the lack of visual improvement due to the relatively flat terrain height maps used. Procedural texturing of the snow particles using Perlin noise are tested and shown to be as fast as using image textures, with nearly 25 fps when rendering 5 million particles using a wind field with resolution 128x32x128 over a 768x768 terrain on a PC with a NVIDIA's GTX480 card. It significantly improves the realism of the rendered snowfall. Ideas for further improvements are also included.

Sammendrag

Å utnytte seg av prosseseringskraften som dagen grafikkort (GPU) tilbyr har blitt et stort forskningsområde. Mengden gentatt uavhengig beregninger i datasimuleringen av fysiske systemer som vind- og snøsimulering gjør at de egner seg veldig godt for GPU parallellisering.

HPC-laben på norges teknisk-naturvitenskapelige universitet (NTNU) har hatt flere masterstudenter som har bidratt til en sanntids-snøsimulator. Denne oppgaven presenterer implementeringen av flere renderingsteknikker til denne snøsimulatoren. Hovedversjonen av snøsimulatoren, som er den versjonen denne oppgaven tar utgangspunkt i, er skrevet i CUDA og kjører på GPU, og dens rendering blir gjort i OpenGL. Bruken av interoperabilitet mellom CUDA og OpenGL er derfor avgjørende for å fylle sanntidskravet til simulatoren. Renderingsteknikkene dekket i denne oppgaven er mesh-tekstrurering og lyssetting, triplanar teksturering, skalar tekstur miksing, Perlin støy tekstur blanding, shadow mapping, avstandståke, skybox, billboarding og prosessuell teksturering. Det strenge sanntidskravet til snøsimulatoren har vært tungtveiende i valget av og implementeringen av de forskjellige renderingsteknikkene.

Resultatene våre viser at skalar miksing, shadow mapping, avstandståke og skybox gir signifikante visuelle forbedringer til snøsimulatoren med relativt lav kostnad, og holder en bildefrekvens på over 24 bilder i sekundet for terreng med oppløsning 1024x1024 noder. Triplanar teksturering viser seg imidlertid til å ikke egne seg så godt til snøsimulatoren på grunn av høy utregningskostnad for ekstra tekstursampling for hvert fragment, samt mangel på visuell forbedring grunnet relativt flate høydekart brukt til terrenget. Prosessuell teksturering av snøpartikler ved hjelp av Perlin støy viste seg å være like raskt som tradisjonell bildeteksturering, og rendret i en hastighet på nesten 25 fps med 5 millioner partikler i et vindfelt med oppløsning 128x32x128 over et 768x768 terreng, og gjorde det rendrede snøfallet mer realistisk. Forslag til forbedringer av snøsimulatoren er også inkludert.

Acknowledgements

I would first like to thank my supervisor for this thesis, Dr. Anne C. Elster, her assistance has been invaluable. Because of her hard work and networking, the HPC-lab has received sponsoring and resources without which this work would not have been feasible. I also wish to thank NVIDIA for their support of HPC-Lab through their CUDA Research Center and CUDA Teaching Center program.

I would also like to thank my brother, Espen Nordahl, who has been a constant source of inspiration, an excellent guide to the immense world of computer graphics, and who has been very patient in answering all my ignorant questions. And lastly, I would like to thank the people with whom I have had the honor to share the HPC-lab, especially Magnus Mikalsen. They have all been very helpful and supporting.

Contents

List of Figures

Chapter 1

Introduction

With the relatively recent availability to more general-purpose computing on graphic processing units (GPGPU), utilizing the raw power of graphic processing units (GPUs) has become a major research topic. To access the high throughput the GPU offers, programs have to be able to take advantage of the parallel nature of the GPU. One class of problems that are suited for this, and an area in which there has been done a lot of research, is computer simulation of physical phenomenon. The amount of repeated independent computation that goes into the simulation of systems such as wind and snow makes them perfect candidates for GPU programming.

At the Heterogeneous and Parallel Computing Lab (HPC-lab) at the Norwegian University of Science and Technology (NTNU) there has, as part of numerous master projects and thesis, been developed such a computer simulation; a snow simulator. The original version of the HPC-lab snow simulator was written in 2006 by Ingar Saltvik [21], and only ran in parallel threads on the CPU. Since then, the program has been ported to CUDA, to be run on the GPU, and this revolutionized its performance, increasing its efficiency by several orders of magnitude. It made it possible to run the simulator in real-time, with millions of snow particles falling on a complex terrain. The history of the snow simulator is covered in depth in Section 2.6.

This report presents the work done in this thesis to improve upon the rendering part of the simulator. This is done by well known rendering techniques for real-time graphics used in games and the special effect industry. Improved texturing for the terrain and snow, in addition to the techniques like distance fog, skybox and shadow mapping, are implemented to increase the realism of the snow simulator scene. Keeping the real-time constraint is difficult, but of the utmost importance for applications like this. One of the most crucial reasons that makes this possible is the utilization of the specific CUDA/OpenGL interoperability operations designed to make their interaction more effective.

1.1 Outline

The following is an outline of the rest of this thesis.

Chapter 2: Background introduces some concepts and background knowledge related to the thesis. GPGPU is presented, and an introduction is given to the graphics language OpenGL, the OpenGL Shading Language, and CUDA. The powerful procedural texturing method Perlin noise is then explained, before finally giving a thorough introduction to the HPC-lab snow simulator.

Chapter 3: Implementation gives an explanation of the implementation of each of the techniques used in this thesis. The specifics of the terrain rendering, shadow mapping, distance fog, the skybox, and snow rendering is covered in detail.

Chapter 4: Results and Discussion presents and discusses the results of each of the performance tests run on the snow simulator. The performance of all the implemented rendering techniques are tested, and the visual results are evaluated.

Chapter 5: Conclusion and Future Work summarizes the thesis and conclusions are drawn. Some suggestions are also given for possible future projects.

Appendix A-D contain the source code of terrain and snow shaders, and are referred to in the text.

Chapter 2

Background

In this chapter, some concepts and background knowledge related to this thesis are introduced. After a short introduction to GPUs in Section 2.1, Section 2.2 and 2.3 describe OpenGL and the OpenGL Shading Language. Section 2.4 presents the CUDA framework and programming model, and its interoperability with OpenGL. In Section 2.5 the concept of Perlin noise is explained, and Section 2.6 introduces the the HPC-lab snow simulator.

2.1 The GPU

Graphic processing units (GPUs) are high-performance, many-core accelerators that are specifically designed for the rendering of 3D graphics. Computer graphics is highly parallel and demands high floating-point throughput, so the GPU are designed to maximize these properties. The high throughputs possible with a GPU compared to that of a CPU, as seen in Figure 2.1, made them attractive to programmers who wanted to utilize their powers for other tasks than computer graphics. Although GPUs were difficult to program at first, they have been made increasingly accessible through programming interfaces and languages such as OpenCL and CUDA. Taking advantage of the GPUs parallel power combined with the more general CPU to accelerate general-purpose scientific and engineering applications is what is called general-purpose computing on graphics processing units (GPGPU).

GPGPU is not suited for any program, however. The high number of cores is the source of the GPUs power, and to be able to harvest this power, the program running on it must be optimized to run in parallel. Since there is so little space set aside for caching and flow control on the GPU, Figure 2.2, optimizing for how memory is used is crucial to the performance of the program. Moreover, the relatively slow transfer rate of the PCI express bus can be the bottleneck when

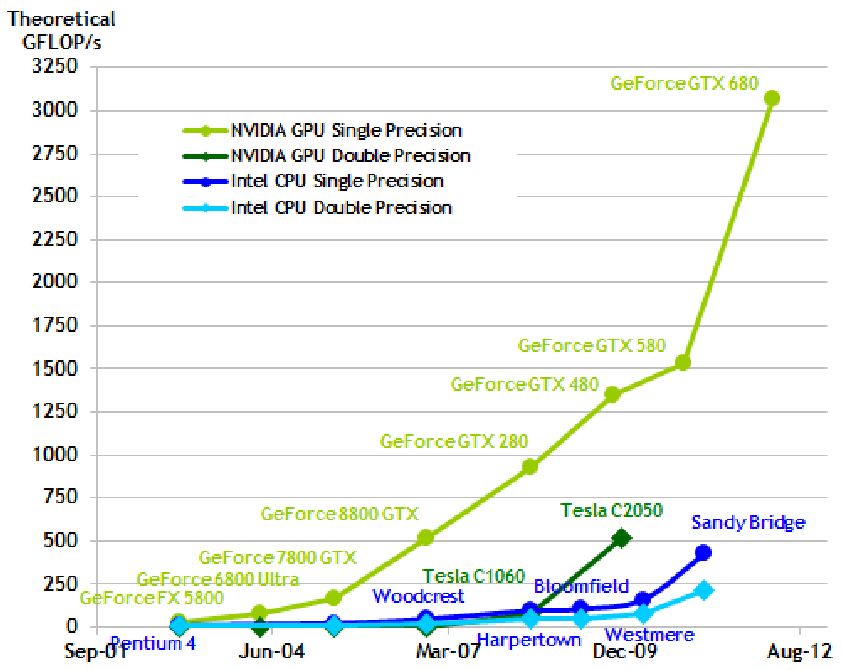


Figure 2.1: Memory Bandwidth for the CPU and GPU (with permission from NVIDIA[17]).

running a program on the GPU, and careful design of what and when to transfer memory to the device is decisive for the final performance.

2.2 OpenGL

Open Graphics Library (OpenGL) is an application programming interface (API) for drawing 2D and 3D graphics using graphics hardware. The API consists of several hundred procedures and functions that are syntactically similar to C, but OpenGL is language-independent and as such there exists several language bindings, like the one for JavaScript called WebGL. OpenGL is also platform independent, concerning itself only with rendering, leaving windowing, and obtaining and managing context to external libraries such as GLUT and GLFW. OpenGL is managed by the non-profit technology consortium Khronos Group.

2.2.1 Execution Model

OpenGL functions as the programmer's interface to graphics hardware. It contains a series of functions that allows the programmer to specify different aspects of the geometry in their scene. The geometry consists of primitives that are either points,

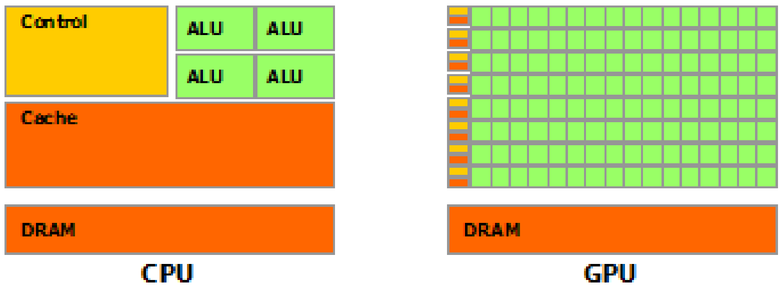


Figure 2.2: The GPU Devotes More Transistors to Data Processing (with permission from NVIDIA[17]).

lines, line segments, patches or polygons. The different primitives are built up of one or more vertices, defining either a point, an endpoint of a line segment, or a corner of a polygon where two edges meet. Each vertex can hold information like color, texture coordinates, and normal vector in addition to its three-dimensional position.

OpenGL is only concerned with processing data in GPU memory. However, it doesn't expose typical C-like pointer type memory allocation. Instead, OpenGL stores data in abstract generic buffers called buffer objects. These buffers objects can have different types, like shader objects containing shader programs, texture objects containing texture data, and vertex array objects that holds a set of vertices. Each object type has a corresponding set of commands which manage objects of that type. The buffer objects contain a data store holding a fixed-sized allocation of GPU memory and provide the mechanisms to allocate, initialize, read from, and write to that memory.

Drawing a 2D image from the 3D geometry is performed by looking at the scene through the lens of a virtual camera. Through a process of clipping and rasterization, the 3D primitives are projected onto a 2D plane according to a projection matrix that defines the camera. This process, called the graphics pipeline, Figure 2.3, is explained in more detail below. Some of these stages are programmable, through what is called shader programs (covered in the following section) and some are fixed-function stages.

2.2.2 The Graphics Pipeline

In the first stage each vertex is processed by the vertex shader. The vertices are then assembled to form geometric primitives, before allowing the optional tessellation and geometry shaders to generate multiple primitives from single input primitives. The results can then be fed back into buffer objects using transform feedback, or continue down the pipeline.

The final primitives output by the geometry shader are clipped to a clip volume before being rasterized and interpolated. In the rasterizer, a series of framebuffer addresses and values are produced, describing the primitives as interpolated two-dimensional values. Each of these fragments are then fed to the next stage, the fragment shader. The fragment shader performs operations, such as depth buffering, masking, texture sampling, and blending of colors, on individual fragments before they are finally written to the framebuffer as pixel values.

This whole process is usually double buffered, which means that the results of each image is not written directly to the screen. Instead, it is written to a second buffer which is only copied to screen when the entire process is completed. In doing so, the flickering and tearing that could otherwise occur is completely avoided.

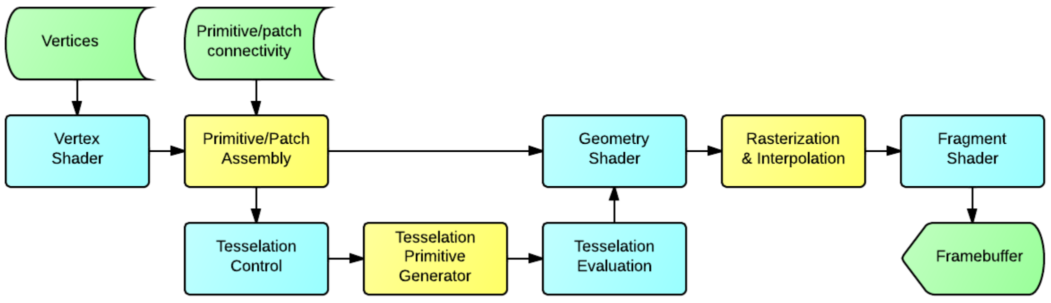


Figure 2.3: The OpenGL graphics pipeline.

2.3 OpenGL Shading Language

OpenGL Shading Language (GLSL) is a language used to write shader programs for OpenGL. It is a high-level shading language based on the syntax of the C programming language, and it was created to give developers a high-level, and more accessible option to more direct control of the graphics pipeline. Shader programs are executed on the GPU, and each type serve a very specific role. GLSL is actually several closely related languages, each one used to create shaders for the different programmable processors contained in the OpenGL processing pipeline. Below, each of these languages, and their typical function, are introduced.

2.3.1 Vertex Shaders

A vertex shader operates on all the incoming vertices, one vertex at a time. For each incoming vertex, the vertex shader outputs a single vertex, ensuring a 1:1 mapping of input vertices to output vertices. The data available to the vertex shader is limited to the user defined attributes of the incoming vertex, e.g. position and color, in addition to any uniform variables. As a result, the vertex shader has no knowledge of the other vertices that make up its primitive, or what type of primitive it belongs to. The most usual functionality of a simple vertex shader is to multiply the incoming vertex' position by the model-view-projection matrix supplied as a uniform available to all vertices. In doing so, each vertex is transformed into clip-space.

2.3.2 Tessellation Shaders

Tessellation is an optional part of the vertex processing stage, and involves subdividing a patch of vertices and computing the values of the vertices generated. The process is divided into three sub-stages, two of them programmable, called tessellation control and tessellation evaluation, with a fixed-function primitive generator in between. The tessellation control shader determines how much tessellation to do, ensuring gaps and breaks do not occur. The tessellation primitive generator then subdivides the input patch based on the computed value, and finally the tessellation evaluation shader takes the tessellated patch and computes the values for the generated vertices.

2.3.3 Geometry Shaders

Geometry shaders are also an optional part of the graphics pipeline. If present, it takes a single primitive as input, and outputs zero of more primitives. It can therefore, unlike vertex shaders, destroy primitives, or create new ones. Geometry shaders can be used to amplify geometry, but is more generally used for layered rendering, allowing a primitive to be rendered to multiple images with minimal work.

2.3.4 Fragment Shaders

Fragment shaders processes each fragment fed to it by the rasterization process. It's inputs are the fragment's window position along with the interpolated outputs from the vertex stage. Similarly to the vertex shader, the fragment shader only has access to the current pixel and knows nothing about the surrounding pixels. The outputs of the fragment shader is the fragment's depth value, used in the depth test, in addition to a four-dimensional color value vector for each of the draw buffers, usually one. The fragment shader is usually responsible for computing the color of it's pixel by sampling textures at the supplied interpolated texture coordinates.

