



Norwegian University of
Science and Technology

Real-Time Snow Simulation

Integrating Weather Data and Cloud
Rendering

Thomas Martin Schmid

Master of Science in Informatics

Submission date: June 2016

Supervisor: Anne Cathrine Elster, IDI

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

Problem Description

Over the years a snow simulator has been developed by several masters' students at the HPC-lab at NTNU. This thesis proposal focuses on improving and adapting this simulator.

The thesis would consider improvements to the terrain as well as the weather simulation. The terrain and weather data may be based on, but would not be restricted to, real-world data.

The thesis may also investigate the adaptations needed for the simulator to be used in more computationally constrained systems such as video games.

Abstract

Over the last decade the NTNU Heterogenous Computing Laboratory (HPC-Lab) at the Norwegian University of Science and Technology (NTNU) has had master students working on a real-time snow simulator. Evolving from a complex and highly parallel Central Processing Unit (CPU) smoke simulation, the simulator now covers models for wind simulation, snow particle physics, and avalanche prediction all computed in parallel on both CPU and Graphics Processing Unit (GPU) using Open Computing Language (OpenCL) or Compute Unified Device Architecture (CUDA). The resulting simulation is rendered using modern techniques and Open Graphics Library (OpenGL).

In this thesis, the wind and in-air snow simulation are improved by removing and reducing simplifications and assumptions. By extending control of the boundary conditions and initial distributions, simulations can be run with external context such as real-world weather information.

The simulation boundary conditions are extended to support interpolation between ground-truth points, and a simple, yet novel approach is introduced for selecting interpolation weights. The interpolation cost is shown to be less than 2ms more than for constant values in the worst case, for a frame time of 26ms, and at less than 1ms of additional loading time. The neighborhood calculation time is smaller than the time to load ground truth points from disk. Snow precipitation rates across the simulation domain are controllable from animated data-sets or procedural functions, allowing the use of radar imaging as a source of simulation data. The additional cost of the rejection sampling is shown to be negligible for reasonable configuration values.

To help visualize the precipitation rates cloud rendering is introduced to the simulator. While previous work focused on terrain and snow particles, the sky was simplified. Animated clouds are shown to be a computationally costly, but affordable, addition to the simulator, improving the quality of the rendered images. The run-time cost of high quality clouds is shown to a frame time increase of less than 50% at typical camera positions, and even last-generation mid-range GPUs maintain frame-rates of over 30HZ. High-end consumer GPUs maintain over 30HZ even in worst-case scenarios and almost 60HZ in normal use.

Finally, the choice of Pseudo-Random Number Generator (PRNG) on both the CPU and GPU are re-evaluated. A change is made necessary to support varied precipitation rates, but also improves the statistical properties of the simulation. The performance cost of the improvement is shown to be negligible at less than a 5% increase in run-time of the CUDA version of the snow particle update kernel.

Sammendrag

Over det siste tiåret har master-studenter ved NTNUs HPC-Lab jobbet med en sanntids-simulasjon av snø. Fra en kompleks røyk-simulasjon som utnyttet parallellitet på CPUer har simulatoren blitt utvidet med modeller for vind, snøflak-fysikk, og sannsynlighet for snøras. Beregninger skjer parallellt tvers av GPU- og CPU-er, ved hjelp av CUDA eller OpenCL. Simulasjonen visualiseres med moderne teknikker gjennom OpenGL.

I denne oppgaven forbedres både vind- og snøflaksimulasjonene ved å redusere eller fjerne forenklinger og antakelser. Ved å utvide brukerens kontroll over grenseverdier og startforhold kan simulasjoner kjøres i sammenheng med eksterne data, som innsamlet vær-data.

Simulasjonens grenseverdier får støtte for interpolasjon mellom verdier fra virtuelle målestasjoner, og en ny, enkel teknikk for vektning av interpolasjonen introduseres. Kostnaden av interpolasjonen vises å være mindre enn 2ms mer enn for konstante verdier, og total kostnad for virtuelle målestasjoner ved oppstart er mindre enn 1ms. Kontroll over nedbørsmengden i simulasjons-området utvides til animerte datasett og matematiske funksjoner, noe som tillater bruk av værradar som informasjonskilde. Tilleggs-kostnaden for re-posisjonering av snøpartikler vises å være ubetydelig for fornuftige parametre.

For å hjelpe visualisering av sannsynligheten for nedbør introduseres rendering av skyer. Tidligere arbeide på visualisering i simulatoren har fokusert på terreng og snø, mens himmelen har vært enkel og statisk. Et animert skylag vises å være beregningstungt, men kostnaden er akseptabel på moderne maskinvare, og forbedrer kvaliteten på simulasjonens visualisering. Kjøretidskostnaden for skyer av høy kvalitet fører til en økning i beregningstid per bilde på mindre enn 50% for vanlige kamera-posisjoner, og selv på en 4 år gammel GPU holdes bilde-frekvensen over 30HZ. Moderne høy-kvalitets forbuker GPUer holder frekvensen over 30HZ selv i verst tenkelig scenario, og nesten 60HZ i vanlig bruk.

Simulatorens valg av generator for pseudo-tilfeldige tall re-evalueres også. En utveksling som nødvendiggjøres av støtten for variert sannsynlighet for nedbør forbedrer også simulasjonens statistiske egenskaper. Kjøretidskostnaden vises å være ubetydelig, ned mindre enn 5% økning i beregnings-tid for oppdatering av snøpartikler i CUDA-utgaven.

Acknowledgements

First and foremost I would like to thank my supervisor for this thesis, Dr. Anne C. Elster, for her assistance, guidance, and the opportunity. I would also like to thank NTNU and NVIDIA's GPU Research Center programs for their support and research equipment donations to the HPC-Lab at IDI. The HPC-Lab is lead by my advisor Dr. Elster, and is where most of this work was done.

Gratitude is also extended to fellow HPC-Lab master student Inge Halsanet, who worked on the snow simulator in parallel, and without whom this work would have been considerably harder. Finally, I would like to thank my parents, without whoms generous support this thesis would never have been possible.

Table of Contents

Problem Description	i
Abstract	iii
Sammendrag	iv
Acknowledgements	v
Table of Contents	x
List of Tables	xi
List of Figures	xiv
1 Introduction	3
1.1 Motivation	3
1.2 Contribution	4
1.2.1 Real-world data integration	4
1.2.2 Visual and miscellaneous improvements	4
1.3 Outline	4
2 Background	7
2.1 Computational fluid dynamics	7
2.1.1 Navier-Stokes	7
2.1.2 Numerical methods	8
2.2 Meteorological measurements	10
2.2.1 Methods	10
2.2.2 Precipitation and cloud coverage	11
2.2.3 Wind	11
2.2.4 Public data	11
2.3 Terrain models	12
2.3.1 USGS DEM Format	13

2.4	Random number generation and noise	13
2.4.1	Random Number Generators	13
2.4.2	Probability Density Functions and Distributions	18
2.4.3	Noise	18
2.5	Voronoi tessellation and their duals	19
2.6	Stereoscopic rendering	19
2.6.1	Anaglyph stereoscopy	20
2.6.2	Polarization stereoscopy	20
2.6.3	Active-shutter stereoscopy	20
2.6.4	Multi-screen stereoscopy	20
2.6.5	Performance implications	21
2.7	GPU and GPGPU computing	21
2.7.1	Hardware model	22
2.7.2	OpenGL	22
2.7.3	CUDA	24
2.7.4	OpenCL	26
2.7.5	Performance analysis	26
2.8	Cloud rendering	27
2.8.1	Ray-marching volumetrics	28
2.9	The HPC-Lab Real-Time Snow Simulator	28
2.9.1	History	28
2.9.2	Technical overview	29
3	Simulator improvements and extensions	35
3.1	Motivation	35
3.2	Wind	36
3.2.1	Importing the values	37
3.2.2	WindSource representation on the GPU	37
3.2.3	Spatial interpolation at boundary points	38
3.2.4	Setting the values	40
3.2.5	Visualization	40
3.3	Precipitation	41
3.3.1	Precipitation distribution	41
3.3.2	Initial distribution	42
3.3.3	Re-positioning	42
3.3.4	PRNGs	45
3.4	Clouds	45
3.4.1	Technique choice	46
3.4.2	Technical details	46
3.4.3	Real-world data integration	49
3.5	Terrain	50
3.6	Other Snow Simulator improvements	51
3.6.1	3D rendering	51
3.6.2	OpenCL sampling improvements	52
3.6.3	Performance analysis	53
3.6.4	Bindless textures	53

3.6.5	Wind simulation stability problem	54
4	Results and Discussion	57
4.1	Performance analysis	57
4.1.1	Tool analysis	58
4.2	Wind	58
4.2.1	Wind advection kernel implications	58
4.2.2	CPU implications	62
4.2.3	Memory packing	62
4.3	Precipitation	63
4.3.1	RNGs	64
4.3.2	Rejection cost	65
4.3.3	Distribution map	66
4.4	Clouds	69
4.4.1	Technique differences	69
4.4.2	Visual artifacts	71
4.4.3	Optimization	71
4.4.4	Interaction with other systems	74
4.5	Terrain	75
4.6	Miscellaneous improvements	75
4.6.1	3D rendering	75
4.6.2	OpenCL sampling improvements	77
4.6.3	Bindless textures	79
4.6.4	Wind simulation stability improvements	79
5	Conclusion and Future Work	81
5.1	Real-world data integration	81
5.2	Miscellaneous improvements	82
5.3	Future work	82
5.3.1	Snow melting	83
5.3.2	Terrain model improvements	83
5.3.3	Wind simulation improvements	83
5.3.4	Extended Multi-GPU support	84
5.3.5	Raytracing	84
5.3.6	NEXRAD support	84
	Bibliography	85
A	Procedural sphere generation	91
B	WindSource neighborhood evaluation	95
C	Wind interpolation advection kernels	97
C.1	Branching version	97
C.2	Constant only version	99
C.3	Nearest-Neighbor only version	100

C.4	Interpolation only version	101
D	Snow redistribution	105
D.1	CUDA implementation	105
D.2	OpenCL implementation	106
E	Cloud shader	109
F	Profiling system	115
F.1	Automation script	115
F.2	TimingSystem class	120
F.3	vul_benchmark	124
G	Terrain pre-processing	133
H	Stereo rendering	137
I	User Manual	141
I.1	Wind	141
I.2	Precipitation	142
I.3	Clouds	142
I.4	Terrain	143
I.5	3D rendering	143
I.6	OpenCL hardware sampling	144
I.7	Performance analysis	144
I.8	Bindless textures	145
I.9	Wind simulation stability problem	145

List of Tables

2.1	Analysis if Eidissen’s floating point RNG	16
2.2	RNG quality analysis	17
2.3	The corresponding terminology of CUDA and OpenCL	26
2.4	The HPC-Lab Snow Simulator’s particle representation	30
3.1	The HPC-Lab Snow Simulator’s GPU WindSource representation	38
4.1	Performance test machines	57
4.2	Wind advection kernel statistics	59
4.3	Wind source pre-processing time	62
4.4	Wind source packing implications	62
4.5	GPU PRNG output visualization	67
4.6	GPU PRNG performance	68
4.7	Cloud technique performance	70
4.8	Cloud pre-marching timings	72
4.9	Cloud step length performance	74
4.10	3D rendering performance	78
4.11	OpenCL hardware vs. software sampling	78
4.12	Bindless texture performance	79
4.13	Wind simulation stability improvement costs	80

List of Figures

2.1	The four steps of the Semi-Lagrangian method	9
2.2	Visual comparison of collocated and staggered grids	9
2.3	CPU vs. CPU performance	33
2.4	Terrain mesh and obstacle representation	34
2.5	Terrain snow accumulation	34
3.1	Wind Source contribution problem	39
3.2	Interpolation neighbors	40
3.3	Rain front	41
3.4	Side redistribution issue	43
3.5	Distribution texture extent	44
3.6	Wind simulation stability issue	55
4.1	Boundary wind techniques	59
4.2	Wind advection kernel statistics	61
4.3	Precipitation probability visualized	63
4.4	Precipitation probabilities at sides	63
4.5	GPU PRNG performance	65
4.6	Rejection method cost	66
4.7	Cloud rendering techniques	69
4.8	Cloud artifacts	72
4.9	Cloud step length performance	73
4.10	Imported terrain examples	76
4.11	Terrain mesh vs. obstacle resolution	77
4.12	Wind simulation stability improvement	80
I.1	Wind type menu entry	141
I.2	Wind source menu entry	141
I.3	Precipitation distribution type menu entry	142
I.4	Precipitation distribution settings menu	142
I.5	Cloud rendering type menu entry	143

I.6	Cloud rendering settings menu	143
-----	---	-----

Acronyms

2D Two dimensional. 19

AES Advanced Encryption Standard. 15

AMD Advanced Micro Devices, Inc.. 17

API Application Programming Interface. 10–12, 22, 26, 27, 37, 52

AVX Advanced Vector Extensions. 62

CPU Central Processing Unit. iii, 3, 14, 15, 22, 24–28, 37, 42, 45, 52, 58, 60, 62, 77, 79

CUDA Compute Unified Device Architecture. iii, 4, 17, 23–28, 30, 32, 44, 45, 51–54, 58, 59, 64, 65, 77, 79, 82, 97, 105, 144, 145

DEM Digital Elevation Model. 13, 50, 133

DirectX Microsoft DirectX. 22, 51

FBM Fractal Brownian Motion. 19, 28, 46, 48, 49, 70, 71, 74

GIF Graphics Interchange Format. 12

GPGPU General-Purpose Graphics Processing Unit. 3, 7, 21, 24, 26, 32, 36

GPS Global Positioning System. 37

GPU Graphics Processing Unit. iii, 3, 7, 13–17, 21–30, 35, 41, 45, 46, 58, 59, 67, 69, 70, 75, 76, 78, 79, 81, 84, 143, 144

GUI Graphical User Interface. 36, 49, 51–53

HMD Head-Mounted Display. 20, 21, 52, 77

HPC-Lab NTNU Heterogenous Computing Labratory. iii, 3, 7, 28, 29, 35, 36, 51, 53, 75, 81

HTML HyperText Markup Language. 12

IPD Inter-Pupillary Distance. 52, 143

LCD Liquid Crystal Display. 20

LCG Linear Congruential Generator. 13–17, 32

LFSR Linear-Feedback Shift Register. 14, 15, 64

LiDAR Light Detection and Ranging. 12, 75, 76

MET Norway The Norwegian Meteorological Institute. 10–12, 49

MRG Multiple Recursive Generator. 14, 16, 64

NEXRAD Next-Generation Radar. 12, 49, 84

NMA The Norwegian Mapggin Authority. 12, 50

NORAD North American Aerospace Defense Command. 10

NSE Navier-Stokes Equations. 7, 8, 31

NTNU Norwegian University of Science and Technology. iii, 3

OpenACC Open Accelerators. 29

OpenCL Open Computing Language. iii, 4, 17, 23, 26–28, 30, 32, 44, 45, 51–53, 62, 64, 65, 77, 78, 82, 97, 105, 106, 144

OpenGL Open Graphics Library. iii, 22–26, 28, 51–54, 77, 79, 143

OpenMP Open Multi-Processing. 29

PCI-E Peripheral Component Interconnect Express. 22

PDF Probability Density Function. 18, 42, 43, 45

POSIX Portable Operating System Interface. 32, 52, 77

PRNG Pseudo-Random Number Generator. iii, 13–19, 28, 32, 42, 43, 45, 55, 58, 64, 65, 67, 68, 77, 81, 105

RAM Random Access Memory. 22

REST Representation State Transfer. 12

- SOAP** Simple Object Access Protocol. 12
- SOR** Successive Over-Relaxation. 9, 54, 55, 79, 80, 82, 145
- SSBO** Shared-Storage Buffer Object. 24
- SSE** Streaming SIMD Extensions. 62
- UBO** Uniform Buffer Object. 24
- UK** United Kingdom. 12
- UKEA** United Kingdom Environment Agency. 50
- US NWS** The US National Weather Service. 11, 12
- USA** United States of America. 10
- USGS** United States Geological Survey. 13, 50, 133
- VR** Virtual Reality. 77, 82
- XML** Extensible Markup Language. 12

Introduction

The HPC-Lab snow simulator is a continuously evolving project at the NTNU HPC-Lab. The complex numerical work of an early highly CPU-parallel smoke simulation has been extended, improved upon and optimized into a full-fledged wind, snow-fall, and avalanche simulation with high quality terrain and snow rendering. Simulations at this scale are computationally expensive, and the use of General-Purpose Graphics Processing Unit (GPGPU) computing is central to the simulator, due to the massive amount of particles and wind simulation volume voxels. This enables the simulation to be run in parallel over millions of invocations, making use of the massively parallel nature of GPUs to gain performance multi-core CPUs. However, the models used for simulation and the visual presentation still contain numerous simplifications. This thesis extends upon previous work by reducing the scope of simplifications, and adding visual representation of some missing features.

1.1 Motivation

The simulation of weather conditions such as snow has practical applications in many areas. By studying wind patterns and snow accumulation it may aid in infrastructure planning, or in predicting dangerous phenomenon such as avalanches, both on mountain and rooftop scale.

The HPC-Labsnow simulator uses a number of techniques originally from the computer graphics field, not from physical modelling. It is desirable to be able to compare the quality of simulation using these techniques with real-world data, to evaluate if the quality of simulation is sufficient. However, prior versions of the simulator contained numerous simplifications on the input data; snow fell uniformly, and wind at the boundaries of the simulation domain was assumed constant in direction and velocity.

Simultaneously, the simulator sees practical use as a demonstration tool for the HPC-

Lab, and as an experimental environment for students. For this purpose, any improvements to visual quality, introduction of variety in rendering or simulation techniques, or new technology is of interest. Care must be taken to maintain both the CUDA and OpenCL versions of the simulator.

More realistic snow simulation would one day allow for prediction of both mountain scale and rooftop avalanches based on weather prediction and gathered data, as well as help plan optimal routes for infrastructure such as roads and railways based on snow accumulation and wind and weather statistics. By improving the coupling with real-world data the simulation may also be verified, enabling improvements to be measured against ground-truth measurements.

1.2 Contribution

The thesis enables the use of real-world data for initial and boundary conditions, paving the way for quality evaluation and improvements based on comparing simulation results with ground truth values. It adds to the visual quality of the simulator with cloud rendering, and includes minor performance improvements, optimization attempts, and suggests a fix to a simulation quality problem.

1.2.1 Real-world data integration

This thesis will extend support to non-uniform precipitation rates across the domain from both imported data sources, uniform data or generated patterns or noise. It will also update the boundary conditions of the wind simulation to support non-uniform wind at the edges of the domain, enabling the use of weather station data as a basis for the domain-internal simulation. For this, a simple algorithm is introduced to map weather station contribution to boundary voxel. Finally, support for high-resolution terrain models is maintained and improved.

1.2.2 Visual and miscellaneous improvements

To help visualize the non-uniform precipitation, and to improve visual fidelity of the simulator, cloud rendering is introduced. Prior work to support stereoscopic rendering is extended to support the newest version of the simulator. Minor performance improvements and suggestions to visual problems with simulation quality are also introduced.

1.3 Outline

The thesis is structured as follows:

Chapter 2 (Background) looks at background information on any fields the thesis touches upon, research used as a basis for the work done, and a look at the snow simulator's history, inner workings and uses.

Chapter 3 (Implementation) describes the work done, with reasoning and discussion of decisions made and alternative paths not taken.

Chapter 4 (Results and Discussion) analyzes performance and quality of the implemented solutions and evaluates the decisions made in Chapter 3.

Chapter 5 (Conclusions and Future Work) sums up the important points from Chapter 4, takes a birds-eye view of the thesis' contributions, and discusses interesting avenues for future improvement

Appendix A (Procedural Sphere Generation) Contains source code for generation of sphere meshes of varying complexity and quality using subdivision.

Appendix B (Neighborhood Evaluation for Wind Sources) Contains source code for the evaluation of wind source neighborhoods for interpolation purposes.

Appendix C (Wind interpolation advection kernels) Contains source code for the various boundary wind calculation techniques supported in the simulator.

Appendix D (Snow redistribution) Contains source code for snow redistribution with respect to a non-uniform distribution map.

Appendix E (Cloud shader) Contains the source code for the cloud rendering ray-marching shader.

Appendix F (Profiling system) Contains the source code for the automated profiling system and related simulator-internal classes and library.

Appendix G (Terrain pre-processing) Contains the pre-processing program to convert terrain models into height-maps.

Appendix H (Stereo rendering) Contains the source code related to the implementation of stereoscopic rendering.

Appendix I (User Manual) Describes the use, configuration parameters, and default settings of all new features of the simulator.

Chapter 2

Background

This chapter is an investigation of previous work and a short survey of fields of study and material related to the thesis. Section 2.1 introduces computational fluid dynamics, which is required to understand the core loop of the snow simulation. Sections 2.2 and 2.3 introduce meteorological measurements and terrain models to understand how real-world data can be integrated, and what data is required. Section 2.4 introduces random number generation and *noise*, constructs core to the implementations presented in Chapter 3. Section 2.5 introduces Voronoi tessellation briefly to help understand the discussion in Section 3.2.3. Section 2.6 introduces stereoscopic rendering, and Section 2.8 introduces cloud rendering techniques. Section 2.7 gives a wide introduction to the GPU and GPGPU computing environment, before the chapter closes with an introduction to the history, uses, and a technical overview of the HPC-Lab snow simulator in Section 2.9.

2.1 Computational fluid dynamics

The study of fluid dynamics is well covered in academia, and the Navier-Stokes Equations (NSE) is a widely used model for the flow of fluid. There are many ways to compute a numerical solution of the NSE, and this section covers the methods relevant to this thesis and their trade-offs.

2.1.1 Navier-Stokes

The NSE is often simplified by assuming that the fluid in question is incompressible; its density is uniform. This assumption rules out pressure waves, but holds well for fluids of low mach numbers [1, Ch. 3]. Given this assumption, the NSE describe flow by operating on two fields; a velocity field u and a pressure field p , both of which typically vary in both time and space. The *NSE for incompressible flow* consists of Equations 2.1 and 2.2,

$$\nabla \dot{\mathbf{u}} = 0 \quad (2.1)$$

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla) \mathbf{u} - \nabla w + v \nabla^2 \mathbf{u} + \mathbf{F} \quad (2.2)$$

$$\frac{1}{p} \nabla \mathbf{p} \equiv \nabla w \quad (2.3)$$

Equation 2.1 ensures the velocity field is divergence-free; the inward and outward flux at any point in the velocity field is zero and flow is conserved. Equation 2.3 is the simplification of incompressibility; if the density is uniform then the thermodynamic work w is the partial derivative of the pressure field \mathbf{p} divided by the density p . Equation 2.2 is a partial differential equation in multiple dimensions. Here, ∇ denotes the vector or spatial partial derivatives, that is $\nabla = (\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z})$ in three dimensions. The equation consists of four terms, which can be broken down as follows, applying the simplification of equation 2.3.

Advection $-(\mathbf{u} \cdot \nabla) \mathbf{u}$ The velocity self-advects, that is the velocity field transports its values along its flow.

Internal stress $-\frac{1}{p} \nabla \mathbf{p}$ Inter-molecular collisions lead to pressure in moving fluid. This pressure leads to a natural acceleration of the fluid inversely proportional with the density of the fluid, p .

Diffusion $v \nabla^2 \mathbf{u}$ This term decelerates flow due to inter-molecular attraction. The thickness or viscosity – denoted v – determines the fluid's resistance to flow.

External forces \mathbf{F} Any force that causes acceleration in the fluid not internal to the fluid itself.

2.1.2 Numerical methods

Numerical solutions of the NSE traditionally employ a Eulerian – or *implicit* – method. The velocity and pressure fields are represented at fixed points the terms of Equation 2.2 are calculated using finite difference schemes.

Foster and Metaxes [2] introduce an implicit approach where the fields are discretized into grids and updated using a finite difference approach.

Stam [3] [4] introduces and refines a semi-Lagrangian approach which can be described in the four steps shown in Figure 2.1.

The important distinction is in the advection step; unlike Foster and Metaxes [2] who use a finite difference approach for the advection step, this approach uses a Lagrangian – or *method of characteristics* – approach. Each grid point is treated as a particle that

addforce → *advect* → *diffuse* → *project*

Figure 2.1: The four steps of the semi-Lagrangian approach described by Stam.

is traced backwards in time using the velocity field to find the velocity at its previous location, advecting that velocity to the grid point.

The projection step, which solves for the effect of viscosity, requires a good numerical solver. However, convergence in most solvers can be slow, so it is desired to reduce the required accuracy in this step. The collocation of the velocity and pressure grids may lead to a number of stability problems [5], and the suggested solution is the use of staggered grids. Figure 2.2 shows the different approaches to grid representations.

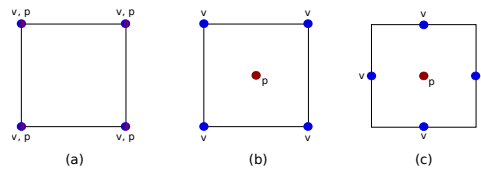


Figure 2.2: Visual comparison of collocated and staggered grids. v denotes velocity values, p denotes pressure values. (a) Collocated grid. (b) Partially staggered grid. (c) Fully staggered grid.

Boundary conditions

The simulation of flow using grids operates on a finite domain. This requires a model for behaviour at boundaries, since neither a finite difference approach nor a backwards interpolation handles values outside the domain. Typical boundary conditions include:

Dirichlet $y(\mathbf{x}) = f(\mathbf{x}), \quad \forall \mathbf{x} \in \partial\Omega$ The value y at the boundary is a defined by some fixed function $f(\mathbf{x})$ for all \mathbf{x} in the boundary domain $\Omega \subset R^n$.

Von Neumann $\frac{\partial y}{\partial \mathbf{n}}(x) = f(\mathbf{x}), \quad \forall \mathbf{x} \in \partial\Omega$ The normal derivative at the boundary is a defined by some fixed function $f(\mathbf{x})$ for all \mathbf{x} in the boundary domain $\Omega \subset R^n$. The normal derivative is defined by $\frac{\partial y}{\partial \mathbf{n}} = \nabla y(\mathbf{x}) \cdot \mathbf{n}(\mathbf{x})$.

For a thorough overview of these, see the theses by Young [6] and Elster [7].

Successive Over-Relaxation

A variant of the Gauss-Seidel method, Successive Over-Relaxation (SOR) is a method for solving linear equations that accelerates convergence. By decomposing the matrix, $A = D + L + U$, the equation can be rewritten as in Equation 2.4, where $\omega > 1$ is a constant called the *relaxation factor*. The left side is then solved for \mathbf{x} using forward substitution. The *relaxation factor* can be selected to significantly increase the speed of

convergence, but only for positive definite matrices, and $0 < \omega < 2$ is it guaranteed to converge. Press et. al. [8] is a practical introduction to the method.

$$(D + \omega L)\mathbf{x} = \omega \mathbf{b} - [\omega U + (\omega - 1)D]\mathbf{x} \quad (2.4)$$

2.2 Meteorological measurements

Many national meteorological institutes make gathered data publicly available. The Norwegian Meteorological Institute (MET Norway) has multiple Application Programming Interfaces (APIs) to pull this data from, providing raw data from ground measurement stations as well as historical data sets and averages. They also offer low-resolution radar imaging as animations over limited time spans. North American Aerospace Defense Command (NORAD) supplies higher resolution (and raw data) from radar imaging in the United States of America (USA). This section will cover some of the data available through these APIs and the formats they are supplied in.

2.2.1 Methods

There are two primary methods in use for gathering weather information. The simplest is the ground station; a collection of stationary measurement equipment typically consisting of a thermometer, a wind measurement device, a barometer and some means for determining precipitation (at least in liquid form). These stations provide raw data of weather as it occurs on the ground, but may also be deployed in-air with balloons.

The second form is the use of radar pulses. These stations transmit low frequency microwaves and analyze the reflections received back. A typical Doppler radar transmits waves of roughly ten times the size of water and ice particles (1-10cm) in bursts of about one microsecond. The occurrence of Rayleigh scattering – the scattering of light due to molecules in the air – at these frequencies means some of these waves will be reflected back towards the radar, and by analyzing the amount of reflection and the time between pulse and reflection, the radar can build a model of coverage around itself. The pulses spread the further from the station they travel, leading to diminishing resolution on the collected data; at 150-200 km distance a single pulse may scan roughly a cubic kilometer. Doviak and Zrnic [9] provide a thorough study on Doppler radar and weather observation in general.

The output from weather radar as available to the public is a reflectivity map, where values are in decibel. The reflectivity perceived by the radar Z_e is given in Equation 2.5, and is a function of the rain droplets' diameter D , the dielectric constant K of the targets and the droplet size distribution N . This function is the truncated Gamma function [10].

$$Z_e = \int_0^{D^{max}} |K|^2 N_0 e^{-\Lambda D} D^6 dD \quad (2.5)$$

From this the precipitation rate R can be calculated. It is additionally a function of the droplets' fall speed v , and is given in Equation 2.6

$$R = \int_0^{D_{max}} N_0 e^{-\Lambda D} \frac{\pi D^3}{6} v(D) dD \quad (2.6)$$

These equations have a simple relation, given in Equation 2.7, where a and b depend on the type of precipitation (and thus the values of Λ , K , N_0 and v).

$$Z = aR^b \quad (2.7)$$

2.2.2 Precipitation and cloud coverage

Precipitation data is available in two forms; radar imaging in the air and as water or snow levels on ground measurement stations. The ground data is available in mm per time period as water, while radar data is either available in raw reflectance as described in Equation 2.5 or in pre-processed precipitation data. Typically this is supplied in image form overlaid a map, however The US National Weather Service (US NWS) supplies raw reflectance values. The format of this data however is sufficiently complex that multiple tools are supplied along it to decode the data, such as the *TRMM Radar Software Library*¹ and *The Python ARM Radar Toolkit*².

When working with image-based output, such as the data available from MET Norway, the resolution of the available data is limited. A single value in these pictures corresponds to roughly one square kilometer, and the precipitation is graded on a 6-stop scale from “none” to “heavy”.

2.2.3 Wind

Wind is measured primarily at ground stations. Some information may be inferred from radar imaging deltas, but the low resolution of such data is an issue.

2.2.4 Public data

There are a number of public data sources for weather information. This section introduces the APIs of MET Norway and US NWS.

¹http://trmm-fc.gsfc.nasa.gov/trmm_gv/software/rs1/ (last accessed: 2016-06-20).

²<http://arm-doe.github.io/pyart/> (last accessed: 2016-06-20).

wsKlima & eKlima

MET Norway makes historical ground weather data available through a Web Services API. The data is supplied as Extensible Markup Language (XML), and requested through Simple Object Access Protocol (SOAP). The historical data is also available through eKlima, a web portal that makes access simpler. This data is returned as HyperText Markup Language (HTML) web pages and formatted in tables. Historical data is available in samples at 6-hour intervals.

MET weatherapi

In addition to the ground data, MET Norway supplies radar images, forecasts and any other publicly available data through a Representation State Transfer (REST) API. This data is returned either in XML form or as binary data. Radar is returned as Graphics Interchange Format (GIF) images or animations.

NWS NEXRAD

The US NWS supply Next-Generation Radar (NEXRAD) data in binary format as discussed in Section 2.2.1. These requests must be made manually, and after some processing time a link to the data is electronically mailed the user.

2.3 Terrain models

Capturing real-world terrain is usually done using Light Detection and Ranging (LiDAR), which utilizes laser light to map distance to ground from some plane- or satellite-mounted imaging device. Satellite-based scans typically suffer from slightly lower resolution than their plane-based counterparts, but the cost amortizes as the covered area grows and they are cheaper.

The Norwegian Mapping Authority (NMA) – a governmental agency responsible for geodesy, surveying, cadastre, cartography, and more – supplies terrain models of most of the country in 5- and 10-meter resolution. Equivalently, the British government supplies plane-based scans at 0.5-, 1- and 2-meter resolution of significant portions of the United Kingdom (UK).

The scanned data is supplied as two-dimensional fields of heights, however due to warp and absolute height values the formats require some pre-processing to turn into the height maps traditionally used in computer graphics, which are simple single-channel images.

2.3.1 USGS DEM Format

The United States Geological Survey (USGS)'s Digital Elevation Model (DEM) is a text-based format defined in [11]. The format is comprised of three records types:

A-records contain meta-information about the elevation data, such as the area's name, zone numbers, resolution and elevation units, the quadrangle containing the area and the resolution of the grid.

B-records contain the elevation data. Each describes a column of the quadrangle, and missing data is denoted with a `void` value of `-32767`.

C-records contain quality control data in the form of root-mean squared error for the data in the `B-records`.

Lien [12] implemented a utility to convert data in the USGS DEM format to 16-bit raw integer height maps, and describes its workings in detail.

2.4 Random number generation and noise

Since computers are deterministic machines, generating true randomness is impossible. However, randomness has many uses in simulation, statistical analysis, encryption and many other fields of computer science. Particularly the uses in statistics and encryption have led to a large selection of PRNGs. Knuth, Ch. 3 [13], provides an introduction to the topic.

In this section, the term noise is taken to mean a non-uniform pattern generated from randomness.

2.4.1 Random Number Generators

This section will examine some major categories of PRNG, explaining their strengths and weaknesses with a particular focus on properties important for GPU implementation. Uniform values are a desired property of the generators discussed.

A typical PRNG has two parts: the state transition function and the output function. Most of the PRNGs discussed in this section focus on the quality of only one of these parts.

Linear Congruential Generators

The Linear Congruential Generators (LCGs) family of PRNGs generate sequences of numbers, X , of the form given in Equation 2.8.

$$X_{n+1} = (aX_n + c) \pmod{m} \quad (2.8)$$

Three values are chosen by the author of the various LCGs; the modulus m which determines the range of the numbers output by the generator, the multiplier a and increment c , both in range $[0, m)$. Given the sequential nature of the generator, a starting value X_0 often called the *seed* is also needed. The special case where $c = 0$ is referred to as a multiplicative congruential generator, or Lehmer RNG after Lehmer [14].