2.3.5 Compute Shaders

Compute shaders are special in that they are not part of the graphics pipeline. While they can be used for rendering, in general they are used to compute arbitrary information, and are in many ways equivalent to CUDA kernels.

2.4 CUDA

Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model for general purpose, high-performance parallel computing. It was created by NVIDIA, and gives developers access to the virtual instruction set and memory of the parallel computational elements in CUDA enabled GPUs. Development of CUDA programs is generally done in CUDA C/C++ and compiled with nvcc, but extensions to other languages, such as Fortran, do exist. By giving access to the highly parallel graphics hardware for more general processing than just rendering, CUDA has paved the way for the development of accelerated non-graphical applications in many fields, such as cryptography, physics simulation, biology, and more.

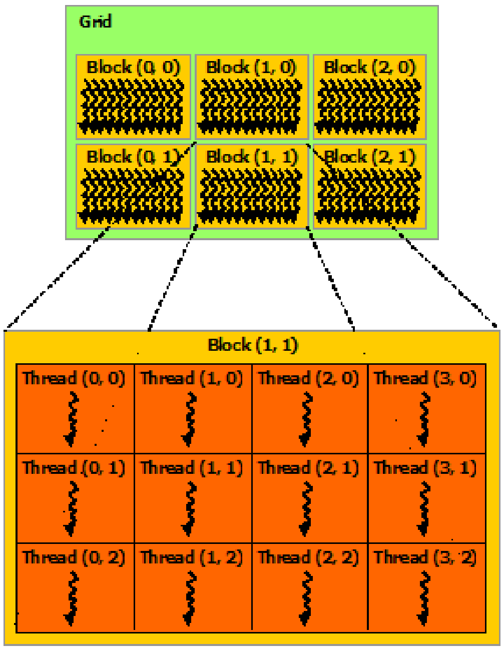


Figure 2.4: Grid of thread blocks (with permission from NVIDIA[17]).

2.4.1 Programming Model

A typical CUDA program i divided into two parts; the host code, that executes on the CPU, and the device code, that executes on the GPU. The device code is written as functions, called kernels, and when called by the host, the kernel is executed in several different CUDA threads on the GPU. The threads in CUDA are organized into blocks that can be up to three dimensions. Within these blocks, the threads can interact with each other via shared memory and sync points. These blocks are again organized in grids which can also be up to three dimensions. This structure is illustrated in Figure 2.4. Each thread can be identified in the grid with its unique thread id within each block and the block id of the block it resides in. Block and grid size, and therefore also the number of threads spawned, is declared when the kernel is called. During execution, the threads of each block are divided into warps of a maximum of 32 threads which are executed simultaneously.

2.4.2 Memory Model

The memory of the host and device are assumed to be completely separate. As kernels operate on the device, and only has access to device memory, there are functions available in the runtime to allocate, deallocate, and copy device memory, as well as transfer data between host memory and device memory. Memory allocation is typically done very similarly to normal C/C++ memory allocation, by using cudaMalloc(), and freeing that memory is done with cudaFree(), similar to the C/C++ syntax for freeing allocated memory.

Threads executing on the GPU have access to several types of memory as seen in Figure 2.5. In addition to registers, each thread has their own private local memory. The block has a memory space called shared memory that all threads in that block can access. The global memory is the largest memory and is accessible to all threads. The two last types of memory are also accessible to all threads, but serves specific roles. Constant memory is rather small and is used for storing constants. The texture memory has some neat features such as automatic interpolation of values, and is generally used as read-only.

2.4.3 CUDA/OpenGL Interoperability

CUDA functions for mapping resources from OpenGL into the address space of CUDA enable interoperability between the two. CUDA can then read data written by OpenGL, and data written by CUDA kernels can be used by OpenGL. The first step to achieve this is to have an active OpenGL context and then creating the CUDA context by calling cudaGLSetGLDevice(). Any OpenGL buffer intended *for* use is CUDA then has to be registered during initialization by calling cudaGraph-

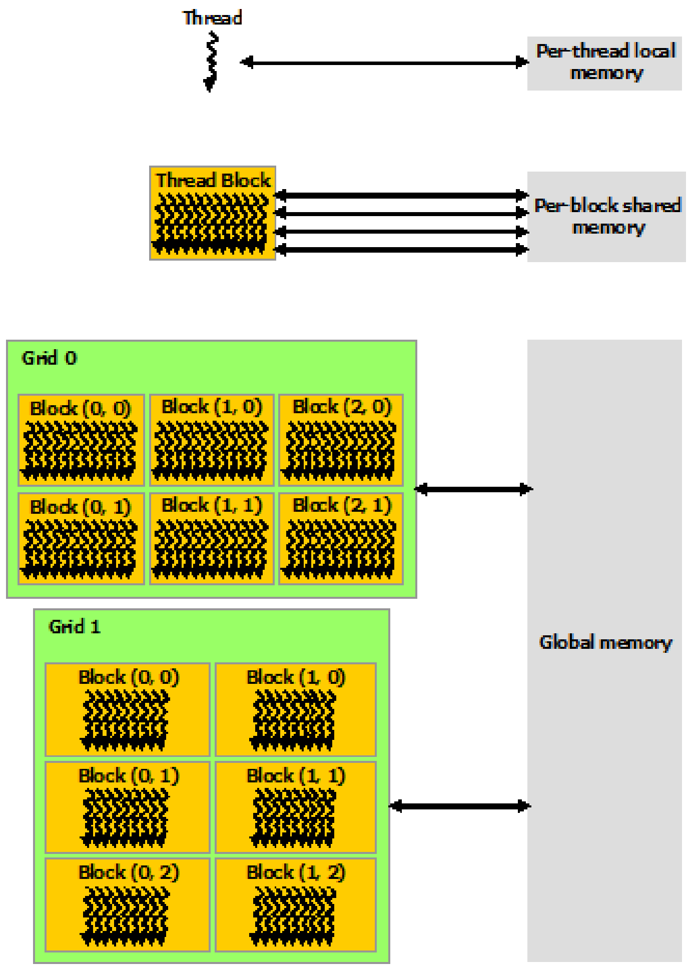


Figure 2.5: CUDA memory hierarchy (with permission from NVIDIA[17]).

icsGLRegisterBuffer(). Any time CUDA want access to a registered OpenGL resource, it has to be mapped to CUDA by calling cudaGraphicsMapResources(). As long as an OpenGL resource is mapped to CUDA, accessing it with OpenGL produces undefined results. When CUDA has done its work on the resource, it therefore has to be unmapped by calling cudaGraphicsUnmapResources() before use by OpenGL. Before freeing the buffer, it should be unregistered by calling cudaGLUnregisterBufferObject().

2.5 Perlin Noise

Perlin noise is a technique for generating natural appearing procedural textures. It was first presented at SIGGRAPH 1985 by Ken Perlin[18], who later won an

Academy Award for Technical Achievement for its development[20]. Perlin noise has proven itself to be extremely useful, also outside the realm of procedural textures. In addition to the generation of textures, Perlin noise can be used to create realistic landscapes, to improve and blend between of existing textures, to produce smooth animation patterns, and a whole lot more, and new uses for it are being found all the time. A sphere textured with Perlin noise in its simplest form is pictures in Figure 2.6.

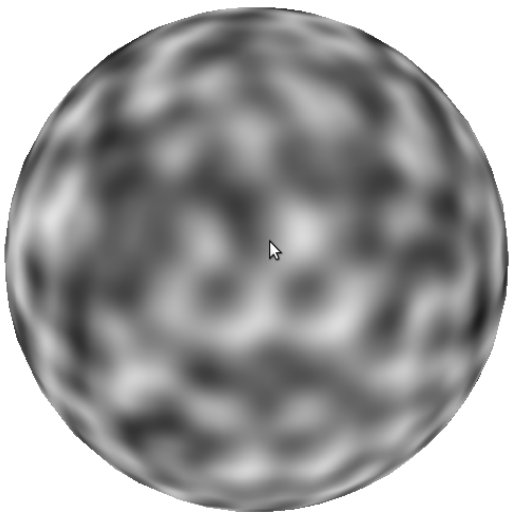


Figure 2.6: Perlin noise.

2.5.1 Algorithm

Compared to how powerful its results are, the basic algorithm to create Perlin noise is surprisingly simple. Given an input point P of n dimensions, look up all surrounding grid points. In n dimensions there are 2^n surrounding grid points, giving the algorithm a complexity of $O(2^n)$. For each of these points, choose a pseudo-random gradient vector G. The gradient vector for a specific grid point has to be the exact same every time for the returned noise to be correct. These gradient vectors are then interpolated between using a blend function to give a smooth final result.

This blend function was originally proposed to be the Hermite blending function $3t^2 - 2t^3$, but in his 2002 article “Improving Noise”[19] Ken Perlin suggested the function $6t^5 - 15t^4 + 10t^3$, Figure 2.7. This new function has the highly desirable quality of having a zero second derivative at its endpoints, which makes the noise function better suited for surface displacement and bump mapping. In

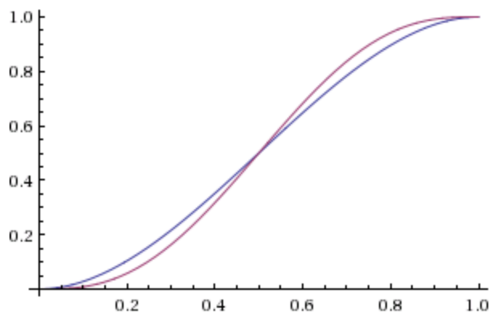


Figure 2.7: The blending functions. Red: $6t^5-15t^4+10t^3$, Blue: $3t^2-2t^3$.

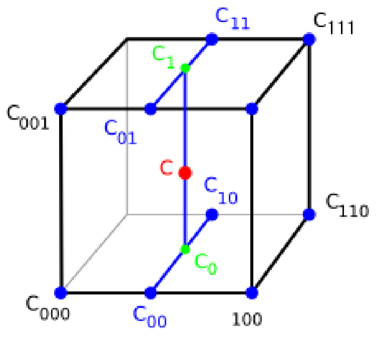


Figure 2.8: 3D interpolation.

the common 3D-case, interpolation is done by first selecting four connecting corner pairs and interpolating between them, then interpolating between the top and bottom pairs of resulting values, and ultimately interpolating between the two last values to get the final result, as illustrated by Figure 2.8.

The output Perlin noise yields is always the same given the same input. An important factor in achieving this is that the gradients are pseudo-random and not truly random. For 3D, Ken Perlin actually recommends to only use 12 gradients, the midpoints of each of the 12 edges of a cube centered on the origin. Associating each grid point with exactly one gradient is done by having a sufficiently large permutation table (Ken Perlin suggests 256), using the position of each grid point as the index to the table, and running the returned value through a modulo function to select the gradient.

Perlin noise is often used to create fractal noise. This is done by summing successively higher octaves of noise at successively lower amplitudes. An example of this can be seen in Figure X. The noise in the figure is generated using the GLSL function in Listing 2.1 with 5 octaves of noise with half the amplitude and double the frequency for each octave. By breaking the loop if the frequency is higher than the sampling rate, as seen in line 11-12, noise that is too complex to be adequately sampled is not generated. This helps avoid unnecessary computation and counteracts aliasing due to breaking the Nyquist limit.

2.6 The HPC-Lab Snow Simulator

In this section, the HPC-lab snow simulator is introduced. First, the history of the snow simulator is presented. Afterwards, some key aspects of the wind and snow simulation is described. And finally, an overview of the current code base is

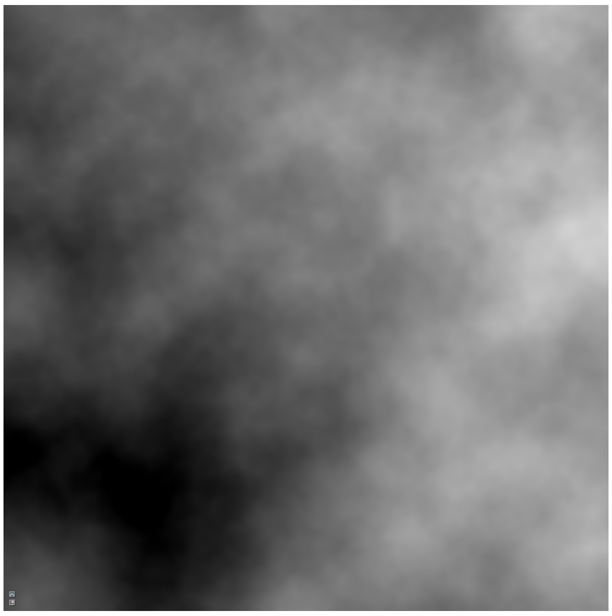


Figure 2.9: Fractal Perlin noise.

Listing 2.1 GLSL Perlin noise based fractional Brownian motion function.

```
float filtered_fBm (vec3 p, int octaves,
        float amplitude, float frequency,
        float lacunarity, float gain) {
    float sum = 0;
    float amp=amplitude, freq=frequency;
    float sampling_rate = 1/abs(dFdx(p.x));
    for (int i = 0;  i < octaves;  i += 1) {
        sum += amp *
            filteredsnoise(p, freq, sampling_rate);
        amp *= gain;  freq *= lacunarity;
        if (freq > sampling_rate)
            break;
    }
    return sum;
}
```

given.

2.6.1 History

The snow simulator has been an ongoing project at the HPC-lab at NTNU for a long time. It got its start in 2006, when Ingar Saltvik implemented the first version of the snow simulator [21]. This version was written to run with parallel threads, but only on the CPU. The frame rate obtained was respectable, but only when run with tens of thousands of particles. A paper by Ingar Saltvik, Anne C. Elster, and Henrik R. Nagel was also published in connection with this thesis [10]. In 2009, Robin Eidissen ported the snow simulator to CUDA [7], and with great care put into optimization for GPU execution, the new performance was several orders of magnitude better than that of the CPU version.

Since then, several master theses and projects have been written about different aspects of the snow simulator. In 2009, Alexander Gjermundsen added the Lattice-Boltzmann method for simulating wind [8]. Joel Chelliah did some optimizations for the NVIDIA Fermi architecture in 2010 [6]. Both Jarle Erdal Steinsland [23] and Frederik Vestre [24] have ported the snow simulator to OpenCL as part of their theses, in 2010 and 2012 respectively. Hallgeir Lien's project in 2011 was the generation of roads in the terrain of the simulator [13]. Kjetil Babington worked on ray-tracing, tessellation, Perlin noise and other visual techniques in 2011 and 2012 [3, 4] to improve the rendering on the terrain and snow. There have also been some projects that have not contributed directly to the snow simulator, but that are still somewhat connected to it. In 2010, Holger Ludvigsen worked on ray tracing using NVIDIA OptiX and published an article in collaboration with Anne C. Elster where, among other things, snow crystal rendering was discussed [14] Another project associated with the snow simulator is \IeC {\O }ystein Eklund Krog thesis on smoothed-particle hydrodynamics avalanche simulation from 2010 [12], and the paper Fast GPU-based Fluid Simulations Using SPH by Krog and Elster [11]. That brings us up to today, where in parallel to this thesis, Magnus Mikalsen is porting the snow simulator to OpenACC.

In the following section, some of the concepts and methods that make the snow simulator tick will be introduced. This will only be to give an overview, however, as each of the topics covered here are explained in much more detail in previous theses where they played a larger role.

2.6.2 Snow Simulation

The model for movement of falling snowflakes is the same as described in Michael Aagaard and Dennis Lerche's master thesis from 2004 [1]. Each snowflake is modeled as a particle with a position and velocity, and there are four different forces

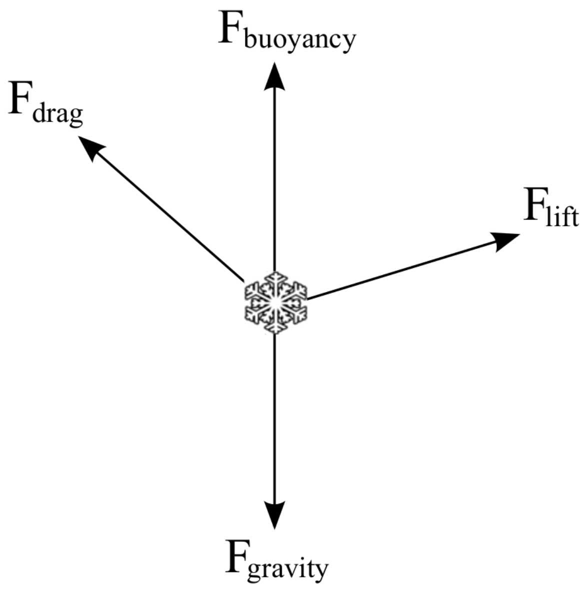


Figure 2.10: The different forces influencing the path of a snowflake.

that affect its path, as seen in Figure 2.10. Firstly, the gravity has a constant pull on the snowflakes in the downward direction. Secondly, there is an upwards buoyancy caused by the difference in density between objects and their surrounding medium. This force is seen as insignificant in this case, and is therefore omitted. Thirdly, the drag force created by the difference in velocity between the snowflake and the wind. Finally, there is a lift force caused by a phenomenon called vortex shedding, where nearby snowflakes and the snowflake itself affect the medium it falls through and causes an irregular and chaotic movement. In the snow simulator, this force is modeled as a vertical spiraling path. To accommodate this, each snowflakes has a radius and an angular velocity that defines the circular force.

Each time-step of the simulation, new positions and velocities are calculated for each individual snow particle. The updated velocity depends on the particle's previous velocity, the gravity, drag, and lift, which is a function of the difference between the velocities of the wind and snow particle. After moving the particle to its new position based on the calculated new velocity, if its height is lower than the height map it is considered to have collided with the ground. What happens then is that the snow level of the terrain is increased according to the pattern in Figure 2.11 and the particles is moved to a random position at the top of the domain. If instead the new position of the particle is outside the domain to any of the sides, it is wrapped around to the opposite side.

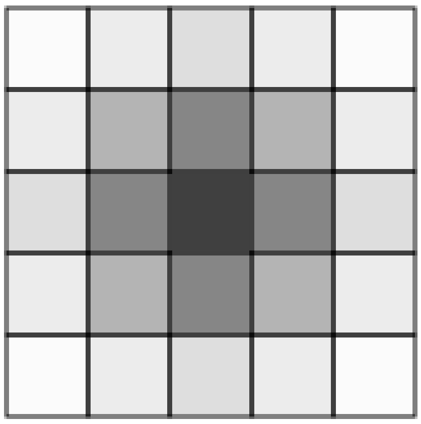


Figure 2.11: Snow level increase. The ratio of color intensity corresponds to the ratio of snow growth at the vertices around the point of collision.

2.6.3 Wind Simulation

The wind field plays an important role in the realistic simulation of falling snow, as it has a large impact on the movement of each particle. To model the wind field, computational fluid dynamics are used. Presented here is a very simple overview of the process, as the specifics of the simulation is not at the core of this thesis. A more thorough description of the physics and math involved can be found in Saltvik's thesis [21].

Navier-Stokes

The Navier-Stokes equations, in the general form, describes the motion of any fluid. In the case of wind simulation, the simplifying assumption that air is an incompressible fluid with a homogeneous density equal to one and zero viscosity, is made. With these simplifications, the wind model can be computed using incompressible Euler equations:

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \nabla p \tag{2.1}$$

$$\nabla \cdot \mathbf{u} = 0 \tag{2.2}$$

Solving these equations is a two step process; self-advection, then projection. In the snow simulator, the advection step is divided into two sub-steps; advection, where the advective forces are calculated and the velocity field is updated, and a step where the Poisson equation is solved, resulting in a pressure field.

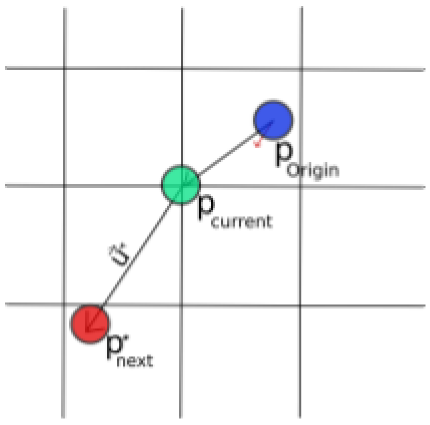


Figure 2.12: Self-advection.

Advection

Fluids in motion transports itself and other quantities along with the flow. This process is called advection. In Equation 2.1, the first term on the right hand side represents the self-advection of the fluid. We consider each grid point as a particle and use an inverted Euler's method to compute the next position as illustrated in Figure 2.12. The current velocity of each point in the grid is followed backwards in time and grid values for direction and magnitude closest to the calculated position are interpolated to get our new velocity. That velocity is then used to find the particle's next position.

Solve Poisson

By utilizing the Helmholtz-Hodge decomposition after combining Equation 2.1 and Equation 2.2, the result is a Poisson equation. This Poisson equation can be expressed as a system of linear equations on the form $Ap = b$, where A is a sparse matrix, p is the pressure field, and b is a vector of scalar values. The first step in solving the equation is to compute b by calculating the discrete divergence for all points.

To solve the Poisson equation, the successive over-relaxation (SOR) method is used, which is a variant of Gauss-Seidel interpolation based on [27]. The process is iterative, and is run five times with a decreasing relaxation factor. The relaxation factor that gives the best convergence rate depends on the problem, and the factor currently used, which is the result of empirical testing, starts at 1.7 and decreases to 1.1 over the iterations. A comprehensive explanation of this method can be found in [8].

Projection

After the pressure field has been computed by the Poisson solver, the only remaining step is projection. This is done by computing the discrete gradient of the pressure field and multiplying it component-wise with the result of the wind advection to give the final result. This result is then copied to texture memory to be used next time around, and in the snow simulation step.

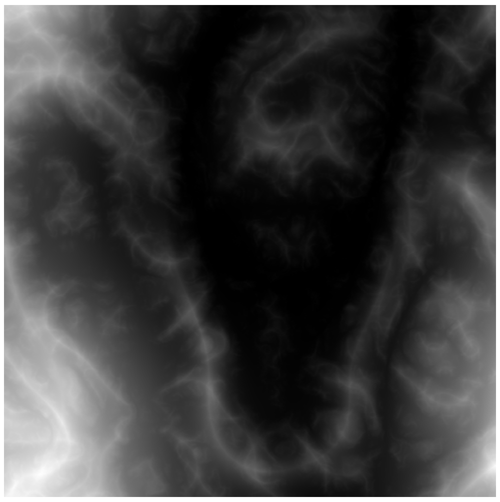
2.6.4 Terrain Interaction

The terrain of the snow simulator is represented by a grid of height values. This model allows for arbitrary terrains, with the one requirement that there are no overhangs or caves as the model cannot represent such formations. Representing terrain in this way is widespread, as it is simple yet powerful. One way of obtaining realistic height maps is to generate them using Perlin noise or similar techniques. Artificial terrains produced in this way can be convincingly realistic and produce astonishing landscapes, Figure 2.13a shows one such height map. Another, possibly more interesting source of height maps is real-world data. The National Aeronautics and Space Administration (NASA) has made their global digital elevation map, which contains height values covering most of our planet, available to the public [2]. With such an enormous dataset readily available, the snow simulator can take pretty much any location in the world as input, and make it snow on the landscape. An example of a real-world height map can be seen in Figure 2.13b.

As mentioned earlier, when a snow particle collides with the terrain, the snow level increases. This snow level is stored in the vertices of the terrain as a height value, and as the snow buildup increases in an area, it is rendered as snow instead of the underlying texture. The terrain also interacts with the wind field in the form of obstacles. At all the points the wind field is defined by, the terrain is sampled to check if the point is above or below it. If the point is below the terrain, it is considered an obstacle, which means that the wind considers it as a solid object.

2.6.5 Code Overview

At the beginning of this semester a lot of work was put in to the improvement the code base. This was done in collaboration with Magnus Mikalsen, who's master thesis also concerns the snow simulator. This spring cleaning was really needed, as the code base suffers from being a program of cobbled together parts from several students working on it at different times for their projects or theses. It was unstructured, with a lot of code that was never even called during execution, so getting an overview was near impossible. The logic of some of the more complex


(a) Computer generated height map.

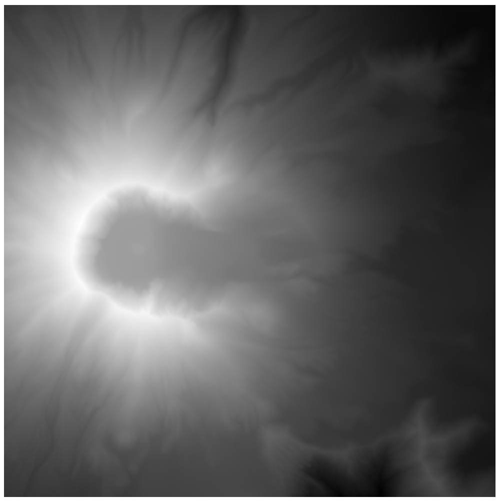

(b) Real-world height map from Mount St. Helens, Washington.

Figure 2.13: Height data visualizations. The height value corresponds to the lightness of the pixel.

parts of the program was difficult to get a grasp of because it was badly commented, in some cases not at all, and the variable names were not very descriptive.

With an aim to improve that, the program was broken down into it's essentials, and then put together again, piece by piece. More thought was put into giving the program a logical structure and making it more modular, so the different pieces such as rendering and simulation could be worked on independently. Readable and commented code was also an aspect of the program that was focused on, making the snow simulator more accessible for future students. With that in mind, the general structure of the current state of the program is presented in the following sections, and is given by the class diagram in Figure 2.14.

The core of the program, and where it all starts every time the program is executed, is the main class. During initialization, the main class creates instances of the terrain, wind, snow and rendering classes, and calls their initialization functions. The main class also contains the main loop, where there are call to the different steps in the simulation process, as well as the main rendering function, plus incrementation updating of the timers and counters.

The first step during initialization is the loading of the configuration file. This is handled by the config class, with assistance from its helper class conf. When the loading process is done, the config class holds important program variables, such as the size of the rendering window, the resolution of the wind field, and the number of snow particles.

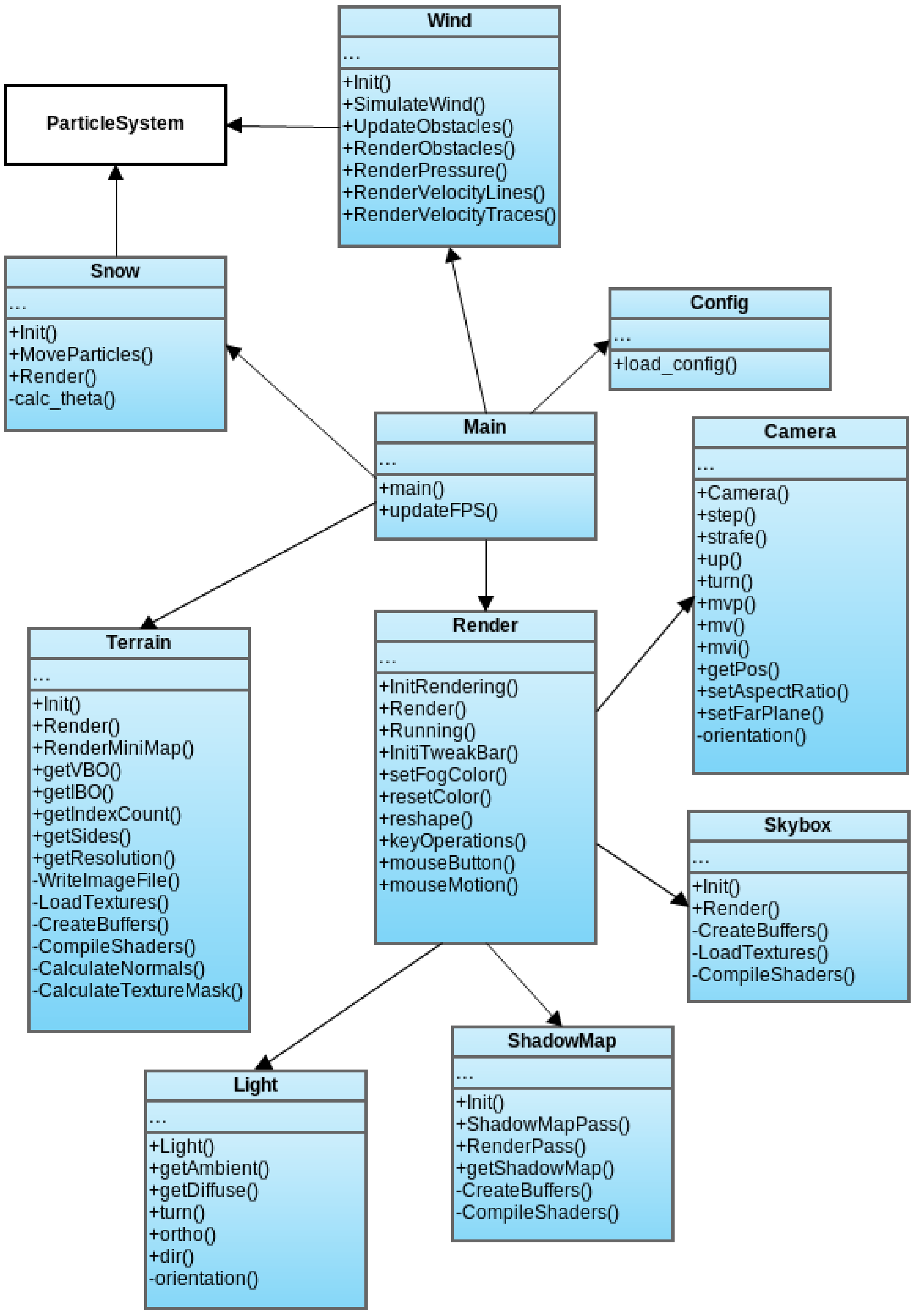


Figure 2.14: Class diagram.

The terrain, wind and snow class are all very similar. They each hold their own buffers, shaders and in some cases textures used in their respective rendering functions. Where they differ is in the type of information they have in their buffers. The terrain buffers has the height map information, the wind buffers hold obstacles, pressure and velocity, and the snow buffers hold the information of each snow particle. The wind and snow classes also have some functionality that the terrain class is lacking, namely functions for simulation, in the wind case updateObstacles() and simulateWind(), and in the snow case, moveParticles().

The render class is in charge of anything graphics-related. It keeps track of the camera position, the skybox, light and shadow map, and it also registers user input and updates the user interface. All calls to rendering within the main loop goes through the render class as well. The render functions of terrain, snow and wind are each called from the main render function within the renderer class. The specifics of each of these rendering functions are defined in their own rendering functions within their own class. Organizing the rendering-specific code in this way makes for a logical sequence of calls during execution, and a clean interface to the graphics code for further development.

The particle system part of the program contains all the actual simulation code that is run on the GPU. SnowSystem and WindSystem are intermediate classes between Snow and Wind and their respective CUDA kernels. They make sure all the relevant data is mapped and ready for the kernel to execute before it sets up the grid and blocks and call the CUDA kernel. After execution, it unmaps any buffers and does any cleanup necessary. An overview of the different classes that make up the CUDA particles system can be seen in Figure 2.15.

Another important new addition to the snow simulator is the new menu system, pictured in Figure X. The menus are created using the library AntTweakBar [ref], and they are initialized and handled by the render class. Having menus like this that enable configuration of simulation and rendering variables in real time makes interaction with the program instant, which is both useful when running the simulator as a demo and during development. Nearly every aspect of the simulator can be changed from the menus, from wind speed and direction and debug rendering, to what snow particle shader is active or which type of fog distance formula. In addition to the menus available during simulation, there is a startup menu that comes up each time the program starts. In this menu, all the variables that are locked during simulation can be set, such as what height map file to load and the number of particles and resolution of the wind field.