The name is obvious upon inspection of (2.8); the sequence is a linear function of the three authored constants a , c and m , and the previous value X_n . A LCG has a fixed period of at most m , since a single input X_n will always return the same X_{x+1} . Knuth, Ch. 3.2.1 pp. 10-26 [13], showed that the full period can only be achieved if three conditions are met: c is relatively prime to m , $a - 1$ is divisible by all prime factors of m , and $a - 1$ is a multiple of 4 if m is a multiple of 4. The sequence remains unchanged, and the seed X_0 is set by the user to determine where in the sequence to start.

Selecting m is a question of performance. The range of X is non-negative, and arithmetic overflow on unsigned integers truncates the most-significant bit thus providing free modulo if m is the size of a machine word. Modern GPUs and CPUs typically implement fused multiply-adders that have a throughput of one per cycle or better as documented in [15], and the combination of these fact make LCGs very fast. This is the main strength of these generators.

However, Gershenfeld, Ch. 5.3.2 [16], shows that not all bits in a generated number of LCGs are equally random. This points to the weakness of LCGs, they do not produce very *good* random numbers. Section 2.4.1 examines PRNG quality and references data that supports this.

Multiple Recursive Generators

Multiple Recursive Generators (MRGs) of order k take the form given in Equation 2.9.

$$X_{n+1} = (a_1X_n + \dots + a_kX_n) \pmod{m} \quad (2.9)$$

They are superficially similar to LCGs but have significantly larger periods. L'Ecuyer [17] introduces the MRG32k3a PRNG which has a period of order 2^{191} .

Linear-Feedback Shift Register

In a Linear-Feedback Shift Register (LFSR) the state is constructed by shifting the previous state over, adding some new bits to one end. The input bit (or bits) in a LFSR is a linear function previous state. The initial state is typically set with a user-specified seed value. The most commonly used linear function is a bit-wise exclusive-or.

The LFSR113 PRNG described in [18] has a 128-bit state where each 32-bit component is self-shifted and exclusive-or'ed as well as exclusive-or'ed with a shifted version of the previously treated component. The final number is a exclusive-or of the four 32-bit components after these changes are made.

The Xorshift family

Marsaglia [19] introduces the Xorshift PRNG, which generates the next number in its sequence by repeatedly taking the exclusive-or of bit-shifted versions of itself; it is a special form of LFSR. Bit-arithmetic like shifting and logical operations are typically fast on modern CPUs but not as fast on GPUs, with large variances between vendors [20]. Straight Xorshift produces better randomness than LCGs but do have statistical problems. A few variations have been introduced that include a non-linear transformation to improve upon them, as in the case of Xorshift* which introduces an invertible multiplication as a final step [19].

Mersenne Twister

The Mersenne twister is worth a mention for historical reasons. Introduced in [21] it was one of the first PRNGs to avoid major problems and has a very large period of $2^{19937} - 1$. It grew popular, even though it also has problems, as seen in Section 2.4.1.

The algorithm has a large state (624 values in the typical implementation) that is initialized using an LCG. Whenever a value is retrieved, a tempering transform is applied to the next value in the state. If the state is out of values to use, the complete state is changed by applying a twisted generalized linear-feedback shift register.

The large state and initial generation cost indicate that the Mersenne Twister does incur significantly higher memory and performance overhead than LCGs.

Matsumoto [22] introduces the Mersenne Twister for Graphics Processor (MTGP) PRNG. While this version performs better on GPUs, the statistical issues of the Mersenne Twister pertaining to linearity remain.

Ciphers

Cryptography requires a PRNG to stand up to adversarial analysis. Given the knowledge of which PRNG and the absence of knowledge about the initial seed, the adversary should have only a negligible advantage in distinguishing output from a random sequence. Typical approaches are block and stream ciphers.

Particularly block ciphers are of interest even when cryptographic security is not required. Advanced Encryption Standard (AES), described in [23] and specified in [24], has hardware support in some CPUs making its performance almost comparable to LCGs and LFSRs while generating good random numbers. GPU support is however lacking.

The Philox PRNG is introduced in [25]. It is based on the Threefish block cipher – introduced in [26] – but significantly reducing the number of rounds, taking advantage of not needing to be cryptographically secure, only statistically good.

Eidissen’s PRNG

The HPC-lab snow simulator has previously used a custom made PRNG due Eidissen [27] when re-positioning snow particles on the GPU. This uses the previous position of the particles as its state, thus removing the need for extra state.

Particle re-positioning always generates a 3-vector $\langle x, y, z \rangle$, where $y = 1$ (particles are inserted as the top of the simulation volume). The x and z values are generated from 12 bits of their previous values and 8 bits from the previous y -value. The 2 least significant bits are zero. These bits are inserted directly into the mantissa of a floating point number with a constant sign and exponent, producing uniform values in range $[0, 1]$

Entropy is introduced into these values by floating point rounding and instabilities in the lower bits of the positions are the particles interact with wind and gravity on the way to the ground. However, if the particle is immediately re-positioned repeatedly (thus no entropy is introduced through intermediate operations on it), the output will not only be predictably, but in fact constant after only 4 sequential calls. Table 2.1 shows the values of x for sequential calls to this PRNG.

call	sign	exponent	mantissa
0	0	01111110	RRRRRRRRRRRRRRRRRRRRRRRR
1	0	01111110	0XXXXXXXXXXXXXXXXYYYYYY00
2	0	01111110	0XXXXXXXXYYYY00111111100
3-N	0	01111110	01111111111001111111100

Table 2.1: The result of sequential calls to Eidissen’s RNG. R means the value is random, X and Y mean bits from the original position’s X and Y values.

Permuted Congruential Generators

For completeness, the PCG generator family introduced in [28] is included. These generators attempt to combined both a strong state transition function and output function to achieve better statistical performance. The state transition function may be a simple LCG or a MRG, while the output function is a new technique dubbed *permutation functions on tuples*.

Quality tests

To test the quality of PRNGs a number of large tests have been developed. The TestU01 suite introduced in [29] implements a large number of these tests and has been used to analyze most if not all PRNG in wide spread use. Table 2.2 is a collection of values from [29], [25], and [28], that show the performance of some variations of the PRNGs discussed in the previous sections on the various test suits within the TestU01 framework.

PRNG	$\log_2 p$	t 32b	t 64b	SmallCrush	Crush	BigCrush
drand48	48	4.1	0.65	4	21	N/A
Xorshift (64-bit)	64	4.0	0.8	1	8	7
LFSR113	113	4.0	1.0		6	6
MRG32k3a	191	10.0	2.1			
MT19937	19937	4.3	1.6		2	2
AES (Key counter mode)	130	10.2	5.2			
PCG XSL RR 128/64	126					
Philox4x32-10	128					

Table 2.2: Results of `TextU01` test suites. Each row is results for a single generator over multiple test suites, where each column denotes the number of p -values outside the range recommended by L’Ecuyer [29]. The two PRNGs below the second line have no timings for the same hardware, however their respective papers claim PCG doubles Xorshift’s performance, and `philox` performs similarly to MRG32k3a.

Superficially it can also be useful to visually inspect the results to look for patterns in the output.

GPU vs CPU

When working in highly parallel environments the naive approach is to have separate states per thread. This can be 4 bytes per thread in the case of a 32-bit LCGs or 19337 bytes per thread in the case of the `Mersenne Twister (19337)`. This is a strong incentive for a simple state transition in PRNGs for the GPU, to the point where the removal of the state completely could be advantageous being replaced by a predictable seed such as the system clock, potentially augmented by thread identifiers when used in parallel. Such an approach requires a strong output function, and cryptographic ciphers deliver this, making them prime candidates for GPU usage. `philox` is an examples of such a PRNGs.

There are libraries that simplify the use of some of the more GPU-suited PRNGs on both the CUDA and OpenCL platforms.

cuRAND is a CUDA library that contains `xorwow`, `mrng32k3a`, `mtgp32`, and `philox 4x32-10` implementations. The `xorwow` generator is a member of the Xorshift family of generators.

clRNG is a OpenCL library developed by Advanced Micro Devices, Inc. (AMD) that contains `mrng31k3p`, `mrng32k3a`, `lfsr113`, and `philox 4x32-10` implementations.

2.4.2 Probability Density Functions and Distributions

The PRNGs discussed above aim to generate uniform randomness, however it is often desirable to have a non-uniform Probability Density Function (PDF), such as a normal distribution. When producing random numbers given a PDF, a basic technique is the rejection method [8, Ch. 7.3]. This technique works by generating uniform random sample v in the domain, then generating an additional uniform random sample p that is compared to the PDF f at v , rejecting v if $f(v) < p$.

The rejection method has simplicity as its primary advantage. The main drawback is a potentially large number of rejected samples if the PDF is highly concentrated. It also disqualifies Eidissen's PRNG from use since it does not handle repeated calls nicely, as discussed in Section 2.4.1.

2.4.3 Noise

In the context of computer graphics the term *noise* refers to a procedurally generated pseudo-random texture. There are a variety of variations suited for different scenarios, and this section describes two commonly used noise types. For further reading on the subject, [30] covers a wide noise modelling in detail.

Perlin noise

Introduced by Perlin [31], this noise uses a combination of polygonal surflets to create a smooth noise. A surflet is a sparse representation of a multidimensional function with smooth discontinuities [32], and in Perlin noise they are constructed by taking the product of a gradient and a falloff function. The gradient has random orientation, and the falloff function should be polynomial and separable. Perlin noise is defined in two dimensions by centering surflets on the integer points of a two-dimensional lattice, each surflet having an extent of 2 in both dimensions. The value of the noise at non-integer positions is a summation of the value at the corners of the integer cell that contains it.

Perlin noise is a special type of a larger category of noise called *value noise*. In general, *value noise* generates a lattice of points with random values and interpolates between these points in the noise function.

Fractal Brownian Motion

By applying the Weyl integral to white noise a rougher smoother noise pattern can be generated. The Weyl integral is an operator of order s defined on a Fourier series by Equation 2.10.

$$\sum_{n=-\infty}^{\infty} (in)^s a_n e^{in\theta} \quad (2.10)$$

This is often simplified to the summation in Equation 2.11 in practice, where $f(x)$ is a simpler noise function that attempts to model white noise. c is a constant near 2, however in practice, it is rarely exactly 2, to compensate for fast, but poor quality, PRNGs.

$$fbm(x, N) = \sum_{n=1}^N \frac{1}{2^n} f(x) c^n \quad (2.11)$$

Fractal Brownian Motion (FBM) relies on successive iterations being dependent, meaning an increasing pattern in one step means the next step is likely to also increase. This leads to larger structures in the noise pattern, however the addition of lower amplitude high frequency noise means this pattern is very suitable for visualizing clouds and fog.

2.5 Voronoi tessellation and their duals

A Voronoi tessellation is a decomposition of space X into a set of regions R_k given a set of seed points P_k for which equation 2.12 holds for some distance function $d(x, P)$.

$$R_k = \{x \in X \mid d(x, P_k) \leq d(x, P_j) \forall j \neq k\} \quad (2.12)$$

This segments the space into regions in which every point is closer to the regions seed point than any other seed point. Aurenhammer [33] provides a detailed survey of Voronoi diagrams.

The dual of a Voronoi tessellation is known as the Delaunay triangulation, and is defined as a triangulation of the set of seed points \mathbf{P} such that no seed point in \mathbf{P} is inside the circumcircle of any triangle in the triangulation $DT(\mathbf{P})$. Connecting the center of these circumcircles produces the Voronoi tessellation. The technique extends into n -dimensional space. Berg et. al., Ch. 9 [34], discusses Delaunay triangulation and introduces an algorithm to calculate it, while Shewchuk [35] describes Triangle, a mesh generator and Delaney triangulator program written in C. The CGAL library³ also support Delaunay triangulation.

2.6 Stereoscopic rendering

Current consumer display technology is confined to Two dimensional (2D) surfaces. Creating the illusion of depth on such displays is accomplished by rendering the scene with two cameras corresponding to the viewers two eyes. By then masking the image so each eye sees the view corresponding to it, the brain can be fooled. This section reviews some of the techniques used to accomplish this masking, particularly in respect to rendering virtual worlds such as games or simulations.

³Available from <http://www.cgal.org/> (last accessed: 2016-06-20).

2.6.1 Anaglyph stereoscopy

One of the oldest techniques uses color to mask the images and dates back to Rollmann [36]. Using differently colored glass or plastic film between either eye and the display, the content can of the left view can be encoded in the right eyes filter only, such that the filter masks the left view's content, and vice versa. This has the advantage that creating glasses with such film is cheap, and that no special projection technology or display is required. However, the color range of the image is drastically reduced.

2.6.2 Polarization stereoscopy

Instead of encoding the images in a single color channel, Kaiser [37] suggests the use of glasses with lenses of different polarization. Two images are superimposed on the same screen, but polarized using two orthogonal filters. The viewer wears linearly polarized glasses that match the polarization filter for their corresponding eye. This allows the full use of the color spectrum, but also requires specialized projection or display equipment capable of polarizing the light differently for each eye's view. The glasses, being passive as the anaglyph ones, are relatively cheap to produce.

2.6.3 Active-shutter stereoscopy

NVIDIA [38] describe the use of active shutter glasses in `Nvidia 3D vision`. Active shutter glasses' lenses are crystallized Liquid Crystal Display (LCD) that block vision in synchronization with the display. Left and right eye's views are displayed in alternation, and the lenses gate the eye so they can only see their respective images. While this does allow for the full color spectrum usage, LCDs are not completely opaque, and their state change is not instant, leading to a general darkening of the view. The alternation of images also requires the display to refresh at double the frequency. Consumer implementations of this techniques such as `NVIDIA 3Dvision` work with any 120Hz display, however consumer standard displays are only 60Hz-capable.

2.6.4 Multi-screen stereoscopy

All the techniques above use a single display surface that is gated from either eye with a filter. However, by mounting the display close enough to the viewer's eyes that the node blocks view of the other display, multi-display techniques can eliminate the need for filters all together. Devices using these techniques are called Head-Mounted Displays (HMDs).

HMDs require the displays to be so close to the retina that the individual pixels may be discerned, a phenomenon known as the *screen-door effect*. Such devices need a much higher pixel density than typical computer displays, and this specialized need has traditionally made them costly. As the smartphone and tablet market started taking interest in high density displays however, this cost has reduced, and numerous HMD products aimed at the consumer price range have launched or aim to launch in 2016 [39].

The illusion of depth in HMDs is significantly stronger than with traditional single-display techniques, and introduce a wide range of technical challenges. As the viewer has a stronger sense of *presence*⁴ in the world being displayed, many users experience motion sickness [41], which the HMD vendors attempt to alleviate by tracking the users head movement and rotation and reflecting it in the virtual world. Input to display latency has been shown to have a significant impact on the motion sickness problem [42], leading the HMD vendors to use separate software paths to render to their devices and often re-project the rendered images in post-processing to minimize head-tracking latency. These problems have lead to a figurative arms race in HMD driver software as they got more complex, which has limited support to Microsoft Windows only.

2.6.5 Performance implications

Most single-display techniques have very predictable performance implications; the final view of the scene as seen by the camera must be rendered twice. However, these views are not independent, and by accounting for this a doubling in frame render time should be avoidable. Any scene culling performed can be shared for a slightly expanded version of either camera's frustum, and any view independent tasks such as shadow map rendering, animation do not need to be duplicated. There may also be some overhead in additional framebuffer swaps (in the case of active-shutter stereoscopy) or post-processing (in the case of coloring the output for anaglyph viewing).

Multi-display techniques have more complex performance implications. Each eye's display may have a lower resolution than typical consumer displays⁵ meaning the rendering overhead is less severe, however the HMD driver software will incur extra costs to do head-tracking, re-projection to account for it, and apply post-process filters that warp the image to compensate for lenses in the HMD devices. These allow for lower density displays (by increasing the eye-display distance), as well as increasing the relative density of the center of the eye's view to the peripheral vision.

2.7 GPU and GPGPU computing

Ever since the early days of arcade games in the 1970s, dedicated hardware has been used to accelerate the rendering of virtual worlds. As the video game industry progressed, the GPU got more powerful; first with dedicated rasterization and filtering hardware to handle three-dimensional worlds, and later with fully programmable shading pipelines. As the hardware got programmable the term GPGPU surfaced, and modern GPUs are used in many fields for compute-heavy applications, not just graphics.

⁴*Presence* in the field of virtual reality is defined as "the degree to which participants feel that they are somewhere other than where they physically are when they experience the effects of a computer-generated simulation [40]

⁵The Oculus Rift CV1 has a resolution of 1080×1200 per eye, while current consumer high end displays are 3840×2160 and industry standard is 1920×1080

2.7.1 Hardware model

GPU come in many forms, from mobile and integrated devices that share a chip with the CPU to dedicated hardware. This section primarily discusses high-performance GPUs, which are dedicated cards due to the heat they generate requiring separate heat sinks and cooling. These cards are highly parallel processing units that prioritize throughput over latency. They excel at floating point arithmetic, and have very high bandwidth.

The card is connected to the CPU and general purpose memory through the Peripheral Component Interconnect Express (PCI-E) bus. The latest available version of PCI-E – 3.0 – has a theoretical maximum bandwidth of 15.754 GB/s (the 4.0 version, expected in 2017, doubles this), which is comparable to CPU to Random Access Memory (RAM) bandwidth, but the latency is several orders of magnitude larger (comparing the the $\sim 100ns$ reported for CPUs RAM access in [43] with the $\sim 10\mu s$ reported for GPUs in [44]). The GPU itself has dedicated memory to avoid round trips to the CPU. The internal theoretical maximum memory bandwidth on the NVIDIA GTX 1080, a high-end GPU from 2016 is 320 GB/s. This is an order of magnitude more than to 34 GB/s of the Intel i7 6770HQ, a high-end CPU from 2016. Figure 2.3b shows the memory bandwidth of GPUs and CPUs over time.

This high memory bandwidth it utilized by going very wide; the GPU is designed around a large amount of threads working in parallel. The real strong point of the GPU is its floating point operation performance. Figure 2.3a show the theoretical floating point performance of high-end GPUs and CPUs over time. The performance does come with some caveats, and these are discussed in Sections 2.7.3 and 2.7.3.

2.7.2 OpenGL

Given its history, the GPU has traditionally been accessed through the usage of graphics APIs such as OpenGL or Microsoft DirectX (DirectX). The wide variety of GPU architectures expose their features to the application by supplying entry points corresponding to OpenGL functions in their drivers that the application calls.

Programming model

The modern graphics pipeline has multiple stages, but before these are addressed a few things must be supplied to the pipeline. The pipeline assumes the rendering of some number of primitives given a buffer with vertex information and either implicitly or explicitly defined indices. Modern versions also require explicitly defined shaders for each shading stage. The result is written to a framebuffer, which may be the default back-buffer that is displayed on any attached display, or a framebuffer-object; a writable texture. The stages of the pipeline follow:

Input assembler Read or generates index data from either an index buffer or implicit information in the draw call, such as when rendering a triangle strip. Also reads vertex data.

Vertex shader Given per-vertex attributes read from the vertex buffer or buffers and uniform read-only parameters shared across all invocations, this programmable stage outputs processed vertex data for the next stage.

Primitive assembly Assembles the vertices that make up primitives into a single unit; the primitive. Also clips and culls the primitive if it falls outside the viewport.

Hull shader, tessellator stage, domain shader, geometry shader These four stages allow programmable transformations to the geometry, such as tessellation or instancing.

Stream out The output of the geometry shader may be streamed out to a buffer ending the pipeline here.

Rasterizer Rasterizes the primitives; this computes barycentric coordinates and interpolates the values. Generates the per-pixel input for the pixel shader stage.

Fragment shader Gets the interpolated vertex data from the rasterizer and generates output pixel colors.

Output merger Performs alpha blending on the output pixels from the pixel shader and writes the output to the attached framebuffer.

All shader stages may access texture memory. Textures are supplied to the GPU through OpenGL as flat arrays of values, but are often laid out in tiled formats such as Morton Order internally. These textures are accessed in the shaders through samplers, which contain information about the layout, the address of the data and information about how to filter the textures. GPUs support nearest-neighbor, bi- and tri-linear filtering as well as varying quality-levels of anisotropic filtering in hardware. When requesting texture data in the shader, this filtering is performed completely transparently before the result is returned.

OpenGL 4.3 introduces compute shaders, which are outside the pipeline. These read and write directly from un-ordered access views, and have an execution model similar to that of OpenCL and CUDA, discussed in Sections 2.7.3 and 2.7.4

Giessen [45] gives a thorough introduction to the modern graphics pipeline.

Memory model

Traditionally, OpenGL has a few different memory types. These include

Texture memory As described in Section 2.7.2, this memory is generally laid out differently on the GPU than it is received, and the creation of it used to implicitly create a sampler. `ARB_separate_shader_objects`⁶ introduced features that allow the application to manually handle this. Individual values of a texture are referred to as *texels*.

⁶https://www.opengl.org/registry/specs/ARB/separate_shader_objects.txt (last accessed: 2016-06-20).

Pixel buffer objects and framebuffer objects Writable textures. It is important to note that these may be either readable or writable in a shader invocation, but not both, as no memory fencing primitives are available to the user in non-compute shaders.

Vertex, index and instance buffers These buffers are used to describe geometry. Vertex and instance buffer in particular have additional information associated with them, related to attribute types, sizes and strides, while the index buffer only has element type as meta information.

Uniform buffers Traditionally uniform parameters and attributes for shaders were set by value, but `ARB_uniform_buffer_objects`⁷ introduced uniform buffers that allow the user to pack the values into buffers themselves. By value is still used, but it is likely that the driver packs this into buffers behind the scenes.

The inclusion of compute shaders in OpenGL 4.3 also introduces shared memory and Shared-Storage Buffer Objects (SSBOs). The latter are similar to Uniform Buffer Objects (UBOs), but may be significantly larger (at up to 16MB, compared to the 16kB of UBOs), are writable from shaders and support atomic operations.

2.7.3 CUDA

To harness the tremendous parallel computing power of the GPGPU, NVIDIA launched the CUDA⁸. Its purpose was to provide a way to use the GPU for general-purpose computing without necessarily using the graphics pipeline. CUDA supports inter-operability with OpenGL, but does not require a OpenGL context.

Execution model

In CUDA, the programmable code that runs on the GPU is called a kernel. Kernels are written in an annotated version of C or C++ called `CUDA C/C++`, but support for other languages such as FORTRAN exists. Unlike OpenGL shader which are compiled by the driver at run-time, CUDA kernels are compiled at compile-time by a separate compiler, the `nvcc`.

Kernels describe the execution as it happens on a single thread. Host code, running on the CPU, executes a kernel by passing it its parameters, as well as a number of threads invocations to run, and the block size to use. CUDA has synchronization primitives and shared memory for threads within a block, but no support for this between blocks. During execution, up to 32 threads from the same block are executed together in a *warp*.

While CUDA kernels may have branches in them, execution in a warp does not diverge. This has the effect that both branches are executed fully – unless **all** threads in the warp

⁷https://www.opengl.org/registry/specs/ARB/uniform_buffer_object.txt (last accessed: 2016-06-20).

⁸CUDA launched as an acronym but NVIDIA has subsequently dropped the expanded name, and CUDA is not the full name of the product

can be determined by the scheduler to take the same path – and only applying side-effects after the fact with conditional moves. This is one of the caveats hinted at in Section 2.7.1.

CUDA makes no guarantees about the execution order of the threads. In practice, the order will vary widely, as this is the primary way the scheduler hides the latency of memory accesses. When all threads in a warp are guaranteed to execute the same code in parallel, they can be run as far as they can before needing the result of a memory fetch, only to suspend the entire warp until all those fetches have gotten returned. This way the large bandwidth may be used by performing the memory requests in bulk, hiding the latency by performing other work while waiting.

Memory model

The CUDA memory model consists of three categories:

Global This is memory in the GPU's global memory. It is large, but accesses incur significant latency. Texture memory is a special case; it is stored in global memory but accessed through the filtering unit.

Shared A smaller block of memory is available near the execution units. This is shared between all threads in a warp. It is significantly closer than global memory.

Local The fastest memory; the registers. Each execution unit has a limited amount, and if threads use too much the warp may contain less threads to compensate.

If register contention is high, a warp may not be filled with the full amount of threads because the register file is shared between all threads of a warp, leading to wasted compute power. However, if register contention is low, registers may be used as cache to improve performance, as described in [46].

Bindless textures

CUDA 7.5 introduced bindless textures. This is a feature primarily motivated by the graphics community, where a large amount of draw calls are frequently made which access different textures. Since texture samplers is state in OpenGL and may not change during a draw call, and the number of samplers bound at any given time was limited, this led to splitting of large draw calls into smaller ones. Every draw call incurs CPU overhead, and this was often a limiting factor in graphics applications. Bindless textures no longer binds samplers to textures as state pre-invocation, but makes them both arguments of the sampling function. This reduces the number of draw calls as the number of different textures accessed in a single draw call no longer has an upper bound⁹. CUDA exposes this functionality by changing the way textures are passed to the kernels. Where pre-7.5 versions required accessing a symbol visible to the kernel (essentially a global `cudaArray` bound to a texture), post-7.5 passes a small texture object as a parameter to the kernel.

⁹There are still limitations related to virtual pages and texture array sizes in OpenGL, but these are not relevant to the CUDA discussion

The feature requires hardware support. For NVIDIA, any card beyond the GeForce 8 series is supported, while AMD supports it in all GCN based hardware and Intel does not support it at all¹⁰. The section "Interactions with NV_gpu_shader5" in ARB_bindless_texture¹¹ indicate that AMD has an additional restriction on OpenGL bindless textures; all threads in a warp must access the same resource (texture).

2.7.4 OpenCL

While CUDA is NVIDIA's proprietary GPGPU compute library, OpenCL is an open standard developed by the Khronos Group; the same body that governs OpenGL. Its functionality is similar to that of CUDA, but the terminology differs slightly, as summarized in Table 2.3.

CUDA term	OpenCL term
GPU	Device
Shared memory	Local memory
Local memory	Private memory
Kernel	Program
Block	Work-group
Thread	Work-item

Table 2.3: The corresponding terminology of CUDA and OpenCL

OpenCL is supported on a wide range of hardware, not only on GPUs, but also on CPUs. In part due to this, and in part due to API design choices, OpenCL setup code is more verbose than CUDA, and while CUDA C/C++ is compiled at compile time, OpenCL C is compiled at run-time by the driver.

OpenGL inter-operability

OpenCL also has OpenGL inter-operability, allowing OpenGL textures to be used in OpenCL programs. This allows for use of hardware filtering to be used. The feature is defined by OpenCL extension `gl_khr_gl_sharing`, and requires at least an OpenGL 3.1 context.

2.7.5 Performance analysis

Profiling CUDA or OpenCL code using the CPU clock is possible but inaccurate; kernel dispatches are asynchronous and the scheduler may reorder operations. The application

¹⁰Naturally, this describes bindless textures in general (through OpenGL extensions) as neither AMD nor Intel support CUDA

¹¹https://www.khronos.org/registry/specs/ARB/bindless_texture.txt (last accessed: 2016-06-20).

can synchronize the GPU and CPU explicitly before and after a call, however this ensures that all work in the GPU pipeline is flushed and stops the driver from performing some of the latency hiding that makes the GPU fast. Both OpenCL and CUDA have event timers for this purpose. In CUDA, two events are created surrounding the section of interest. When the timing information is wanted, which may be immediately or significantly later, the latter event is used as a synchronization barrier, and the elapsed time between them can be queried. In OpenCL API calls take events as arguments, which may then be used to query profiling information such as the start and end time of the event.

The CUDA framework also ships with both graphical and command-line profiling tools that provide detailed information on kernel execution timers and counters.

2.8 Cloud rendering

The accurate rendering of light transport in participating media has seen extensive research in the computer graphics research. Jarosz [47] provides a good overview of the theory, while [48], [49], and [50] describe some of the advanced techniques used in offline rendering, accounting for multiple scattering and anisotropy. Participating media is modeled as a micro-particles. The change in radiance due to photons' interactions with these particles is considered not on individual photons, but along rays through the medium, and is calculated as four individual terms:

Absorption covers the loss of photons due to absorption into the micro-particles.

Emission covers the gain of photons due to excitation by the micro-particles.

Out-scattering covers the loss of photons due to reflection away from the ray by micro-particles.

In-scattering covers the gain of photons due to reflection into the ray as photons travelling along other rays collide with the micro-particles.

One ray's out-scattering is another's in-scattering. The net loss defined by out-scattering and absorption is called *extinction*. Chapter 4 in [47] includes formulae describing these terms.

In real-time applications, clouds have traditionally been handled using simpler techniques. It is usually known whether the camera can be inside the clouds or not, and Quilez [51] describes a technique for the simpler case where it may not. In this case, the clouds can be rendered as a layer on the skybox, and a simple accumulating ray-march can be performed using noise or a texture generated offline. In the case that the camera may be close to or inside the cloud layer, Harris [52] describes the use of translucent billboards. Here, simple noise-textured primitives – typically a quad – are rendered in place of the clouds; they are oriented towards the camera in the vertex shader and the GPU hardware support for alpha blending is used to accumulate contribution over multiple quads. The problem of varying cloud density becomes a problem of distributing these primitives such that when

blended together the color corresponds to the cloud density. While this approach creates an illusion of volume in the cloud layer the skybox-based technique does not, it still suffers from problems when the camera is inside the clouds, as Harris highlights in Ch. 3.1.1 of [52].

The increasing arithmetic performance of GPUs have enabled the use of ray-marching in volumetric textures or noise within a volume in real-time applications. Notably, Wróński [53] describes ray-marching through a volumetric texture centered on the camera to render fog in contemporary video games, while Quilez [54] generates FBM on the GPU while ray-marching to render high quality clouds.

2.8.1 Ray-marching volumetrics

This technique consists of constructing a ray – an origin and a forward direction – and stepping along the directional vector from the origin, adding the contribution of the volumetric data as it is encountered. In the context of cloud rendering, the ray is constructed from the camera and a near-plane where the volumetric data starts. As the ray is marched, the scattering of the participating media is evaluated and the resulting color contribution to the ray is accumulated.

2.9 The HPC-Lab Real-Time Snow Simulator

This thesis improves on the previous work by former HPC-Lab students on the real-time snow simulation initially presented by Saltvik [55]. This section provides a brief historical overview of previous work on the simulator, a technical overview of the implementation prior to this thesis, and an overview of the simulator’s current and potential uses and applications.

2.9.1 History

The HPC-Lab Snow Simulator was initially described by Saltvik in [55] and [56]. This thesis introduced the snow model and based its wind simulation on a smoke simulation by Vik [57] [58], and the work of Stam [3], [4]. It was implemented on the CPU using hyper-threading for performance, with a software renderer. Eidissen [27] ported the simulator to the GPU using CUDA, introducing an OpenGL renderer and stereo rendering component. The re-positioning of snow particles was changed to use Eidissen’s PRNG discussed in Section 2.4.1. The GPU version could handle significantly higher resolution wind fields. Vestre [59] ported the GPU version to OpenCL, effectively removing the GPU vendor lock-in of CUDA. This thesis intended to make the simulator support mobile platforms, however while the port to OpenCL help enable this, mobile GPUs typically support OpenGL ES which differs from regular OpenGL and the simulator does not appear to have every been tested on a more mobile device than a laptop computer. The thesis also

attempted to improve performance by sorting particles on the GPU, and while this did not have the intended performance enhancing effect, it is interesting research.

In 2013 Babington [60] and Nordahl [61] improved the simulator's visualization. Snow particle rendering using procedural noise instead of predefined sprites, tri-planar texture-projection on terrain, shadow mapping, and dynamic level of detail of the terrain were all added. In [59] complicated and undocumented code was mentioned as a hindrance, and [61] covers a refactoring and documentation effort to alleviate this for future projects, modularizing simulation and rendering for simpler replacement of components. Lien [12] added the pre-processing tools to import real-world terrain data as discussed in Section 2.3.1. Mikalsen [62] ported the simulation to Open Accelerators (OpenACC), a directive-based parallel programming standard similar to Open Multi-Processing (OpenMP) with GPU support – as well as producing a sequential version of the simulator –, however since this work was done in parallel to the work in [61], it was not properly merged back into the main code-base built on by subsequent projects.

Following work by Krog [63] [64] on avalanche movement – implemented independently of the HPC-Lab snow simulator – Boge [65] investigated avalanche prediction. The snow accumulation was remodeled to use a fine-grained grid normal to the ground, which is used to simulate fracture progression using a finite element method. The structure stores snow temperature and humidity for use in the avalanche prediction, but due to time constraints these properties do not correctly propagate through the structure. Visualization of the calculated probability of shear and powder avalanches were added to the renderer.

Prior to this thesis, Kerr [66] did some general documentation, code refactoring, added CMake – a multi-platform build system – support, and fixed a large number of outstanding bugs.

2.9.2 Technical overview

This section outlines the architecture and algorithms of the HPC-Lab Snow Simulator as it was prior to the work in this thesis.

Snow model

In air, the snow is modeled as individual particles as described in Ch. 2.4.1 of [27]. Each particle is influenced by four forces:

$\mathbf{F}_{gravity}$ The gravitational force on the particle, determined by its mass m , which is randomized at simulator initialization, and the gravitational constant g , which is configurable at run-time. It is directed down along the negative Y-axis.

\mathbf{F}_{lift} As the particles fall they interfere, and the resulting vorticity applies a chaotic force on the particles. This leads to irregular motion. As this force is caused by turbulence at a scale the simulator is unable to handle at real-time, it is modeled with a visually pleasing replacement; \mathbf{V}_{circ} . This is a circular velocity component that diminishes

as the particle's velocity increases, described by a radius R and an angular velocity ω .

F_{drag} The particle does not fall in a vacuum, thus a drag force is applied. Eidissen [27] shows that the drag force is a function of a particle's mass, its terminal velocity $-V_{max,y}$ and the difference in velocity between the particle and its surrounding fluid. The terminal velocity, like the mass, is randomized per particle at simulator initialization.