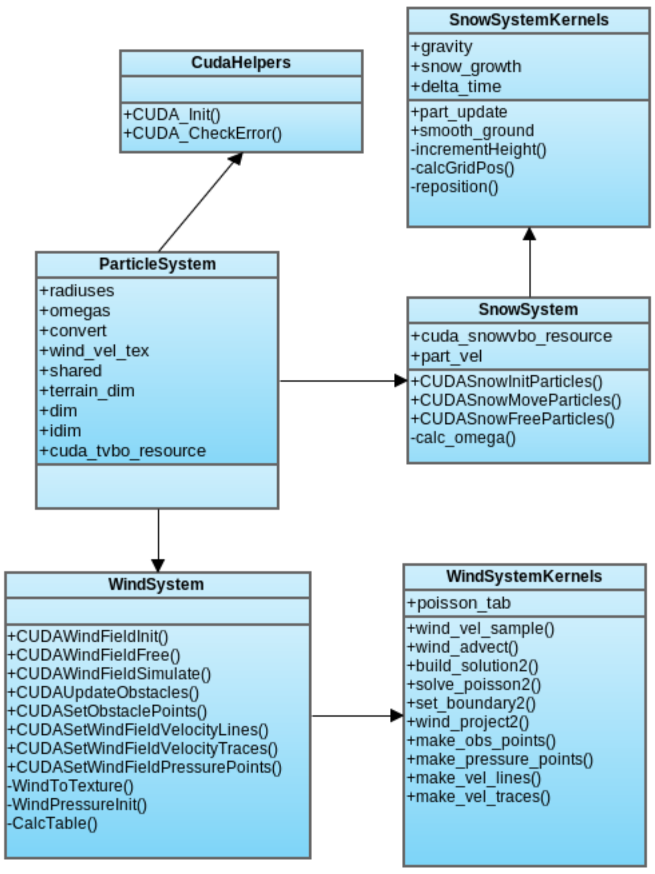
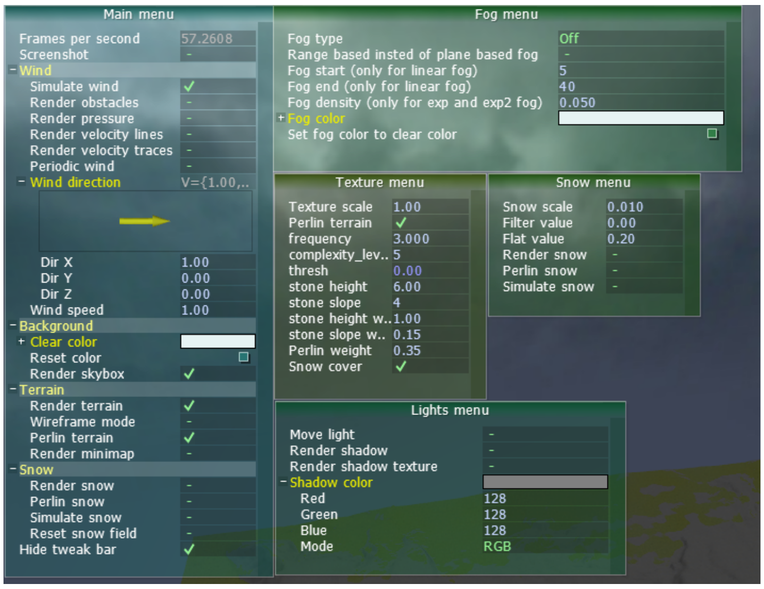


Figure 2.15: CUDA particle system class diagram.



Main menu

Frames per second	57.2608
Screenshot	-
- Wind	
Simulate wind	✓
Render obstacles	-
Render pressure	-
Render velocity lines	-
Render velocity traces	-
Periodic wind	-
- Wind direction	V={1.00,..
Dir X	1.00
Dir Y	0.00
Dir Z	0.00
Wind speed	1.00
- Background	
+ Clear color	⬜
Reset color	☐
Render skybox	✓
- Terrain	
Render terrain	✓
Wireframe mode	-
Perlin terrain	✓
Render minimap	-
- Snow	
Render snow	-
Perlin snow	-
Simulate snow	-
Reset snow field	-
Hide tweak bar	✓

Fog menu

Fog type	Off
Range based insted of plane based fog	-
Fog start (only for linear fog)	5
Fog end (only for linear fog)	40
Fog density (only for exp and exp2 fog)	0.050
+ Fog color	⬜
Set fog color to clear color	☐

Texture menu

Texture scale	1.00
Perlin terrain	✓
frequency	3.000
complexity_lev..	5
thresh	0.00
stone height	6.00
stone slope	4
stone height w..	1.00
stone slope w..	0.15
Perlin weight	0.35
Snow cover	✓

Snow menu

Snow scale	0.010
Filter value	0.00
Flat value	0.20
Render snow	-
Perlin snow	-
Simulate snow	-

Lights menu

Move light	-
Render shadow	-
Render shadow texture	-
- Shadow color	▨
Red	128
Green	128
Blue	128
Mode	RGB

Figure 2.16: AntTweakBar menus.

Chapter 3

Implementation

This chapter describes the implementation of each of the implemented rendering techniques. First, in Section 4.3, the terrain rendering is presented in detail, including the mesh, lighting, different texturing methods, and the explanation of a simple terrain shader. The implementation of shadow mapping, distance fog and the skybox is then covered Section 3.2, 3.3, and 3.4. Finally, Section 3.5 breaks down the snow particle rendering.

3.1 Terrain Rendering and Texturing

As previously mentioned, the height data the program is supplied with comes in the form of a grid of height values. Visualizing this data as a three-dimensional terrain is then a question of converting this data to a series of triangles, and rendering these triangles to screen with fitting textures. This process is not as straight forward as it sounds, and this section explains in detail how this is implemented in the snow simulator.

3.1.1 The Mesh

The first step is to convert the incoming data to a mesh of triangles. A height map is in essence just a series of scalar values - the height values, which gives us the height, or the y-value, of each vertex. The width and depth values, x and z, then have to be deducted from the location of the value in the list and the given resolution. For example, if we have a height map with resolution 8, and element 0 is the first value in our list, then the height values corresponding to each of the corners of the terrain will be given by element 0, element 7, element 56, and element 63.

The result of the previous step is a set of vertices, and the next step is to

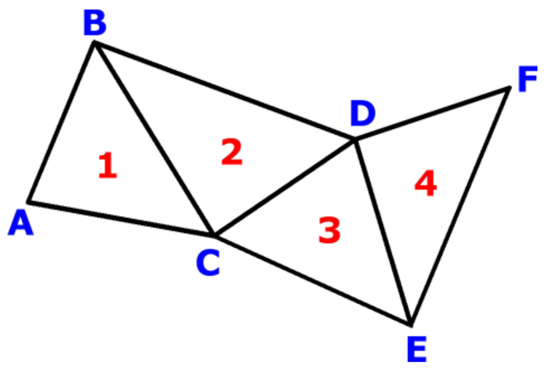


Figure 3.1: Triangle strip.

organize these vertices into triangles. The simplest way of doing this would be to draw one triangle at a time, with three vertices as input. Giving the vertices themselves as input is wasteful, however, because most the vertices will be shared between several triangles, and a simple draw function would require duplicate vertices. What is much more memory efficient is to use what is called an indexed draw. When using an indexed draw, the vertices used is decided by an index buffer, which contains pointers to vertices in the vertex buffer. That means that each vertex is only stored once in memory, and all that is needed to use it is a pointer. Another way to utilize memory more efficiently when rendering a triangle mesh is by using a triangle strip. The way a triangle strip works is by reusing the previous two vertices for every vertex in the call after the second to get a new triangle. Take for example the triangle strip in Figure3.1, there vertex A, B and C form triangle number 1. Vertex D then combines with vertex B and C to form triangle 2, triangle 3 in the formed by C, D and E, and 4 by D, E and F. By utilizing these techniques in our triangle mesh rendering we only have to store each vertex in memory once, and we reduce the number of pointers by close to a factor of three. The mesh rendered without textures, as a wire-frame, is seen in Figure 3.2.

3.1.2 Textures

Texturing an arbitrary terrain to make it look realistic is no easy task, but several techniques have been developed to make the job more manageable. It is still an enormous field, however, so a thesis with a relatively broad scope, like this one, can only hope to cover a small fraction of it.

There are several sources to textures. They can be generated using procedural texturing techniques, or they can be painted by a texture artist to fit the specific need of the scene at hand. Both these approaches require countless hours spent

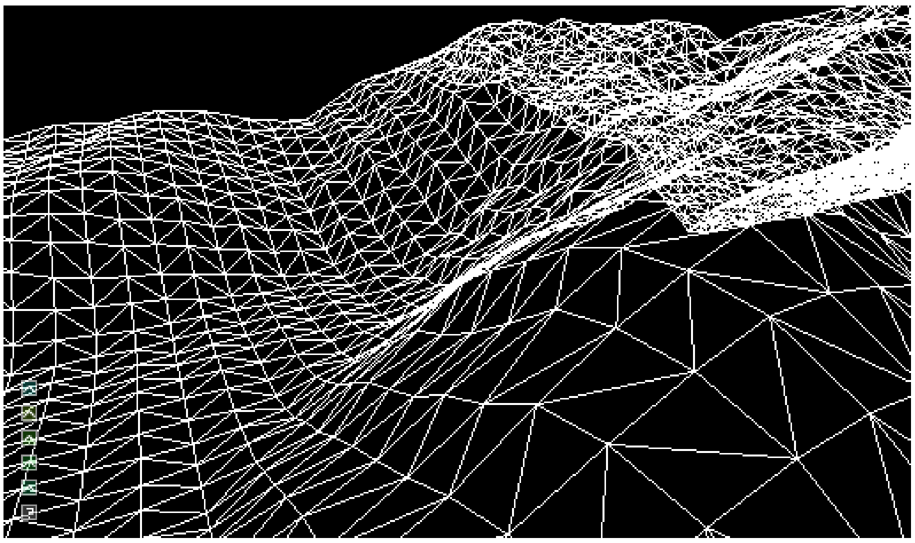


Figure 3.2: Terrain mesh.

by trained professionals, but luckily there are simpler ways to handle the problem. Although the result might not be as good as if it was done by a professional, finding fitting textures online, on websites such as cgtextures.com[5], is a more feasible option for a project like this. When trying to find the perfect texture, there are a few things to keep in mind. Firstly, the texture should not be very large as that would take up too much memory. It also needs to not be too small, as a certain level of detail is needed for a satisfactory result. Because of the intended use, the texture should be able to be tiled, meaning that connecting the texture edge to edge produces a seamless result. Another important characteristic is that the texture does not contain any striking visual elements, as that would make the result less visually pleasing because of obvious repetition.

3.1.3 Lighting

The lighting model used in the snow simulator is lambertian reflectance. It modeles diffuse reflection, which means the brightness is only dependent on the incident angle of the light and the direction of the normal vector of the surface. This makes it cheap to compute in real-time, as the brightness is the same regardless of the positions of the observer. In addition to being computationally cheap, it models the way light reflects in a terrain setting in the real world quite well. Although snow has some shimmering because of its crystal form, and grass has some specular properties that gives it some highlights, most of the light reflected off these surfaces will be diffuse, and in the case of rock, almost all of its reflected light is diffuse. This is because by a combination of their rough surfaces and subsurface scattering, which causes the light that hits it scatter in all directions.

As mentioned above, the two factors of the brightness of a surface are the properties of the light, namely incident angle and intensity, and the normal vector of the surface. The light properties are given during initialization and are stored in the light class. These variables can change during execution of the program, and are therefore uploaded to the graphics device every frame in the form of a uniform. The surface normals need to be computed from the mesh, and are stored in each of the vertices. Geometrically, attributing a vertex with a normal is not strictly correct, as a vertex only represents a point, and points do not have a surface. Representing the geometry in this way is a useful simplification for rendering, however, and the way the vertex normals are computed is by adding together the normals of all the connecting triangles and then normalizing the result. During rendering, these normals are then interpolated between in a technique called Phong shading, and the final brightness of the surface is calculated in each pixel.

3.1.4 Simple Shader

This is a good time to look at the source code for the most basic terrain shaders to see how these techniques are used in the snow simulator.

The most important function of a vertex shader is to make sure the incoming vertex is transformed into the correct coordinate system. In the case of the basic snow simulator terrain vertex shader, seen in Appendix A, the incoming vertex has a w-coordinate that represents the snow field. This has to first be added to the y-coordinate before the transformation because the total height at this point is the sum of the height of the terrain and the snow height. The snow height also has to be passed on to the fragment shader in order to render the correct color based on whether the terrain is covered by snow or not. After all this is done, the model view projection matrix is finally multiplied by the homogeneous vertex position vector. The final operation of this vertex shader is to normalize the vertex normal before passing it to the fragment shader. This is done to ensure that the interpolation between normals is actually based on normalized vectors and not weighted one way or the other.

The simple fragment shader for terrain rendering, is responsible for returning the correct color value for its pixel. To calculate this color, there are two important factors, texture and light. When it comes to texture, there are three different types of material the pixel could represent, snow, grass and rock. The terrain is considered to be completely covered in snow if its snow height is over 0.01. Between a snow height of 0 and 0.01, the snow blending factor is defined as the smooth Hermite interpolation of the GLSL smoothstep function[9], as seen in line 46 in Appendix A. By using a flat white color instead of sampling a snow texture to represent snow, one texture sampling operation is avoided, giving a slight speedup

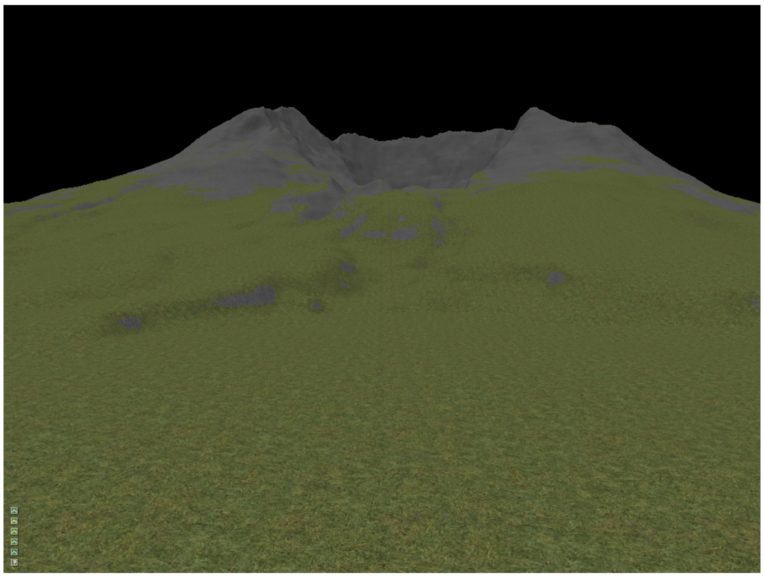


Figure 3.3: Terrain rendered with simple shaders.

for minimal cost. If the terrain is not covered completely by snow, the grass and rock textures have to be sampled using the scaled interpolated position returned by the vertex shader. These textures are then blended together based on the height and slope of the terrain; if the terrain is either high enough or in a steep enough slope it will be rendered with a rock texture, leaving the flat and low terrain having a grass texture. Then if the area is partly covered by snow, a factor of white is blended in.

A light factor is then computed based on the direction of the light and the normal vector of the fragment. The way this is done is by taking the dot product of the normalized interpolated normal vector and the direction of the light. This operation will in effect return the cosine of the angle between the two, because the length of both of the vectors is one. That means that the only time the fragment gets any diffuse light is if this angle is between -90 and 90 degrees, and the most light if the angle is 0. Any other case would return a value of 0 or less, resulting in no diffuse light. The fragment would not be completely black, however, as that would leave large portions of the terrain invisible. There is instead an ambient factor to the light, meaning that even the darkest areas have some color. A screen capture of a terrain rendering using the simple terrain shaders is seen in Figure 4.5a.

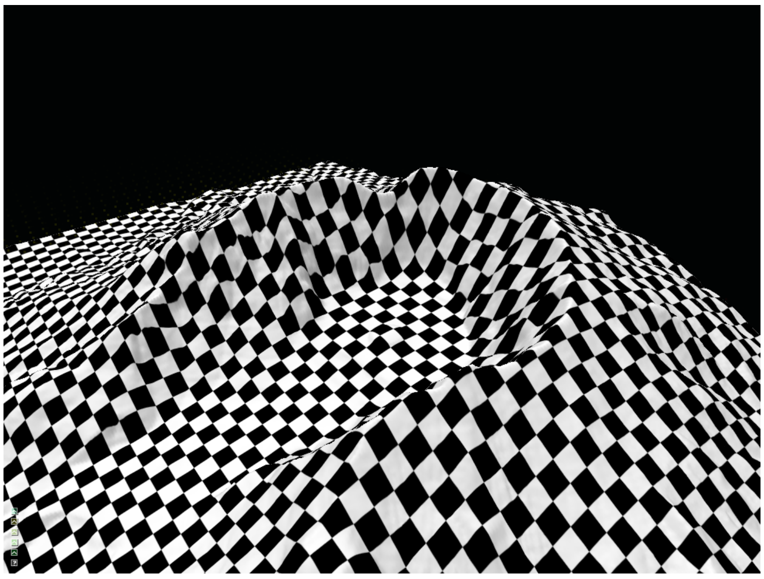


Figure 3.4: Texture stretching.

3.1.5 Triplanar Texturing

In the simple terrain shaders, the textures are only mapped to the terrain in the xz plane. This is unproblematic as long as the terrain is relatively flat. However, if the terrain contains steep slopes, it could lead to aliasing due to stretching, as seen in Figure 3.4 where the checker pattern is distorted by the slope of the terrain. One way to fix this problem is to use a technique called triplanar texturing.