F_{buoyancy} The difference in density between objects and their surrounding medium causes buoyancy, however Eidissen [27] argues the force for snow flakes is small enough to be ignored.

A particle is then described by two fields of 4-wide floating point vectors, as described in Table 2.4. In addition, two arrays of 32 radii R and angular velocities ω are used to calculate V_{circ} . These are indexed by the GPU thread index during particle update, and filled with random values at startup (given the ranges described in [27]).

Float component	0	1	2	3
Field 0	V_x	V_y	V_z	$V_{max,y}$
Field 1	P_x	P_y	P_z	θ

Table 2.4: V denotes velocity, P denotes position. θ is the current angle around the center of rotation for V_{circ} .

The snow particles' velocity and position are updated after the wind simulation step by the integration described in Equations 2.13 and 2.14, where $\mathbf{a} = \frac{\mathbf{F}_{gravity} + \mathbf{F}_{drag}}{m}$. The wind value needed in both this integration and the advection step of the wind simulation take advantage of GPU hardware sampling of texture memory for fast tri-linear interpolation of values in the CUDA version, while the OpenCL version does this manually in the kernels.

$$\mathbf{p}^{t+\Delta t} = \mathbf{p}^t + \mathbf{V}_{snow}^t + \mathbf{V}_{circ}^t \Delta t + \frac{1}{2} \mathbf{a} \Delta t^2 \quad (2.13)$$

$$\mathbf{V}_{snow}^{t+\Delta t} = \mathbf{V}_{snow}^t + \mathbf{a} \Delta t \quad (2.14)$$

When snow particles hit an obstacle, they are re-positioned to the top of the simulation volume. The y-position is set to the height of the highest non-boundary voxel, while the position in the xz-plane is randomly selected. The particle's other properties (velocity, terminal velocity, mass, rotation) are maintained. When particles hit the outer boundaries of the simulation volume (any of the vertical planes bounding it), they wrap.

Wind simulation

Eidissen [27] makes the assumption of zero viscosity, simplifying the NSE to the “incompressible Euler equation with constant and uniform density”. Additionally, it is assumed that the force of gravity on the fluid (the air) is negligible compared to the advection and pressure forces. This leads to Equation 2.15.

$$\frac{\partial \mathbf{v}}{\partial t} = -(\mathbf{v} \cdot \nabla) \mathbf{v} - \nabla p \quad (2.15)$$

Following the work by Stam [4] and Vik [58], Eidissen [27] solves this in three steps:

Advection Calculate $\mathbf{F}^{(n)} = \mathbf{v}^{(n)} - \delta t (\mathbf{v} \cdot \nabla) \mathbf{v}$, the intermediate velocity field.

Solve Poisson Use the intermediate field $\mathbf{F}^{(n)}$ to solve $\Delta p^{(n+1)} = \frac{1}{\delta t} \nabla \cdot \mathbf{F}^{(n)}$.

Projection Calculate the pressure force $-\nabla p$ and modify the intermediate velocity field $\mathbf{F}^{(n)}$ to get the $\mathbf{v}^{(n+1)}$, which is divergence free.

Dirichlet boundary conditions are used at the world bounds, and the value set at the boundary is constant for a single iteration step, but may vary between steps. Internal obstacles, such as the terrain, have boundary condition $\mathbf{v} \cdot \mathbf{n} = 0$, that is the velocity component normal to the domain is zero. For the pressure field, Von Neumann boundary conditions are applied by setting the pressure at a boundary equal to the value of a neighboring (non-boundary) voxel.

Snow particles are not considered in the wind simulation

Terrain model

The simulator has three different but connected terrain models. Upon initialization, a height-map (16-bit unsigned integer values) is used to create a terrain mesh. This mesh has 4-component vectors in its vertices (x, y, z, h) – where h is the amount of snow buildup at the vertex. This is the representation used for snow accumulation and rendering. For the wind and pressure simulation, the voxel fields of velocity and pressure are accompanied by an obstacle grid; a bit-field per cell that contains Boolean values for obstruction of the cell itself and all 27-neighbors. This map is created from mesh representation and is updated periodically. Finally, the avalanche prediction uses a fixed grid that uses the normal vectors at the corresponding mesh vertices in its simulation. Figure 2.4 shows the mesh vertices and corresponding simulation obstacle representation, while Figure 2.5 shows snow accumulation.

Avalanche prediction

Described in detail in [65], the avalanche layer in the simulator operates on a fixed grid using a finite element method. The grid filled with initial values based on terrain and no

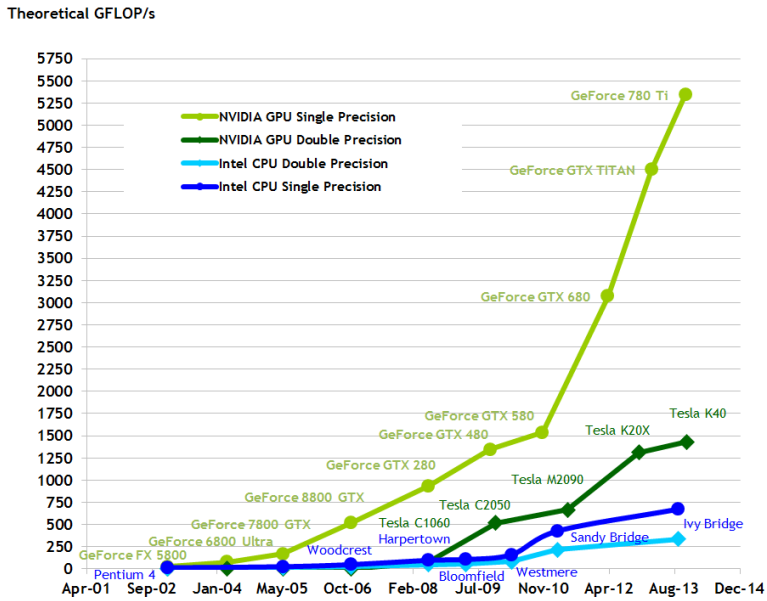
snow at the start of simulation, and each subsequent simulation step a single layer of the grid is updated, propagating from top to bottom. Each voxel contains a 24-bit vector of the forces applied to it, as well as a 5-bit temperature and a 3-bit humidity scaled to restrictive ranges. These values are set similarly to the boundary conditions in the wind field; they are constant in a time step but may vary across steps. Since only a single layer is evaluated per step, the values may vary during the down-propagation of forces. A simple noise function is used to vary the temperature and humidity given a range configurable at run-time.

Rendering

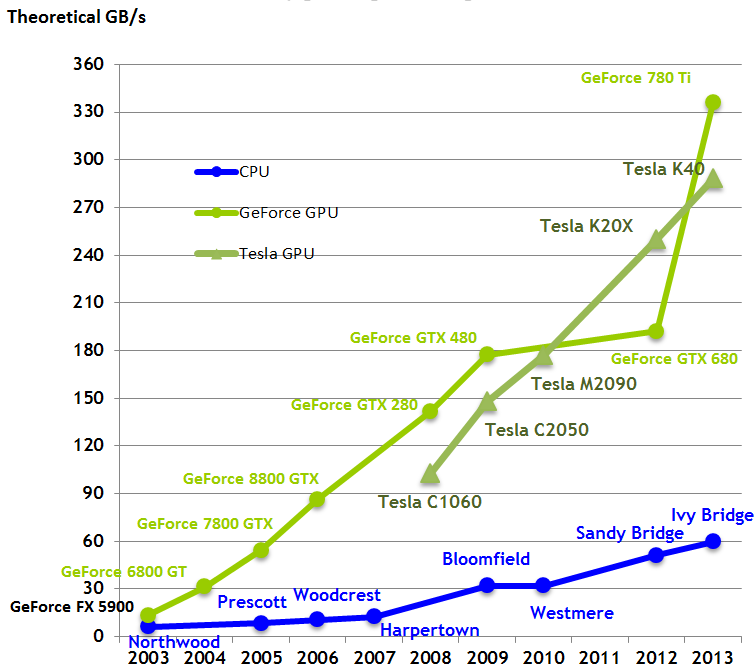
The rendering techniques used in the simulator are discussed in detail in [61], and the configuration system allows for fast switching between visualization styles and realistic techniques.

Building and profiling

Kerr [66] added support for the `CMake` build system, and the `GLFW` library is used for window management and input events. However, the simulator uses a few non-portable features beyond the `GPGPU` libraries; the Portable Operating System Interface (POSIX) PRNG `drand48` – a simple LCG – is used in the snow particle initialization, and the profiling code uses the POSIX `clock_gettime` function. The profiling code performed full-frame timings of a limited number of frames if compiled with a specific pre-processor flag. For more advanced timings, previous theses used `CUDA` or `OpenCL` specific tools or specialized, and since removed, timing code.



(a) Floating-point operations per second.



(b) Memory bandwidth.

Figure 2.3: Comparison of GPU and CPU performance over time. Green lines are GPUs, blue lines are CPUs. Figures ©NVIDIA, used with permission.

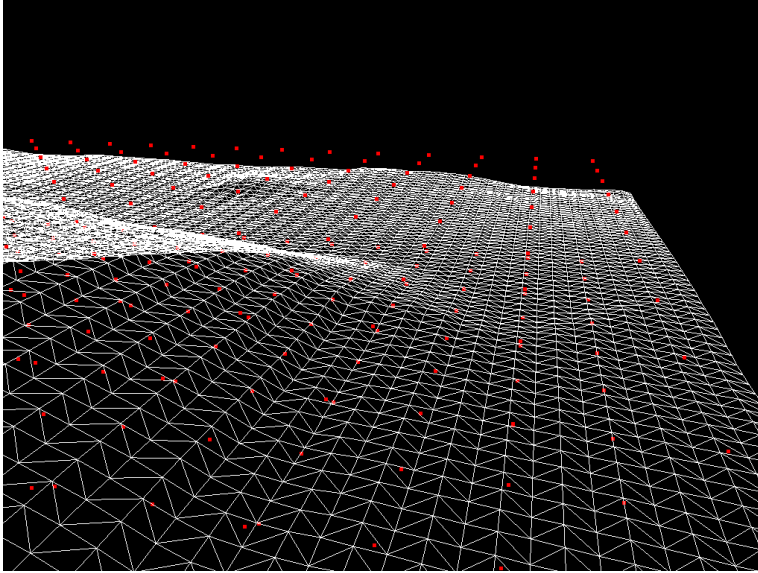


Figure 2.4: The terrain mesh of Mt. St. Helens terrain, at 768^2 vertices (white), compared to the obstacle map (voxel centers) at $128 \times 32 \times 128$ voxels (red).

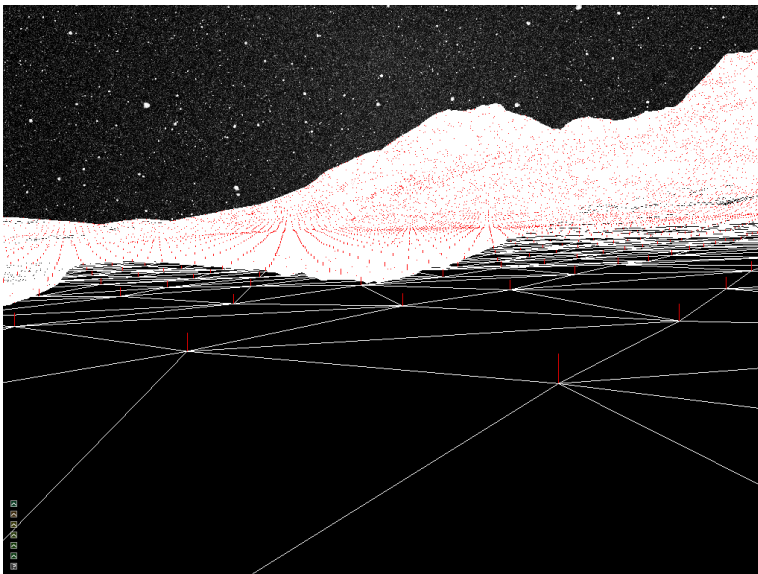


Figure 2.5: Snow accumulation on the Mt. St. Helens terrain after a period of uniform snowfall. The red lines indicate the accumulation of snow at the vertices.

Simulator improvements and extensions

This chapter examines the improvements and extensions to the HPC-Lab snow simulator made as part of this project. It begins with a short section describing motivating the changes made, followed by 3 sections devoted to the integration of real-world weather data and related work. Section 3.2 describes the methods and implementation of ground truth wind sources for boundary data. Section 3.3 covers varied precipitation probabilities across the simulation domain, and Section 3.4 details the related method and implementation of cloud rendering. In Section 3.5 the simulator's handling of terrain models is refreshed and updated. Finally, Section 3.6 examines various other improvements to the simulator.

3.1 Motivation

To motivate the work in this thesis, this section examines the historical uses of the HPC-Lab Snow Simulator, as well as the potential uses with certain attainable improvements.

The use of the wind simulation techniques described in Section 2.9.2, as well as the constraint to real-time, have limited the uses of the simulator in the past. The fluid simulation technique is based on the work of Stam [4], which is primarily intended for use in computer graphics and video games, and as such values believable visual results over physical accuracy. The real-time constraint and the memory constraints of GPUs set an aggressive upper bound on the resolution of the simulation. As a result, small scale turbulence is lost, and the drag force \mathbf{F}_{drag} is replaced with the visually pleasing but physically inaccurate circular velocity component \mathbf{V}_{circ} .

However, the rendering quality improvements of Babington [60] and Nordahl [61] as

well as the stereoscopic rendering capabilities have made the simulator a valuable demonstration and recruitment tool for the HPC-Lab. The porting to various libraries and the heterogeneous compute environments supported make it a valuable tool to compare performance on a real-world application. Work on improving it has also motivated interesting results on GPGPU computing in general, such as Ch. 5 in [59].

Its use as a simulation tool is however slightly limited by the lack of a few key features. Random weather data such as boundary wind, temperature, and humidity not only limit its power as a predictive tool, but also make it hard to validate the results obtained in the simulator with real world results. The resolution of the simulation volume, currently limited by memory requirements of the voxel grids, may be of too low granularity for large scale terrain such as the mountain in which avalanches are common, and for such large scale environments uniform snowfall may not be accurate enough. Rooftop avalanches in populated areas could be handled, however city architecture is not adequately handled by the two-dimensional world-view of a height-map. Simulation over longer time periods may also want to take snow smelting into account.

It is important to note that the limitations above are not due to oversight or negligence, but simplifications and optimizations made necessary by the immensely complex nature of other aspects of the simulator, the real-time requirement, and in some cases hardware limitations. The work in this thesis builds upon the complex numerical models and work of previous students to alleviate some of these missing features by addressing the integration of real-world weather data, hoping to pave the way for a wider area of use in the future. While this is the primary focus, significant effort is used to maintain current uses. The visual quality of the rendering of the new features, as well as improvements to the performance and demonstration capabilities of the existing features, are areas of significant focus in the following.

3.2 Wind

Section 2.1.2 introduced typical boundary conditions, and Section 2.9.2 explains how these relate to the HPC-Lab snow simulator. Internally in the volume, the wind is simulated, however the wind at the boundary is set to a constant value across the domain; either a user set value (selected in the configuration Graphical User Interface (GUI) or set in the configuration file) or a *periodic wind* setting that linearly interpolated between a hard-coded set of four values at 15 second intervals.

Wind measurements are primarily available at ground measurement stations, as covered in Section 2.2.1. The following sections cover a new boundary wind mode – interpolated measurement station values – and the changes made to the simulator to support it.

3.2.1 Importing the values

As mentioned in Section 2.2.4, there are multiple APIs that offer weather data from ground measurement stations to the public, and these are offered in varying formats. Supporting all these formats in the simulator is neither feasible nor required; instead a simple text-based file format is defined to supply only the values required. Internally in the simulator, each ground measurement stations is called a `WindSource`, and it consists of a fixed position, a time interval between samples and a list of samples; 3-vectors of wind velocity. The number of samples is not known at compile time¹, and instead of complicating the individual `WindSource` input files, a meta-file lists the sources.

To explain this system, consider the meta-file `winddata.txt`, containing N lines. Each line is the path of a `WindSource` input file (relative to the running simulator or fully qualified). When the simulator is asked to use this meta-file, it creates N `WindSource` objects. For each of these, the file at the path listed in the corresponding line of the meta-file is parsed. These files – the `WindSource` input files – have a leading line containing constant values: the 3-vector describing its fixed position in the simulation domain (in the wind simulation voxel fields’ coordinate system), the number of milliseconds between samples, and the number of samples, S . This line is followed by exactly S lines, each containing a single 3-vector describing the wind velocity at the `WindSource` at that sample’s time.

The simulator internally uses a domain-relative coordinate system. Importing values from a public data-set is likely to yield absolute location (in the form of Global Positioning System (GPS) coordinates), however this must manually be converted into in-simulator coordinates to match their location relative to the terrain. Thus, a pre-processing script separate of the terrain pre-processing script still requires manual input, and for the test data used to test performance, the values were manually copied over or generated from random values.

3.2.2 WindSource representation on the GPU

The boundary value is set at the end of the `wind_advect` kernel to avoid a separate kernel invocation for the small subset of voxels that lie on the boundary. Two new parameters are passed to this kernel; a floating point array of 8 values per `WindSource` and an unsigned integer containing the number of wind sources. The most significant bit of this unsigned integer is used to switch interpolation modes, as discussed below. Table 3.1 describes the packing of wind sources into the floating point array.

The neighbor information is a bit-mask and is expanded upon in Section 3.2.3; at time of writing only 32 bits are used and thus the number of wind sources supported is capped at 32, however the unused 32 bits in the last value may be used to expand this to 64 without additional memory cost. The wind velocity is temporally interpolated between the sample values imported on the CPU before upload. This interpolation is performed every simulation step.

¹There is a known upper bound, as explained below.

Float	0	1	2	3
0-3	P_x	P_y	P_z	Neighborhood bit-mask
4-7	V_x	V_y	V_z	Unused

Table 3.1: V denotes velocity, P denotes position.

The 4 unused bytes are included as padding for vector architectures, where having the position and velocity vectors aligned on 16-byte boundaries may help performance, and as Section 4.2.3 shows there is no significant performance cost on scalar architectures. The capped source count ensures no more than 128 bytes are wasted as a result, and the space for future expansion of neighbor count or other data is practical.

3.2.3 Spatial interpolation at boundary points

When setting the boundary value at any given point a decision has to be made about which wind sources affect that point; not all wind sources' values are relevant for a given point. Figure 3.1 shows some configurations of wind sources relative to a boundary point in two dimensions, and which samples are likely to contain relevant data for that point.

The simplest scheme is a nearest-neighbor approach; each boundary point is set to value of the closest wind source to that point. This technique is implemented primarily for debugging purposes. The next approach is to interpolate between the values of **all** wind sources based on their distance from the source. However, as Figure 3.1 highlights, not all wind sources are relevant. Instead, a smarter approach is suggested; only the wind source closest to the point, and all wind sources that are *direct neighbors* of this source, are used in the interpolation.

This requires a definition of *direct neighbors*. It is reasonable to base this on a Voronoi segmentation of the domain; any boundary point falls into a volume, and for a wind source to be considered in the interpolation no more than a single plane of separation in the Voronoi segmentation may lie between the boundary point and the source. To compute the Voronoi segmentation – or its dual – in three dimensions is non-trivial, as covered in Section 2.5. Attempts were made to use the CGAL library, however the library is sizable and complex, and while a Delaunay triangulation would provide the information desired, a full triangulation is not needed. Only the information on which sources to use for interpolation is sought, and a lot of work would go to waste.

A simpler approach

For the small number of wind sources the simulator may encounter (the space covered by a simulation is typically on the order of a few kilometers squared, making it unlikely that the upper bound of 32 measurement stations is exceeded) it may be possible that a simpler approach with worse computational complexity proves acceptable. The information that is sought is the neighborhood relationship between wind sources, and this is not required to

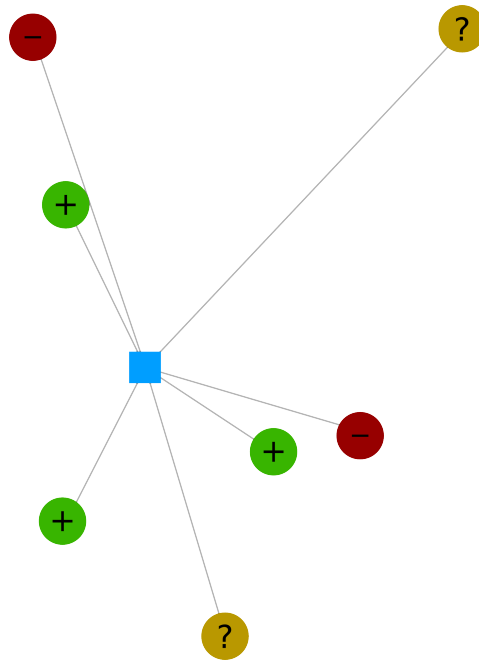


Figure 3.1: A problematic case of interpolation. A boundary cell (square) and multiple sources that may give information about its true value. Green (+) sources are likely to contain good information on the value at the sample point, yellow (?) may contain good information, while red (-) is unlikely to contribute anything not already encoded in the other sources.

be defined like a Voronoi segmentation. Instead, definition 1 is proposed, and visualized in Figure 3.2.

Definition 1. *Two wind sources a, b are considered direct neighbors if and only if at no point p on the straight line l between them there exists a third wind source $c \notin (a, b)$ that is closer to l than either a or b .*

This definition leads to a very simple algorithm, the pseudo-code for which is included in Listing 3.1. The C++ implementation is included in Appendix B.

Listing 3.1: Pseudo-code for neighbor-relations finding algorithm

```

for S in sources :
  for N in sources where N != S :
    Neighbor = True
    for P in sources where P != N && P != S :
      A = closest_point_on_line( P, line( S, N ) )
      D = distance( P, A )
      If D < distance( S, A ) or D < distance( A, N ) :
        Neighbor = False
    Neighborhood_of_s = all N for which Neighbor is True

```

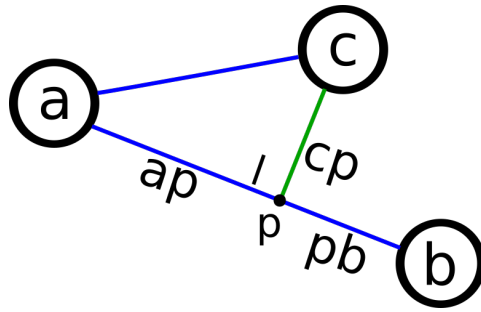


Figure 3.2: Consider the neighborhood relationship of a towards b and c ; c is a neighbor, while b is not, because the distance cp is smaller than either of ap and pb .

This implementation of the algorithm is $\mathcal{O}(n^3)$ in n wind sources, however the constant factor is very small. Additionally, the neighborhood relationship commutes so only half iterations of the second loop are required. Since the wind sources are spatially fixed, this relation only has to be computed once at load time.

3.2.4 Setting the values

Given the wind source data and the boundary points in the kernel, the final value still needs to be interpolated. The simulator now supports three different strategies; the value can be fixed across the domain as in previous versions, it can be interpolated or it can be set to the value of the nearest wind source. In the prior case, the wind source count is zero. Otherwise the closest wind source is calculated by iterating over all sources calculating the distance to them from the boundary voxel in question. These values are cached in local storage. If the nearest-neighbor approach is to be used, the most significant bit of the wind source count is 1, and the wind value is set to the value of the closest source.

In the case of interpolation, the distance from the boundary voxel to the closest source and all neighbors of that source – d_{total} – is summed by iterating over the wind sources, skipping those with a zero bit in the neighbor bit-field and reusing the cached distances to avoid re-computation. Finally, the velocities of these sources are accumulated in a weighted sum, where the weight w_{source} of a source is given by $w_{source} = \frac{d_{source}}{d_{total}}$, and d_{source} is the distance to the source in question. These weights sum to 1, and the final value is the wind velocity of all contributing sources linearly interpolated based on distance. The complete source of the new `wind_advect` kernel is included in Appendix C.1.

3.2.5 Visualization

The wind sources can be rendered as spheres for visualization and debugging. These spheres are generated at run-time by creating an equilateral tetrahedron and repeatedly subdividing each face into 4 equilateral triangles, extruding vertices to *radius* distance from the center of the sphere. This allows the generation of arbitrary accurate spheres,

however in practice, only 4 subdivisions are performed. The code to generate the spheres is included in Appendix A.

3.3 Precipitation

When simulating areas of multiple square kilometers, treating precipitation as uniform over the entire area is a significant simplification. Figure 3.3 shows a rain front, and showcases how treating precipitation over the full area covered by the image would not capture this weather phenomenon.



Figure 3.3: A weather front, showcasing the issue with the uniform precipitation simplification. Image released under CC0 (Public Domain) license by user *sandid* on website <https://pixabal.com> (last accessed: 2016-06-20).

There are three factors that determine precipitation levels in the simulator; the initial distribution of snow particles, the redistribution of particles that hit the ground or leave the domain, and the total number of particles. The latter is fixed for implementation reasons – adding and deleting particles would require resizing GPU arrays which is costly – and can be ignored if precipitation levels are considered as relative to this value.

3.3.1 Precipitation distribution

A non-uniform distribution of particles is required for varying precipitation. As the simulation is assumed to be entirely below the cloud layer – no snow particles are formed in the simulation domain, they all enter from outside it – this can be described by providing a probability of any point on the boundary to introduce a new particle. Maintaining the constraint that the snow particle count is fixed, this can be modeled with a $p(x, y, z) \in [0, 1]$

probability given a three dimensional boundary position. As described in detail in Sections 3.3.2 and 3.3.3, re-positioning can be limited to the top² plane of the simulation.

The PDF of the initial positioning and re-positioning of a snow particle then becomes a two dimensional field of zero-to-one values; an ideal format for representation as a texture. This also enables the use of hardware filtering to interpolate values at sub-*texel* granularity. This texture is referred to as the *distribution map* in the following.

3.3.2 Initial distribution

Previous versions of the simulator have used `drand48` for the initial distribution of snow particles in all three dimensions. As soon as the simulation starts, any particle below the terrain would be redistributed, leading to a large amount of particles reentering the simulation at the very top of the domain in the second time step of simulation. The average height of the terrain in the example scene seen in Figure 3.5 – the Mount St. Helens region in Washington, USA – is $\sim 38\%$ of scene height, which implies that an initially uniform distribution of particles leads to a re-positioning of $\sim 38\%$ of all particles in the first time step of simulation.

The CPU-side PRNG is changed to the 32-bit reference implementation – `pcg32` – from [28], as part of an attempt to remove all usages of the `drand48` PRNG. This includes the initial snow distribution, radii, terminal velocity, and rotation, as well as the cloud noise texture initialization, and the temperature and humidity values used for avalanche prediction.

Additionally, a change to the initial distribution domain is proposed; particle positions below the terrain are rejected, effectively reducing the domain of valid initial positions using the rejection method. A maximum number of rejections is introduced as a compile-time constant, `MAX_REPROJECTION_ATTEMPS`, after which a position is accepted even if outside the valid domain. This is acceptable due to the changes to re-positioning described in Section 3.3.3, and motivated by the possibility of low precipitation levels across the entire domain. Should the chance of precipitation be zero across the entire domain an infinite loop is avoided using this concession.

3.3.3 Re-positioning

Once the simulation is running there are two ways in which a re-positioning can be triggered; a particle hits the ground or leaves the domain. In the latter case, the particle has previously wrapped the domain, that is its position in the direction it left the domain is taken modulo the domain size. While giving a visually pleasing result, this is problematic if the precipitation is not uniform, as it may wrap particles from high precipitation sections into sections with no precipitation at all. Simply forcing a re-positioning of all particles leaving the domain introduces an additional problem as re-positioning has previously enforced a fixed position along the axis of gravity; particles always enter the domain at the

²The plane at the opposite direction of gravity; in the simulator this is the maximum *y*-value in the domain

top when fully re-positioned. If no particles enter the domain at the side from which the wind blows, this leads to sections of the ground getting very little to no snow accumulation, as seen in Figure 3.4.

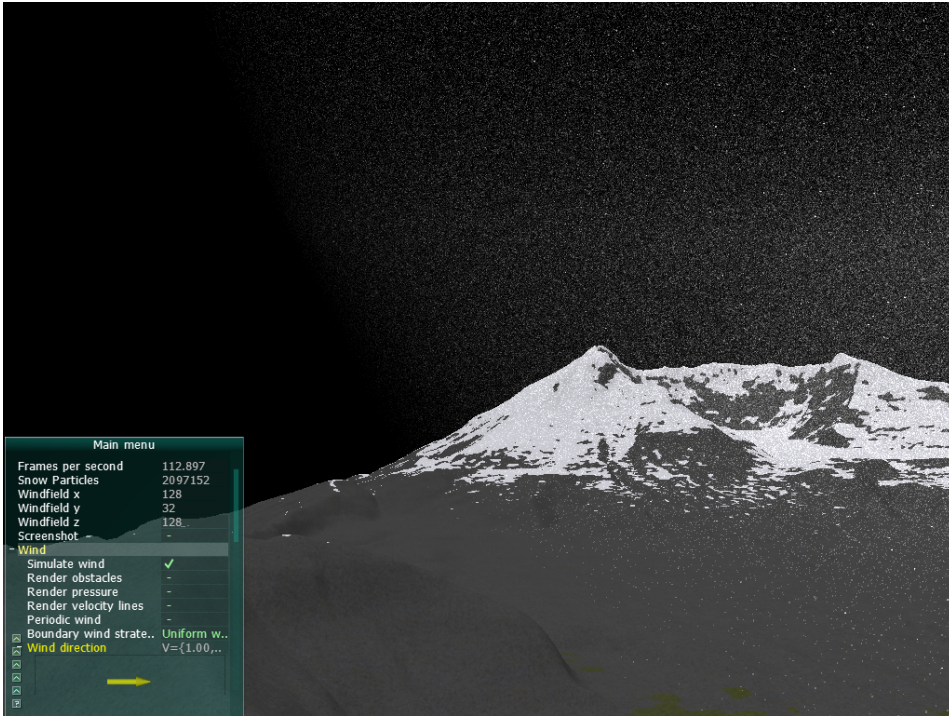


Figure 3.4: The lack of redistribution of particles at the sides leads to lack of snow on the side the wind is coming from.

The proposed solution to this problem is an alteration of the old wrapping approach. When a particle leaves the domain its position is wrapped as previously, however this new position is not unconditionally accepted. A backwards interpolation of the position using the current velocity of the particle is performed to find its xz -position in the re-positioning plane. The probability of precipitation at this point is then looked up in the *distribution map*; if valid the position is used, otherwise the particle is fully re-positioned. As this interpolation may produce positions outside the domain, the *distribution map* should extend beyond the domain's xz -plane; the current implementation assumes it to be double the size of the domain, centered over the domain center. Figure 3.5 shows how the distribution map extends beyond the simulation domain.

Full re-positionings are also applied to the case of snow particles hitting the ground. In this case, previous simulator versions used Edissen's PRNG to re-position the particle into the top-most plane of the domain. As a non-uniform PDF must now be considered, the rejection method is implemented. The `reposition` function is maintained, however its use in the particle update kernel is replaced with calls to a new `redistribute`

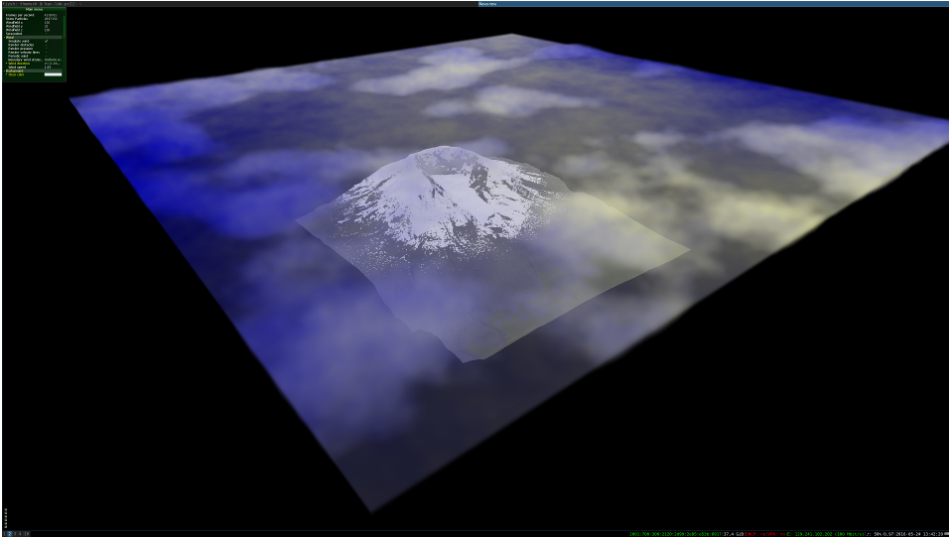


Figure 3.5: The distribution texture, here as a flat cloud layer with color visualization, extends beyond the simulation domain, which is the size of the terrain seen beneath.

function. The CUDA and OpenCL implementations of this function are included in Appendix D. The function performs an initial call to `reposition` before using the rejection method on it using the *distribution map*. It repeats this until either a valid position is found or a maximum number of attempts, `DISTRIBUTION_REJECTION_ATTEMPS`, is reached. Unlike the initial positioning, once this maximum number of attempts is reached, the invalid value is not accepted. Instead, the particle is positioned far outside the domain, effectively marking it as inactive. To do this, the x -position is set to -10000 ; the valid domain of x is non-negative, but a particle's position may enter the negative during simulation with the right wind conditions, so simply using the sign bit is insufficient. A large enough value must be used that it is reasonable to assume that the particle cannot have reached that position in a valid simulation step. As an inactive particle is outside the domain, the next iteration of the simulation will attempt to re-position it, ensuring all particles are considered at any iteration, even if not all particles are visible to the camera or in the domain.

The visual effect of multiple re-positioning attempts is that the amount of snow particles in any cell of the top field of the domain given a fixed probability p_{cell} for that cell may still vary, as the probability of any particle being in that cell is $p_{any} = \frac{p_{cell}}{p_{total}}$, where p_{total} is the sum of probabilities for all cells. Thus, if it only snows in half the domain (with a chance of 1.0 in all cells in that half) the amount of snow particles per cell of that half is double the amount per cell if it snows (with chance 1.0 in all cells) in the entire domain. If this effect is undesired – p_{any} is wanted independent of p_{total} – the `DISTRIBUTION_REJECTION_ATTEMPS` constant can be set to 1, however as p_{total} becomes small this leaves a large amount of particles inactive in any given simulation frame.

3.3.4 PRNGs

As discussed in Section 2.4.1 Eidissen’s PRNG does not handle immediate reevaluations on the same position well and a different PRNG must be used. A number of different PRNGs of various quality and suitability to GPU implementation were discussed in Section 2.4.1, which also mentions two libraries of interest. The libraries, `cuRAND` for the CUDA back-end and `clRNG` for the OpenCL back-ends respectively, have been implemented into the simulator. Selecting PRNG is now performed using a selection of pre-processor macros, enabling comparison of quality and performance of the various new generators, as well as the old one. The use of the various PRNGs is implemented for the `reposition` function only; the `redistribute` function also requires uniform random numbers to reject against the PDF, but these are always supplied by a `philox` generator. This is in part because the old generator cannot be used for this purpose at all (it does not neatly generate a single value in the correct range, nor handle multiple subsequent calls), and in part due to the issues some of these generators have, as discussed in Section 2.4.1. The uniformity of these numbers are paramount to the validity of the rejection method, and thus only the best PRNGs are suitable candidates. All the PRNGs made available by the ones the libraries (except the `cuRAND` version of the `Mersenne Twister` have been integrated into the simulator for testing and comparison, however for reasons discussed in Section 4.3.1, the `philox 4x32-10` generator is the preferred default.

The generators are seeded per thread, with a combination of the GPU clock and thread identifier in the CUDA implementation, and the CPU clock and thread identifier in the OpenCL implementation, due to a lack of access to the GPU clock from within kernels. This means the CUDA potentially has more variety in seed, as the clock value is likely to vary at least between warps, whereas the CPU clock value passed to the OpenCL kernels is invariant across all warps of a kernel invocation.

Issues

While CUDA kernels are compiled at program compilation time, the OpenCL programs are compiled at run-time using the GPU driver’s internal compiler. This led to problems with the `clRNG` library, as some compilers did not support features in the pre-processor that the library relies on. A debug print command had to be manually altered in every included header (and the source for the compiled library, so as to match) to make the library work on the test machined of Chapter 4. This complicates upgrading the library, as the changes must be reapplied on every update.

3.4 Clouds

When dealing with non-uniform precipitation additional visualization techniques are required. It is desirable for the user of the simulator to quickly gain insight into the precipitation probabilities across the domain, and while the snow particles themselves provide

a decent insight into recent trends, and the ground accumulation gives a good average over time, inspecting instant values is difficult. Additionally, Section 3.1 highlights the use of the simulator as a demonstration tool which motivates a visually pleasing technique that is easily accessible. Clouds, while an imperfect indicator of precipitation, provide a simple and intuitive visual indicator. This section describes the implementation of cloud rendering in the simulator, both as a visualization tool and as a pleasing visual effect.

3.4.1 Technique choice

Section 2.8 introduces a number of techniques for cloud rendering that can be classified into two categories; billboard-based and ray-marched techniques. The prior have a lower computational cost, but have lower visual quality and may have artifacts if the camera is inside the cloud layer. The simulator has a free-flying camera, and together with the focus on GPU computing techniques central to the simulator history this motivated the implementation of ray-marched clouds for visualization purposes.

When considering stereoscopic rendering, the billboarding technique may also present as problematic. Billboards are flat primitives oriented to the camera. This causes two issues with stereoscopic rendering: there are two cameras and the billboard must either be oriented towards some third virtual camera between them so the geometry being rendered is the same, or oriented towards each camera for their pass leading to different geometry but simpler instancing. Either way, the flat surface has no depth information in the cloud, leading to a loss of depth perception that the ray-marching does not have.

Finally, it is noted that the clouds are not taken into account in the shadow map rendering, due to their high cost as discussed in Section 4.4

3.4.2 Technical details

Four different cloud visualization modes have been implemented:

Vanilla This is a pure visually pleasing rendering of clouds. The density is purely based on FBM and the color is calculated using Equation 3.1.

Precipitation as color The density is FBM based, but the color is calculated using Equation 3.3, where blue is used to indicate the precipitation levels, and p is the precipitation probability.

Precipitation as height The density is a mixed function of FBM for visually pleasing detail and the precipitation probability p , where the density proportional to both p and the height difference to the center of the cube. The color is calculated using Equation 3.1.

Precipitation as threshold The density of the cloud is a pure function of the height difference to the center of the cube and the precipitation probability p . A threshold $C_{threshold}$ is set by the user, and if p is below this threshold the density is zero. The color is calculated using Equation 3.1.

Rendering overview

The cloud layer is rendered as a cube of configurable height, spanning the full xz -plane of the *distribution map* as described in Sections 3.3.1 and 3.3.3, and centered at the center of the top plane of the domain. The ray-marching is based on [54]; in the fragment shader of the cube a ray is constructed based on the world position of the rasterized point and the camera position, which is then marched through the volume of the cube a maximum number of steps or until it exits the cube. Special care must be taken in case the camera is inside the cube to start marching at its position and not at the rasterized point of the cube surface (which will be a back-face).

At each step of the march, the density of the cloud is evaluated, and this density combined with the density difference between subsequent steps is used to update the accumulated color of the ray. Fully evaluating light transport in participating media is costly, and the color accumulation is not physically accurate; it is selected to give a visually pleasing result at a very low computational cost (or simply to visualize precipitation probabilities in certain visualization modes). The low cost is important, as this formula is evaluated at every step of the ray traversal. The formula used is due Quilez [54], and is a linear combination of a light gray with under-saturated blues and a darker gray with over-saturated blues, with increasing saturation of blues as the ray marches deeper and a strong orange hue added if the difference in density is large. This last term attempts to simulate the shimmer seen at the sun's incident side of clouds. The density at a step is used to determine the colour contribution at that point. The full formula for the RGBA color value at step $k + 1$ given a density d and a difference in density δd is listed in Equation 3.1.

$$x_{rgb} = (C_1 * (1 - d) + C_2 * d) \times C_3 \times 1.4 + C_4 \times \delta d) \times d \times 0.4 \quad (3.1a)$$

$$c_{rgba}^{(k+1)} = \langle x_{rgb}, d \times 0.4 \rangle \times (1 - c_{rgba}^{(k)}) \quad (3.1b)$$

$$C_1 = \langle 1.0, 0.95, 0.8 \rangle, C_2 = \langle 0.25, 0.3, 0.35 \rangle,$$

$$C_3 = \langle 0.65, 0.7, 0.75 \rangle, C_4 = \langle 1.0, 0.6, 0.3 \rangle$$

Step length

The step length s varies with the accumulated colour and is given in Equation 3.2. It is a function of the alpha value of the color and a minimum step distance s_{min} . This step length increases as the alpha approaches 1, that is the step size increases as the contribution of the steps to the final color decreases, and is an important optimization as it avoids computation in low-interest regions towards the back-end of the ray. At the front-end of the ray it does not help however, and a lot of small steps may be preformed until the first non-zero density part of the volume is encountered. Another optimization attempts to solve this by performing a *pre-march*; the ray is marched in larger steps without evaluating the color until the first non-zero density value is encountered, at which point the last step is rolled back and that position is used as a starting point for the regular march.

$$s = \max(s_{min}, \frac{1}{1 - c_a}) \quad (3.2)$$

$$x_{rg} = (1 - p) \times d \times 0.4 \quad (3.3a)$$

$$x_b = (0.8 \times (1.0 - d) + 0.35 \times d) \times (1.05 + 0.3 \times \delta d) \times d \times 0.4 \quad (3.3b)$$

$$c_{rgba}^{(k+1)} = \langle x_{rgb}, d \times 0.4 \rangle \times (1 - c_{rgba}^{(k)}) \quad (3.3c)$$

Density function

The density functions for the various visualization modes are given in Equations 3.5, 3.6, and 3.7, where $f(x, y, z)$ is defined in Equation 3.4. Here fbm is a FBM function of various number of octaves and $p(x, z)$ is the probability of precipitation at a given position of the *distribution map*. All values are forced into a $[0, 1]$ domain. C_{scale} denotes the scale of the noise pattern, and is eight times the cube's size along the largest axis and proportional along the others. C_{offset} is an offset into the noise function that is moved by the wind and is coherent across frames. This creates the illusion of the cloud layer moving with the wind. This offset is used in every density evaluation and must also be spatially coherent, so the drift and additional cost of maintaining a texture of offsets is prohibitively large. Instead, a single value is used, and it is moved by the wind value at the xz -center in the top-most layer of the domain, interpolated as described in Section 3.2.3.

$$f(x, y, z) = 0.8 \times fbm(\langle x, y, z \rangle \times C_{scale} + C_{offset}) \quad (3.4)$$

$$d_{vanilla}(x, y, z) = 0.7 - \sqrt{|y - 0.5| + 0.05} / \sqrt{0.5} + f(x, y, z) \quad (3.5)$$

$$d_{height} = 0.7 - \sqrt{|y - 0.5| + 0.05} / \sqrt{p(x, z)} + f(x, y, z) \quad (3.6)$$

$$d_{threshold} = \begin{cases} 0.7 - \sqrt{|y - 0.5| + 0.05} / \sqrt{p(x, z)} & \text{if } p(x, z) > C_{threshold} \\ 0.0 & \text{if } p(x, z) \leq C_{threshold} \end{cases} \quad (3.7)$$

The FBM function is implemented with 2, 3, 4, and 5 octaves. The noise function used is due Quilez [67], and is included in Listing 3.2. It is a look-up table based value noise that uses Hermite interpolation in the noise function. The lattice is a two-channel texture containing one channel of white noise (generated using the PCG32 generator), while the second channel is a replication of the first channel shifted by a constant amount (modulo texture size). This saves a texture lookup at a different UV by amortizing both values into one fetch.

Listing 3.2: Values noise function used for FBM function

```
// Created by inigo quilez – iq/2013
// License Creative Commons Attribution –NC–SA 3.0 UP.
float noise( in vec3 x )
{
    vec3 p = floor(x);
    vec3 f = fract(x);
    f = f*f*(3.0-2.0*f);

    vec2 uv = (p.xy+vec2(37.0,17.0)*p.z) + f.xy;
    vec2 rg = texture2D(iChannel0,(uv+0.5)/256.0,-100.0).yx;
    return mix(rg.x,rg.y,f.z);
}
```

When performing the ray marching in the shader, the number of octaves used in the FBM function may gradually be reduced from five to two. This optimization aims to reduce the cost of the density lookup as the contribution of the ray decreases, but may cause artifacts as discussed in Section 4.4.2.

The complete cloud shader is included in Appendix E.

3.4.3 Real-world data integration

Using real-world precipitation data simplifies to supplying a precipitation texture in this implementation. The simulator takes a series of single-channel images as input (or a constant single texture) that is interpolated in time. The texel values are linearly interpolated, leading to artifacts if the temporal resolution is low. The image series has a base name, a shared extension, an image count, and a length in seconds per image. The files are named `[name][index].[extension]`, where `index` starts at 0 and is monotonically increasing. When a texture series' end is reached, the simulation may end or wrap the index, depending on user selection in the GUI or configuration file.

The varying formats and resolutions of available radar data and weather images motivated the separation of conversion to this simple single-value texture format into pre-processing steps. For testing, images from MET Norway were used. These radar images have a 6-value color-scale with channels that overlap the background map, and instead of writing a tool to separate out the data, the level and color curves feature of a typical image editing suite – in this case *GNU Image Manipulation Program*³ – were used to create the image series.

An attempt was made to write a Python tool to convert NEXRAD binary data format discussed in Sections 2.2.4 and 2.2.2, however multiple attempts at compiling and reading the available data with *the python arm radar toolkit* failed, and a lack of documentation of the format made writing a custom parser difficult.

³<http://www.gimp.org> (last accessed: 2016-06-20).

3.5 Terrain

The format described in 2.3.1 – the de-facto standard for terrain models available from public sources – is a text-based format. It is both spatially inefficient and slow to parse. This motivates storage of simulation terrain in other formats, such as raw binary or image-based height-maps. As the terrain is likely to be static between many simulation runs, it is natural to perform a conversion offline before running the simulation.

A conversion script in Python due Lien [12] was found in the snow simulator source code repository. This script did not work as intended; it struggled to parse the high-resolution data-sets available via the United Kingdom Environment Agency (UKEA)⁴, and on the data-sets it did read it produced garbage results. As the script was only a few hundred lines of source code, and a full understanding of the format is required both to debug and to implement, a re-implementation was conducted.

The re-implementation ignores the `C-records`, as the error has no practical implication on the conversion, and does not readjust the quadrangle defined in the `A-records`, instead assuming that the data supplied in the `B-records` fit onto a grid. In the rare case this is not true, adjusting the data would leave similar gaps in the resulting height-map, and adjusting for orientation is not needed as the simulator currently has no concept of absolute orientation⁵.

The re-implementation is included in full in Appendix G. It reads the raw data from the USGS DEM file, scales the height to the full range of the 16-bit format, adjusts for a user supplied minimum height if desired and writes the output to a square raw height-map. If the data is not square or some data-points are missing, the missing data is set to minimum height.

Available test data

The script was tested on data from the UKEA as well as data from NMA. The latter data in some cases had formatting problems; the rows of `B-records` were not separated by newlines but by varying amounts of spaces. Due to the size of these text files – the full Dovre area 5M data-set is $\sim 150MB$ – and the fact that all data is on a single line, even minimal text editors such as `vim` and `nano` took multiple minutes to load and edit them. A small, ad-hoc script was written that managed to fix those files in which the spacing was constant –although is had to be adapted for each spacing, that is for each data-set–. Those data-sets not fixed by this script were discarded due to the time cost of fixing these issues.

⁴LIDAR Composite DSM, 1m datasets, available from <https://data.gov.uk/dataset/lidar-composite-dsm-1m1> (last accessed: 2016-06-20)

⁵This may be useful if a dynamic day/night cycle is implemented in the future for sun simulation

3.6 Other Snow Simulator improvements

During the work described above, the examination of the snow simulator history and uses in Sections 2.9.1 and 3.1, and its use during demonstrations and at events, a number of small issues and missing features were encountered and resolved. This section covers these changes.

Subsection 3.6.1 covers support for stereoscopic rendering for demonstration purposes. Subsection 3.6.2 investigates the use of hardware filtering in the OpenCL code path for performance gain. To simplify performance analysis and automate longer testing runs, a profiling automation system is integrated into the simulator, and is described in Subsection 3.6.3. Subsection 3.6.4 described the use of bindless textures in the CUDA code path as an attempted optimization, while Subsection 3.6.5 investigates a stability problem encountered in the wind simulation as the simulation volume's resolution increased.

3.6.1 3D rendering

Babington [60] implemented stereoscopic rendering, however this version of the HPC-Lab snow simulator was not used as a basis for later work, as covered in Section 2.9.1. This section covers the re-implementation of stereoscopic rendering in the current version of the simulator.

Hardware

The implementation targets NVIDIA `3D vision` active shutter glasses as described in Section 2.6.3. This limits support to NVIDIA hardware, and as the simulator uses OpenGL it is further limited to the Quadro range of hardware⁶. Any 120hz-capable monitor or projector suffices.

Software

Rendering in stereoscopic mode is a startup setting and can be enabled either through a configuration file or in the startup GUI. The setting cannot be changed at run-time as it requires a recreation of the OpenGL context – which in turn requires a recreation of the application window – to select a stereoscopic framebuffer format. Recreating the context is significantly complicated by sharing OpenGL resources with the OpenCL or CUDA simulation and the `3D vision` hardware supports switching between stereo and mono externally. In light of this, it was deemed unnecessary to support run-time context recreation.

The OpenGL context is recreated once; when transitioning from the startup GUI to the simulation. If stereoscopic rendering is enabled, the framebuffer is created with quad-

⁶The OpenGL driver only supports stereoscopic framebuffers on the professional range of hardware, while the DirectX drivers on windows also support the consumer range, `GeForce`

buffering instead of dual-buffering. Here, each eye has a dual-buffered framebuffer to render to. When selecting framebuffer to draw to with `glDrawBuffer`, the usual `GL_BACK` is replaced with `GL_BACK_LEFT` and `GL_BACK_RIGHT`. A new class – `StereoCamera` – was implemented that extends the old `Camera` class. It features an additional parameter, the Inter-Pupillary Distance (IPD), as well as a method `advanceEye` that selects the correct back-buffer to draw to. The rendering function now loops over the number of eyes – 2 in this case – and renders everything that is to be stereo twice. The shadow map rendering is hoisted outside this loop, as it is independent of the viewer’s camera. As the GUI is rendered at fixed depth of zero to always be visible it has no desirable depth information, however it must still be rendered twice to be visible to both eyes. Appendix H contains the relevant source code of the rendering function and stereo camera class.

Challenges

The creation of an OpenGL context is handled by the GLFW library, however creation of a quad-buffered 3D framebuffer on Linux did not work in GLFW 2.8. The library was updated to GLFW 3.1.2, however the major version change brought changes that propagated into other dependencies; `AntTweakBar` – the GUI library in use – is not officially maintained for GLFW3. An unofficial, updated version was integrated and is statically compiled into the simulator.

Virtual reality support was briefly investigated, however at the time the simulator still relied on POSIX functionality and the accessible hardware – a Oculus DK1 HMD – only had Windows drivers.

3.6.2 OpenCL sampling improvements

Section 2.7.2 mentions hardware texture filtering. The CUDA back-end of the simulator takes advantage of this when looking up wind velocity both in the snow particle update kernels and the wind advection kernel, however the OpenCL version did not.

Previously, the wind velocity had been maintained in an OpenGL buffer that was shared with OpenCL through the OpenCL-OpenGL inter-operability API. This buffer was sampled with the manual tri-linear filtering function `sample_trilinear` in Appendix A.1.1 of [59], which added this manual sampling code to the simulator. It also includes a texture sampling version, however this is neither mentioned in the text itself, nor is the code in there CPU-side code to use it in the snow simulator source code repository.

Instead of creating an OpenGL buffer to hold wind velocities, and sharing it with `clCreateFromGLBuffer`, a texture is now created in the OpenGL context that is shared with `clCreateFromGLTexture`. In the OpenCL kernels, these are sampled with a call to `read_imagef`, which takes both the texture memory and a sampler object as a parameter; the latter is a set of parameters describing addressing mode, whether coordinates are normalized, and filter mode. To use hardware sampling the corresponding filter mode must be set; for tri-linear filtering this is `CLK_FILTER_LINEAR`.

The usage of software or hardware filtering is selected with a compile-time define,

enabling both automated performance analysis as described in Section 3.6.3 and support for OpenCL devices that lack OpenGL-inter-operability support or support for hardware filtering.

3.6.3 Performance analysis

When analyzing the performance impact of a change to the simulator, it must be run with various configurations and potentially recompiled with different pre-processor commands between each run. Additionally, the kernels or sections of code that are of interest must be timed and the timings must be analyzed. Conditions must otherwise be stable, and when storing the results for later analysis or reproduction it is important to note all related software and hardware information. Doing all this manually, as appears to have previously been done on work related to the HPC-Lab snow simulator, is tedious.

Appendix F reproduces parts of an automated system built into the snow simulator as part of this thesis. The system consists of the `Python` script in Listing F.1, a header-only library previously written by the author included in Listing F.3, and the `TimingSystem` class of the snow simulator, shown in Listings F.1 and F.2. The script defines different test profiles, each consisting of a set of configuration permutations and pre-processor commands for the C++-compiler, `nvcc`, or the driver's OpenCL compiler. The timing class consists of a set of "timing events"; named sections of the program that are timed only if the configuration file specifies it. When the script re-compiles the program in `AUTOMATED_TEXT_VERSION`-mode, a number of frames to run each test is specified. The simulator is started without the startup GUI, runs for the specified number of frames, then exits. The benchmarking library calculates mean, median and standard deviation for all timings collected and prints these, which are then annotated by the script with the configuration and pre-processor permutations of the run. Finally, the script also notes hardware configuration, operating system, and driver versions, before returning the configuration and compiled version to its previous state.

The implementation of new test profiles requires adding them to the script, and the introduction of new timing sections in the code requires adding them to the timing system and calling into that system in the code in question. While this is intrusive, it is simple and still requires comparable amounts or less work than the old system of manually writing the timing code for each such section. The primary benefit of this system is that a test profile can be launched and left to run unsupervised, and the data can be collected as a text file at any time after completion. Testing a large number of configuration permutations is also made simple by specifying a default configuration and only enumerating the deltas from this per permutation to test. As these permutations are given as arrays in python, they could be generated automatically if desired.

3.6.4 Bindless textures

This section describes the implementation of bindless texture, as described in Section 2.7.3. As mentioned, support for bindless textures was introduced in CUDA 7.5, and

the use of this feature requires compute capability 3.0 or higher⁷.

The simulator uses texture memory for the wind velocity field, in addition to a number of textures are used in the OpenGL rendering pipeline. Only the wind velocity field is also handled by CUDA and of primary interest in this section.

The wind velocity texture is created as a `cudaTextureObject_t` instead of a `texture`. This object is a combination of a resource description – `cudaResourceDesc` – which contains type and reference to the data storage buffer (in the wind velocity texture’s case this is a `cudaArray`), and a texture description – `cudaTextureDesc` – which contains filtering, address, and read modes for the texture. This descriptor is then passed to each kernel that uses it.

OpenGL also supports bindless textures on certain hardware through the *ARB_bindless_texture* extension, but the textures in the rendering path were not moved to this feature for reasons discussed in 4.6.3.

3.6.5 Wind simulation stability problem

Figure 3.6 contains a screenshot of stability problems related to the numerical method discussed in Section 2.1.2. The simulator uses a partially staggered grid and performs 5 iterations of SOR in its Poisson solver, yet there are still visible patterns in both falling and accumulated snow. This section discusses two simple solutions to this problem. All figures in this section are generated on a $128 \times 32 \times 128$ volume with uniform precipitation probability.

Improve the Poisson solver

This problem was discovered before a demonstration of the simulator at a university recruitment event. Strapped for time to get stereoscopic rendering working, the SOR solver was moved from 5 to 8 iterations and the coefficients were tweaked until the problem was no longer noticeable. The problem appeared due to increased wind simulation volume resolution relative to solver accuracy, and the additional SOR iterations scale solver result accuracy to make up the lost ground.

Stochastic sampling

The wind simulation stores wind velocity at voxel centers, where a sample is a point sample of the wind field. If the goal is a believable visual result – which is the basis for [4] which Eidissen [27] and Vik [57] [58] based this simulator on – then some numerical accuracy of any given step in the simulation may be sacrificed as long as the results is visually correct over time. By sampling at some randomized point within the volume instead of the center, the amortized sampling position will still be the center of the voxel,

⁷The Kepler architecture is the first to support it



Figure 3.6: A visible pattern in accumulated snow due to stability problems in the SOR solver.

while the jitter in sample position within a single frame completely removes the problem. An implementation of this approach using the `philox` PRNG is presented in Listing 3.3.

Listing 3.3: Stochastic lattice sampling in wind advection

```
// PRNG setup
curandStatePhilox4_32_10_t state;
int id = z * dim.z + y * dim.y + x;
curand_init((unsigned long long)clock() + id, 0, 0, &state);
// Select random offset
float dx = curand_uniform(&state) * 0.5f - 0.5f;
float dy = curand_uniform(&state) * 0.5f - 0.5f;
float dz = curand_uniform(&state) * 0.5f - 0.5f;
// Self-advect backwards in time (reuse v for position)
v->x = (float)x - dt * v->x - dx;
v->y = (float)y - dt * v->y - dy;
v->z = (float)z - dt * v->z - dz;
// Sample the wind texture at that position
v = wind_vel_sample(wind_vel_tex, *v);
```


Chapter 4

Results and Discussion

In this chapter the quality, performance cost, and problems of the implementation choices in Chapter 3 are presented and discussed. Section 4.1 presents the methodology of analysis, and an evaluation of the quality of the tools presented in Section 3.6.3. Sections 4.2, 4.3, 4.4 and 4.5 cover their respective weather-related sections in Chapter 3, while Section 4.6 collects the various minor changes discussed in Section 3.6.

4.1 Performance analysis

Table 4.1 lists the hardware and software configurations used for testing.

ID	CPU OS	GPU GPU driver	CUDA ver.
M0	Intel Xeon E3-1200 Linux 3.16.0-31-generic	NVIDIA GeForce GTX 980 352.39	7.5.17
M1	Intel Xeon E3-1200 Linux 3.16.0-31-generic	NVIDIA Quadro 5000 352.39	7.5.17
M2	Intel Core i7 3930K Linux 4.5.4-1-ARCH	NVIDIA GeForce GTX 670 364.16	7.5.17

Table 4.1: Characteristics of the performance test machines.

Kernel and frame timings were collected using the performance analysis tools described in Section 3.6.3. Occupancy and register usage data was collected using the NVIDIA NSight and NVIDIA Visual Profiler tools. Attempts were made to use NVIDIA's Linux Graphics Debugger tool to analyze rendering performance at the sub-frame level, however this tool reliably crashes during the context recreation dis-

cussed in Section 3.6.1. CPU timing of rendering calls only measure CPU overhead, as they do not block until the operation completes on the GPU, so rendering performance is analyzed based on full frame timings; time between first rendering operation (the rendering of the shadow map) and framebuffer presentation (framebuffer swap).

4.1.1 Tool analysis

The tools presented in Section 3.6.3 output CPU, GPU, and software information, time of the test, the configurations and permutations used and timings with average, median, mean, and a simple histogram to console or file. Having this information automatically gathered made the post-analysis of performance tests run months prior possible. The script also enabled the running of a larger test battery – such as the performance comparisons GPU of PRNGs presented in Section 4.3.1 – on a machine without constant manual operation; the test battery could be started and left unsupervised¹ until completion.

The timing system does not enforce sync points for sub-frame timings in the rendering system, which means it cannot measure the cost of individual draw calls or passes. Enabling this, by forcing a pipeline flush inside the CPU-side timing blocks, is possible, but will poison full-frame timings. If added, it should be a configurable option, and noted in the output of the profiling script. The tool does not perform the full analysis of kernel register usage, occupancy and memory access efficiency that tools like `NSight` provide. While some of this information may be available through CUDA at no extra cost, the observed performance of the simulator when `NSight` performs such analysis is an order of magnitude worse than under normal operation. Because this data is primarily used to understand performance observations, it makes sense to use it as a supplementary tool; first kernel timings are observed with the automated script, second the result is analyzed and more detailed kernel statistics are used to explain the observed timings.

4.2 Wind

Figure 4.1 shows the three different boundary wind techniques. The two techniques that use wind sources use two sources, both visualized as pink spheres. The lines indicate wind velocity values at voxels. The images are taken below the terrain so the non-boundary voxels are obstructed and the velocity in these cells is zero. Since only the boundary cells are of interest to this discussion, this eliminates visual clutter.

4.2.1 Wind advection kernel implications

Table 4.2 and Figure 4.2 contain run-times, register usage per thread, and GPU occupancy – the proportion of active warps to maximum active warps possible – for the various techniques on the two test machines **M0** and **M2**. The kernels are listed in C; the top half are separate kernels for the three techniques, while the bottom half use a shared kernel

¹In practice supervision is advised so interfering external factors can be identified

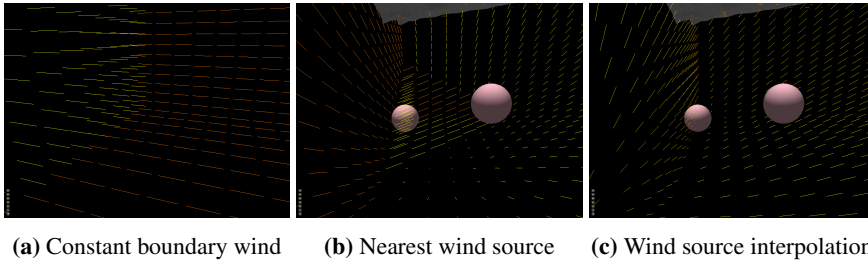


Figure 4.1: Boundary wind techniques. The lines indicate directionality of wind velocity, their length the speed.

with branching. These values were collected on a wind field of size $128 \times 32 \times 128$ using NVIDIA visual profiler for the CUDA code-path. Timings are absolute value ranges for more than 100 invocations². Both the minimum and maximum number of wind sources are considered.

Kernel	R	O_{max}	O_2	O_{32}	$t_2 \mu s$	$t_{32} \mu s$
Machine M0						
Constant	27	93.8	85.9 – 87.6		197 – 210	
Nearest-Neighbor	29	93.8	78.6 – 81.9	76.0 – 77.7	241 – 252	714 – 784
Interpolated	31	93.8	77.6 – 83.9	70.1 – 77.5	253 – 267	920 – 1117
Machine M2						
Constant	31	93.8	85.8 – 86.6		197 – 210	
Nearest-Neighbor	31	93.8	79.3 – 82.4	76.5 – 77.9	241 – 254	719 – 783
Interpolated	31	93.8	78.1 – 84.1	73.1 – 75.9	253 – 267	928 – 1098
Machine M2						
Constant	25	93.8	89.4 – 89.9		257 – 262	
Nearest-Neighbor	26	93.8	80.1 – 82.9	50.6 – 55.6	347 – 452	1752 – 1995
Interpolated	33	70.3	59.7 – 60.8	50.6 – 52.2	447 – 469	2289 – 2398
Machine M2						
Constant	30	93.8	89.6 – 89.9		243 – 279	
Nearest-Neighbor	30	93.8	80.1 – 82.0	50.8 – 54.4	345 – 439	1797 – 1992
Interpolated	30	93.8	76.9 – 77.2	56.4 – 56.8	384 – 434	2113 – 2132

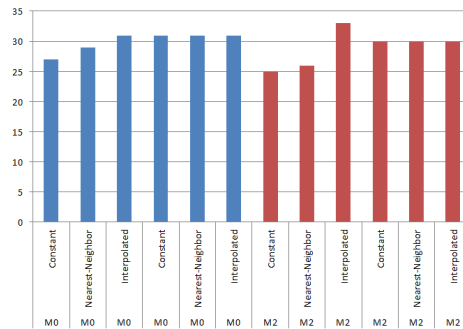
Table 4.2: Statistics on the various wind advection kernels. The top half of each machine are separate kernels for the various techniques, the rest are a shared kernel with run-time branching on the GPU. R denotes registers used per thread, O_{max} denotes theoretical maximum occupancy, and O_x denotes achieved occupancy at x wind sources. t_x denotes time used per invocation at x wind sources. Occupancy is given in percent. The constant boundary wind kernels are invariant of wind source count.

When the different code paths are shared in a kernel with run-time branching, the simpler techniques have higher register usage, as the compiler must be conservative and assume the worst case. It is interesting that the interpolated version with no branching

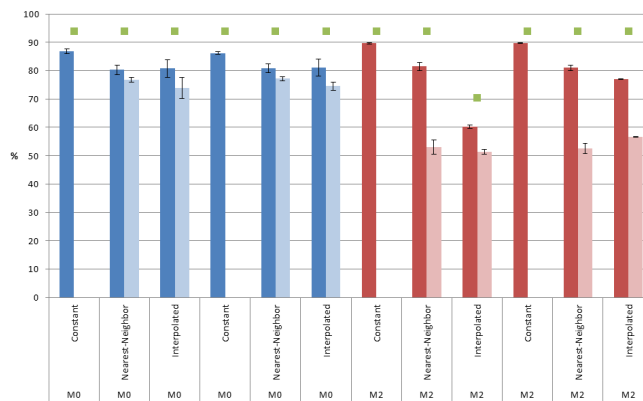
² actual invocation count varies, as the tests were manually run with NVIDIA visual profiler.

uses more registers than the branching version on **M2**, and given the run-times this is likely the compiler applying an optimization that ends up hurting performance due to the significantly lowered occupancy it results in. From the run-times it is obvious that run-time branching has no additional cost over separate kernels; performance is similar except in the discussed case where occupancy drops for the interpolated technique. Due to the similar performance, lack of CPU side branching into multiple kernel invocations, less code duplication due to the other parts of the `wind_advect` kernel being shared across techniques, and smaller overall code footprint the shared kernel is preferable.

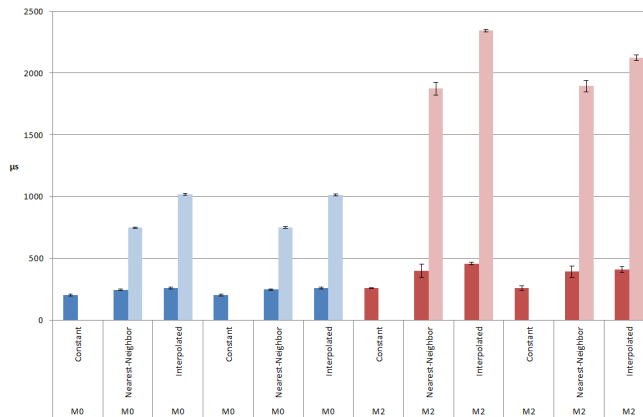
The cost of the wind advection kernel increases up to an order of magnitude on **M2** as the number of wind sources reaches the upper limits of the implementation. This is due to both more computation and more memory bandwidth needed to interpolate wind source values. An alternative approach would be to pre-compute the interpolated values at every boundary point at the fixed wind source sample timings, and only interpolate temporally in the kernel. This would eliminate the need to calculate distances to each source every frame – instead requiring this computation only when wind source sample times are reached – at the cost of additional bandwidth. It may also benefit performance to cache wind source data in shared memory, however this was not investigated further due to time constraints.



(a) Register usage per thread.



(b) Warp occupancy.



(c) Kernel run times.

Figure 4.2: Statistics for the various wind advection kernel implementations for test machines **M0** and **M2**. For source-based kernels, the lighter bars are for runs with 2 sources, the darker for runs with 32 sources. The green squares in (b) are theoretical maximum occupancy.