Triplanar texturing works by mapping the textures not only in the xz plane, but also in the xy and yz plane based on the slope of the terrain, illustrated in Figure 3.5, where the three projection planes are colored green (xz), blue (xy), and red (zy). In the most extreme case, where the surface is vertical, instead of having a horribly stretched texture, the texture will be projected flat onto the surface. Between the extremities, the different projections are blended to give smooth transitions, as there would otherwise be undesirable aliasing due to sharp edges between them. The technique is implemented in the complex terrain shaders as seen in B, line 108-112. This implementation is based on an example given in chapter 1 of GPU Gems 3 [15].

3.1.6 Scalar Mixing

Scalar mixing is a relatively low-cost technique for decreasing apparent tiling. By blending together the same texture with different coordinate sizes, the repetition is somewhat broken up, and is no longer as easily noticeable by the human eye.

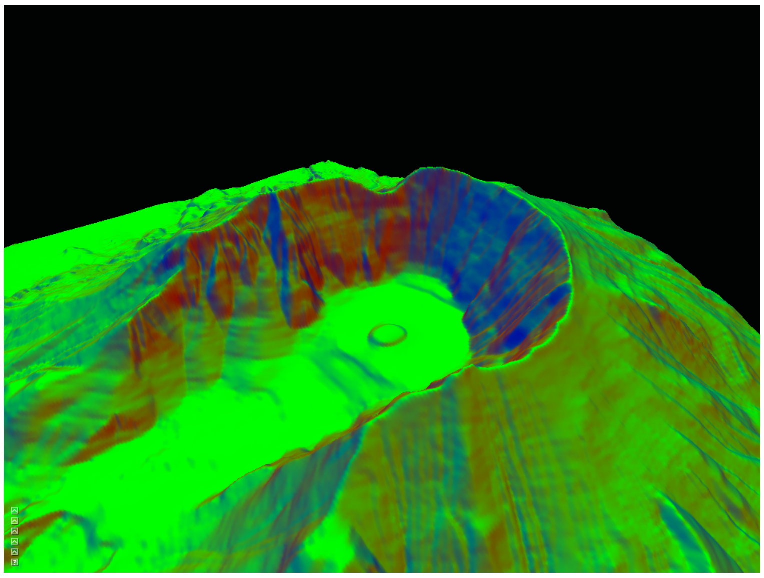


Figure 3.5: Triplanar texturing.

Because the textures being blended are actually the same texture, but with different coordinates, the method does not require more memory than simple texturing, only more sampling of the same texture.

The method is best suited for use on materials that contain mostly smaller detailing, such as grass, sand or dirt. Striking visual details in the texture is also detrimental to the visual result as it can shine through the mixing, so finding fitting textures is essential. This also makes surfaces such as rocks and other textures that contain deep markings to not look natural using this technique.

The most basic way to implement scalar mixing is by scaling the texture coordinate by a constant factor. A good such factor is four, and simply mixing a texture with itself scaled by 0.25 gives a surprisingly good result. By using a negative scaling factor, the texture will in effect get flipped, and this is an easy way to get even better results without doing any extra work. Another way to improve upon the technique is to also rotate the texture by some amount. This rotation makes it even harder to pick out any repetitions, and the final result will appear even more natural.

3.1.7 Perlin Noise

Another way to break up the repetitions in the tiled textures is to shift the hue and saturation of the color in areas specified using Perlin noise. In the real-world, there will always be some variations in color in large areas with the same type of surface. In a grassy field for example, there will always be darker and lighter spots, and

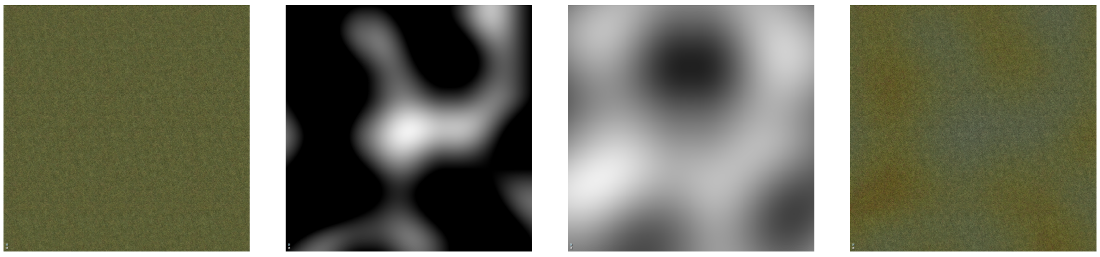

(a) Base texture. (b) Hue noise. (c) Saturation noise. (d) Resulting texture.

Figure 3.6: Hue and saturation shift using Perlin noise.

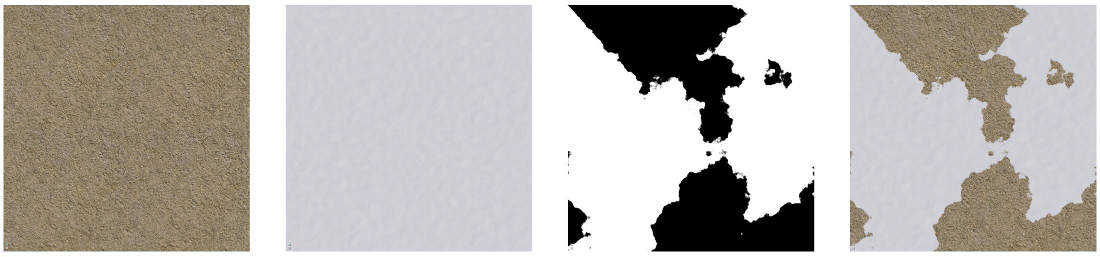

(a) Soil texture. (b) Snow texture (c) Noise function. (d) Blended texture.

Figure 3.7: Texture blending using Perlin noise.

areas that have a more red- or yellowish color. This type of effect can be imitated by shifting the hue and saturation of the texture based on a noise function. Figure 3.6 illustrates this concept by taking an area textured with a tiled grass texture, two channels of Perlin noise, and shifting the hue and saturation of the texture color in the areas specified by the noise.

The default color representation in GLSL is RGB. Therefore, before shifting the hue or saturation of a color, it has to first be converted into the HSV representation. HSV is a different way to represent RGB colors, composed of three values corresponding to the hue, saturation and value of the color. After converting the color to HSV, changing the hue and saturation is as easy as scaling or shifting the individual values. When the color values have been changed, the color has to be converted back to the standard RGB representation, because that is the representation the values are interpreted as in the framebuffer.

Perlin noise can also be utilized to improve the way the terrain is textured by combining it with the height and slope values when calculating the blending of different textures. This gives the transition between textures a pseudo-random and more natural look. Figure 3.7 shows how this can be done by using a soil and a snow texture and blending between them based on a complex noise function.

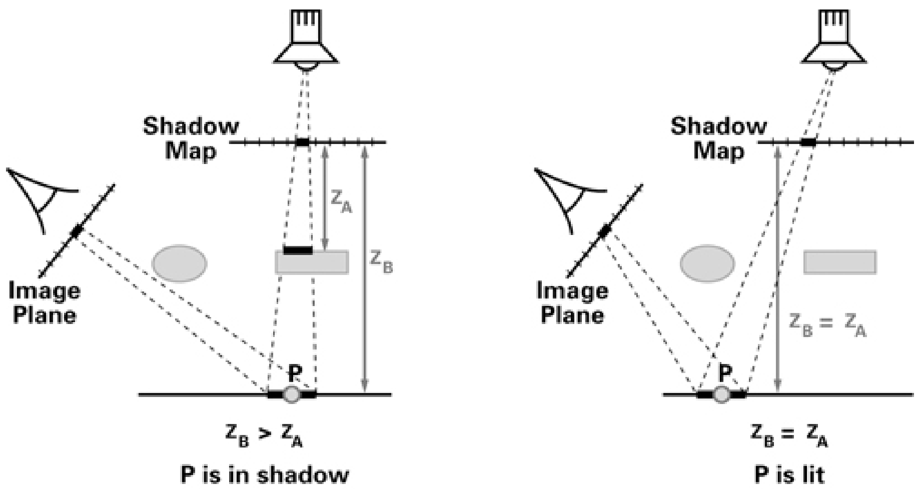


Figure 3.8: The Shadow Mapping Depth Comparison. (with permission from NVIDIA[16]).

3.2 Shadow Mapping

Shadow mapping is an inexpensive method for adding shadows to a 3D scene. It was first introduced by Lance Williams in his 1978 article "Casting curved shadows on curved surfaces" [26], and it is still a popular shadowing technique for both real-time applications and for computer generated special effects. While it is not as accurate as other techniques, such as shadow volumes, it is faster and does not require any preprocessing, which means it is better suited for dynamic scenes where the geometry changes over time.

The way the technique works is by creating a depth image from the point of view of the light source in an extra rendering pass before rendering the final image. During rendering, the distance of each fragment from the light source is then compared to this depth image to decide if it is to be shadowed or not. Figure 3.8 illustrates this concept, where in the figure to the left, the distance from the light source and the value in the depth map differs, meaning the fragment is in the shadow, and in the figure to the right, the two values are the same, meaning that the fragment should be lit.

3.2.1 Shadow map pass

In the case of the snow simulator, the only light source is the sun, which is simplified to be considered infinitely far away. To emulate this, the projection used in the rendering pass where the shadow map is computed is an orthographic projection instead of perspective projection. This orthographic projection is optimized to

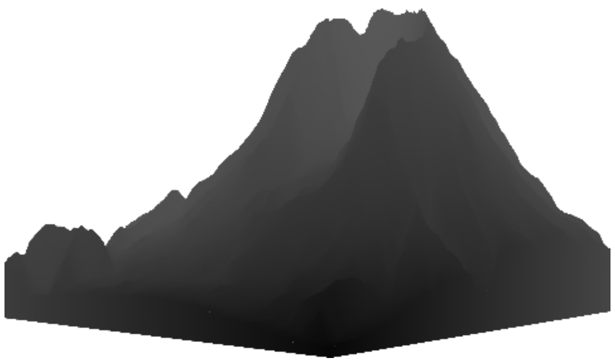


Figure 3.9: Gray-scale rendering of a depth image used for shadow mapping.

only cover the area needed to fit the bounding box of the terrain. Having provided the shadow map shader with the correct model view projection matrix, the actual computation of the z-buffer is quite simple. The height of the snow field still has to be taken into account, but other than that, the vertex shader is the simplest possible. A fragment shader is also not needed, because the only output we care about is the depth value. In addition to rendering the terrain mesh, the shadow map also has to render the bottom edges of the bounding box to make sure the terrain is not lighted from below when the angle of incident is steep.

The output of the shadow map pass has to be handled in a special way, since it is not to be rendered to screen as it normally would. Instead, the depth buffer has to be written to a texture to be used in the second pass. The way this is done is by using what is called a framebuffer object and attaching a texture to it. Before running each shadow map pass, the framebuffer object then has to be bound to the framebuffer, and when the pass is done, the framebuffer object has to be unbound again. The depth texture is then ready to be used as input to the terrain rendering pass. The gray-scale rendering of one of these shadow maps is seen in Figure 3.9.

3.2.2 Rendering pass

In the rendering pass, the coordinate of the terrain from the point of view of the light source has to be computed in addition to the normal model view projection in the vertex shader, see Appendix B. This coordinate is needed to sample the shadow texture at the correct position. When comparing the distance and the depth buffer, a bias has to be added to counteract shadow acne aliasing. If the bias is not added, some of the fragments would incorrectly be considered to be in the shadow because the resolution of the depth map is limited. Figure 3.10 illustrates this nicely, where the three pixels are sampled at different locations, causing a striped pattern.

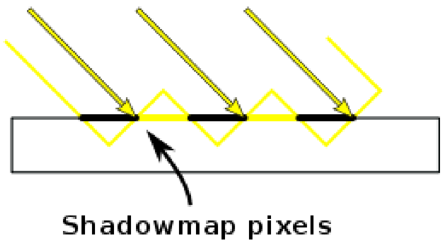


Figure 3.10: Shadow acne.

3.2.3 Anti-aliasing

Although the shadows produced by shadow mapping in the snow simulator looks smooth from a distance, once the camera is close to the shadowed area, aliasing due to the limited resolution of the shadow map texture quickly become apparent, as shown in Figure 3.11. There are several methods to counteract this, but there was sadly not enough time to implement any of them. As it stands, the shadow map implementation of the snow simulator is more just a proof of concept that anything, as the visual result are less than ideal. A relatively simple technique for shadow map anti-aliasing is percentage-closer filtering, where the depth texture is sampled several times with a statistical distribution to minimize the effects of the pixelated shadow and create a smoother transition. Another, more complex, technique is cascaded shadow maps, where several depth maps are created at different resolutions dependent on the view frustum, with higher resolutions closer to the camera, thereby making the closest shadows less jagged.

3.3 Distance Fog

Fog is a natural phenomenon that obscures visibility and makes details in the distance difficult to make out. This effect is caused by the scattering and absorption of light in the minute particles of water that constitutes fog. This effect can be simulated in computer graphics using some simple techniques, to great effect. Without a fog effect, computer graphics scenes can appear unrealistically sharp. This is especially true for large outdoor scenes. Adding a distance fog when rendering the terrain of the snow simulator can increase the realism of the scene, especially because heavy snow obscures vision of the distance in a similar way to

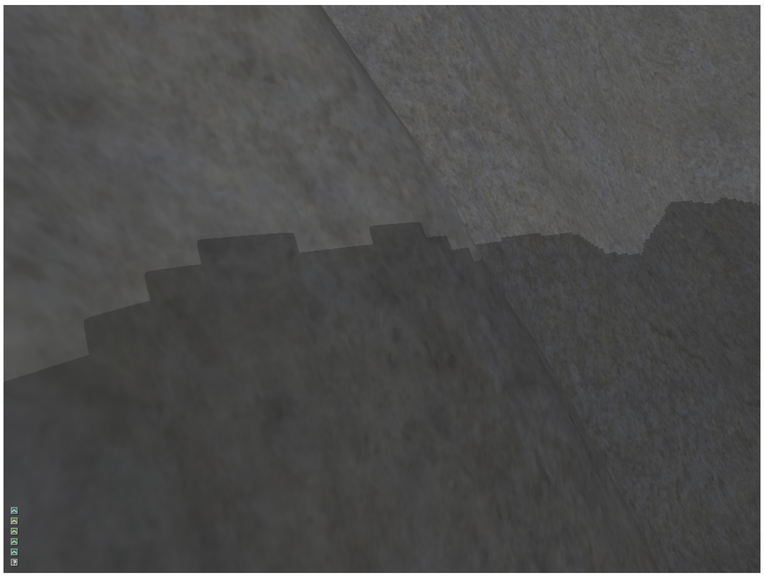


Figure 3.11: Shadow aliasing.

the effect of fog, and the weather is also often a bit hazy when it is snowing. It can also improve the performance by enabling culling of fully fogged objects.

3.3.1 Visibility drop off formulae

The basic idea behind distance fog is that the further an object is from the camera, the more the more fogged it should be. The fog is calculated in the fragment shader, where a fog color is blended with the final color based on some fog factor decided by the distance from the camera. The formulae used to decide the rate at which the fog increases differ. In the snow simulator, three different equations for calculating the fog cover have been implemented: linear, exponential and exponential squared, all plotted in Figure 3.12.

Linear

$$fog = \frac{end - distance}{end - start}$$

The linear equation interpolates linearly between the distances ‘fog start’ and ‘fog end’. Before ‘fog start’ there is no fog’, and after ‘fog end’ nothing but fog is rendered.

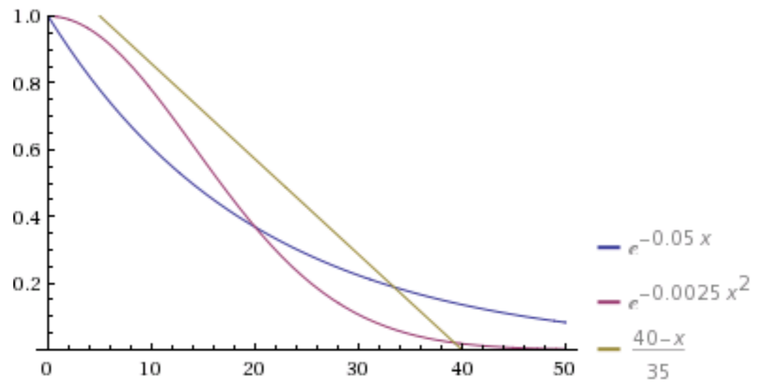


Figure 3.12: Drop off formulae. 'fog start' = 5, 'fog end' = 40, 'fog density' = 0.05.