4.2.2 CPU implications

The neighborhood calculation was run on two `WindSource` collections of 2 and 32 sources, each source containing 6 samples. Table 4.3 contains the timings for computing the neighborhood information for all sources with the algorithm described in Listing 3.1 on test machine **M0**. It is noteworthy that even with the upper limit of 32 sources, the time required to calculate the neighbor relations is less than the time it takes to load the sources. This indicates that the simple approach is sufficiently fast at realistic wind source numbers even with a naive implementation.

Machine	Source count	Load	Neighborhood compute
M0	2	68 μ s	< 1 μ s
M0	32	390 μ s	299 μ s
M2	2	468 μ s	1 μ s
M2	32	4918 μ s	395 μ s

Table 4.3: Time used to load wind source data from file, and computing the neighborhood relationships of all the sources.

4.2.3 Memory packing

As discussed in Section 3.2.2, the memory packing of the wind sources includes 4 bytes of unused padding. This was originally intended for future expansion of the neighborhood information, but may also be useful if the avalanche prediction system is updated to use temperature and humidity interpolated from wind sources instead of fixed values.

A test with tight packing (7-floats per wind source) versus the suggested packing was performed: Table 4.4 contains run-times over 301 invocations of the `wind_advect` kernel with 32 wind sources and $128 \times 32 \times 128$ wind simulation volume, on test machine **M0**.

Packing	Avg	Min	Max
8-float	978 μ s	902 μ s	1102 μ s
7-float	972 μ s	925 μ s	1099 μ s

Table 4.4: Run-times for the `wind_advect` kernel using different memory packing strategies for the wind sources.

Even on the scalar architecture tested, the performance difference is negligible (< 1%), and the memory wasted is limited to 128 bytes under current constraints to wind source count. As OpenCL may also run on vector architectures (such as CPUs with Streaming SIMD Extensions (SSE) or Advanced Vector Extensions (AVX) instruction set support) where 16-byte alignment for vectors is important, the 8-float packing is preferable.

4.3 Precipitation

Figure 4.3 shows the snow accumulation on the terrain given a checkerboard distribution pattern and no wind. It can be observed that the pattern on the ground matches the input distribution, validating the correct behavior of precipitation probability control.

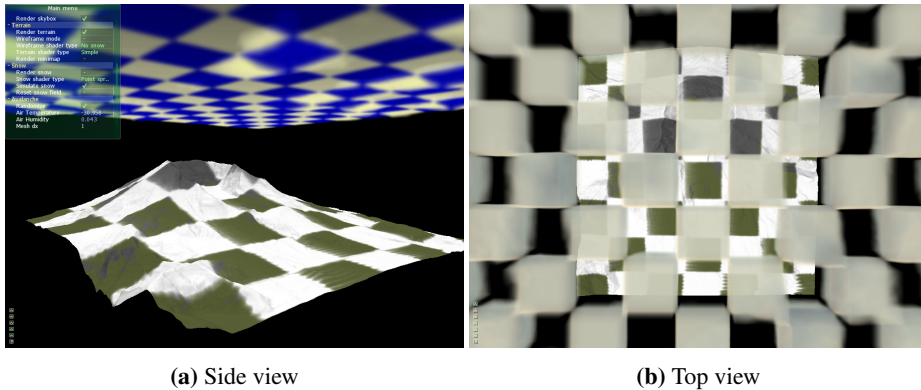


Figure 4.3: A checkerboard precipitation pattern visualized with cloud color (a) and height (b) and the snow accumulation observed after running the simulator for some time.

Figure 4.4 shows similar results with uniform wind along the x -axis; this is intended to show that the re-positioning along the simulation volumes vertical sides, described in 3.3.3, correctly accounts for distribution. The checkerboard pattern selected is larger, as the variance in snow particle mass leads to spread in particles as the wind impacts them. Making the pattern larger makes it more obvious in the face of the softer boundaries introduced by this variance.

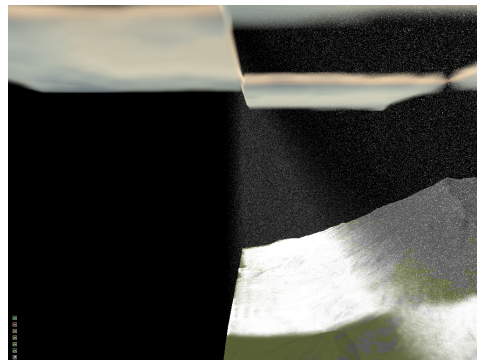


Figure 4.4: The snow accumulation along the side of the domain at which the wind enters for a checkerboard precipitation pattern.

4.3.1 RNGs

The quality of the PRNGs implemented are evaluated with statistical models in the work summarized in 2.4.1. This section investigates their suitability for use in the snow simulator, first by inspection of snow particle distribution printed to images, and finally by comparing their performance implications.

Quality

Table 4.5 contains image sequences of snow particles distribution for the 3 first re-positioning attempts of the various PRNGs supported by the simulator. To generate the images, the snow particle update kernel was altered to re-position every particle every frame with a uniform precipitation probability of 1.0 – that is every re-positioning attempt is successful –, and the resulting snow particle distribution was printed. This simulates the behavior of multiple re-positioning attempts in a single time step over multiple, simplifying the printing of distributions to images. The images color is normalized, so the pixel with the most particles is always of full brightness. Only the cuda version of Eidissen’s PRNGs is included as the OpenCL implementation is identical.

Eidissen’s PRNG fails as expected, but notably the CUDA `XORWOW`, and to a lesser extent the CUDA `MRG32`, results also exhibit visible patterns. This is likely due to the strategy used to seed the state for the individual threads. The others have no visual problems.

Performance

Table 4.6 and Figure 4.5 show timing information for the snow particle update kernel using the various PRNGs for re-positioning. The OpenCL versions use the same generators in redistribution, while the CUDA versions all use `philox` for this. For these tests, the particle count is set to $2 * 1024^2$, and the scene has a uniform precipitation probability 0.5. There are no OpenCL results for test machine **M2** due to a problem with the OpenCL support on this machine. The PRNGs identified as problematic in the last section are ignored in the follow, but it should be noted that in addition to showing light patterns the CUDA `mrng32k3a` implementation is also slower than the others³.

Among the CUDA implementations, the `philox` generator stands out as both the second fastest behind Eidissen’s and the only one that handled multiple re-positioning gracefully. Among the OpenCL candidates there are more candidates that exhibit the quality desired. All `clRNG` generators are notably slower than Eidissen’s. The two MRG variations and the LFSR generator exhibit similar performance (`mrng32k3a` is slightly faster), while the `philox` implementation is slightly slower. It is worth noting that `LFSR113` is not crush-resistant, as discussed in Section 2.4.1.

³This difference is markedly bigger on **M0** than **M2** over multiple test runs. This is likely a driver-related problem as the drivers differ while the CUDA version, and thus `cuRAND` implementation, are the same.

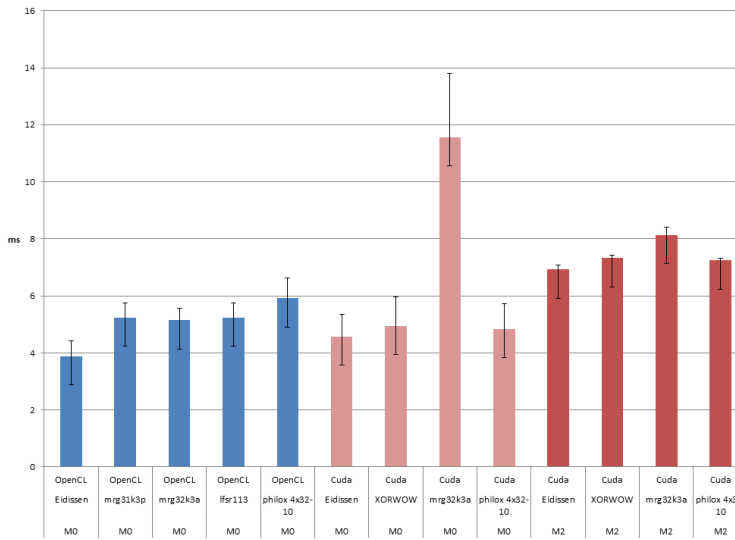


Figure 4.5: Run times for the snow update kernel with std. deviation for the various PRNGs used for re-positioning. Blue are OpenCL, red are CUDA for the two test machines **M0** and **M2**

For the CUDA code path, the `philox` generator is the obvious choice. It has statistically good quality and its performance is almost on par with the specialized PRNG previously used. An argument could be made for the `mrg32k3a` implementation to be used in the OpenCL back-end, as it is the fastest of the high quality generators. However, it is desirable that the two compute back-ends produce similar results. Sharing the generator between the implementation back-ends is seen as more valuable than the relatively small performance gain of using `mrg32k3a` over `philox`.

While all generators are accessible to the user through pre-processor defines, the simulator defaults to using the `philox` generator regardless of back-end.

4.3.2 Rejection cost

Redistribution with rejection has a performance cost; multiple positions must be generated and the precipitation probability texture (the *distribution map*) must be sampled at these positions and compared to a random value. As the number of re-positioning attempts increases and the probability of precipitation decreases, the performance deteriorates, however the effect is only significant when the probability is very close to zero and for large numbers of re-positioning attempts.

This is observable in Figure 4.6, which plots the performance of the snow particle update kernel for various combinations of re-positioning attempt counts and uniform precipitation probabilities (the probability of rejection of a generated position). These tests use the same settings as the ones in Section 4.3.1, but all use the `philox` PRNG. Timings are for the first 1000 frames, and this matters as with time the probability of

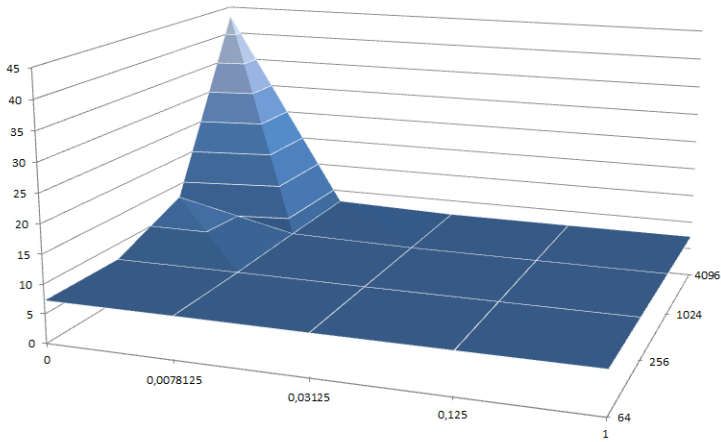


Figure 4.6: Frame times (vertical) of the snow particle update kernel in *ms* for 4 million particles with various precipitation probabilities (horizontal, range $[0, 1]$) and rejection attempts (depth, range $[64, 4096]$).

inactive particles increases (when precipitation probability is less than 1), which increases the number of re-positioning that take place over time, in turn degrading performance.

4.3.3 Distribution map

The radar data publicly available is of low temporal resolution, on the range of multiple minutes. When coupled with the simple linear interpolation used between distribution maps, the distribution does not appear to move “with the wind” when inspected. To appear as if in motion, the movement of features in the map needs to no more than a few texels per map. Kim et. al. [68] present a technique that uses feature detection to describe motion and use this motion for better interpolation between frames in video. Similar approaches can be used to generate better distribution maps between the available data, however the processing time for the algorithm presented is on the order of seconds per frame for required resolutions. This reinforces the decision to keep interpolation simple within the simulation and allow for better interpolation as a pre-processing step; computation is expensive, but the additional storage space for more distribution maps on disk is cheap.

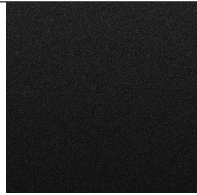

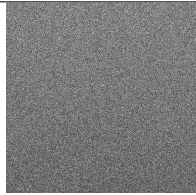
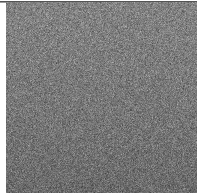
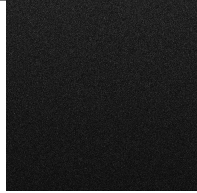
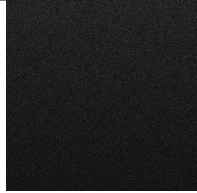
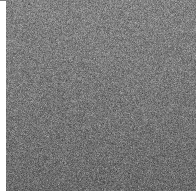
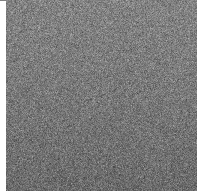
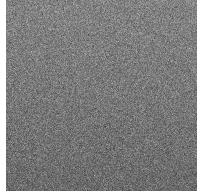
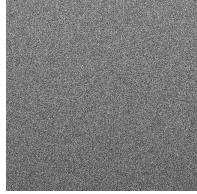
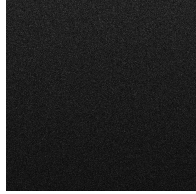
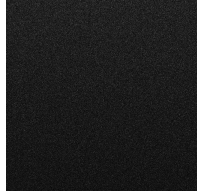
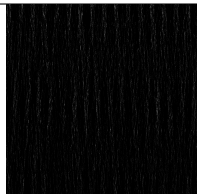
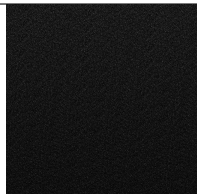
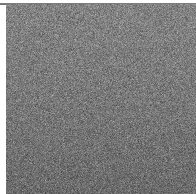
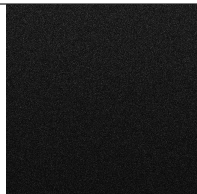
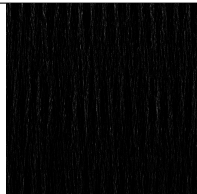
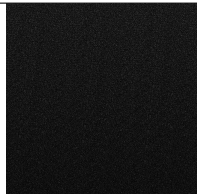

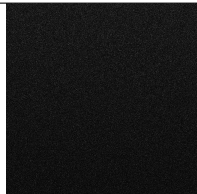
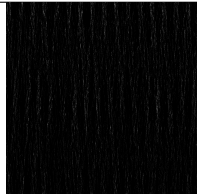
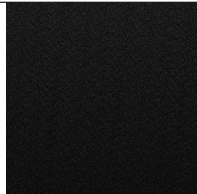
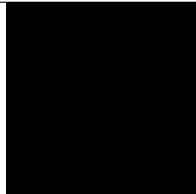
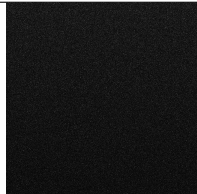
OpenCL				
PRNG	mrg31k3p	erg32k3a	lfsr113	philox
1st				
2nd				
3rd				
CUDA				
PRNG	XORWOW	erg32k3a	eidissen	philox
1st				
2nd				
3rd				

Table 4.5: Immediate redistribution attempts for the various GPU PRNG implementations. Brighter pixels indicate the part of the domain it maps to contains more particles. Pixel color is normalized.

Machine M0			
PRNG	Back-end	Average time	Std. Dev.
Eidissen	OpenCL	3.87ms	0.55ms
mrg31k3p	OpenCL	5.23ms	0.53ms
mrg32k3a	OpenCL	5.14ms	0.43ms
lfsr113	OpenCL	5.23ms	0.51ms
philox 4x32-10	OpenCL	5.91ms	0.71ms
Eidissen	Cuda	4.58ms	0.76ms
XORWOW	Cuda	4.94ms	1.02ms
mrg32k3a	Cuda	11.56ms	2.25ms
philox 4x32-10	Cuda	4.84ms	0.88ms
Machine M2			
PRNG	Back-end	Average time	Std. Dev.
Eidissen	Cuda	6.92ms	0.17ms
XORWOW	Cuda	7.32ms	0.10ms
mrg32k3a	Cuda	8.12ms	0.28ms
philox 4x32-10	Cuda	7.24ms	0.09ms

Table 4.6: Run-time of snow update kernels with various PRNGs used for re-positioning.

4.4 Clouds

The ray-marching approach, while widely used in offline rendering on volumetric data [48], is not yet widely used in real-time applications due to its cost. The results presented in this section show that while the cost is still large, high-end consumer GPUs may be able to handle it if the cloud rendering is allowed to take up a large portion of the frame time.

4.4.1 Technique differences

Figure 4.7 shows the various techniques implemented on a 128×128 distribution map over the Mt. St. Helens terrain.

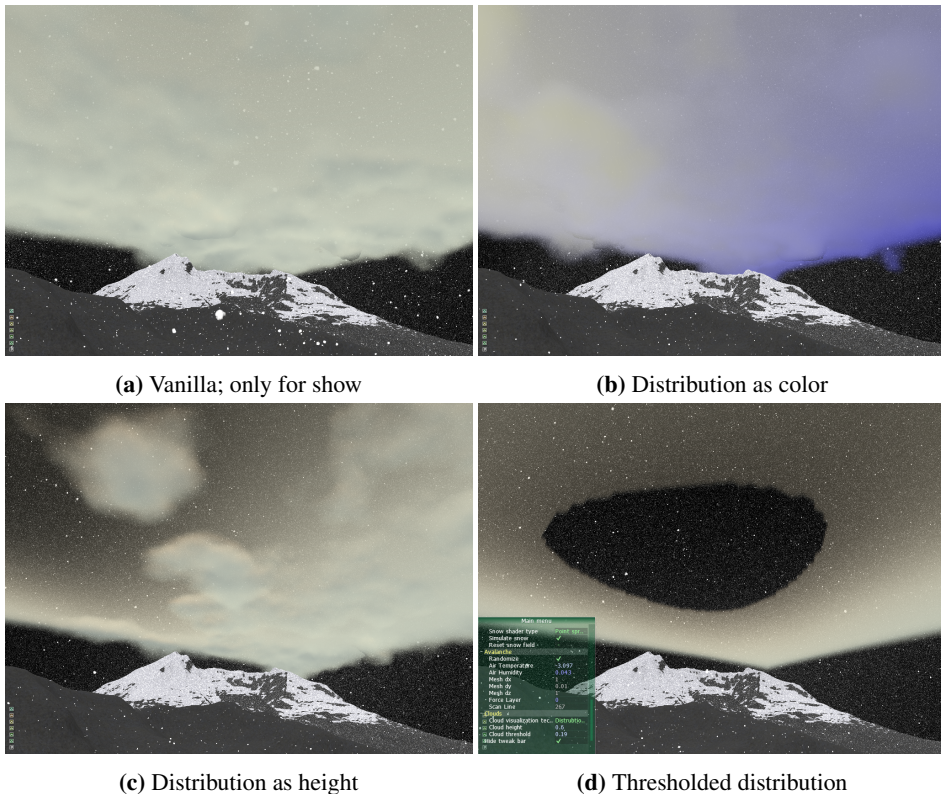


Figure 4.7: The four cloud visualization techniques implemented with precipitation probability (distribution) from Perlin noise. For (d) the threshold is set to 0.19.

Table 4.7 shows frame timings for the various cloud techniques implemented. Two runs per technique are performed with the camera placed for typical scenarios (close to the ground) and worst case performance (at one end of the cloud layer box, looking into

it parallel with the edge of the cloud)⁴. Pre-marching was used (128 steps), and the step lengths and counts for the varying octaves of FBM were $fbm_5 = \langle 32, 0.4 \rangle$, $fbm_4 = \langle 32, 0.4 \rangle$, $fbm_3 = \langle 24, 0.6 \rangle$, and $fbm_2 = \langle 64, 1.0 \rangle$. Distribution probability was constant across the scene, and set to 0.5. Rendering resolution was 1680×1050 , simulation volume $128 \times 32 \times 128$ and the *Perlin terrain shader* setting was used, on the Mt. St. Helens terrain at 768^2 vertices.

Worst-case camera placement				
Style	M0: Avg.	M0: Std. Dev.	M2: Avg.	M2: Std. Dev.
No clouds	11.15ms	0.25ms	16.88ms	0.49ms
Vanilla	24.06ms	0.66ms	47.0ms	1.4ms
Colour vis.	24.51ms	0.42ms	43.9ms	3.1ms
Height vis.	24.75ms	0.57ms	46.2ms	2.5ms
Thresholded	15.66ms	0.27ms	24.84ms	0.46ms
Typical camera placement				
Style	M0: Avg.	M0: Std. Dev.	M2: Avg.	M2: Std. Dev.
No clouds	12.12ms	0.28ms	19.71ms	0.45ms
Vanilla	17.61ms	0.42ms	28.61ms	0.50ms
Colour vis.	17.72ms	0.35ms	28.37ms	0.49ms
Height vis.	17.68ms	0.36ms	28.53ms	0.56ms
Thresholded	13.52ms	0.28ms	21.99ms	0.33ms

Table 4.7: Performance of the implemented cloud visualization styles in milliseconds over 1000 frames.

It is noteworthy that in the worst case, with these settings, the performance on test machine **M2** is no longer real-time. The performance is highly GPU bound, and the GPU in **M2** is a mid-range consumer device. In typical camera situations, where the camera is near the ground and the clouds cover $\sim \frac{1}{3}$ of the screen, performance is real-time even on this device, however by looking at the difference of frame times with clouds and without, a cost of $\sim 9ms$ per frame may still be prohibitive for use in domains such as games. For the snow simulator, the cost of other parts of the rendering is low enough that this is acceptable.

Another thing to note is the difference in performance between the thresholded height visualization, and vanilla versions. These differ only in their density function, where the thresholded version has no FBM evaluation and the vanilla version has no distribution texture lookup, and the height visualization version has both. Notably, the vanilla and height visualization versions have very similar performance, while the thresholded version is significantly faster, indicating that the shader is heavily bound by FBM evaluation, and the additional texture lookup to look up distribution values is not prohibitive.

⁴Worst-case camera coordinates are $\langle 38, 25, 63 \rangle$, typical case coordinates are $\langle 38, 3, 63 \rangle$

4.4.2 Visual artifacts

Figure 4.8 shows some visual artifact that occurs due to optimizations on the ray-marching step length, compared to an image of intended behavior. A large step length will prevent the high-frequency noise from being adequately sampled, leading to a visually uninteresting cloud. However, a smaller step size increases the required number of steps. In order for the shader to be optimized (and potentially in-lined) properly by the compiler, a fixed number of steps in the loop is preferred, with potential for early outs in cases where not all iterations are needed. If the step size is small enough that the end of the volume is not reached, artifacts may occur. As discussed in Section 3.4.2, variable step length is used to improve performance, and care must be taken to ensure that the step length is large enough that these artifacts are minimized. The problem is most common at local extremes of cloud height encountered early in the march, when viewed along the cloud top or bottom. In these cases, the pre-march (at large step length) ends and the color becomes non-zero, but because the ray only spends a small amount of steps inside non-zero density, the color is still close to zero, and the step length remains small. This leads to the march terminating before the end of the cloud is reached, and a jarring black outline can be seen if the ray would reach non-zero density further along if not terminated.

4.4.3 Optimization

Due to the costly nature of ray-marching a volume that spans more than the entire simulation domain, some optimizations were implemented in the shader. This section evaluates the performance versus quality trade-off of varying step length, and the performance gain from the pre-marching optimization described in Section 3.4.2.

Pre-marching

The pre-marching optimization has no impact on visual quality in itself, but allows for more aggressive step length increases and FBM octave decreases by increasing the probability that the high frequency sampling of the ray is not wasted outside the cloud. Table 4.8 includes timings without pre-marching and with various step lengths and maximum step counts. In these test, the density map is set to 0.5 across the entire domain, and the ray is marched 256 steps of no less than 0.5 length through 5 octave FBM. Pre-marching must always be performed on the highest octave noise to guarantee correct results, which in this case is the same 5 octave FBM. The timings are over 1000 frames, and the camera was set to the same position as in the *worst case* timings in Section 4.4.1; all other settings are shared with these tests, for vanilla clouds.

From these values, there is a small observable pure performance benefit to pre-marching if the step length is larger than that of regular marching. In the case where the step length is the same as with the regular march, performance degrades. This is more significant than the one extra marching step (the final pre-marching step that is inside is undone before regular marching is performed) should indicate, and likely explained by threads in a warp staying synced to the worst thread of the wrap. However, if the step size is increased for

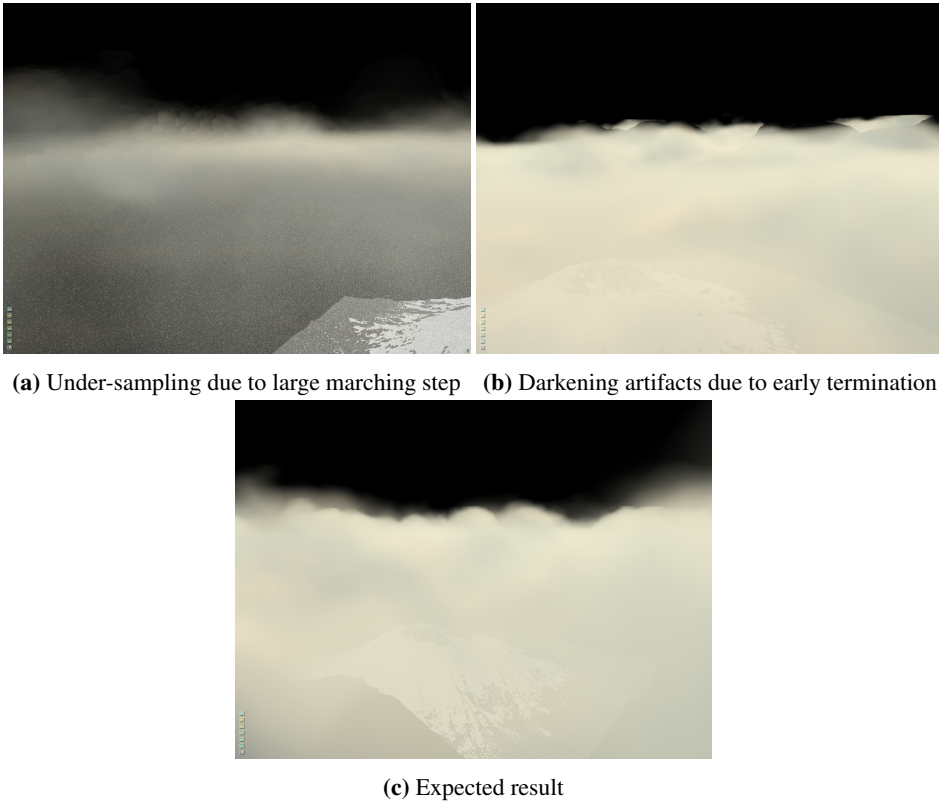


Figure 4.8: Artifacts and issues related to ray-marching step length

Step count	Step length	Avg. time	Std. dev.
0		24.75ms	1.6ms
128	0.5	25.65ms	2.0ms
128	1.0	22.63ms	1.7ms
128	2.0	23.09ms	2.0ms
256	0.5	25.25ms	2.5ms
256	1.0	22.84ms	2.1ms
256	2.0	22.04ms	2.0ms

Table 4.8: Full frame times at various levels of pre-marching granularity and length on test machine M0

the pre-marching step, a performance improvement is observed.

Increasing the number of steps does not appear to have an impact, given that total step length – $steplength \times stepcount$ – is large enough that the first non-zero density part of the cloud, or the end of the cloud volume, is reached. In these tests, the cloud volume

along the camera is 126 units long, so all but the $\langle 128, 0.5 \rangle$ case are guaranteed to be large enough. Due to the camera placement in the cloud, reaching the end of the pre-marching loop without a non-zero density value is unlikely even in this case.

Step length

Table 4.9 and Figure 4.9 contain timings for a number of different step length configurations. The settings are as for the pre-marching discussion above, however pre-marching is enabled at step length 1.0 and 192 steps. Higher octave noise is always sampled before lower octave noise, that is if multiple octaves are used, these manifest in code as multiple loops placed sequentially from highest octave to lowest octave.

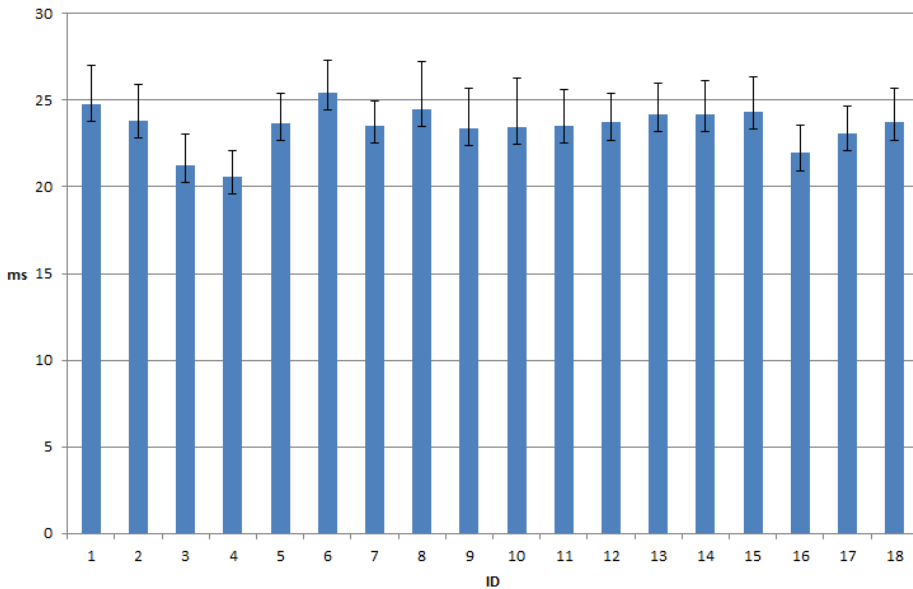


Figure 4.9: Timings of the full rendering frame for the various test settings in Table 4.9.

The performance observed is very similar for all combinations of values, and except cases 1 and 2 they all exhibit some degree of artifacts. Case 2 handles this test only due to the camera placement; the worst-case ray distance in the test scene requires a total marched distance of 182 units. However as the tested values all cover similar ranges – with the exception of the first –, quality and performance comparisons on the test scene are fair. These values indicate that the optimization of using cheaper noise and longer (minimum) step lengths as the ray is traversed do not have a significant impact on performance, while they do introduce minor artifacts compared to sampling at small steps along the complete ray. The different noise functions in fact introduce a popping artifact between frames when the first non-zero density of the ray moves, as different sections of the ray density values change frequency. However, if the system is constrained the per-

ID	Step counts	Step lengths	Avg time	Std. dev.	Artifacts
1	$\langle 512, 0, 0, 0 \rangle$	$\langle 0.4, 0, 0, 0 \rangle$	24.78ms	2.2ms	
2	$\langle 256, 0, 0, 0 \rangle$	$\langle 0.4, 0, 0, 0 \rangle$	23.82ms	2.1ms	
3	$\langle 256, 0, 0, 0 \rangle$	$\langle 0.7, 0, 0, 0 \rangle$	21.27ms	1.8ms	minor
4	$\langle 256, 0, 0, 0 \rangle$	$\langle 1.0, 0, 0, 0 \rangle$	20.58ms	1.5ms	minor
5	$\langle 128, 0, 0, 0 \rangle$	$\langle 0.4, 0, 0, 0 \rangle$	23.66ms	1.7ms	major
6	$\langle 128, 64, 0, 0 \rangle$	$\langle 0.4, 0.4, 0, 0 \rangle$	25.44ms	1.9ms	minor
7	$\langle 128, 64, 0, 0 \rangle$	$\langle 0.4, 0.7, 0, 0 \rangle$	23.52ms	1.4ms	minor
8	$\langle 128, 64, 0, 0 \rangle$	$\langle 0.4, 1.0, 0, 0 \rangle$	24.46ms	2.8s	major
9	$\langle 64, 64, 0, 0 \rangle$	$\langle 0.4, 0.4, 0, 0 \rangle$	23.41ms	2.3ms	major
10	$\langle 64, 64, 0, 0 \rangle$	$\langle 0.4, 0.7, 0, 0 \rangle$	23.47ms	2.78ms	major
11	$\langle 64, 64, 64, 0 \rangle$	$\langle 0.4, 0.4, 0.7, 0 \rangle$	23.52ms	2.1ms	minor
12	$\langle 64, 64, 32, 0 \rangle$	$\langle 0.4, 0.4, 0.7, 0 \rangle$	23.71ms	1.7ms	minor
13	$\langle 64, 64, 64, 64 \rangle$	$\langle 0.4, 0.5, 0.6, 0.7 \rangle$	24.21ms	1.8ms	minor
14	$\langle 64, 64, 64, 32 \rangle$	$\langle 0.4, 0.5, 0.6, 0.7 \rangle$	24.22ms	1.9ms	minor
15	$\langle 128, 32, 32, 32 \rangle$	$\langle 0.4, 0.6, 0.8, 1.0 \rangle$	24.33ms	2.0ms	minor
16	$\langle 32, 24, 16, 128 \rangle$	$\langle 0.4, 0.4, 0.7, 1.0 \rangle$	21.95ms	1.6ms	minor
17	$\langle 128, 0, 0, 128 \rangle$	$\langle 0.4, 0, 0, 0.4 \rangle$	23.06ms	1.6ms	minor
18	$\langle 192, 0, 0, 64 \rangle$	$\langle 0.4, 0, 0, 1.0 \rangle$	23.71ms	2.0ms	minor

Table 4.9: Timings for various combinations of minimum step lengths and counts, and FBM octave counts on test machine **M0**. Step lengths and counts are given as $\langle fbm_5, fbm_4, fbm_3, fbm_2 \rangle$ tuple. The *Artifacts* column is a subjective measure; it describes the severity the visual artifacts described in Section 4.4.2, if present

formance requirements for sampling at maximal frequency along the entire ray cannot be met, a concession must be made in quality. Either the sampling distance must be increased (at the cost of under-sampling artifacts), the noise octaves decreased (at the cost of visual fidelity) or the combined approach used (at the cost of popping and darkening artifacts). If the camera is placed inside the clouds, as in these test, the prior two mentioned artifacts are less obnoxious, but if the camera is placed at a greater angle to the cloud surface, both popping and darkening artifacts are less likely to occur, and a combined approach with increasing step count and length as octaves decrease prove useful. Test case 16 is an example of such a configuration, and along with cases 3 and 4 (which exhibit under-sampling), it sets itself apart by being faster than the rest.

4.4.4 Interaction with other systems

Shadow maps rendering as performed in the simulator uses a depth comparison to determine if a pixel is in shadow or not. This is only performed in the *complex* and *Perlin* ground shaders. *In shadow* is a binary state, but clouds are not either fully opaque or fully transparent. Due to this, clouds are not accounted for in the terrain shading.

If a shadow mapping technique that handles partial transparency is introduced in the

future, rendering the clouds at full quality may still be too costly. Lower octave noise may be good enough for this purpose however, and under-sampling should also prove less of an issue, so it may be possible to include the clouds in a shadow mapping pass with significantly lower performance cost than in the final rendering pass.

4.5 Terrain

Figure 4.10 contains images of the simulator running on three terrain models created from Google Maps⁵ height-map images, and $1m$ and $0.5m$ granularity LiDAR scans respectively. The high resolution data-sets highlight problems with a height-map representation in high-frequency areas such as forests and cities; while trees and buildings are captured, the vertical nature of buildings or layer nature of trees cannot be accurately represented with a height-map. This is being addressed in parallel to this thesis by Halsauet [69].

Another issue is highlighted in Figure 4.11. Here the resolution of the terrain mesh can be compared to the resolution of the wind simulation voxels (red marks obstructed voxels).

The pre-processing script introduced in Section 3.5 takes $5.25s$ to convert the London Heathrow terrain above into a raw height-map on test machine **M0**, indicating that pre-processing this data is still beneficial to reduce loading times of the simulator itself.

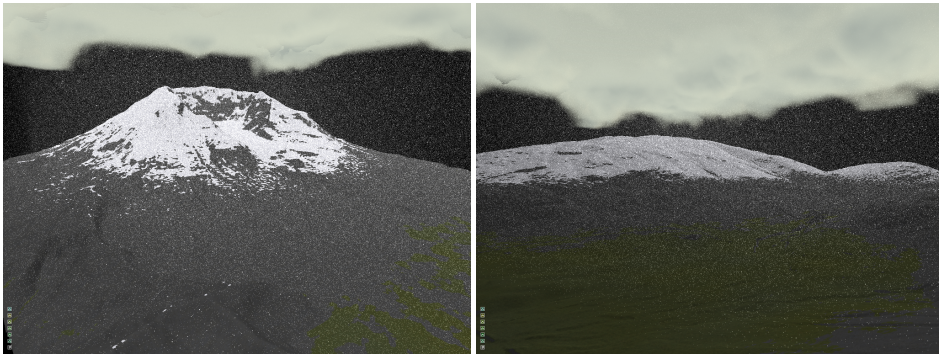
4.6 Miscellaneous improvements

This section discusses the implications of the various implementations described in Section 3.6. These changes are primarily performance-related.

4.6.1 3D rendering

The performance implications of quad-buffered stereoscopic rendering are fairly predictable; the scene must be rendered twice, with the exception of the reuse of the shadow map. As can be seen in Table 4.10, the frame times double when the simulation is turned off (but snow particles are still rendered). By comparing the performance of cases with and without shadow maps but other settings identical (f.ex. 3 and 4), it is evident that the shadow map pass costs so little it fails to register on these benchmarks, so the double frame times is perfectly in line with expectations. The cloud rendering is prohibitively expensive on the slightly older GPU in test machine **M1** at the test resolution of 1680×1050 . However, as the 3D implementation is motivated by demonstration session for the HPC-Lab, the performance must be evaluated in this perspective, and these demonstrations are typically performed using an external processor of the lower resolution 1024×768 . While this is only 45% of the pixels of the test setup, the target of 120Hz – or $8.33ms$ – is unlikely to

⁵<https://maps.google.com> (last accessed: 2016-06-20)



(a) Mt. St. Helens, WA, USA – 768^2 vertices

(b) Lake district, UK – 1024^2 vertices



(c) Heathrow, London, UK – 2048^2 vertices

Figure 4.10: Terrains imported from (a) Google Maps height-maps, (b) 1m-, and (c) 0.5m-resolution LiDAR scans.

be met by all but the lowest fidelity settings tested, with significantly less particles than tested here (2 million). GPUs can be upgraded, and in this eventuality the performance is dominated by the vast majority of rendering time being spent in the cloud rendering. While this is problematic, the alternative to the costly ray-marching is to use billboarding, which has its own issues with stereoscopic rendering, as discussed in Section 3.4.1. The optimizations mentioned for low-performance systems in Section 4.4.3 may be applied with success in a controlled environment such as a demonstration booth, as the camera can be set to a position where the artifacts are unlikely to be problematic.

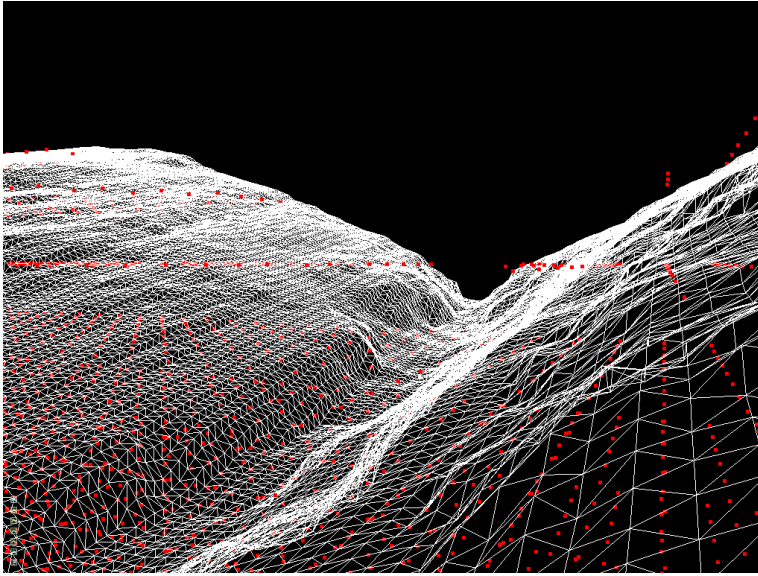


Figure 4.11: The terrain mesh of Lake District terrain, at 1024^2 vertices, compared to the obstacle map at $128 \times 32 \times 128$ voxels.

Hardware support

The lack of OpenGL 3Dvision support on consumer hardware is limiting to current implementation. The primary issue is that a high-end professional range card must be used to render the clouds at full fidelity and high screen resolutions, while the usual advantages of such cards (such as better double precision support) are not needed. This hardware is significantly more expensive than the equivalent performance (on this simulation) in consumer hardware.

Additionally, the lack of Linux drivers for high-end consumer HMDs are currently the primary deterrent to implementing Virtual Reality (VR) support. Windows support in the simulator has been brought nearer by the replacement of the POSIX-only PRNG used on the CPU-side of the simulation, but there are still issues to overcome with the timing system and incompatible versions of CUDA Microsoft's C++ compiler⁶.

4.6.2 OpenCL sampling improvements

Due to incompatibilities in OpenCL and CUDA support on test machine **M2**, and OpenCL kernel compilation errors on **M1** likely due to hardware support issues, this performance

⁶Microsoft's newest compiler suite – Visual Studio 2015 – is not supported by CUDA 7.5, the latest release. While the simulator should compile with older versions, repeated attempts to downgrade to the previous (and supported) version Visual Studio 2012 on test machine **M2** (which also has Microsoft Windows installed) did not prove successful.

ID	Terrain shader	Shadows	Clouds	Avg. Time	Std. Dev.
<i>With simulation</i>					
2D rendering					
1	Simple			16.77ms	0.27ms
2	Complex			33.33ms	0.27ms
3	Perlin			32.97ms	0.31ms
4	Perlin	Y		32.97ms	0.28ms
5	Perlin	Y	Vanilla	55.28ms	0.53ms
3D rendering					
6	Simple			22.91ms	0.29ms
7	Complex			56.17ms	0.36ms
8	Perlin			55.41ms	0.34ms
9	Perlin	Y		55.39ms	0.36ms
10	Perlin	Y	Vanilla	101.72ms	2.0ms
<i>Without simulation</i>					
2D rendering					
11	Simple			5.72ms	0.026ms
12	Complex			22.36ms	0.043ms
13	Perlin			22.17ms	0.044ms
14	Perlin	Y		22.15ms	0.046ms
15	Perlin	Y	Vanilla	44.41ms	0.071ms
3D rendering					
16	Simple			11.80ms	0.062ms
17	Complex			44.83ms	0.20ms
18	Perlin			44.68ms	0.11ms
19	Perlin	Y		44.66ms	0.11ms
20	Perlin	Y	Vanilla	90.63ms	0.16ms

Table 4.10: Full frame times with and without stereoscopic rendering for a variety of settings on test machine **M1**, for the Mt. St. Helens terrain.

was only compared on machine **M0**.

Filtering strategy	Particle update		Wind advection	
	Avg.	Std. Dev.	Avg.	Std. Dev.
Hardware	10.09ms	0.091ms	161μs	4μs
Software	15.85ms	0.19ms	161μs	4μs

Table 4.11: Comparison of kernel times for the affected kernels using hardware and software trilinear filtering of the wind velocity texture in the OpenCL simulation.

A significant improvement in performance of the snow particle update kernel can be observed from the timings in Table 4.11 when hardware sampling is used. This is likely partially because the filtering does not use compute unit time, but also because the GPU

scheduler can better coalesce the texture fetches. The use of texture memory means the data may be stored in a filtering- friendly tiled layout instead of linearly. This would in turn reduce the number of cache lines read per fetch from the kernels as well.

4.6.3 Bindless textures

Bindless texture in the CUDA kernels was tested on test machines **M0** and **M2** only, as the GPU in test machine **M1** does not have support for the feature. Table 4.12 contains kernel timings with and without the feature. The test is run at similar rendering settings (1680 × 1050 resolution) and for 2×1024^2 particles.

Machine & Strategy	Particle update		Wind advection	
	Avg.	Std. Dev.	Avg.	Std. Dev.
M0 Bindless	7.26	0.087	0.33	0.68
M0 State-full	7.25	0.079	0.39	0.95
M2 Bindless	7.29	0.097	0.43	1.13
M2 State-full	7.30	0.10	0.47	1.23

Table 4.12: Time measurements of the relevant textures and full frame using bindless and state-full textures. All times are in *ms*.

There was no significant change in performance between the settings, indicating that the simulator does not have sufficient state changes in sampler and texture bindings to cause significant CPU overhead; the feature primarily helps reduce overhead when the driver validates state changes to texture or sampler bindings, and the number of such changes in the cuda part of the simulation is very low, as only a single texture is used.

The rendering pipeline uses a more textures, depending on which techniques are selected in the configuration, but the number of texture-related state changes is still in the low double-digits. Everitt et. al. [70] shows that the problem the feature addresses is prevalent in situations orders of magnitude larger by showing marked improvements in a test case with 10000 quads using different textures. Since the OpenGL extensions required are not widely supported, and after observing the lack of performance impact on the CUDA kernels, bindless support for the rendering pipeline was not implemented.

4.6.4 Wind simulation stability improvements

Section 3.6.5 describes two potential solutions to the wind simulation stability problem. Figure 4.12 shows the visual result of the solutions, and Table 4.13 contains kernel timings for the affected kernels over 1000 frames compared to the old implementation.

Increasing the SOR iteration count alone did no improve on the old version. It is likely that a different (decomposition based) solver would need to be implemented, and it is possible that performance improvements on GPUs in recent years make this feasible in real-time, unlike when Eidissen [27] originally chose the SOR solver.

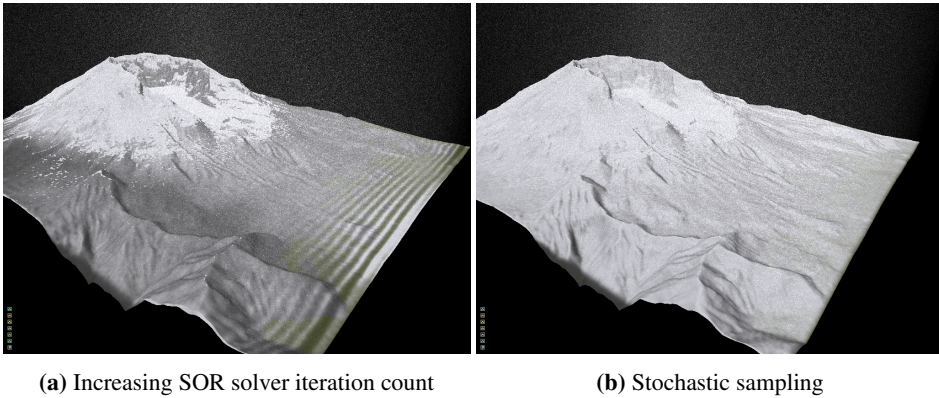


Figure 4.12: The effect of the proposed solutions to the wind simulation stability problem. The coefficients are in (a) are $\langle 1.7, 1.5, 1.2, 1.1, 1.1, 1.05, 1.02, 1.01 \rangle$.

Solution	Wind advect kernel	Poisson solver kernels
Old	$0.22 \pm 0.12ms$	$0.58 \pm 0.14ms$
SOR iteration increase	$0.25 \pm 0.073ms$	$1.09 \pm 0.40ms$
Stochastic sampling	$0.22 \pm 0.071ms$	$0.59 \pm 0.23ms$

Table 4.13: The stability problem solutions' effect on the wind simulation kernel run times.

The stochastic approach completely alleviates the problem, and as a visually pleasing solution it is perfect, as it does not even affect performance noticeably. No attempt was made to prove its physical validity however, and as such it is a band-aid solution only recommended for visual applications, such as demonstrations of video games. For simulation runs where the snow accumulation accuracy is of primary interest, it is not recommended to use this solution without first showing that it produces valid results.

Conclusion and Future Work

As the processing power of GPUs increase, more and more complex real-time simulations are made possible. Accurate snow simulations that include real weather data are of interest for scientific, engineering, and entertainment purposes. The work performed as part of this thesis to enhance the HPC-Lab snow simulator can be separated into two parts: the work to enable use of real-world data as a basis for simulation, and the numerous minor changes made to the simulator in an effort to improve it's performance and visual quality, as well as simplify future work on it.

5.1 Real-world data integration

The support for varied precipitation probability not only enables the use of interesting source data for snow fall, but also motivated the change of the PRNG used in the simulator to ones of higher quality. While the precipitation data available is of low resolution – temporally and spatially – compared to the terrain models supported, the future may bring more detailed data, or pre-processing may be used to infer or insert detail. The simulation handles scaling resolution of this data seamlessly. Adding cloud rendering both improves the visual quality of the sky, as the upper edge of the simulation volume is now less visually jarring, and proved useful as a visualization tool for the precipitation probabilities. The use of ray-marching shows that high-end consumer GPUs are capable of using this technique in real-time – with the high-end test machine (**M0**) maintaining frame-rates of over 30HZ even in the worst-case scenario –, however the cost may still be prohibitive for use in applications where volumetric data only comprises a smaller part of the overall rendering budget, such as most video games.

Prior work to support high-resolution terrain data was re-integrated into the simulator tool chain, and as new terrain models of up to half-meter accuracy were tested, a number of problems with the terrain representation emerged.

The wind simulation's only inputs are the boundary wind and the terrain. To enable the simulation to reproduce real-world conditions, uniform boundary wind is insufficient. The integration of wind sources enables varied wind to better account for obstacles and variations in wind outside the simulation domain. The algorithm introduced to map boundary voxels to contributing wind sources is shown to be sufficiently fast for the small number of wind sources that may be expected, although the computational complexity is likely to be problematic for larger numbers. For the current limitation of 32 wind sources, the cost of calculating neighborhood relations is less than the time it takes to load the source data from a solid state disk. The interpolation technique, as well as the data format for wind sources, can be extended to support temperature and humidity for use in the avalanche prediction model. The in-kernel interpolation of neighbor samples is shown to be at best 1.2x and at worst 4x the cost of using a constant value, however even in this case the kernel in question amounts to $< 10\%$ of the total frame time.

5.2 Miscellaneous improvements

The OpenCL version's performance is brought significantly closer to the CUDA version by the introduction of hardware filtering when sampling the wind velocity texture. On average, the snow particle update kernel is shown to have a speedup of 1.57.

Attempts to improve performance by using bindless textures yielded no gains. This is attributed to the low number of texture state changes in the simulator; the problem the technique solves simply is not present.

Stereoscopic rendering is re-introduced into the latest version of the simulator. The performance cost of the new cloud rendering is an issue on the older hardware available for testing which was unable to maintain the required frame-rate to be considered real-time, however the visual performance is good, and as hardware performance scales ray-marching should prove superior to the alternatives discussed. Limited driver support and hardware access prevented the support of VR integration.

Issues related to the quality of the SOR solver used in the wind simulation were attempted solved. Increasing the iteration count proved unsuccessful, while using stochastic sampling points as a basis for interpolation in the wind advection appears to solve the problem. This solution is sufficient for uses where visual quality is important and simulator accuracy may be sacrificed, however it is not guaranteed to be accurate. Enabling lower quality simulation while still yielding good visual results suggests it may be useful for computer graphics and video game applications.

5.3 Future work

This section suggests some improvements to the simulator suggested by the work in this thesis, or by problems encountered along the way.

5.3.1 Snow melting

Chang and Ryoo [71] use shadow mapping to simulate snow melting. The accumulated snow in the simulator and existing support for shadow mapping make this an interesting avenue of research. The cloud layer is currently not taken into account when rendering the shadow map, and the density of clouds may not map cleanly to precipitation values, so a separate cloud density map may be of interest. This data can be extracted from radar data.

To properly handle melting in the avalanche prediction model, temperature and humidity should be interpolated from ground truth points similarly to the boundary wind, and vary with light contribution based on the shadow map. If further accuracy is wanted, phase change to water, and transport of water along the terrain under the snow may be simulated, however this complicates the simulation, as another fluid simulation must be performed in or under the snow layer to transport the water, and this water must be taken account for as a source of temperature change in the avalanche prediction model. To visualize water transport, river rendering may prove useful as well, which would further improve the visual quality of the simulator.

5.3.2 Terrain model improvements

The height-map based terrain is shown to be problematic in Section 4.5. The parallel work of Halsauet [69] may solve this problem by supporting arbitrary geometry, however the terrain models available from public sources are still constrained to grid-based approaches. It is likely that manual labor must be used to separate the geometry that can be represented using the current terrain rendering approach (with tri-planar blending), and geometry that needs different rendering techniques, such as trees and buildings. By supporting arbitrary geometry the avalanche prediction model may prove useful in city-scape terrains to predict roof avalanches.

5.3.3 Wind simulation improvements

The current wind simulation volume is a uniform grid, but not all regions of a simulation contain features at the same scale. Along geometry, fine-grained vorticity may cause vortexes that lead to important features such as snow cornices. To support small-scale vorticity the current uniform grid approach would require the entire domain to be simulated at that scale, which has both computational and memory requirements current hardware does not support. Multi-resolution approaches are an interesting approach to this problem.

Balme et. al. [72] present an approach in which the simulation domain is an octree that is subdivided in interesting regions, spending the majority of computational power and memory on those regions that are deemed interesting. They use wavelet analysis to determine which regions will have interesting features. This approach should be integrated decently with the terrain model introduced in [69], which use should be investigated for the simulation domain.

5.3.4 Extended Multi-GPU support

The simulator has support for separate simulation and rendering GPUs, as is required to work with certain scientific GPUs that don't have display output ports. By dividing the simulation work between multiple GPUs the simulation volume resolution may be increased further. The current limitation of the simulation volume size is a combination of memory requirements and computational cost of the wind simulation, and as the work of Spampinato et. al. [73] shows, multi-GPU solutions may be well suited to alleviate these issues.

5.3.5 Raytracing

The cloud rendering introduced in this thesis uses ray-marching, a performance optimization of full ray-traced rendering. Ludvigsen [74] investigates the use of NVIDIA's OptiX ray-tracing platform for real-time use. As GPUs get faster, and potentially with use of multi-GPU solutions, moving the entire rendering to ray-tracing may be feasible for improved quality even under real-time constraints.

5.3.6 NEXRAD support

NEXRAD makes radar data available in binary form, and while, for reasons detailed in Section 3.4.3, this thesis does not cover support for such data, it should be considered for future addition. A pre-processing script using the available tools could convert the data, and any other formats that may be available in the future, into the distribution textures the simulator now supports with little to no changes to the core simulator code.

Bibliography

- [1] D. J. Acheson, *Elementary fluid dynamics*. Oxford University Press, 1990.
- [2] N. Foster and D. Metaxas, “Realistic animation of liquids,” *Graph. Models Image Process.*, vol. 58, pp. 471–483, Sept. 1996.
- [3] J. Stam, “Stable fluids,” in *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH ’99, (New York, NY, USA), pp. 121–128, ACM Press/Addison-Wesley Publishing Co., 1999.
- [4] J. Stam, “Real-time fluid dynamics for games,” in *the Game Developers Conference*, 2003.
- [5] M. Lieb, “A full multigrid implementation on staggered adaptive cartesian grids for the pressure poisson equation in computational fluid dynamics,” Master’s thesis, Technische Universität München, 2008.
- [6] J. David M. Young, *Iterative Methods for Solving Partial Difference Equations of Elliptic Type*. PhD thesis, Harvard University, Mathematics Department, Cambridge, MA, USA, May 1950.
- [7] A. C. Elster, *Parallelization Issues and Particle-in-cell Codes*. PhD thesis, Ithaca, NY, USA, 1994. UMI Order No. GAX95-11889.
- [8] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C (2Nd Ed.): The Art of Scientific Computing*. New York, NY, USA: Cambridge University Press, 1992.
- [9] R. J. Doviak and D. S. Zrnic, *Doppler radar and weather observations: Second edition*. Dover books on engineering, Dover publications, 2006.
- [10] M. K. Yau and R. R. Rogers, *A short course in cloud physics*. Butterworth-Heinemann, 1989.

-
- [11] USDOJ, “Standards for digital elevation models: Part 2 specifications,” January 1998. <http://nationalmap.gov/standards/pdf/2DEM0198.PDF> (last accessed: 2016-06-20).
- [12] H. Lien, “Procedural generation of road for use in the snow simulator,” 2011. Master’s thesis pre-project.
- [13] D. E. Knuth, *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.
- [14] D. H. Lehmer, “Mathematical methods in large-scale computing units,” in *Proceedings of the Second Symposium on Large Scale Digital Computing Machinery*, (Cambridge, United Kingdom), pp. 141–146, Harvard University Press, 1951.
- [15] A. Fog, “Instruction tables: List of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus,” January 2016. http://www.agner.org/optimize/instruction_tables.pdf (last accessed: 2016-06-20).
- [16] N. A. Gershenfeld, *The nature of mathematical modeling*. Cambridge, New York: Cambridge University Press, 1999.
- [17] P. L’Ecuyer, “Good parameters and implementations for combined multiple recursive random number generators,” *Operations Research*, vol. 47, pp. 159–164, 1998.
- [18] P. L’Ecuyer, “Tables of Maximally-Equidistributed Combined LFSR Generators,” *Mathematics of Computation*, vol. 68, no. 225, pp. 261–269, 1999.
- [19] G. Marsaglia, “Xorshift rngs,” *Journal of Statistical Software*, vol. 008, no. i14, 2003.
- [20] J. Hruska, “Amd destroys nvidia at bitcoin mining, can the gap ever be bridged?,” 2013. <http://www.extremetech.com/computing/153467-amd-destroys-nvidia-bitcoin-mining> (last accessed: 2016-06-20).
- [21] M. Matsumoto and T. Nishimura, “Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator,” *ACM Trans. Model. Comput. Simul.*, vol. 8, pp. 3–30, Jan. 1998.
- [22] M. Saito, “A variant of mersenne twister suitable for graphic processors,” *CoRR*, vol. abs/1005.4973, 2010.
- [23] J. Daemen and V. Rijmen, *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002.
- [24] “Specification for the advanced encryption standard (aes).” Federal Information Processing Standards Publication 197, 2001. <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf> (last accessed: 2016-06-20).

-
- [25] J. K. Salmon, M. A. Moraes, R. O. Dror, and D. E. Shaw, "Parallel random numbers: As easy as 1, 2, 3," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, (New York, NY, USA), pp. 16:1–16:12, ACM, 2011.
- [26] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker, "The skein hash function family." Submission to NIST (Round 3), 2010. <http://www.skein-hash.info/sites/default/files/skein1.3.pdf> (last accessed: 2016-06-20).
- [27] R. Eidissen, "Utilizing gpus for real-time visualization of snow," Master's thesis, Norges teknisk-naturvitenskapelige universitet, 2009.
- [28] M. E. O'Neill, "Pcg: A family of simple fast space-efficient statistically good algorithms for random number generation." Unpublished, <http://www.pcg-random.org/pdf/toms-oneill-pcg-family-v1.02.pdf> (last accessed: 2016-06-20), 2015.
- [29] P. L'Ecuyer and R. Simard, "Testu01: A c library for empirical testing of random number generators," *ACM Trans. Math. Softw.*, vol. 33, Aug. 2007.
- [30] J. K. Helsing and A. C. Elster, *Noise Modeler: An Interactive Editor and Library for Procedural Terrains via Continuous Generation and Compilation of GPU Shaders*, pp. 469–474. Cham: Springer International Publishing, 2015.
- [31] K. Perlin, "Improving noise," *ACM Trans. Graph.*, vol. 21, pp. 681–682, July 2002.
- [32] V. Chandrasekaran, M. B. Wakin, D. Baron, and R. G. Baraniuk, "Surflets: a sparse representation for multidimensional functions containing smooth discontinuities," in *Information Theory, 2004. ISIT 2004. Proceedings. International Symposium on*, pp. 563–, June 2004.
- [33] F. Aurenhammer, "Voronoi diagrams—a survey of a fundamental geometric data structure," *ACM Comput. Surv.*, vol. 23, pp. 345–405, Sept. 1991.
- [34] M. d. Berg, O. Cheong, M. v. Kreveld, and M. Overmars, *Computational Geometry: Algorithms and Applications*. Santa Clara, CA, USA: Springer-Verlag TELOS, 3rd ed. ed., 2008.
- [35] J. R. Shewchuk, "Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator," in *Applied Computational Geometry: Towards Geometric Engineering* (M. C. Lin and D. Manocha, eds.), vol. 1148 of *Lecture Notes in Computer Science*, pp. 203–222, Springer-Verlag, May 1996. From the First ACM Workshop on Applied Computational Geometry.
- [36] W. Rollmann, "Zwei neue stereoskopische methoden," *Annalen der Physik*, vol. 166, no. 9, pp. 168–187, 1853.
- [37] J. B. Kaiser, *Make your own stereo pictures*. Macmillan, 1955.
-

-
- [38] NVidia, “Nvidia 3d vision pro and stereoscopic 3d.” Whitepaper, http://www.nvidia.com/docs/IO/40505/WP-05482-001_v01-final.pdf (last accessed 2016-06-20), 2010.
- [39] C. Hall and E. Betters, “Best vr headsets to buy in 2016, whatever your budget,” 2016. <http://www.pocket-lint.com/news/132945-best-vr-headsets-to-buy-in-2016-whatever-your-budge> (last accessed: 2016-06-20).
- [40] K.-E. Bystrom, W. Barfield, and C. Hendrix, “A conceptual model of the sense of presence in virtual environments,” *Presence: Teleoper. Virtual Environ.*, vol. 8, pp. 241–244, Apr. 1999.
- [41] J. LaViola and J. Joseph, “A discussion of cybersickness in virtual environments,” *SIGCHI Bull.*, vol. 32, pp. 47–56, Jan. 2000.
- [42] J. Carmack, “Latency mitigation strategies,” 2013. Original article has been removed, link to third party reproduction <https://www.twentymilliseconds.com/post/latency-mitigation-strategies/> (last accessed: 2016-06-20).
- [43] P. Norvig, “Teach yourself programming in ten years,” 2014. <http://norvig.com/21-days.html#answers> (last accessed: 2016-06-20).
- [44] R. J. Hovland, “Latency and bandwidth impact on gpu-systems,” Master’s thesis, Norges teknisk-naturvitenskapelige universitet, 2008.
- [45] F. Giessen, “A trip through the graphics pipeline 2011,” 2011. <https://fgiesen.wordpress.com/2011/07/09/a-trip-through-the-graphics-pipeline-2011-index/> (last accessed: 2016-06-20).
- [46] T. L. Falch and A. C. Elster, “Register caching for stencil computations on gpus,” in *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2014 16th International Symposium on*, pp. 479–486, IEEE, 2014.
- [47] W. Jarosz, *Efficient Monte Carlo Methods for Light Transport in Scattering Media*. PhD thesis, UC San Diego, September 2008.
- [48] A. Bouthors, F. Neyret, N. Max, E. Bruneton, and C. Crassin, “Interactive multiple anisotropic scattering in clouds,” in *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games, I3D ’08*, (New York, NY, USA), pp. 173–182, ACM, 2008.
- [49] O. Elek, T. Ritschel, A. Wilkie, and H.-P. Seidel, “Interactive cloud rendering using temporally coherent photon mapping,” *Computers & Graphics*, vol. 36, pp. 1109–1118, 2012.
- [50] O. Elek, T. Ritschel, C. Dachsbacher, and H.-P. Seidel, “Principal-ordinates propagation for real-time rendering of participating media,” *Computers & Graphics*, vol. 45, 2014.

-
- [51] I. Quilez, “Dynamic clouds,” 2005. Interactive demo, <http://www.iquilezles.org/www/articles/dynclouds/dynclouds.htm> (last accessed: 2016-06-20).
- [52] M. J. Harris, “Real-time cloud rendering for games,” in *Proceedings of Game Developers Conference*, pp. 21–29, 2002.
- [53] B. Wronski, “Volumetric fog: Unified compute shader based solution to atmospheric scattering,” in *ACM Siggraph*, 2014.
- [54] I. Quilez, “Volumetric clouds,” 2013. Interactive demo, <https://www.shadertoy.com/view/XslGRr> (last accessed: 2016-06-20).
- [55] I. Saltvik, “Parallel methods for real-time visualization of snow,” Master’s thesis, Norges teknisk-naturvitenskapelige universitet, 2006.
- [56] I. Saltvik, A. C. Elster, and H. R. Nagel, “Parallel methods for real-time visualization of snow,” *Applied Parallel Computing. State of the Art in Scientific Computing, Lecture Notes in Computer Science*, pp. 218–227, 2007.
- [57] T. Vik, “Real-time visual simulation of smoke,” Master’s thesis, Norges teknisk-naturvitenskapelige universitet, 2004.
- [58] T. Vik, A. C. Elster, and T. Hallgren, “Real-time visualization of smoke through parallelizations,” *Advances in Parallel Computing*, vol. 13, pp. 371–378, 2004.
- [59] F. M. J. Vestre, “Enhancing and porting the hpc-lab snow simulator to opencl on mobile platforms,” Master’s thesis, Norges teknisk-naturvitenskapelige universitet, 2012.
- [60] K. Babington, “Terrain rendering techniques for the hpc-lab snow simulator,” Master’s thesis, Norges teknisk-naturvitenskapelige universitet, 2012.
- [61] A. Nordahl, “Enhancing the hpc-lab snow simulator with more realistic terrains and other interactive features,” Master’s thesis, Norges teknisk-naturvitenskapelige universitet, 2013.
- [62] M. A. Mikalsen, “Openacc-based snow simulation,” Master’s thesis, Norges teknisk-naturvitenskapelige universitet, 2013.
- [63] Ø. E. Krog, “Gpu-based real-time snow avalanche simulations,” Master’s thesis, Norges teknisk-naturvitenskapelige universitet, 2010.
- [64] Ø. E. Krog and A. C. Elster, “Fast gpu-based fluid simulations using sph,” in *International Workshop on Applied Parallel Computing*, pp. 98–109, Springer Berlin Heidelberg, 2010.
- [65] O. L. Boge, “Avalanche simulations using fracture mechanics on the gpu,” Master’s thesis, Norges teknisk-naturvitenskapelige universitet, 2014.
- [66] I. Kerr. Personal communication, 2015.
-

-
- [67] I. Quilez, “Noise - value - 3d,” 2013. Interactive demo, <https://www.shadertoy.com/view/4sfGzS> (last accessed: 2016-06-20).
- [68] K. Kim, M. Kim, D. Kim, and W. W. Ro, “True motion compensation with feature detection for frame rate up-conversion,” in *Image Processing (ICIP), 2015 IEEE International Conference on*, pp. 2260–2264, Sept 2015.
- [69] I. E. Halsauet, “Snow simulation with terrain interactions,” Master’s thesis, Norges teknisk-naturvitenskapelige universitet, 2016, expected. Title is tentative. Advisor: Anne C. Elster.
- [70] C. Everitt, G. Sellers, J. McDonald, and T. Foley, “Approaching zero driver overhead,” in *Game Developers Conference*, 2014.
- [71] J.-K. Chang and S.-T. Ryoo, “Real-time rendering of snow accumulation and melt under wind and light,” 2015.
- [72] J. Balme, E. Brown-Dymkoski, V. Guerrero, S. Jones, A. Kessler, A. Lichtl, K. Lung, W. Moses, K. Museth, N. Roberson, *et al.*, “Extreme multi-resolution visualization: A challenge on many levels,” 2015.
- [73] D. G. Spampinato, A. C. Elster, and T. Natvig, “Modelling multi-gpu systems.,” in *Parallel Computing: From Multicores and GPU’s to Petascale*, vol. 19, pp. 562–569, IO Press, 2010.
- [74] H. Ludvigsen and A. C. Elster, “Real-time ray tracing using nvidia optix,” *Eurographics Short Papers*, pp. 65–68, 2010.