Exponential (insert equation here)

$$fog = e^{-density*distance}$$

The visibility drop off of the exponential equation is more rapid than that of the linear one. The rate is controlled by the 'fog density' variable, the higher the value, the quicker the drop off.

Exponential squared

$$fog = e^{-(density*distance)^2}$$

This is the same equation as the exponential one, except the exponent is squared, and the drop off is even quicker. Again, 'fog density' controls the rate of change.

3.3.2 Distance Calculation

There are also different ways to measure distance. The cheapest, and most common way of doing it is to use the distance from the viewpoint in the z-direction, the depth. This is called plane based distance, as a given distance value forms a plane in front to the viewpoint. Because the calculated plane is not the same distance from the camera at all points, but differs from fragment to fragment, the technique can lead to visual artifacts when the observer turns. Objects in the peripheral vision that were just outside the fog can be completely covered in fog if the observer rotates so the object is straight ahead.

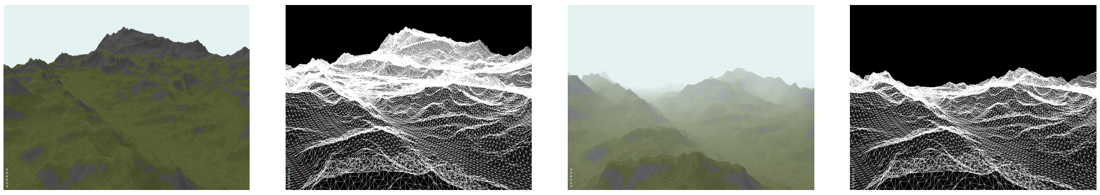

(a) Without fog. (b) Underlying geometry. (c) With fog. (d) Culled geometry.

Figure 3.13: Culling geometry hidden by fog.

The way to fix this would be to calculate the actual distance from the viewpoint instead of just the distance in the z-direction, a technique which is called range based fog. A given distance will then form a sphere around the viewpoint. The consequence of this is that whichever way the observer turns, all objects will be covered in a constant amount of fog. As seen in Appendix B, this distance has to be calculated in the vertex shader and then passed to the fragment shader where the actual fog calculation takes place.

3.3.3 Culling

A neat quality of distance fog is that it enables the culling of objects that are completely covered in fog. In the of linear drop off, the far clipping distance can be set to the same value as 'fog end', as everything behind that would not be rendered anyway. By using this optimization, the performance gain can be enormous, and in some older games it was a necessity to be able to fill the real-time demand while still having complex outdoor scenes. Figure 3.13 shows an example of how much geometry can be saved in the snow simulator when using a thick fog.

3.4 Skybox

A skybox is a technique for making outdoor scenes look bigger than they really are. The idea behind it is to render a cube centered around the viewpoint, and texture this cube with an image of the distant terrain and the sky. The cube will always be stationary with respect to the viewer, so the viewer can never reach the horizon. This imitates the real world, where objects far in the distance, such as mountains and clouds, appear stationary when the observer moves relatively short distances.

Texturing the skybox is done with cube maps consisting of six individual textures that are designed to projected onto each of the faces of a cube. The textures of the cube map have to be carefully aligned to produce a seamless result, or the

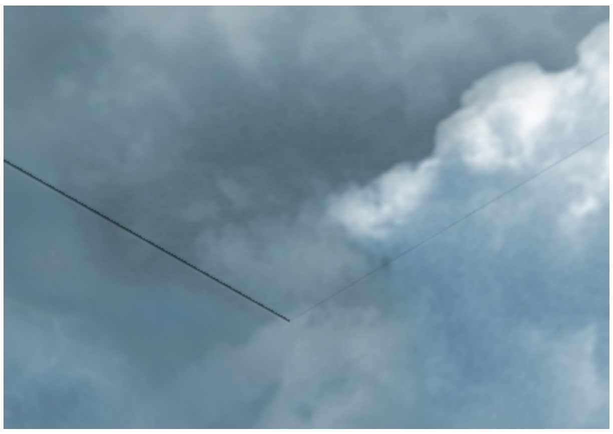


Figure 3.14: The corner of a skybox where the cube map is not aligned correctly, causing a seam.

geometry of the cube will be visible to the observer, as seen in Figure 3.14. A skybox texture is often a combination of the sky and some sort of terrain that acts like an extension of the environment.

Because the technique has a very low performance cost it is often used in games that need to be able to render large outdoor scenes while still filling the real-time requirement. Some recent games also improves upon the concept by changing the texture over time, either as a result of the character moving, or to simulate the passing of time or a change in weather. If the terrain always covers the ground completely and the bottom of the skybox can never be seen, a small optimization can be made where only the top half of the cube is actually in use. A technique very similar, but not as widespread to skybox also exists, called skydome, where the only difference is that a sphere is used instead of a cube.

The snow simulator utilizes the library Simple OpenGL Image Library (SOIL) [22] to handle the loading and management of its cube maps. SOIL has some very handy functions made especially for doing exactly that, which significantly simplifies the implementation of the skybox. The cube itself is set up as straightforward as possible around the origin, and then moved to be centered around the camera during rendering. Even though the skybox looks infinitely big when rendered, its actual size is irrelevant. By setting the transformed depth value of each vertex to be equal to its w component, the perspective divide ensures that the final z value always will be 1.0. This means that the skybox always fails the depth test against the other objects in the scene, and it is therefore rendered at the very back, behind everything else.

3.5 Snow Rendering

The snow particles in the snow simulator are simplified to be represented as points in space. They do not have any volume or direction, and the only data in each particle that is relevant to rendering is their position, which updates every time step. Because there can be several million particles to render in a scene at any time, it is of the utmost importance that the rendering code is fast.

The quickest way to render a point would be to render it as a colored pixel. This approach would not create a very realistic visual result, however, as snowflakes that are closer to the viewpoint should take up a larger part of the screen than snowflakes that are further away, thus creating a sense of perspective. The most widespread way to achieve this, popular in both games and real time simulations, is to use a technique called billboarding.

3.5.1 Billboarding

The idea behind the billboarding technique is to render the point as a quad that always faces the camera, making the particles look the same from all directions. One way to accomplish this is by running each vertex through a geometry shader as part of their rendering. The billboarding snow shader can be seen in Appendix C. As seen there, the geometry shader takes points as input, and generates triangle strips as output. Triangle strips are used instead of quads as the GLSL geometry shader output formats are restricted to points, line strips or triangle strips.

To make the billboard alway face the camera, the direction between the point and the camera has to be calculated. From this vector, the vectors used to displace the different vertices are derived using cross products. Each of the four displaced vertices are then transformed by the model view projection matrix and given their respective texturing coordinates, before being passed on to the fragment shader. Using a geometry shader to do all the calculation necessary to generate the new vertices can be costly, but another method for producing almost exactly the same result is to render the snowflakes as point sprites.

Point sprites are an OpenGL feature that allows for points of different sizes that can be textured, defined by a single vertex. The point can be set to be any size, measured in pixels, in the GLSL vertex language built-in output variable gl_PointSize [9]. The size calculation, which in the case of geometry billboarding would be taken care of by the perspective projection, would therefore have to be taken care of by the vertex shader. As seen in Appendix D line 17-20, the way this is done in the snow simulator is by measuring the distance between the incoming vertex position and an offset after the perspective divide. The correct size of each snowflake is thereby found, giving a correct perspective. One weakness of the point sprite method is that the minimum size of a rendered point is one pixel. Particles

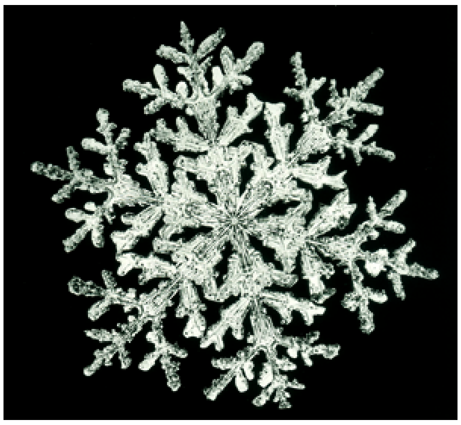


Figure 3.15: Snow crystal texture.

that would otherwise not occupy a whole pixel is therefore rendered incorrectly. The snow simulator handles this by fading distant snowflakes into the background by making them translucent, multiplying in an alpha factor to the final color.

3.5.2 Snow Texturing

Having produced produced an area of correct size for each particle, now comes the problem of texturing that area. The simplest way to do that would be to texture all the particles with the picture of a snow crystal, like the one in Figure 3.15. This is the method that has been used in previous versions of the snow simulator, and it works, but it has some big problems. Firstly, a falling particle of snow hardly ever consists of a single snow crystal. It is instead a collection of several snow crystals, connecting together into different shapes and in different sizes. That leads into the second problem, which is that using the same picture as the texture for all the snow particles means that they all look exactly the same which makes for a very unrealistic result. Using a collection of different textures, with a more accurate representation of what falling snow looks like, would begin to solve these problems, but that approach has issues. The number of different textures that can be used at a time is limited by the amount of memory available, and when performance is as important as it is in the case of the snow simulator, this number is relatively low. That means that each texture would have to be repeated several times when rendering a large number of particles, which is not desirable, so a different approach is needed.

Problems like this happens to be the perfect use case for procedural textures. The Perlin noise implementation described in Section 2.5 allows for generation of pseudo-random gradient noise that can be used to create a set of unique textures for each of the snow particles. Listing 3.1 shows the function used to generate snow

Listing 3.1 Procedural snow texture function.

```
float snow(vec3 pos) {
    float sampling_rate = 1/fwidth(pos.x);
    if (2.0 > sampling_rate)
    return flat_value;
    float noise = filtered_fBm(pos,
            5, 1.0, 2.0, 2.0, 0.5, sampling_rate);

    float x = pos.x*2-1;
    float y = pos.y*2-1;
    float dist = 1-(x*x+y*y);

    float weighted = ((dist*0.5)+(noise*0.5)-0.5)*2;

    return max(1-exp(-5*weighted),0.0);
}
```

textures in a fragment shader of the snow simulator. It combines five different octaves of noise of decreasing amplitudes to create a fractional Brownian motion and combines this fractional noise with the distance function from the center, which is then amplified to create the final texture. An example of a texture produced with this functions can be seen in Figure 3.16.

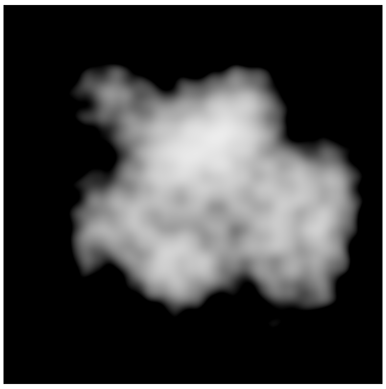


Figure 3.16: Procedural snow particle texture generated with Perlin noise.

Chapter 4

Results and Discussion

This chapter presents tests for the implemented rendering techniques and discusses their results. The speeds of the snow simulator when rendering and not rendering are compared in Section 4.2 to evaluate the cost of rendering. Different terrain shaders are tested at increasing terrain resolutions in Section 4.3, and in Section 4.4 the snow rendering is tested with a varying number of snow particles. Section 4.5 then evaluates the visual results of the terrain, snow particles, distance fog, the skybox, and shadow mapping.

4.1 Test Setup

All the tests in this chapter are run in a 64-bit Ubuntu 12.04 environment on a computer with an Intel® Core™ i7 CPU 930 processor, 12 GB of DDR3 memory and a GeForce GTX 480 GPU with NVIDIA driver version 310.32 running CUDA 5.0. The tests are only run on one system because the simulation part of the snow simulator is unchanged from previous versions. Comparisons of the snow simulator run on several different systems can be found in [4, 6, 8, 7]. The focus of this thesis is to improve the rendering code, both the realism of the visual output and its performance. Comparing the performance on different systems is therefore not going to be as interesting as studying the relative efficiency of the different implemented rendering techniques.

4.2 Simulation and Rendering

This test was run with 4.000.000 snow particles, a wind field with resolution 128x32x128 and the height map 'helens768.raw' with a resolution of 768x768. The terrain was rendered using the 'simple' shader and the snow particles were rendered as point sprites textured with procedural texturing. The window size was

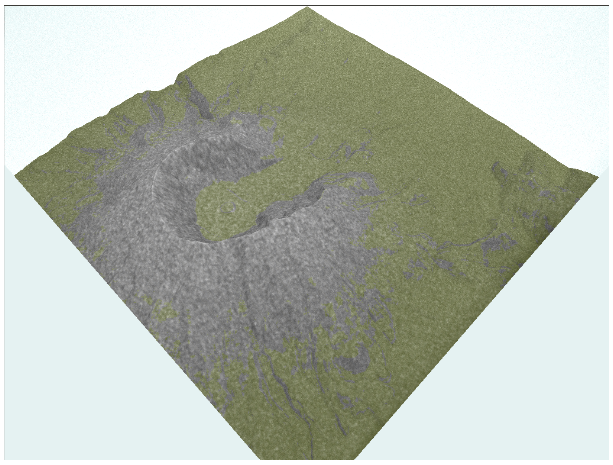


Figure 4.1: Camera positioned so the entire terrain and all the snow particles are rendered.

1024x768, and the camera was positioned so the entire terrain and all the snow particles are visible, as seen in Figure 4.1. In the ‘no rendering’ case, the rendering step of the snow simulator was skipped altogether, leaving the only code to be run the actual wind and snow simulation. In each case the simulator was left to run for 15 seconds.

Even when not rendering, the snow simulator measures the number of frames that would be rendered per second, or the simulation steps taken per second. In this test, when not rendering anything, the average frame rate was 72.602839, meaning that each frame took an average of 0.013773 seconds to compute, as seen in the above bar chart, Figure 4.2. When rendering, however, this frame rate dropped to an average of 29.596117, giving a seconds per frame value of about 0.033788.

Adding the rendering step to the simulator at this problem size therefore increases the average time taken to for each frame by a factor of nearly 2.5. This result emphasizes the fact that rendering is something that should not be taken lightly, and that it is of the utmost importance to have fast and effective rendering code. Increasing the problem size makes will increase the amount of time spent updating the wind field and snow particles positions relative to the time spent rendering, as can be observed in the tests below, and using more complex rendering techniques to visualize the simulation tilts this distribution in the opposite direction.

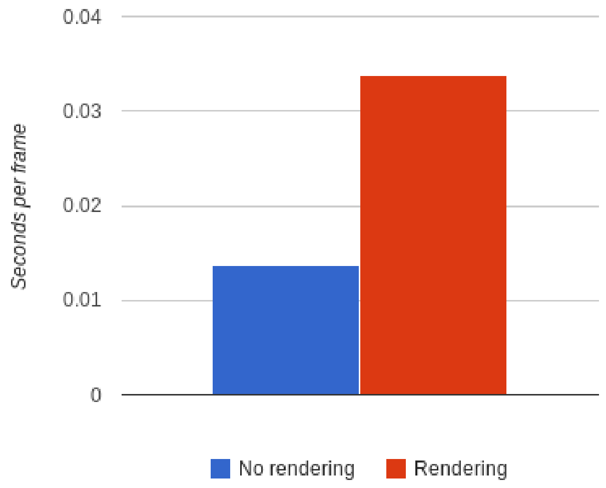


Figure 4.2: Comparison of frame rates when only running the simulation and when rendering.

4.3 Terrain Rendering

In this test the simulation was run with 4.000.000 snow particles and a wind field with resolution 128x32x128. The window size was 1024x768, and the camera was positioned so the entire terrain was within the view frame. The snow particles were rendered as point sprites textured with procedural textures. The height map used in this test was ‘trondheim4096.raw’ imported at different resolutions from 256x256 to 4096x4096. This test compares the different rendering techniques implemented for the rendering of the terrain by their average frame rate at different terrain resolutions. In the case using shadow mapping, the resolution of the shadow map was 2048x2048. In each case the simulator was left to run for 15 seconds.

The first thing that becomes clear from the results of this test, seen in Figure 4.3, is that the most limiting factor for how fast the terrain can be rendered is the number of vertices the terrain is defined by. In fact, after the terrain resolution passes the 2048x2048 point, all the rendering techniques except shadow mapping converge to have close to the same performance. Modern films run at a frame rate of 24 frames per second (fps), which is well within the bounds of what humans can process, measured to be 10-12 images per second. 24 fps is therefore a good threshold to aim for when analyzing the results. It can be observed that regardless of the rendering technique, this threshold is passed somewhere between a terrain resolution of 1024x1024 and 2048x2048.