Procedural sphere generation

This appendix chapter includes the source code for procedural mesh generation of spheres by repeated subdivision of a tetrahedron's faces into 4 equilateral triangles that are extruded to the sphere's radius.

```
//Constructs the vertices of a sphere of given radius with given vertex count
void Wind::GenerateUploadSphere(float **verts_out ,
                                unsigned short **indices_out ,
                                unsigned int *vert_count ,
                                unsigned int *face_count ,
                                unsigned int recursion_level ,
                                float radius) {
    unsigned int faces = 4, vertices = 4;
    for(unsigned int i = 1; i <= recursion_level; ++i) {
        vertices += faces * 3;
        faces *= 4;
    }

    *vert_count = vertices;
    *face_count = faces;
    float *verts = (float*)malloc( sizeof(float) * 3 * vertices );
    unsigned short *indices = ( unsigned short* )malloc(
        sizeof(unsigned short) * 3 * faces );

    // Construct tetrahedron
    float vdist = radius / sqrtf( 3.0 );
    float *tverts = &verts[vertices * 3 - 4 * 3];
    unsigned short *tindices = &indices[faces * 3 - 4 * 3];
    tverts[0] = vdist; tverts[1] = vdist; tverts[2] = vdist;
    tverts[3] = vdist; tverts[4] = -vdist; tverts[5] = -vdist;
    tverts[6] = -vdist; tverts[7] = vdist; tverts[8] = -vdist;
    tverts[9] = -vdist; tverts[10] = -vdist; tverts[11] = vdist;
```

```

tindices[0] = 0; tindices[1] = 1; tindices[2] = 2;
tindices[3] = 0; tindices[4] = 1; tindices[5] = 3;
tindices[6] = 0; tindices[7] = 2; tindices[8] = 3;
tindices[9] = 1; tindices[10] = 2; tindices[11] = 3;

// Refine the sphere by dividing each face into 4 equilateral triangles
// and extruding vertices to radius
unsigned int level_vert_count = 4, level_faces = 4;
for(unsigned int i = 1; i <= recursion_level; ++i) {
    unsigned int last_vert_count = level_vert_count;
    level_vert_count += level_faces * 3;
    level_faces *= 4;
    float *nverts = &verts[vertices * 3 - level_vert_count * 3];
    unsigned short *nindices = &indices[faces * 3 - level_faces * 3];
    for(unsigned int idx = 0, vidx = 0, iidx = 0;
        idx < (level_faces / 4) * 3;
        idx += 3) {
        unsigned short idx0 = tindices[idx    ];
        unsigned short idx1 = tindices[idx + 1];
        unsigned short idx2 = tindices[idx + 2];

        unsigned int idx3 = vidx / 3;

        float midx = (tverts[idx0 * 3    ] + tverts[idx1 * 3    ]) / 2.f;
        float midy = (tverts[idx0 * 3 + 1] + tverts[idx1 * 3 + 1]) / 2.f;
        float midz = (tverts[idx0 * 3 + 2] + tverts[idx1 * 3 + 2]) / 2.f;
        float scale = radius / sqrtf( midx * midx + midy * midy + midz * midz );
        midx *= scale;
        midy *= scale;
        midz *= scale;
        nverts[vidx++] = midx;
        nverts[vidx++] = midy;
        nverts[vidx++] = midz;
        midx = (tverts[idx0 * 3    ] + tverts[idx2 * 3    ]) / 2.f;
        midy = (tverts[idx0 * 3 + 1] + tverts[idx2 * 3 + 1]) / 2.f;
        midz = (tverts[idx0 * 3 + 2] + tverts[idx2 * 3 + 2]) / 2.f;
        scale = radius / sqrtf( midx * midx + midy * midy + midz * midz );
        midx *= scale;
        midy *= scale;
        midz *= scale;
        nverts[vidx++] = midx;
        nverts[vidx++] = midy;
        nverts[vidx++] = midz;
        midx = (tverts[idx1 * 3    ] + tverts[idx2 * 3    ]) / 2.f;
        midy = (tverts[idx1 * 3 + 1] + tverts[idx2 * 3 + 1]) / 2.f;
        midz = (tverts[idx1 * 3 + 2] + tverts[idx2 * 3 + 2]) / 2.f;
        scale = radius / sqrtf( midx * midx + midy * midy + midz * midz );
        midx *= scale;
        midy *= scale;
        midz *= scale;

```

```
nverts[vidx++] = midx;
nverts[vidx++] = midy;
nverts[vidx++] = midz;

nindices[iidx++] = idx0 + (level_faces / 4) * 3;
nindices[iidx++] = idx3;
nindices[iidx++] = idx3 + 1;
nindices[iidx++] = idx1 + (level_faces / 4) * 3;
nindices[iidx++] = idx3 + 2;
nindices[iidx++] = idx3;
nindices[iidx++] = idx2 + (level_faces / 4) * 3;
nindices[iidx++] = idx3 + 1;
nindices[iidx++] = idx3 + 2;
nindices[iidx++] = idx3;
nindices[iidx++] = idx3 + 2;
nindices[iidx++] = idx3 + 1;
}
tverts = nverts;
tindices = nindices;
}

*verts_out = verts;
*indices_out = indices;
}
```

Appendix **B**

WindSource neighborhood evaluation

This appendix chapter contains the source code for the novel neighborhood evaluation approach for wind sources.

```
// Potential contains all wind sources, count is the number of wind sources
// in potential that are used. Exclude is a bitmask that identifies which
// neighbors to exclude from consideration, usually only ourselves.
uint32_t WindSource::FindNeighbors(WindSource potential[32],
                                   uint32_t count,
                                   uint32_t exclude)
{
    uint32_t neighbors = 0;

    for(uint32_t i = 0; i < count; ++i) {
        if(exclude & (1<<i)) continue;
        // Calculate a line between this and it
        // We define p0 to be this point
        float d[3] = {
            potential[i].pos[0] - this->pos[0],
            potential[i].pos[1] - this->pos[1],
            potential[i].pos[2] - this->pos[2]
        };
        float dist = std::sqrt(d[0]*d[0] + d[1]*d[1] + d[2]*d[2]);
        if( dist == 0.f || dist == -0.f ) {
            printf("Two sources are in identical positions."
                  "Ignoring their neighbor-relations.");
            continue;
        }
        bool closer = false;
```

```

for(uint32_t j = 0; j < count; ++j) {
    if(j == i) continue;
    if(exclude & (1<<j)) continue;
    // Calculate the distance from this source to the line from above
    float p[3] = {
        potential[j].pos[0] - this->pos[0],
        potential[j].pos[1] - this->pos[1],
        potential[j].pos[2] - this->pos[2]
    };
    float dotPD = d[0]*p[0] + d[1]*p[1] + d[2]*p[2];
    float t = dotPD / ( dist * dist );
    // If we're outside the line segment we must be further away than
    // either end point, so we're not a neighbor
    if(t < 0.0 || t > 1.0) continue;

    // Find the closest point
    float pt[3] = {
        this->pos[0] + d[0] * t,
        this->pos[1] + d[1] * t,
        this->pos[2] + d[2] * t
    };
    float distPT = std::sqrt((pt[0] - potential[j].pos[0]) *
                               (pt[0] - potential[j].pos[0]) +
                               (pt[1] - potential[j].pos[1]) *
                               (pt[1] - potential[j].pos[1]) +
                               (pt[2] - potential[j].pos[2]) *
                               (pt[2] - potential[j].pos[2]));
    // If potential[j] is closer to the line from this to potential[i]
    // than that point is from either this or potential[i], then
    // potential[i] is not a neighbor, since it is hidden by
    // potential[j]'s influence.
    if(distPT < std::min( t, 1.f - t ) * dist) {
        closer = true;
        break;
    }
}
// If none of the other points obscure it, this point is a neighbor of
// the point this method is called on.
if(!closer) {
    neighbors |= 1 << i;
}
}
return neighbors;
}

```

Wind interpolation advection kernels

This appendix chapter includes the various versions of the `wind_advect` kernel in the wind simulation. Only the CUDA version is included, as the OpenCL versions are equivalent. Note that only the code in the `else` clauses are original to this thesis, except the constant version (Section C.2) which is entirely prior work (no code by the author).

C.1 Branching version

```

__global__ void wind_advect(cudaTextureObject_t wind_vel_tex ,
                           cudaPitchedPtr u,
                           cudaPitchedPtr p,
                           cudaPitchedPtr obs ,
                           const dim3 dim,
                           const float dt,
                           float4 boundary,
                           uint source_count,
                           float *sources) {
    INIT(u); // calculate x, y, z coordinates
    float4 *v = (float4 *)shared + x; // use shared memory as temp storage
    int mask = REF(int, obs, x, y, z); // obstacle mask
    *v = REF(float4, u, x, y, z);
    volatile int zero = 0;
    tiny
    if(x > zero && x < dim.x-1 &&
        y > zero && y < dim.y-1 &&
        z > zero && z < dim.z-1) {
        // self-advect backwards in time

```

```

v->x = (float)x - dt * v->x;
v->y = (float)y - dt * v->y;
v->z = (float)z - dt * v->z;

*v = wind_vel_sample(wind_vel_tex, *v);

// check for obstacle
if(mask & (VOX_SELF | VOX_LEFT | VOX_RIGHT)) v->x = zero;
if(mask & (VOX_SELF | VOX_ABOVE | VOX_BELOW)) v->y = zero;
if(mask & (VOX_SELF | VOX_UP | VOX_DOWN)) v->z = zero;
} else {
uint source_count_pure = source_count & 0x7fffffff;
// We use shared memory as local storage here; it's not actually shared!
float wind_source_distances[32];
// Calculate distance to sources
for(uint i = 0; i < source_count_pure; ++i) {
float dx = sources[i*8] - (float)x;
float dy = sources[i*8+1] - (float)y;
float dz = sources[i*8+2] - (float)z;
wind_source_distances[i] = sqrt(dx * dx + dy * dy + dz * dz);
}
// Find the closest source
uint us = 0;
float mindist = 3.40282347E+38F; // FLT_MAX
for(uint i = 0; i < source_count_pure; ++i) {
if( wind_source_distances[i] < mindist ) {
mindist = wind_source_distances[i];
us = i;
}
}
if( source_count & 0x80000000 ) {
// Nearest neighbor only
v->x = sources[us*8+4];
v->y = sources[us*8+5];
v->z = sources[us*8+6];
} else {
// Calculate total distance to all neighbors and closest
uint32_t *neighbors = (uint32_t*)&sources[us*8+3];
float dist = mindist;
for(uint i = 0; i < source_count_pure; ++i) {
if( *neighbors & ( 1 << i ) ) {
dist += wind_source_distances[i];
}
}
// Calculate average wind
float3 vel;
vel.x = 0;
vel.y = 0;
vel.z = 0;
for(uint i = 0; i < source_count_pure; ++i) {

```

```

        if( *neighbors & ( 1 << i ) || i == us ) {
            float f = wind_source_distances[i] / dist;
            vel.x += sources[i*8+4] * f;
            vel.y += sources[i*8+5] * f;
            vel.z += sources[i*8+6] * f;
        }
    }
    v->x = vel.x;
    v->y = vel.y;
    v->z = vel.z;
}
} else {
    *v = boundary;
}
}
REF(float4 , u, x, y, z) = *v;
// also set pressure field to 0
REF(float , p, x, y, z) = 0.0f;
}

```

C.2 Constant only version

```

__global__ void wind_advect(cudaTextureObject_t wind_vel_tex ,
                            cudaPitchedPtr u,
                            cudaPitchedPtr p,
                            cudaPitchedPtr obs ,
                            const dim3 dim,
                            const float dt,
                            float4 boundary,
                            uint source_count ,
                            float *sources) {
    INIT(u); // calculate x, y, z coordinates
    float4 *v = (float4 *)shared + x; // use shared memory as temp storage
    int mask = REF(int , obs, x, y, z); // obstacle mask
    *v = REF(float4 , u, x, y, z);
    volatile int zero = 0;

    if(x > zero && x < dim.x-1 &&
        y > zero && y < dim.y-1 &&
        z > zero && z < dim.z-1) {
        // self-advect backwards in time
        v->x = (float)x - dt * v->x;
        v->y = (float)y - dt * v->y;
        v->z = (float)z - dt * v->z;

        *v = wind_vel_sample(wind_vel_tex , *v);

        // check for obstacle
        if(mask & (VOX_SELF | VOX_LEFT | VOX_RIGHT)) v->x = zero;
    }
}

```

```

    if(mask & (VOX_SELF | VOX_ABOVE | VOX_BELOW)) v->y = zero;
    if(mask & (VOX_SELF | VOX_UP | VOX_DOWN)) v->z = zero;
} else {
    *v = boundary;
}
REF(float4 , u, x, y, z) = *v;
// also set pressure field to 0
REF(float , p, x, y, z) = 0.0f;
}

```

C.3 Nearest-Neighbor only version

```

__global__ void wind_advect(cudaTextureObject_t wind_vel_tex ,
                           cudaPitchedPtr u,
                           cudaPitchedPtr p,
                           cudaPitchedPtr obs ,
                           const dim3 dim,
                           const float dt,
                           float4 boundary ,
                           uint source_count ,
                           float *sources) {
    INIT(u); // calculate x, y, z coordinates
    float4 *v = (float4 *)shared + x; // use shared memory as temp storage
    int mask = REF(int , obs, x, y, z); // obstacle mask
    *v = REF(float4 , u, x, y, z);
    volatile int zero = 0;

    if(x > zero && x < dim.x-1 &&
        y > zero && y < dim.y-1 &&
        z > zero && z < dim.z-1) {
        // self-advect backwards in time
        v->x = (float)x - dt * v->x;
        v->y = (float)y - dt * v->y;
        v->z = (float)z - dt * v->z;

        *v = wind_vel_sample(wind_vel_tex , *v);

        // check for obstacle
        if(mask & (VOX_SELF | VOX_LEFT | VOX_RIGHT)) v->x = zero;
        if(mask & (VOX_SELF | VOX_ABOVE | VOX_BELOW)) v->y = zero;
        if(mask & (VOX_SELF | VOX_UP | VOX_DOWN)) v->z = zero;
    } else {
        uint source_count_pure = source_count & 0x7fffffff;
        // We use shared memory as local storage here; it's not actually shared!
        float wind_source_distances[32];
        // Calculate distance to sources
        for(uint i = 0; i < source_count_pure; ++i) {
            float dx = sources[i*8 ] - (float)x;
            float dy = sources[i*8+1] - (float)y;

```

```

        float dz = sources[i*8+2] - (float)z;
        wind_source_distances[i] = sqrt(dx * dx + dy * dy + dz * dz);
    }
    // Find the closest source
    uint us = 0;
    float mindist = 3.40282347E+38F; // FLT_MAX
    for(uint i = 0; i < source_count_pure; ++i) {
        if( wind_source_distances[i] < mindist ) {
            mindist = wind_source_distances[i];
            us = i;
        }
    }
    // Nearest neighbor only
    v->x = sources[us*8+4];
    v->y = sources[us*8+5];
    v->z = sources[us*8+6];
}
REF(float4 , u, x, y, z) = *v;
// also set pressure field to 0
REF(float , p, x, y, z) = 0.0f;
}

```

C.4 Interpolation only version

```

--global-- void wind_advect(cudaTextureObject_t wind_vel_tex ,
                           cudaPitchedPtr u,
                           cudaPitchedPtr p,
                           cudaPitchedPtr obs ,
                           const dim3 dim,
                           const float dt,
                           float4 boundary ,
                           uint source_count ,
                           float *sources) {
    INIT(u); // calculate x, y, z coordinates
    float4 *v = (float4 *)shared + x; // use shared memory as temp storage
    int mask = REF(int , obs , x, y, z); // obstacle mask
    *v = REF(float4 , u, x, y, z);
    volatile int zero = 0;

    if(x > zero && x < dim.x-1 &&
        y > zero && y < dim.y-1 &&
        z > zero && z < dim.z-1) {
        // self-advect backwards in time
        v->x = (float)x - dt * v->x;
        v->y = (float)y - dt * v->y;
        v->z = (float)z - dt * v->z;

        *v = wind_vel_sample(wind_vel_tex , *v);
    }
}

```

```

// check for obstacle
if(mask & (VOX_SELF | VOX_LEFT | VOX_RIGHT)) v->x = zero;
if(mask & (VOX_SELF | VOX_ABOVE | VOX_BELOW)) v->y = zero;
if(mask & (VOX_SELF | VOX_UP | VOX_DOWN)) v->z = zero;
} else {
uint source_count_pure = source_count & 0x7fffffff;
// We use shared memory as local storage here; it's not actually shared!
float wind_source_distances[32];
// Calculate distance to sources
for(uint i = 0; i < source_count_pure; ++i) {
float dx = sources[i*8 ] - (float)x;
float dy = sources[i*8+1] - (float)y;
float dz = sources[i*8+2] - (float)z;
wind_source_distances[i] = sqrt(dx * dx + dy * dy + dz * dz);
}
// Find the closest source
uint us = 0;
float mindist = 3.40282347E+38F; // FLT_MAX
for(uint i = 0; i < source_count_pure; ++i) {
if( wind_source_distances[i] < mindist ) {
mindist = wind_source_distances[i];
us = i;
}
}
// Calculate total distance to all neighbors and closest
uint32_t *neighbors = (uint32_t*)&sources[us*8+3];
float dist = mindist;
for(uint i = 0; i < source_count_pure; ++i) {
if( *neighbors & ( 1 << i ) ) {
dist += wind_source_distances[i];
}
}
// Calculate average wind
float3 vel;
vel.x = 0;
vel.y = 0;
vel.z = 0;
for(uint i = 0; i < source_count_pure; ++i) {
if( *neighbors & ( 1 << i ) || i == us ) {
float f = wind_source_distances[i] / dist;
vel.x += sources[i*8+4] * f;
vel.y += sources[i*8+5] * f;
vel.z += sources[i*8+6] * f;
}
}
v->x = vel.x;
v->y = vel.y;
v->z = vel.z;
}
REF(float4 , u, x, y, z) = *v;

```

```
// also set pressure field to 0  
REF(float , p, x, y, z) = 0.0f;  
}
```

Appendix D

Snow redistribution

This appendix chapter contains the snow redistribution functions for top and side redistribution of snow particles. Prior versions of the simulator simple called re position directly; this code steps in in its place and takes the distribution probability map into account. Both the CUDA and OpenCL implementations are included; in the CUDA version, the code looks identical for all PRNG versions, but in the OpenCL version the function footprints differ. Only the `philox` version is included, as the other versions are identical with the exception of the type of the PRNG streams and calls to the `clRNG` library functions.

D.1 CUDA implementation

```
#ifndef DISTRIBUTION_REJECTION_ITERATIONS
#define DISTRIBUTION_REJECTION_ITERATIONS 12
#endif

__device__ void redistribute_top(float4 &pos) {
    curandStatePhilox4_32_10_t state;
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    curand_init((unsigned long long)clock() + tid, 0, 0, &state);
    int i = 0;
    float chance;

    do {
        reposition(pos);
        chance = tex2D(part_dist, pos.x + SCENE_X * 0.5,
                               pos.z + SCENE_X * 0.5);
    } while(curand_uniform(&state) > chance &&
            i++ < DISTRIBUTION_REJECTION_ITERATIONS);
    if(i == DISTRIBUTION_REJECTION_ITERATIONS) {
```

```

    // NOTE(schmid): We put it outside the volume on the negative
    // side (which marks it as inactive in the shaders), and far
    // enough that it is likely to be outside the frustum even if
    // the camera looks towards it and thus get culled early.
    pos.x = -10000.f;
}
}

__device__ void redistribute_side(float4 &pos, float4 &vel) {
    curandStatePhilox4_32_10_t state;
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    curand_init((unsigned long long)clock() + tid, 0, 0, &state);

    // Backwards interpolate based on velocity to the cloud layer
    float h = SCENE_Y - 2.0 - pos.y;
    float t = h / vel.y;
    float4 p = pos;
    p.x -= vel.x * t;
    p.y -= vel.y * t;
    p.z -= vel.z * t;
    // Look up the distribution chance; we clamp to edge here, so if
    // we're outside the volume, we assume the distribution at the
    // edge fits. This looks better than redistribtuion it from the
    // top (where in high wind speeds we have issues with lack of snow
    // at the edges) and the distribution is undefined, but assuming
    // continuation from the edges is fairly reasonable.
    float chance = tex2D(part_dist, p.x + SCENE_X * 0.5,
                                   p.z + SCENE_X * 0.5);
    if( curand_uniform(&state) > chance ) {
        // Redistribute from top next iteration
        pos.x = -10000.0;
    }
}
}

```

D.2 OpenCL implementation

The `philox` implementation is included, but the OpenCL version uses whichever generator is used in `reposition` in redistribution as well.

```

#ifndef DISTRIBUTION_REJECTION_ITERATIONS
#define DISTRIBUTION_REJECTION_ITERATIONS 12
#endif

#include <clrng/philox432.clh>
void reposition_top(float4 *pos,
                   int4 scene_dim,
                   __global clrngPhilox432HostStream *streams,
                   uint seed,
                   __read_only image2d_t part_dist) {

```

```

int gid = get_global_id(0);

clrngPhilox432Stream work_item_stream;
clrngPhilox432CopyOverStreamsFromGlobal(1, &work_item_stream,
                                          &streams[ gid ]);
work_item_stream.current.ctr.L.lsb += seed - gid;
int i = 0;
float chance;

do {
    float x = clrngPhilox432RandomU01(&work_item_stream);
    float z = clrngPhilox432RandomU01(&work_item_stream);
    pos->x = x * scene_dim.x;
    pos->z = z * scene_dim.z;
    pos->y = scene_dim.y - 2.0f;
    chance = read_imagef(part_dist, part_dist_sampler,
                        (float2)(pos->x + scene_dim.x * 0.5,
                                pos->z + scene_dim.z * 0.5)).x;
} while( clrngPhilox432RandomU01(&work_item_stream) > chance &&
        i++ < DISTRIBUTION_REJECTION_ITERATIONS );
if(i == DISTRIBUTION_REJECTION_ITERATIONS) {
    pos->x = -10000.f;
}
}

void reposition_side(float4 *pos,
                    float4 *vel,
                    int4 scene_dim,
                    __global clrngPhilox432HostStream *streams,
                    uint seed, __read_only image2d_t part_dist) {
    int gid = get_global_id(0);
    clrngPhilox432Stream work_item_stream;
    clrngPhilox432CopyOverStreamsFromGlobal(1, &work_item_stream,
                                          &streams[ gid ]);
    work_item_stream.current.ctr.L.lsb += seed - gid;

    float h = scene_dim.y - 2.0f - pos->y;
    float t = h / vel->y;
    float x = pos->x - vel->x * t;
    float y = pos->y - vel->y * t;
    float z = pos->z - vel->z * t;
    float chance = read_imagef(part_dist, part_dist_sampler,
                                (float2)(pos->x + scene_dim.x * 0.5,
                                        pos->z + scene_dim.z * 0.5 ) ).x;
    if( clrngPhilox432RandomU01(&work_item_stream) > chance ) {
        pos->x = -10000.f;
    }
}

```

Appendix E

Cloud shader

This appendix chapter includes the cloud rendering shader. It is invoked on a box centered at the top of the rendering domain, and performs a ray-march through itself from the closest intersection point or the camera to the point at which the ray leaves the box (or until the loop terminates). The function `noise` is not original code (as credited in the comment).

```
#ifdef _VERTEX_
layout (location = 0) in vec3 in_vert;

uniform mat4 mvp;

out vec3 world_pos;

void main()
{
    world_pos = in_vert; // Our vertices are already in world space
    gl_Position = mvp * vec4(in_vert, 1.0);
}
#endif

#ifdef _FRAGMENT_

in vec3 world_pos;

out vec4 colour;

uniform sampler2D distribution;
uniform sampler2D noise_tex;
uniform vec3 eye_pos;
uniform vec3 cube_min;
uniform vec3 cube_max;
```

```

uniform vec3 light_dir;
uniform vec3 offset;
#ifdef VISUALIZE_THRESHOLD
uniform float threshold;
#endif

// Simple noise function not used for simulation so speed is
// valued over quality.
// Function by Inigo Quilez, https://www.shadertoy.com/view/XslGRr
// License Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported L
float noise(in vec3 x) {
    vec3 p = floor(x);l
    vec3 f = fract(x);
    f = f * f * (3.0 - 2.0 * f);

    vec2 uv = (p.xy + vec2(37.0, 17.0) * p.z) + f.xy;
    vec2 rg = texture2D( noise_tex, (uv + 0.5)/256.0, -100.0).yx;
    return -1.0+2.0*mix(rg.x, rg.y, f.z);
}

// Fractal brownian motion, 5 octaves
float fbm5(in vec3 p) {
    float v;
    v = 0.50000 * noise(p); p *= 2.01;
    v += 0.25000 * noise(p); p *= 2.05;
    v += 0.12500 * noise(p); p *= 2.03;
    v += 0.06250 * noise(p); p *= 2.02;
    v += 0.03125 * noise(p);
    return clamp(v, 0.0, 1.0); // Make sure we're in [0,1] range
}

// Fractal brownian motion, 4 octaves
float fbm4(in vec3 p) {
    float v;
    v = 0.50000 * noise(p); p *= 2.01;
    v += 0.25000 * noise(p); p *= 2.05;
    v += 0.12500 * noise(p); p *= 2.03;
    v += 0.06250 * noise(p);
    return clamp(v, 0.0, 1.0); // Make sure we're in [0,1] range
}

// Fractal brownian motion, 3 octaves
float fbm3(in vec3 p) {
    float v;
    v = 0.50000 * noise(p); p *= 2.01;
    v += 0.25000 * noise(p); p *= 2.05;
    v += 0.12500 * noise(p);
    return clamp(v, 0.0, 1.0); // Make sure we're in [0,1] range
}

// Fractal brownian motion, 2 octaves
float fbm2(in vec3 p) {

```

```

float v;
v = 0.50000 * noise(p); p *= 2.01;
v += 0.25000 * noise(p);
return clamp(v, 0.0, 1.0); // Make sure we're in [0,1] range
}

// Integrate color given a density and a density difference one step towards
// the sun
void integrate( inout vec4 col, in float den, in float dif, in vec2 uv ) {
    vec4 c;
#ifdef VISUALIZE_COLOR
    float d = texture2D( distribution, uv ).r;
    c.r = 1.0 - d;
    c.g = 1.0 - d;
    c.b = mix( 0.8, 0.35, den ) * ( 1.05 + 0.3 * dif );
    c.a = den;
#else
    c = vec4( mix( vec3( 1.0, 0.95, 0.8 ),
                  vec3( 0.25, 0.3, 0.35 ), den ), den );
    c.rgb *= vec3( 0.65, 0.7, 0.75 ) * 1.4
              + vec3( 1.0, 0.6, 0.3 ) * dif;
#endif

    c.a *= 0.4;
    c.rgb *= c.a;
    col += c * ( 1.0 - col.a );
}

// Look up density at a given position with noise of a given level of quality
#ifdef VISUALIZE_HEIGHT
#define density(pos, FBM) \
    clamp( 0.7 - sqrt( abs(pos.y - 0.5) + 0.05 ) / \
           sqrt( texture2D( distribution, pos.xz ).r ) + \
           0.8 * FBM(pos * noise_scale + ofs), 0.0, 1.0 )
#else
#ifdef VISUALIZE_THRESHOLD
#define density(pos, FBM)\
    ( texture2D( distribution, pos.xz ).r >= threshold ? \
      ( 0.7 - sqrt( abs( pos.y - 0.5 ) + 0.05 ) / \
        sqrt( texture2D( distribution, pos.xz ).r ) ) : 0.0 )
#else
#define density(pos, FBM)\
    clamp( 0.7 - sqrt( abs(pos.y - 0.5) + 0.05 ) / \
           sqrt( 0.5 ) + 0.8 * FBM(pos * noise_scale + ofs), 0.0, 1.0 )
#endif
#endif

#define check_bounds(pos)\
    if( pos.x > cube_max.x || pos.x < cube_min.x || \
        pos.y > cube_max.y || pos.y < cube_min.y || \

```

```

        pos.z > cube_max.z || pos.z < cube_min.z ) break

#define premarch(STEPS, FBM)\
    for( uint i = 0; i < STEPS; ++i ) {\
        vec3 x = eye_dir * t + p;\
        check_bounds(x);\
        vec3 pr = (x - cube_min) * ics;\
        float den, dif;\
        den = density(pr, FBM);\
        if( den > 0.01 ) break;\
        t += 1.0;\
    }

#define march(STEPS, FBM, MINSTEP)\
    for( uint i = 0; i < STEPS; ++i ) {\
        if( col.a > 0.99 ) break;\
        vec3 x = eye_dir * t + p;\
        check_bounds(x);\
        vec3 pr = (x - cube_min) * ics;\
        float den, dif;\
        den = density(pr, FBM);\
        if( den > 0.01 ) {\
            vec3 sp = ((x + light_dir * 1.0) - cube_min) * ics;\
            dif = clamp((den - density(sp, FBM)) / 0.6, 0.0, 1.0);\
            integrate( col, den, dif, pr.xz );\
        }\
        float tmp = 1.0 - col.a;\
        t += max(MINSTEP, 0.1/tmp);\
    }

void main() {
    vec3 eye_dir = normalize(world_pos - eye_pos);
    vec3 p = world_pos;
    vec3 cs = cube_max - cube_min;
    vec3 ics = 1.0 / cs;
    float max_size = max(cs.x, max(cs.y, cs.z));
    vec3 csm = cs / max_size;
    vec3 noise_scale = 8.0 * csm;
    vec3 ofs = offset / noise_scale;

    // Test if eye is inside; if so, start at the eye,
    // not the intersection
    if( eye_pos.x > cube_min.x && eye_pos.x < cube_max.x &&
        eye_pos.y > cube_min.y && eye_pos.y < cube_max.y &&
        eye_pos.z > cube_min.z && eye_pos.z < cube_max.z ) {
        p = eye_pos;
    }

    vec4 col = vec4( 0.0 );
    float t = 0.01;

```

```
premark( 192, fbm5 );
t = max( 0.01, t - 1.0 );
float t0 = t;

march( 32, fbm5, 0.4 );
march( 24, fbm4, 0.4 );
march( 16, fbm3, 0.7 );
march( 128, fbm2, 1.0 );

colour = col;
}
#endif
```


Appendix **F**

Profiling system

This appendix chapter describes the automated profiling system introduced into the snow simulator. Section F.1 includes the Python profiling script that is used to re-compile and run the simulator, while Section F.2 includes the `TimingSystem` class that performs the timing internally. Section F.3 contains a header-only library written by the author independently of the thesis used to calculate statistics from timing collections and printing the data to console.

F.1 Automation script

Only two profiles are included for brevity; it should be possible to read from these how more are added.

```
#!/usr/bin/python
# Automatically profiles the snow simulator with all the permutations
# defined for the type of profile specified.
```

```
import sys, time
from subprocess import call, check_output

CONFIG_FILE = "../data/config.txt"
CONFIG_FILE_BACKUP = "../data/config_profile_backup.txt"
CUDA_BIN = "snow-cuda"
CL_BIN = "snow-cl"

COMPILE_OUTPUT_TEMPORARY = "profile_compile_out.tmp"
BINARY_STATUS = { CUDA_BIN: [], CL_BIN: [] }

FRAME_COUNT = 1000
```

PROFILE_TEST = 0x1

PROFILE_TEX = 0x2

help_text = """

Snow simulator profiler v0.1.0

Usage: python """ + sys.argv[0] + """ [options] [profile]

Available options:

-f [OUTPUT_FILE]

Redirects stdout to the given file

-i [ITERATIONS]

Sets the number of frames to profile over. Defaults to 1000.

-h

Prints this help

Available profiles:

Tex

Tests the performance of bindless textures compared to status quo.

test

Tests that the system works. Profiles the CUDA and CL versions with the current configuration file.