The biggest difference between the simple terrain shader and the complex terrain shader, which implements triplanar projection, is the amount of times a texture is sampled. In triplanar texturing, each texture is sampled three times as

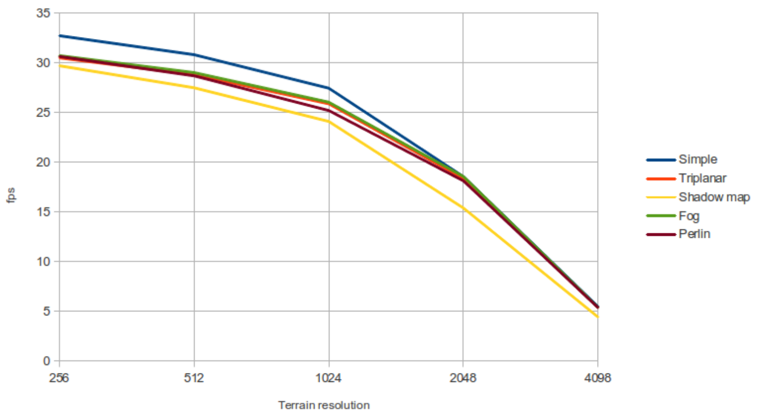


Figure 4.3: Terrain rendering test results.

much as with traditional texturing, one time for each plane the textures are projected onto the terrain with. This makes triplanar texturing more computationally expensive than simple texturing, which can be seen in the reduced frame rate observable in the graph.

The distance fog is implemented in the complex shader, which means it comes combination with triplanar texturing. The additional computation it adds to the rendering is insignificant, however, as can be seen in how the lines for triplanar and fog rendering are almost the exact same. In this test, the optimization where the geometry hidden by the fog is culled was disabled to show how much the fog itself influences the rendering speed. If this optimization was enabled, it would help reduce the amount of vertices that need to be rendered and thereby increasing the frame rate significantly. This speedup would be highly variable, however, as it is dependent on the position of the observer, the complexity of the geometry in its proximity, the thickness of the fog and formula used to calculate it, and many other factors.

The implementation of Perlin noise hue and saturation shifting as well as texture blending is significantly slower than the simple shader. Its decreased speed compared to the simple shader comes from its additional texture sampling, the computational cost of converting the colors back and forth between different representations, as well as combining several octaves of noise. The Perlin shader does not implement triplanar texture projection, but does clock in at about the same speed as the shader that does for all terrain resolutions. The combination of the two would improve the visual result, but at the cost of further decreased rendering speed.

Shadow mapping is, similarly to distance fog, implemented in the complex shader, on top of triplanar texturing. Compared to the other rendering techniques in this test, the results show that it is significantly slower, by what is close to a

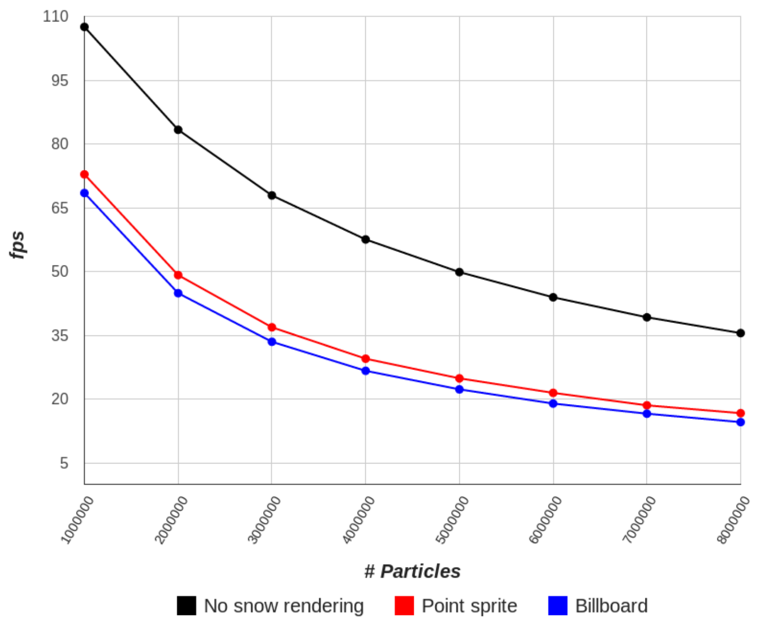


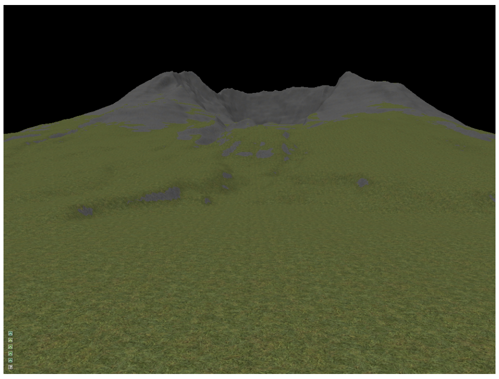
Figure 4.4: Snow rendering test results.

constant factor. This can be explained by the fact that shadow mapping requires an additional rendering pass before the terrain rendering. While implementing the improvements mentioned in Section 3.2 would further decrease the frame rate when using shadow mapping, it would not be by a large amount, because the most costly part of its calculation is the overhead connected with running a separate rendering pass.

4.4 Snow Rendering

This test is run with a varying number of snow particles to test the performance of different snow rendering techniques. The wind field resolution was set to 128x32x128 during this test. The window size was 1024x768, and the camera was positioned so the entire terrain and all the snow flakes was within the view frame. The height map used was ‘helens768.raw’ with a resolution of 768x768 and the terrain was rendering using the simple shader. In each case the simulator was left to run for 15 seconds and the average frames per second was measured.

In addition to testing with no rendering, rendering as point sprites, and billboard rendering, two different texturing methods were tested. Both procedural texturing with Perlin noise and texturing all the particles with the same image gave the same result, regardless of number of particles or if point sprites or billboard was used. Procedural texturing is therefore considered a strict upgrade from


(a) Terrain rendered with the simple shaders.

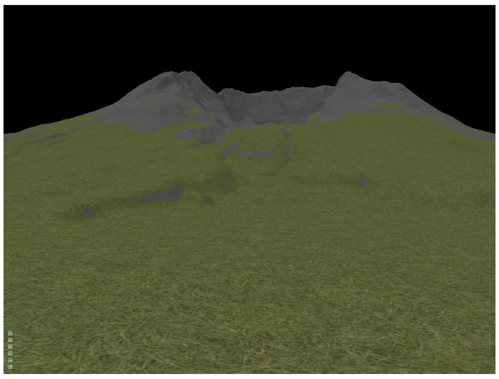

(b) Terrain rendered with the complex shaders.

Figure 4.5: Terrain renderings using different shaders.

the previous implementation because of its improved realism.

Rendering the snow particles is expensive. Compared to not rendering them, the average time taken to produce a frame is increased by a factor between 1.5 and 2, dependent on the method used and the number of particles when using this specific wind field and terrain. Finding the most effective rendering technique is therefore very important. As seen in the graph in Figure 4.4, using point sprites is significantly faster than using billboarding for rendering the snow particles. The 24 fps threshold is passed by billboarding at five million particles, while point sprites renders five million particles with a frame rate of close to 25. The most important factor in the speed difference is the number of vertices needed in the two methods. Point sprites are defined by only one vertex, while billboarding generates four vertices for every particle.

4.5 Visual Results

Figure 4.5a and 4.5b shows a terrain rendered with respectively the simple shaders (Appendix A), and the complex shaders (Appendix B). The difference between the terrain rendered with the simple and complex shader is not very noticeable. In fact, it is not the triplanar texturing that is most apparent, it is the scalar mixing of the textures to counteract repetition that makes the biggest difference. This height map in particular, but also most height maps produced from real-world data, does not have extreme enough slopes to make texture stretching a big problem. The difference in performance between the two shaders and the lack of visual improvement makes the simple shader much more appealing than the complex shader when it comes to terrain rendering in the snow simulator, because

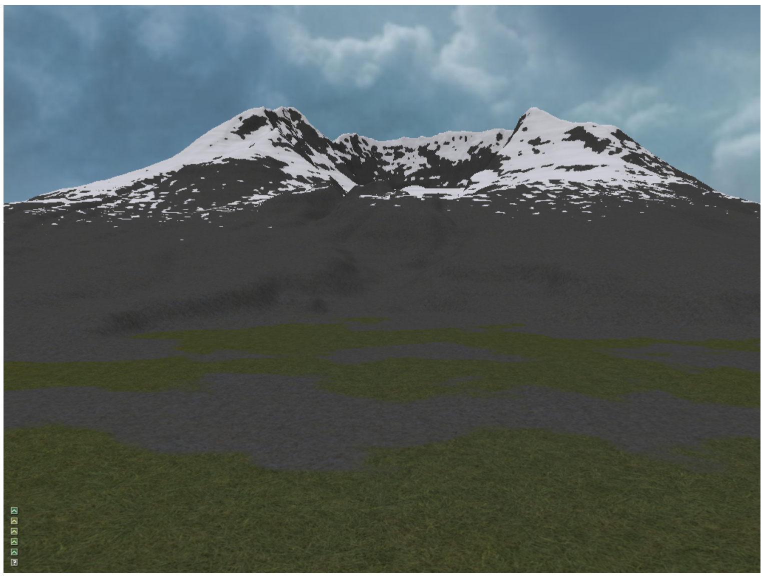


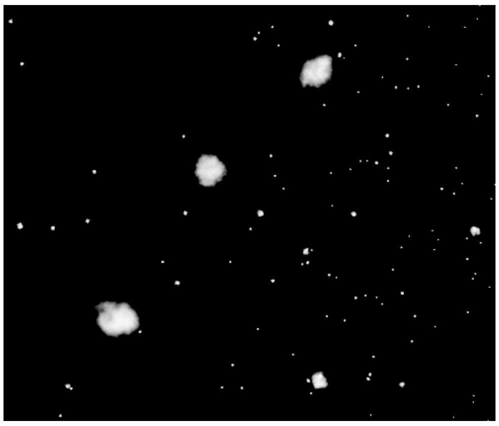
Figure 4.6: Terrain rendered with the Perlin shaders.

of the real-time requirement.

Figure 4.6 shows the same terrain rendered with the Perlin shader, and using the skybox as a background. The snow at the top of the mountain is not caused by the snow field, but is an artistic addition to make the rendering look more realistic. The distribution of grass, stone and snow is computed with Perlin noise, and makes for natural looking formations. Although it not apparent from this angle, Perlin noise is also used to adjust the hue and saturation values of the grass texture. These techniques drastically improves the visual result, but it does come at the cost of decreased frame rate. The skybox also improves the realism of the scene, and it comes at a minimal cost. As the skybox only consist of the eight vertices of its cube, and a cube map to texture it, it is very computationally cheap compared to the improvements it brings to the scene. The added time taken to render it is statistically insignificant compared to the time taken in simulation, and snow and terrain rendering.

Figure 4.7a and 4.7b shows the rendering of some snow particles on a black background textured with respectively Perlin noise procedural texturing and a single image texture. Where the procedurally textured particles have unique textures with natural shapes, the particles textured with an image look artificial and the repetition is obvious. Because the computational cost of both the texturing methods are comparable, but procedural texturing produces a much better visual result, the choice between them is simple is the case of the snow simulator. The Perlin noise based method is preferred.

Figure 4.8 shows a rendering of a scene from the snow simulator with distance


(a) Snow textured with procedural textures.

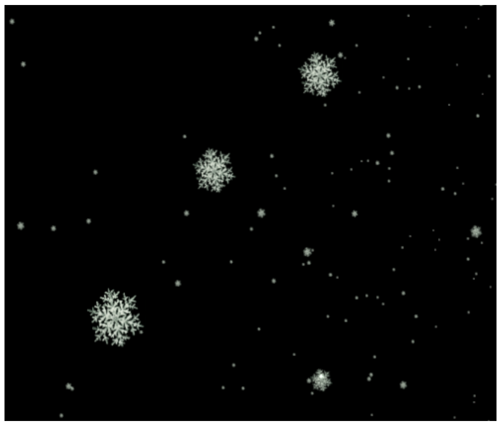

(b) Snow textured with an image of a snow crystal.

Figure 4.7: Snow textured using different texturing techniques.

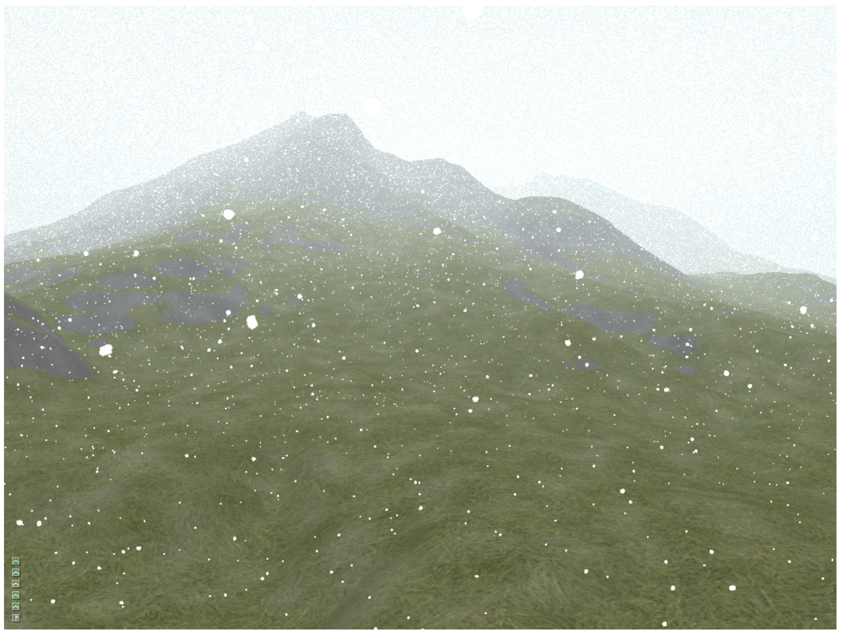


Figure 4.8: Scene from the snow simulator rendered with distance fog.

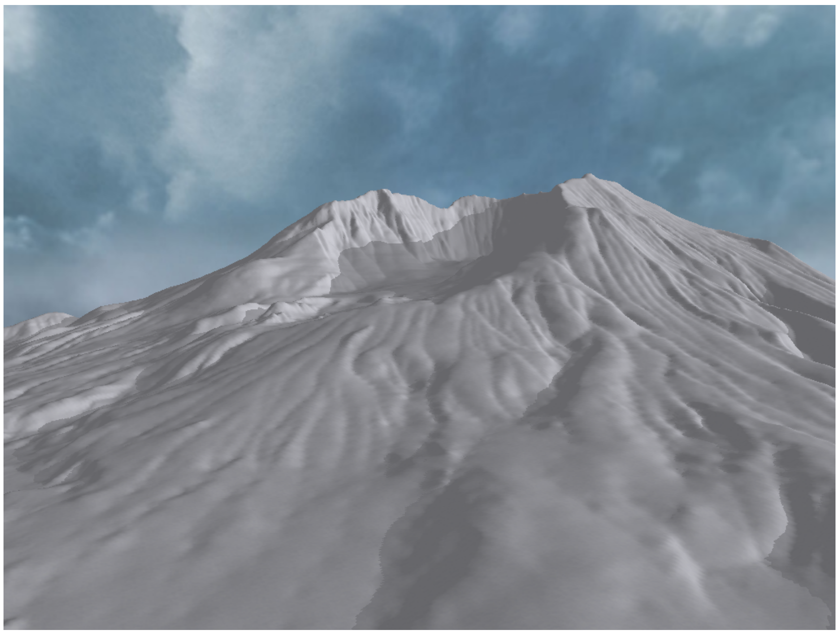


Figure 4.9: Terrain rendered with shadow mapping.

fog and four million snow particles. The visibility drop off formula used is the exponential squared one with a density factor of 0.05. The distance fog creates a sense of depth in the scene, where objects further from the observer are blurred more than close ones. Computer renderings can often be artificially clear where detail in the far distance appear too distinguishable, and the fog helps counteract this issue.

Seen in Figure 4.9 a snow covered terrain rendered using shadow mapping. The light source is placed somewhere to the right of the viewpoint, making the right side of the crater cast a shadow on the left side. At a significantly lower cost than any global illumination methods the shadow mapping here produces a much more realistic representation of the light than simple Phong shading. Although the shadow map is appears quite pixelated when the observer is close, when viewed from a distance the result looks smooth. The shadowing effect works best when the scene rendered is that of a clear day, and combining shadow mapping with distance fog does therefore not produce the most realistic renderings. Even the cube map used in this rendering is not that well suited for the weather conditions implied by the light and shadow.

Chapter 5

Conclusion and Future Work

As part of this thesis, a number of rendering techniques have been implemented, and with varying results. Some of the implementations were not very successful, take for example the lack of visual improvement with triplanar texturing, or the blocky result of the shadow mapping. Other parts had fantastic outcome, however, and the improvements to the rendering, and the code of the simulator as a whole were numerous.

Texturing with scalar mixing and Perlin noise blending made a big difference in the outcome of the terrain rendering, making the landscape more realistic. The procedural texturing of the snow particles using Perlin noise were also an enormous improvement over the previous implementations. The skybox added realism to the scenes at virtually no cost to the performance, and the fog made for a less unrealistically clear rendering of distant objects. The culling of geometry completely covered by the fog improved the performance significantly, leaving more of the processing time to the actual simulation.

Some of the methods made the snow simulator run slightly slower, but a lot of them didn't have a significant influence on the performance. Keeping the graphics code computationally cheap was maybe the most important aspect of this work, while still being able to produce improved visual results. As the results of the tests show, the real-time requirement of the snow simulator were not hampered by the new rendering code, and the visuals improvement are apparent.

A lot of time was also put into improving the code structure and readability of the code during the work on this thesis. The final result is a cleaner, more easily understandable and more accessible code base. While the performance gain of this work is nonexistent, the code has become easier and faster to work with and improve, which will make the coding for future projects and thesis on the snow simulator more effective. A testament to the improved modular structure of the code is how easy it was for Magnus Mikalsen to include the improved graphics code in his OpenACC version of the snow simulator.

5.1 Future Work

There are still a lot of improvements that can be made to the graphics of the snow simulator. First of all, the methods implemented as part of this thesis can be developed further, like the improved shadow mapping techniques were mentioned in Section 3.2. To be able to import arbitrary geometry, like buildings or trees, into the environment of the snow simulator to see how they interact with the wind and snow could also be interesting. Another, more exciting, area to explore could be to implement support for Oculus Rift, the next-generation virtual reality headset. Even though the product is not yet released, the development kit is available for pre-order form their website [25], and a lot of upcoming games have confirmed that they are supporting this new technology.

The model of the snow simulator is also in need of some upgrades. The biggest flaw in the accuracy of the snow simulator in its current state is the way the snow field is represented. Limiting the complexity of the snow field to being stored as a height value in each vertex of the terrain makes some of its behaviour unrealistic. An improved model for the snow field is represented could also be tied into how the wind interacts with snow on the ground. This also ties into the smoothed-particle hydrodynamics avalanche simulator developed at the HPC-lab by Øystein Eklund Krog in 2010 [12]. Combining this simulation with the snow simulator would require some work, especially with the snow field representation, but it would add a new *dimension* to the simulator.

Bibliography

[1] M. Aagaard and D. Lerche. Realistic modelling of falling and accumulating snow. Master's thesis, Aalborg University, Denmark, jun 2004.

[2] The National Aeronautics and Space Administration. Advanced spaceborne thermal emission and reflection radiometer global digital elevation model version 2. [Online; accessed 30-May-2013].

[3] Kjetil Babington. Real-time ray tracing for the hpc-lab snow simuator. Master's thesis, NTNU, Trondheim, dec 2011.

[4] Kjetil Babington. Terrain rendering techniques for the hpc-lab snow simulator. Master's thesis, NTNU, Trondheim, jun 2012.

[5] cgtextures.com. [cg textures] - textures for 3d, grpahic design and photoshop! [Online; accessed 2-Jun-2013].

[6] Joel Chelliah. The ntnu hpc snow simulator on the fermi gpu. Master's thesis, NTNU, Trondheim, dec 2010.

[7] Robin Eidissen. Utilizing gpus for real-time visualization of snow. Master's thesis, NTNU, Trondheim, feb 2009.

[8] Aleksander Gjermundsen. Lbm vs. sor solvers on gpus for real-time snow simulations. Technical report, NTNU, Trondheim, dec 2009.

[9] Khronos Group. The opengl shading language, version: 4.30. [Online; accessed 25-May-2013].

[10] Anne C. Elster Ingar Saltvik and Henrik R. Nagel. Parallel methods for real-time visualization of snow. *B. Kågstrom et al. (Eds.): PARA 2006, LNCS 4699 of Lecture Notes in Computer Science*, pages 218–277, 2007.

[11] Øystein E. Krog and Anne C. Elster. Fast gpu-based fluid simulations using sph. *PARA'10 Proceedings of the 10th international conference on Applied Parallel and Scientific Computing*, 2:98–109, 2012.

[12] Øystein Eklund Krog. Gpu-based real-time snow avalanche simulations. Master's thesis, Norwegian University of Science and Technology, Trondheim, jun 2010.

[13] Hallgeir Lien. Procedural generation of roads for use inthe snow simulator. Master's thesis, NTNU, Trondheim, 2011.

[14] Holger Ludvigsen and Anne C. Elster. Real-time ray tracing using nvidia optix. *Eurographics 2010*.

[15] Hubert Nguyen, editor. *GPU Gems 3*, chapter 1. Addison-Wesley Professional, aug 2007.

[16] NVIDIA. The cg tutorial. [Online; accessed 4-Jun-2013].

[17] NVIDIA. Cuda c programming guide. [Online; accessed 26-May-2013].

[18] Ken Perin. An image synthesizer. *Computer Graphics (Proceedings of ACM SIGGRAPH 85)*, 19(3):287–296, July 1985.

[19] Ken Perin. Improving noise. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2002)*, 21(3):681–682, July 2002.

[20] Ken Perlin. Academy award for technical achievement. [Online; accessed 28-May-2013].

[21] Ingar Saltvik. Parallel methods for real-time visualization of snow. Master's thesis, NTNU, Trondheim, jun 2006.

[22] SOIL. Simple opengl image library. [Online; accessed 5-Jun-2013].

[23] Jarle Erdal Steinsland. Porting the opencl snow simulator to opencl for mobile applications. Master's thesis, NTNU, Trondheim, dec 2010.

[24] Frederik Magnus Johansen Vestre. Enhancing and porting the hpc-lab snow simulator to opencl on mobile platforms. Master's thesis, NTNU, Trondheim, jun 2012.

[25] Oculus VR. Oculus vr home page. [Online; accessed 9-Jun-2013].

[26] Lance Williams. Casting curved shadows on curved surfaces. *ACM SIGGRAPH Computer Graphics*, 12(3):270–274, August 1978.

[27] David M. Young. *Iterative Solution of Large Linear Systems.* Courier Dover Publications, 2003.

Appendix A

Simple Terrain Shaders

```
#version 400 core

#ifdef _VERTEX_

uniform mat4 mvp;

layout (location = 0) in vec4 in_position;
layout (location = 1) in vec3 normal;

out vec3 out_position; out vec3 n;
out float snow_height;

void main() {
    out_position = in_position.xyz;
    snow_height = in_position.w;
    out_position.y += snow_height;
    gl_Position = mvp * vec4(out_position, 1.0);
    n = normalize(normal);
}
#endif

#ifdef _FRAGMENT_

uniform sampler2D grass;
uniform sampler2D rock;
uniform float tex_scale;

in vec3 out_position;
in vec3 n;
in float snow_height;

out vec4 finalColor;

struct Light {
    vec4 diffuse;
    vec4 ambient;
    vec3 dir;
};

uniform Light light;

void main() {
    // snow
    vec4 blended_color = vec4(1.0, 1.0, 1.0, 1.0);

    float snow_blend = smoothstep(0.0, 0.01, snow_height);
    if (snow_blend < 1.0) {
        vec2 coord = out_position.zx * tex_scale;
        // sample
        vec4 grass_col = texture(grass, coord);
        vec4 rock_col = texture(rock, coord);
        // blend
        float slope = 1-dot(normalize(n), vec3(0.0, 1.0, 0.0));
        float blend_value = 0.0;
        if (out_position.y*slope > 1) blend_value = 1.0;
        if (out_position.y > 12) blend_value = 1.0;
        if (slope > 0.3) blend_value = 1.0;
        blended_color = mix(grass_col, rock_col,
                    smoothstep(0.3, 0.7, blend_value));
        blended_color = mix(blended_color,
                    vec4(1.0, 1.0, 1.0, 1.0), snow_blend);
    }

    // Light
    float diffuseFactor = max(dot(normalize(n),light.dir), 0.0);
    vec4 TotalLight = diffuseFactor * light.diffuse + light.ambient;

    finalColor = blended_color * TotalLight;
}

#endif
```

Appendix B

Complex Terrain Shaders

```
#version 400 core

#ifdef _VERTEX_

uniform mat4 mvp;
uniform mat4 light_bias_mvp;
uniform vec3 cam_pos;

layout (location = 0) in vec4 in_Position;
layout (location = 1) in vec3 normal;

out vec3 wsCoord; out vec3 n;
out float snow_height;
out vec4 ShadowCoord;
out float dist;

void main() {
    wsCoord = in_Position.xyz;
    snow_height = in_Position.w;
    wsCoord.y += snow_height;
    gl_Position = mvp * vec4(wsCoord, 1.0);

    dist = length(wsCoord-cam_pos);

    n = normalize(normal);

    ShadowCoord = light_bias_mvp * vec4(wsCoord, 1.0);
}

```

```
#endif

#ifdef _FRAGMENT_
// Texturing
uniform sampler2D grass;
uniform sampler2D rock;
uniform sampler2D snow;
uniform float tex_scale;
// Shadow map
uniform sampler2D ShadowMap;
uniform vec3 shadow_color;
uniform bool shadow;
uniform vec2 texmapscale;
uniform float bias;
uniform float offset;
// Fog
uniform int fog_type; // 0 = off, 1 = linear, 2 = exp, 3 = exp2
uniform bool fog_range; //Range based instead of plane based fog
uniform float fog_start;
uniform float fog_end;
uniform float fog_density;
uniform vec3 fog_color;

in vec3 wsCoord; in vec3 n;
in float snow_height;
in vec4 ShadowCoord;
in float dist;

out vec4 finalColor;

struct Light {
    vec4 diffuse;
    vec4 ambient;
    vec3 dir;
};

uniform Light light;

void main() {
    vec2 coord1 = wsCoord.yz * tex_scale;
    vec2 coord2 = wsCoord.zx * tex_scale;
    vec2 coord3 = wsCoord.xy * tex_scale;

    vec4 grass_col1 = mix(texture(grass, coord1),
                texture(grass, coord1 * -0.25f), 0.5f);
    vec4 grass_col2 = mix(texture(grass, coord2),
                texture(grass, coord2 * -0.25f), 0.5f);
    vec4 grass_col3 = mix(texture(grass, coord3),
                texture(grass, coord3 * -0.25f), 0.5f);

    vec4 rock_col1 = mix(texture(rock, coord1),
                texture(rock, coord1 * -0.25f), 0.5f);
    vec4 rock_col2 = mix(texture(rock, coord2),
                texture(rock, coord2 * -0.25f), 0.5f);
    vec4 rock_col3 = mix(texture(rock, coord3),
                texture(rock, coord3 * -0.25f), 0.5f);

    float slope = 1-dot(normalize(n), vec3(0.0, 1.0, 0.0));
    float blend_value = 0.0;
    if (wsCoord.y*slope > 1) blend_value = 1.0;
    if (wsCoord.y > 12) blend_value = 1.0;
    if (slope > 0.3) blend_value = 1.0;

    vec4 col1 = mix(grass_col1, rock_col1,
                smoothstep(0.3, 0.7, blend_value));
    vec4 col2 = mix(grass_col2, rock_col2,
                smoothstep(0.3, 0.7, blend_value));
    vec4 col3 = mix(grass_col3, rock_col3,
                smoothstep(0.3, 0.7, blend_value));

    col1 = mix(col1, texture(snow, coord1),
                smoothstep(0.0, 0.01, snow_height));
    col2 = mix(col2, texture(snow, coord2),
                smoothstep(0.0, 0.01, snow_height));
    col3 = mix(col3, texture(snow, coord3),
                smoothstep(0.0, 0.01, snow_height));

    // Triplanar projection
    vec3 blend_weights = abs(n);
    blend_weights = (blend_weights - 0.2)*7;
    blend_weights = max(blend_weights, 0);
    blend_weights /= (blend_weights.x +
                blend_weights.y + blend_weights.z);

    vec4 blended_color = col1.xyzw * blend_weights.xxxx +
                         col2.xyzw * blend_weights.yyyy +
                         col3.xyzw * blend_weights.zzzz;

    // Light
    float diffuseFactor = max(dot(normalize(n),light.dir), 0.0);
    vec4 TotalLight = diffuseFactor *
                    light.diffuse + light.ambient;

    // Shadow
    if (shadow) {
        vec4 visibility = vec4(1.0);
        if (texture(ShadowMap, ShadowCoord.xy).z <
                    ShadowCoord.z-bias)
            visibility = vec4(shadow_color, 1.0);
        TotalLight = visibility * diffuseFactor *
                light.diffuse + light.ambient;
    }

    // Fog
    float fog_dist = gl_FragCoord.z / gl_FragCoord.w;
    if (fog_range) {
        fog_dist = dist;
    }

    float fog_factor = 0.0;
    if (fog_type == 1) {
        // Linear
        fog_factor = 1.0-clamp( (fog_end - fog_dist)/
                (fog_end - fog_start), 0.0, 1.0);
    } else if (fog_type == 2) {
        // Exp
        fog_factor = clamp( 1.0-exp(-fog_density *
                fog_dist), 0.0, 1.0);
    } else if (fog_type == 3) {
        // Exp2
        fog_factor = clamp( 1.0-exp(-fog_density *
                fog_density * fog_dist * fog_dist), 0.0, 1.0);
    }

    finalColor = mix(blended_color * TotalLight,
                vec4(fog_color, 1.0), fog_factor);
}

#endif
```

Appendix C

Billboarding shader

```
#version 400 core

#ifdef _VERTEX_

layout (location = 0)
in vec4 vert;

void main() {
    gl_Position = vec4(vert.xyz, 1.0);
}

#endif

#ifdef _GEOMETRY_

layout (points) in;
layout (triangle_strip) out;
layout (max_vertices = 4) out;

uniform mat4 gWVP;
uniform vec3 eye;
uniform float scale;

out vec2 TexCoord;

void main() {
    vec3 Pos = gl_in[0].gl_Position.xyz;
    vec3 toCamera = normalize(eye - Pos);
    vec3 right = cross(toCamera, vec3(0.0, 1.0, 0.0));
    vec3 up = cross(right, toCamera);

    Pos -= (right * 0.5 * scale);
    Pos -= (up * 0.5 * scale);
    gl_Position = gWVP * vec4(Pos, 1.0);
    TexCoord = vec2(0.0, 0.0);
    EmitVertex();

    Pos += up * scale;
    gl_Position = gWVP * vec4(Pos, 1.0);
    TexCoord = vec2(0.0, 1.0);
    EmitVertex();

    Pos -= up * scale;
    Pos += right * scale;
    gl_Position = gWVP * vec4(Pos, 1.0);
    TexCoord = vec2(1.0, 0.0);
    EmitVertex();

    Pos += up * scale;
    gl_Position = gWVP * vec4(Pos, 1.0);
    TexCoord = vec2(1.0, 1.0);
    EmitVertex();

    EndPrimitive();
}

#endif

#ifdef _FRAGMENT_

uniform sampler2D colormap;

in vec2 TexCoord;

out vec4 FragColor;

void main() {
    FragColor = texture(colormap, TexCoord);
}

#endif
```

Appendix D

Point sprite shader

```
#version 400 core

#ifdef _VERTEX_

uniform mat4 gWVP;
uniform float scale;
uniform float alpha_scale;

layout (location = 0) in vec4 vert;

out float alpha;

void main() {
    vec4 pos = gWVP * vec4(vert.xyz, 1.0);
    gl_Position = pos;

    vec4 pos_ = gWVP * vec4(vert.xyz + vec3(0.0, 1.0, 0.0), 1.0);

    float size = distance(pos.y/pos.w, pos_.y/pos.w);
    gl_PointSize = size * scale;
    alpha = max(0.3, min(1.0, size * alpha_scale));
}

#endif

#ifdef _FRAGMENT_

uniform sampler2D colormap;

```

```
in float alpha;

out vec4 FragColor;

void main() {
    FragColor = texture(colormap, gl_PointCoord)*alpha;
}

#endif
```