"""

```
def parse_args():
```

```
    # Parse arguments
```

```
    profiles = 0
```

```
    i = 1
```

```
    while i < len(sys.argv):
```

```
        if sys.argv[i][0] == '-':
```

```
            if sys.argv[i][1] == 'f':
```

```
                sys.stdout = open(sys.argv[i+1], 'w')
```

```
                i += 1
```

```
            elif sys.argv[i][1] == 'h':
```

```
                print( help_text )
```

```
            elif sys.argv[i][1] == 'i':
```

```
                global FRAME_COUNT
```

```
                FRAME_COUNT = int(sys.argv[i+1])
```

```
                i += 1
```

```
        else:
```

```
            elif sys.argv[i] == 'test':
```

```
                profiles |= PROFILE_TEST
```

```
            elif sys.argv[i] == 'Tex':
```

```
                profiles |= PROFILE_TEX
```

```
            else:
```

```
                print("Unknown option_" + str(sys.argv[i]) +  
                      " . See -h for help.")
```

```
    i += 1
```

```
    if profiles == 0:
```

```
        print("No profiles specified , will do nothing . See -h for help.")
```

```
    return profiles
```

```

# Recompile with optional additional defines.
# defines is an array of tuples of (NAME, VALUE), where value can be None.
def recompile(version, defines):
    print("Recompiling_" + str(version) + "_(additional_defines:_" +
          str(defines) + "_)")
    sys.stdout.flush()
    f = open(COMPILE_OUTPUT_TEMPORARY, 'w')
    call(["make", "clean"], stdout=f, stderr=f, cwd="..")
    defs = ""
    for (d, v) in defines:
        if v == None:
            defs += "-D" + str(d) + "_"
        else:
            defs += "-D" + str(d) + "=" + str(v) + "_"
    call(["rm", "CMakeCache.txt"], stdout=f, stderr=f, cwd="..")
    call('cmake_-DNVCC.DEFINES="' + defs[:len(defs)-1] + "_.',
         shell=True, stdout=f, stderr=f, cwd="..")
    if call('make_' + version + '_CXX.DEFINES="' + defs[:len(defs)-1] + "_.',
           shell=True, stdout=f, stderr=f, cwd="..") == 0:
        f.close()
        call(["rm", COMPILE_OUTPUT_TEMPORARY])
    else:
        f.close()
        f = open(COMPILE_OUTPUT_TEMPORARY, 'r')
        print("Compilation_failed_._Output_follows_\\n")
        print(f.read())
        f.close()
        call(["rm", COMPILE_OUTPUT_TEMPORARY])
    sys.stdout.flush()

def setup():
    # Save away the old configuration file
    print("Saving_the_configuration_file")
    sys.stdout.flush()
    call(["cp", CONFIG_FILE, CONFIG_FILE_BACKUP], stdout=sys.stdout)
    # Mark that the binaries need to be recompiled
    global BINARY_STATUS
    BINARY_STATUS[CUDA_BIN] = []
    BINARY_STATUS[CL_BIN] = []
    # Print PC info
    print("Gathering_PC_information")
    print("OS:")
    sys.stdout.flush()
    call(["uname", "-a"], stdout=sys.stdout)
    print("\n_____")
    print("CPU:")
    sys.stdout.flush()
    call(["lscpu"], stdout=sys.stdout)
    print("\n_____")
    print("GPU:")

```

```

sys.stdout.flush()
lspci = check_output(["lspci"])
for l in lspci.split('\n'):
    if l.find("VGA") != -1:
        if l.find("NVIDIA") != -1:
            call(["nvidia-smi"], stdout=sys.stdout)
        else:
            call(["lspci", "-v", "-s", l.split('_')[0]], stdout=sys.stdout)
    print("\n")
# Print original config file
print("\n-----")
print("Original_configuration_file_(with_no_permutations_applied):\n")
f = open(CONFIG_FILE, 'r')
conf = f.read()
f.close()
print(conf)
print("\n-----")
sys.stdout.flush()

def post():
    # Restore the oldconfig file
    print("Restoring_the_configuration_file")
    sys.stdout.flush()
    call(["cp", CONFIG_FILE.BACKUP, CONFIG_FILE], stdout=sys.stdout)
    # Recompile vanilla version
    if len(BINARY_STATUS[CUDA_BIN]) != 0:
        recompile(CUDA_BIN, [])
    if len(BINARY_STATUS[CL_BIN]) != 0:
        recompile(CL_BIN, [])

def run_profile( name, perms ):
    print("Running_" + name + "_profiling_permutations:\n")
    sys.stdout.flush()
    start = time.time()
    f = open(CONFIG_FILE, 'r')
    conf = f.read()
    orig_conf = conf
    f.close()
    i = 0
    for (b, d, p, c) in perms:
        # Recompile if it has custom defines or the binary
        # does not have the defines we want
        defines = d + [("AUTOMATED_TEST_VERSION", None),
                       ("TIMING_FRAME_COUNT", FRAME_COUNT)]
        for di in defines:
            if di not in BINARY_STATUS[b]: # New directive, recompile!
                recompile(b, defines)
                BINARY_STATUS[b] = defines
            break
    # Alter the config file

```

```

for key, value in p.iteritems():
    pos = conf.find(key)
    if pos == -1:
        conf += "\n" + key + "_" + str(value) + "\n"
    else:
        pos += len(key)
        end = pos
        while conf[end] != '\n':
            end += 1
        conf = conf[:pos] + "_" + str(value) + conf[end:]
        # Uncomment it
        pos -= len(key)
        if conf[pos-1] == '#':
            conf = conf[:pos-1]+conf[pos:]
# Write the config file
f = open(CONFIG_FILE, 'w')
f.write(conf)
f.close()
# Run the simulator
print("Permuatation_" + str(i) + ":_:" + str(p) +
      ( str(d) if len(d) != 0 else "" ) + "\n")
print("Version:_:" + str(b) + "\t")
sys.stdout.flush()
call(["./" + b], cwd="..", stdout=sys.stdout)
for cmd in c:
    call(cmd, shell=True, cwd="..", stdout=sys.stdout)
print("\n\n")
# Reset to original config for next iteration
conf = orig_conf
i += 1
print("Finished_" + name + "_profiling_in_" +
      str(time.time() - start) + "_seconds\n\n")
sys.stdout.flush()

# Permutations. These are tuples of
#   Binary to test,
#   list of preprocessor directives (whcih force a recompile if not empty)
#   dictionary of configuration values to set
#   postprocessing commands
test_perms = [ ( CUDA_BIN, [], { }, [] ),
               ( CL_BIN,  [], { }, [] ) ] # Simple test of the system
tex_perms = [ ( CUDA_BIN, [
                { 'time_kernel_snow_update': 1,
                  'time_kernel_wind_advect': 1 } ],
               [] ),
               ( CUDA_BIN, [ ("LEGACY_TEXTURES", None) ],
                { 'time_kernel_snow_update': 1,
                  'time_kernel_wind_advect': 1 } ],
               [] )
]

```

```

# Main entry point
profiles = parse_args()
if profiles != 0:
    setup()
    if profiles & PROFILE_TEST: run_profile("Tool_Test", test_perms)
    if profiles & PROFILE_TEX: run_profile("Bindless_Texture", tex_perms)
    post()

```

F.2 TimingSystem class

Listing F.1: Header

```

#ifndef TIMING_SYSTEM_H
#define TIMING_SYSTEM_H

#include <stdlib.h>
#include <math.h>
#include <stdio.h>
#include <stdint.h>
#include <stdexcept>

enum TimingEventType {
    TIMING_FULL_FRAME,

    TIMING_KERNEL_SNOW_PARTICLE_UPDATE,
    TIMING_KERNEL_SNOW_SMOOTH_GROUND,

    TIMING_KERNEL_WIND_ADVECT,
    TIMING_KERNEL_WIND_BUILD_SOLUTION,
    TIMING_KERNEL_WIND_POISSON,
    TIMING_KERNEL_WIND_BOUNDARY,
    TIMING_KERNEL_WIND_PROJECT,

    // NOTE(schmid): This contains the swap, so if vsync is enabled,
    // it is inaccurate (contains the v-blank wait)
    TIMING_RENDER_FULL,
    // NOTE(schmid): These are not guaranteed to contain a flush,
    // so it is CPU side timings. GPU timings require a gpu profiler
    // or explicit flush after each of these.
    TIMING_RENDER_SHADOWS,
    TIMING_RENDER_TERRAIN,
    TIMING_RENDER_SKYBOX,
    TIMING_RENDER_SNOW,
    TIMING_RENDER_CLOUDS,

    TIMING_EVENT_TYPE_Count // Must be the last argument!

```

```

};

class TimingSystem {
public:
    TimingSystem( uint64_t frames );
    ~TimingSystem ();

    void SetElapsedTime( TimingEventType type , uint64_t elapsed );
    void CalculateSetElapsedTime( TimingEventType type ,
                                  struct timespec start ,
                                  struct timespec end );
    void PrintAllStats( uint32_t histogram_bucket_count = 20 );
    void PrintStats( TimingEventType type ,
                    uint32_t histogram_bucket_count = 20 );
    void Reset( TimingEventType type );
    void ResetAll ();

    void Start ();
    void Stop ();
    bool Running ();

    static uint64_t CalculateElapsedTime( struct timespec start ,
                                          struct timespec end );

private:
    uint64_t *mTimeArrays[ TIMING_EVENT_TYPE_Count ];
    uint64_t mTimeArrayTops[ TIMING_EVENT_TYPE_Count ];
    uint64_t mFrameCount;

    bool mRunning;

    static const char *TimingEventNames[ TIMING_EVENT_TYPE_Count ];
};
#endif // TIMING_SYSTEM_H

```

Listing F.2: Source

```

#include "TimingSystem.h"
#define VUL_DEFINE
#include "vul_benchmark.h"

const char *TimingSystem::TimingEventNames[ TIMING_EVENT_TYPE_Count ] = {
    "Full frame",
    "Snow particle update kernel",
    "Snow ground smoothing kernel",
    "Wind advection kernel",
    "Pressure solution build kernel",
    "Poisson kernel total",
    "Boundary reset kernel total",
    "Wind projection kernel",

```

```

    "Full render",
    "Shadow pass",
    "Terrain pass",
    "Skybox pass",
    "Snow pass",
    "Cloud pass"
};

TimingSystem::TimingSystem( uint64_t frames )
{
    mRunning = false;
    mFrameCount = frames;
    for( uint32_t i = 0; i < TIMING_EVENT_TYPE_Count; ++i ) {
        mTimeArrays[ i ] = NULL;
        mTimeArrayTops[ i ] = 0;
    }
}

TimingSystem::~TimingSystem()
{
    for( uint32_t i = 0; i < TIMING_EVENT_TYPE_Count; ++i ) {
        if( mTimeArrays[ i ] != NULL ) {
            delete [] mTimeArrays[ i ];
        }
    }
}

void TimingSystem::SetElapsedTime( TimingEventType type, uint64_t elapsed )
{
    if( !mRunning ) {
        return;
    }
    if( mTimeArrays[ type ] == NULL ) {
        mTimeArrays[ type ] = new uint64_t[ mFrameCount + 1 ];
    }
    if( mTimeArrayTops[ type ] > mFrameCount ) {
        printf( "Types %d frame %d\n", (int)type, (int)mTimeArrayTops[ type ] );
        throw std::runtime_error( "Timing array is out of room" );
    }
    mTimeArrays[ type ][ mTimeArrayTops[ type ] ++ ] = elapsed;
}

uint64_t TimingSystem::CalculateElapsedTime( struct timespec start,
                                             struct timespec end )
{
    return ( ( end.tv_sec - start.tv_sec ) * 1000000 ) +
           ( ( end.tv_nsec - start.tv_nsec ) / 1000 );
}

void TimingSystem::CalculateSetElapsedTime( TimingEventType type,
                                           struct timespec start,

```

```

                                struct timespec end)
{
    if( !mRunning ) {
        return;
    }
    uint64_t elapsed = ( ( end.tv_sec - start.tv_sec ) * 1000000 )
                      + ( ( end.tv_nsec - start.tv_nsec ) / 1000 );
    this->SetElapsedTime(type , elapsed);
}

void TimingSystem::PrintAllStats(uint32_t histogram_bucket_count)
{
    for( uint32_t i = 0; i < TIMING_EVENT_TYPE_Count; ++i ) {
        if( mTimeArrays[i] != NULL ) {
            this->PrintStats((TimingEventType)i, histogram_bucket_count);
        }
    }
}

void TimingSystem::PrintStats(TimingEventType type ,
                              uint32_t histogram_bucket_count)
{
    if( mTimeArrays[type] == NULL ) {
        throw std::runtime_error("Attempted to print timing statistics "
                                "for untyped event type");
    }

    vul_benchmark_result res;
    res.mean = vul_benchmark_mean( mTimeArrays[type], 0,
                                  mTimeArrayTops[type] );
    res.median = vul_benchmark_median( mTimeArrays[type], 0,
                                       mTimeArrayTops[type] );
    res.std_deviation = vul_benchmark_standard_deviation(
        mTimeArrays[type], 0,
        mTimeArrayTops[type], res.mean );
    printf( "%s: %f mean (ms), %lu median (micros), %f std.dev. (ms)\n",
            TimingEventNames[type],
            res.mean / 1000.0,
            res.median,
            res.std_deviation / 1000.0 );
    if( histogram_bucket_count ) {
        vul_benchmark_print_histogram_micros( mTimeArrays[type], 0,
                                              mTimeArrayTops[type],
                                              histogram_bucket_count );
    }
}

void TimingSystem::Reset(TimingEventType type)
{
    mTimeArrayTops[type] = 0;
}

```

```

}

void TimingSystem::ResetAll()
{
    for( uint32_t i = 0; i < TIMING_EVENT_TYPE_Count; ++i ) {
        mTimeArrayTops[i] = 0;
    }
}

void TimingSystem::Start()
{
    mRunning = true;
}

void TimingSystem::Stop()
{
    mRunning = false;
}

bool TimingSystem::Running()
{
    return mRunning;
}

```

F.3 vul_benchmark

```

/*
 * Villains' Utility Library – Thomas Martin Schmid, 2016. Public domain?
 *
 * This file contains auxilliary functions for benchmarking.
 * @TODO: Proper statistics; at the moment this does the programmer's
 * hacky equivalent; specify iteration count and calculate mean,
 * median and standard deviation.
 * @TODO: Plotting (bar diagram of all iterations, histogram, smooth graf)
 * nanovg should be useful for this; hide the whole thing behind a define.
 *
 * ? If public domain is not legally valid in your legal jurisdiction
 * the MIT licence applies (see the LICENCE file)
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
 * NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
 * BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
 * ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
 * CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
 * THE SOFTWARE.
 */
#ifdef VUL_BENCHMARK_H
#define VUL_BENCHMARK_H

```

```

#include <stdarg.h>
#include <stdint.h>

// Define VUL_DEFINE in exactly ONE compilation unit (C/CPP file)
// that includes this file to define it.

typedef struct vul_benchmark_result {
    double mean;
    uint64_t median;
    double std_deviation;
} vul_benchmark_result;

typedef struct vul_benchmark_histogram {
    uint32_t *buckets, bucket_count, bucket_max;
    uint64_t smallest, largest;
} vul_benchmark_histogram;

#endif // VUL_BENCHMARK_H
/**
 * Helper function used to find the median. This is the median-of-median
 * algorithm, and all the median-function does is pretend the
 * k-argument doesn't exist.
 */
#ifdef VUL_DEFINE
uint32_t vul__benchmark_select( uint64_t *times, uint32_t left, u
                               int32_t right, uint32_t k );
#else
uint32_t vul__benchmark_select( uint64_t *times, uint32_t left,
                               uint32_t right, uint32_t k )
{
    uint64_t order[ 5 ], time;
    uint32_t i, l, count, median;
    int32_t j;

    count = right - left;
    if( count < 5 )
    {
        // Just find the k-th element the hard way
        l = 1;
        order[ 0 ] = times[ left ];
        for( i = 1; i < count + 1; ++i )
        {
            time = times[ left + i ];
            if( time < order[ l - 1 ] ) {
                for( j = l - 1; j >= 0 && time < order[ j ]; --j )
                {
                    order[ j + 1 ] = order[ j ];
                }
                order[ ++j ] = time;
                ++l;
            }
        }
    }
}

```

```

        } else {
            order[ l++ ] = time;
        }
    }
    for( i = 0; i < 5; ++i ) {
        if ( order[ k ] == times[ left + i ] ) {
            return left + i;
        }
    }
}

for( i = 0; i < ( count / 5 ) + 1; ++i )
{
    l = 5 * i; // Left bound of new array
    j = ( l + 4 ) > right ? right : ( l + 4 ); // Right bound of new array

    median = vul_benchmark_select( times , l , j , 2 ); // 0-indexed k

    time = times[ median ];
    times[ median ] = times[ i ];
    times[ i ] = time;
}

return vul_benchmark_select( times , 0 , count / 5 , count / 10 );
}
#endif // VUL_DEFINE

/**
 * Finds the median element of an array of times. O(n)
 */
#ifndef VUL_DEFINE
uint64_t vul_benchmark_median( uint64_t *times , uint32_t left ,
                               uint32_t right );
#else
uint64_t vul_benchmark_median( uint64_t *times , uint32_t left ,
                               uint32_t right )
{
    return times[ vul_benchmark_select( times , left , right ,
                                        ( right - left ) / 2 ) ];
}
#endif // VUL_DEFINE

/**
 * Calculates the mean of an array of times. O(n)
 */
#ifndef VUL_DEFINE
double vul_benchmark_mean( uint64_t *times , uint32_t left ,
                           uint32_t right );
#else
double vul_benchmark_mean( uint64_t *times , uint32_t left ,

```

```

                                uint32_t right )
{
    uint32_t i;
    double avg, count;

    count = ( double )( right - left );
    avg = 0;
    for( i = left; i < right; ++i )
    {
        avg += ( double )times[ i ] / count;
    }

    return avg;
}
#endif // VUL_DEFINE

/**
 * Calculates the stad_deviation of an array of times given the
 * mean of the array. O(n)
 */
#ifndef VUL_DEFINE
double vul_benchmark_standard_deviation( uint64_t *times ,
                                         uint32_t left ,
                                         uint32_t right ,
                                         double mean );
#else
double vul_benchmark_standard_deviation( uint64_t *times ,
                                         uint32_t left ,
                                         uint32_t right ,
                                         double mean )
{
    uint32_t i;
    double d, dev, inv_count;

    inv_count = 1.0 / ( double )( ( right - left ) - 1 );
    dev = 0;
    for( i = left; i < right; ++i )
    {
        d = ( double )times[ i ] - mean;
        d *= d;
        dev += d * inv_count;
    }

    return sqrt( dev );
}
#endif // VUL_DEFINE

#ifndef VUL_DEFINE
void vul__benchmark_create_histogram( vul_benchmark_histogram *hist ,
                                     uint64_t *times , uint32_t left ,

```

```

uint32_t right, uint32_t buckets );
#else
void vul__benchmark_create_histogram( vul_benchmark_histogram *hist,
uint64_t *times, uint32_t left,
uint32_t right, uint32_t buckets )
{
    double s, l, r, t;
    uint32_t i, j;
    hist->bucket_count = buckets;

    // Find range
    hist->smallest = -1;
    hist->largest = 0;
    for( i = left; i < right; ++i ) {
        hist->smallest = times[ i ] < hist->smallest ?
            times[ i ] : hist->smallest;
        hist->largest = times[ i ] > hist->largest ?
            times[ i ] : hist->largest;
    }
    s = ( double )( hist->largest - hist->smallest ) / ( double )buckets;
    if( hist->largest - hist->smallest < ( uint64_t )buckets ) {
        uint32_t hb = buckets / 2;
        double med = ( double )hist->smallest + ( s * ( double )hb );
        s = 1.0;
        hist->smallest = med - s * ( double )hb;
        hist->largest = med + s * ( double )hb;
    }

    // Fill buckets
    hist->buckets = ( uint32_t* )malloc( sizeof( uint32_t ) * buckets );
    l = ( double )hist->smallest;
    r = ( double )hist->smallest + s;
    for( i = 0; i < buckets; ++i ) {
        hist->buckets[ i ] = 0;
        for( j = left; j < right; ++j ) {
            t = ( double )times[ j ];
            if( t >= l && ( t < r || ( i == buckets - 1 &&
                t == ( double )hist->largest ) ) ) {
                ++hist->buckets[ i ];
            }
        }
        l = r;
        r += s;
    }

    // Find largest bucket
    hist->bucket_max = 0;
    for( i = 0; i < buckets; ++i ) {
        hist->bucket_max = hist->buckets[ i ] > hist->bucket_max ?
            hist->buckets[ i ] : hist->bucket_max;
    }
}

```

```

    }
}
#endif // VUL_DEFINE

/**
 * Print a histogram to stdout with a given number of buckets.
 */
#ifdef VUL_DEFINE
void vul_benchmark_print_histogram_millis( uint64_t *times ,
                                           uint32_t left ,
                                           uint32_t right ,
                                           uint32_t buckets );
#else
void vul_benchmark_print_histogram_millis( uint64_t *times ,
                                           uint32_t left ,
                                           uint32_t right ,
                                           uint32_t buckets )
{
    uint32_t i, j, ml;
    uint64_t v, s;
    double l, r, ds;
    vul_benchmark_histogram hist;

    // Calculate it
    vul_benchmark_create_histogram( &hist , times , left , right , buckets );

    // Print legend
    printf( "Time (ms) | Count |0" );
    ml = ( uint32_t )log10( ( double )hist.bucket_max );
    for( i = 1; i < buckets - ml; ++i ) printf( " " );
    printf( "%d|\n" , hist.bucket_max );
    printf( "-----|-----" );
    for( i = 0; i < buckets; ++i ) printf( "-");
    printf( "|\n" );

    s = hist.bucket_max / buckets;
    ds = ( double )( hist.largest - hist.smallest ) / ( double )buckets;
    l = ( double )hist.smallest;
    r = l + ds;
    for( i = 0; i < buckets; ++i ) {
        v = 0;
        printf( "%02.1f-%02.1f | %d" , l, r, hist.buckets[ i ] );
        ml = ( uint32_t )log10( ( double )hist.buckets[ i ] );
        for( j = 0; j < 5 - ml; ++j ) printf( " " );
        printf( "|\n" );
        l = r;
        r += ds;
        for( j = 0; j < buckets && hist.buckets[ i ] > v; ++j ) {
            printf( "*" );
            v += s;
        }
    }
}

```

```

    }
    for( ; j < buckets; ++j ) printf(" ");
    printf("\n");
}
printf("\n");

// Clean up
free( hist.buckets );
}
#endif // VUL_DEFINE

/**
 * Print a histogram to stdout with a given number of buckets.
 */
#ifndef VUL_DEFINE
void vul_benchmark_print_histogram_micros( uint64_t *times, uint32_t left,
                                           uint32_t right, uint32_t buckets );
#else
void vul_benchmark_print_histogram_micros( uint64_t *times, uint32_t left,
                                           uint32_t right, uint32_t buckets )
{
    uint32_t i, j, ml;
    uint64_t v, s;
    double l, r, ds;
    vul_benchmark_histogram hist;

    // Calculate it
    vul__benchmark_create_histogram( &hist, times, left, right, buckets );

    // Print legend
    printf( "Time (ms) | Count |0" );
    ml = ( uint32_t )log10( ( double )hist.bucket_max ) + 1;
    for( i = 1; i < buckets - ml; ++i ) printf(" ");
    printf( "%d|\n", hist.bucket_max );
    printf( "-----|-----|" );
    for( i = 0; i < buckets; ++i ) printf("-");
    printf("\n");

    s = hist.bucket_max / buckets;
    ds = ( double )( hist.largest - hist.smallest ) / ( double )buckets;
    l = ( double )hist.smallest;
    r = l + ds;
    for( i = 0; i < buckets; ++i ) {
        v = 0;
        printf( "%02.2f-%02.2f | %d", l / 1000.0, r / 1000.0, hist.buckets[ i ] );
        ml = ( uint32_t )log10( ( double )hist.buckets[ i ] );
        for( j = 0; j < 5 - ml; ++j ) printf(" ");
        printf( "|");
        l = r;
        r += ds;
    }
}

```

```
    for( j = 0; j < buckets && hist.buckets[ i ] > v; ++j ) {
        printf(" *");
        v += s;
    }
    for( ; j < buckets; ++j ) printf(" ");
    printf("|\\n");
}
printf("\\n");

// Clean up
free( hist.buckets );
}
#endif // VUL_DEFINE
```


Appendix **G**

Terrain pre-processing

This appendix chapter includes the pre-processing script in Python used to create RAW 16-bit height-maps from USGS DEM data. The user interface is loosely based on the work of Lien [12].

```
#!/usr/bin/python

import argparse, sys, struct, itertools, numpy, math

def read_header(lines):
    '''USGS DEM (.asc) header parser. Header contains the number of
        columns and rows, the cellsize in meters that an entry describes
        and the value used to mark an entry for which no data is
        available (a "nodata" entry).'''
    ncols_line = lines[0].split(' ')
    ncols = ncols_line[len(ncols_line)-1]
    nrows_line = lines[1].split(' ')
    nrows = nrows_line[len(nrows_line)-1]
    csize_line = lines[4].split(' ')
    csize = csize_line[len(csize_line)-1]
    ndata_line = lines[5].split(' ')
    ndata = ndata_line[len(ndata_line)-1]
    return { 'ncols': int(ncols),
            'nrows': int(nrows),
            'cellsize': float(csize),
            'nodata': float(ndata) }

def read_data(header, lines):
    '''Read the height data for all remaining lines (all lines except
        the header), while maintaining a running minimum and maximum
        height record for used in scaling. Writes and error if number of
        rows or columns does not match the header information given.'''
```

```

data = []
mn = 1e20
mx = -1e20
y = 0
for l in lines:
    if len(l) == 0:
        continue
    y += 1
    row = []
    line = l.split(' ')
    x = 0
    for w in line:
        if w == '':
            continue
        v = float(w)
        x += 1
        if v != header['nodata']:
            mn = min(mn, v)
            mx = max(mx, v)
    row.append(v)
    if x != header['ncols']:
        sys.stderr.write("Row contained wrong number of valid entries. " +
            "Found " + str(x) + " instead of " +
            str(header['ncols']) + "\n")
    data.append(row)
if y != header['nrows']:
    sys.stderr.write("Wrong number of valid rows. Found " +
        str(y) + " instead of " +
        str(header['nrows']) + "\n")
return { 'min': mn,
        'max': mx,
        'step': header['cellsize'],
        'size': (header['ncols'], header['nrows']),
        'data': data,
        'nodata': header['nodata'] }

```

```

def scale_height(data, minheight=None):
    '''Scale the height from actual height values to the 2**16 range
    of unsigned shorts, using the full range unless a minimum height
    difference is given (in which case, the larger of the actual range
    and the given minimum range is used).'''
    z_scale = data['max'] - data['min']
    if minheight != None:
        z_scale = max(z_scale, float(minheight))
    sys.stderr.write("Z-scale of heightmap is " + str(z_scale) + "\n")
    raw = []
    for y in range(data['size'][1]):
        row = []
        for x in range(data['size'][0]):
            v = data['data'][y][x]

```

```

        if v == data['nodata']:
            v = data['min']
        row.append(max(min(int(((v-data['min']) / z_scale) * 2**16),
                           2**16-1), 0))

    raw.append(row)
    return raw

def write_height(row, size):
    '''Write the raw (scaled) heightdata to a file/stdout. The size of
    the output is the smallest power of 2 that fits the input'''
    p2x = int(pow(2, math.ceil(math.log(size[0], 2))))
    p2y = int(pow(2, math.ceil(math.log(size[1], 2))))
    sys.stderr.write("Output resolution (%d x %d)\n" % (p2x, p2y))
    for y in range(size[1]):
        sys.stdout.write(struct.pack("%dH" % size[0],
                                     *(e for e in itertools.islice(row[y],
                                                                     0,
                                                                     size[0]))))

    for x in range(p2x-size[0]):
        sys.stdout.write(struct.pack("H", 0))

    for y in range(p2y-size[1]):
        for x in range(p2x):
            sys.stdout.write(struct.pack("H", 0))

def entry(argv):
    parser = argparse.ArgumentParser(description = "Converts " +
        "the USGS DEM (.asc) format to a 16bit " +
        "(unsigned short) RAW format.")
    parser.add_argument("-f", "--inputfile", type=str,
        help="Input USGS DEM .asc file")
    parser.add_argument("-o", "--outputfile", type=str,
        help="Output RAW file")
    parser.add_argument("-m", "--minheight", type=str,
        help="Minimum height scale of the heightdata")
    args = vars(parser.parse_args(argv))

    must_close_infile = False
    must_close_outfile = False

    if args['inputfile']:
        sys.stdin = file(args['inputfile'], "r")
        must_close_infile = True
    if args['outputfile']:
        sys.stdout = file(args['outputfile'], "wb")
        must_close_outfile = True

    lines = sys.stdin.read().split('\n')

    sys.stderr.write("Parsing header\n")

```

```
header = read_header(lines[:6])

sys.stderr.write("Parsing data\n")
data = read_data(header, lines[6:])

if must_close_infile:
    sys.stdin.close()

sys.stderr.write("Calculating heightfield\n")
raw = scale_height(data, args['minheight'])

sys.stderr.write("Writing raw file\n")
write_height(raw, data['size'])

if must_close_outfile:
    sys.stdout.close()

if __name__ == "__main__":
    if len(sys.argv) > 1:
        exit(entry(sys.argv[1:]))
    else:
        exit(entry([]))
```

Appendix H

Stereo rendering

This appendix chapter lists an abbreviated version of the snow simulator main rendering function that highlights how stereoscopic rendering is implemented. It also includes the `StereoCamera` `advanceEye` and `mvp` functions, as the prior is called from the loop, and its state change is used in the second.

All timing-related code, as well as screenshot support, has been cut for brevity. Only the cloud-rendering call and the loop and related variables are original code, the rest is the result of previous work on the simulator.

```
/*
 * Calculate the model-view-perspective matrix.
 * Augment the camera position with the inter-pupillary distance
 * of the camera along the right-vector.
 */
glm::mat4 StereoCamera::mvp() const {
    glm::mat4 camera = glm::perspective(_fieldOfView,
                                        _viewportAspectRatio,
                                        _nearPlane,
                                        _farPlane);

    glm::mat4 ori = orientation();
    camera *= ori;
    glm::vec4 right = glm::inverse(ori) * glm::vec4(1.f, 0.f, 0.f, 1.f);
    float factor = (_currentEye == CAMERA_EYE_RIGHT ? 0.5 : -0.5);
    glm::vec3 delta = _pos + _ipd * factor * glm::vec3(right);
    camera = glm::translate(camera, -delta);
    return camera;
}

/*
 * Set internal state for which eye is in use, and change draw buffer
 */
```

```

void StereoCamera::advanceEye() {
    if( _currentEye == CAMERA_EYE_LEFT ) {
        _currentEye = CAMERA_EYE_RIGHT;
        glDrawBuffer(GL_BACK_RIGHT);
    } else {
        _currentEye = CAMERA_EYE_LEFT;
        glDrawBuffer(GL_BACK_LEFT);
    }
}

/*
 * Main rendering function called from main.
 */
void Render(Terrain* terrain , Wind* wind , Snow* snow ,
            Clouds *clouds , TimingSystem* times) {
    int eyes = 1;

    keyOperations();

    if (conf.camera_3dvision) {
        eyes = 2;
    }

    // Render shadow map only once
    if (conf.render_shadow) {
        shadowMap->ShadowMapPass(terrain , light);
    }
    // Render everything else per-eye
    for( int i = 0; i < eyes; ++i ) {
        if (conf.camera_3dvision) {
            dynamic_cast<StereoCamera*>(camera)->advanceEye();
        }

        if (!conf.wireframe) {
            glClearColor(conf.clear_color[0],
                        conf.clear_color[1],
                        conf.clear_color[2], 1.0f);
        } else {
            glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
        }
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        if (conf.render_terrain) {
            terrain->Render(camera , light , shadowMap->getShadowMap());
        }
        if (conf.render_skybox) {
            skybox->Render(camera);
        }
        if (conf.render_obstacles) {
            wind->RenderObstacles(camera);
        }
    }
}

```

```
    }
    // Pressure has to be rendered after the skybox to
    // make the blending work correctly
    if (conf.render_pressure) {
        wind->RenderPressure(camera);
    }
    if (conf.render_velocity) {
        wind->RenderVelocityLines(camera);
    }
    if (conf.render_wind_sources) {
        wind->RenderWindSources(camera, light);
    }
    if (conf.render_snow) {
        snow->Render(camera);
    }
    if (conf.cloud_type != CLOUD_OFF) {
        clouds->Render(camera, light);
    }
    if (conf.render_shadow_texture) {
        shadowMap->RenderPass();
    }
    if (conf.render_minimap) {
        terrain->RenderMiniMap(camera);
    }
    if (conf.tweak_bar) {
        TwDraw();
    }
}

glfwSwapBuffers(window);
glfwPollEvents();
}
```

Appendix I

User Manual

This appendix chapter contains a simple description of the steps required to use the functionality introduced in Chapter 3. The sections below mirror the sections from Chapter 3 directly, except the motivation section.

I.1 Wind

The boundary wind changes manifest in two options: the type of interpolation/source to use, and if sources are used the source data files. Figures I.1 and I.2 show these entries in the start-up menu. The file path is to the meta-file described in Section 3.2, and contains a list of the actual source files. Listings I.1 and I.2 contain examples of the meta file and the file it points to. Note that once the simulation is started, the selection of *Uniform wind* is final, while if one of the two source-based strategies is used, these two can be toggled (but *Uniform wind* may not be selected without restart).

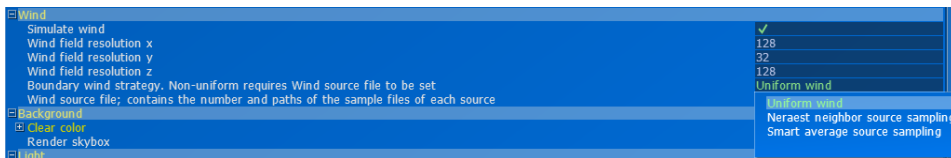


Figure I.1: The expanded entry (*boundary wind strategy*) is used to select the boundary wind type to use. *Uniform wind* is the old approach, and is the default setting.



Figure I.2: The *Wind source file* entry takes the relative or absolute path of a wind source meta-file. This setting is only used if the wind strategy is not *Uniform wind*

Listing I.1: Wind source meta-file example with two sources

```
data / wind_samples .1
data / wind_samples .2
```

Listing I.2: Wind source data file example with 4 samples spaced at 1 minute intervals

```
4.0 , 3.2 , 1.5 60000 4
1.0 0.0 0.0
0.4 0.0 0.3
0.0 0.1 0.8
-0.3 0.0 0.5
```

I.2 Precipitation

The precipitation distribution entries in the start-up menu are shown in Figures I.3 and I.4. Strategy can be changed after start-up, however it is important that if *Texture sequence* is used, the number of textures is ≥ 2 , and the path to the textures is valid. When texture sequences are used, the texture path takes the base name (for example the base name `data/dist.png` expects the actual textures to be at `data/distN.png`, where N is the zero-based frame index). If a constant texture is used, the path is the full path to that texture. If *Wrap* is set to false, the simulation is terminated when the time of the last frame is reached. All settings except the texture paths can be changed from the snow distribution menu in the running simulator.

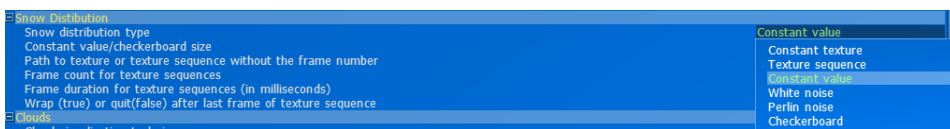


Figure I.3: The expanded entry (*Snow distribution type*) is used to select the strategy to use. The default is *Perlin noise*.



Figure I.4: All precipitation distribution settings. If the *Constant value* strategy is used, only the two first entries have meaning.

I.3 Clouds

The cloud rendering settings are shown in Figures I.5 and I.6. If the *No clouds* option is used, none of the settings (except the type) are used. The *Cloud threshold* setting is only

used with the *Distribution thresholded* type. Cloud height determines the maximum height of the cloud volume, and is defined in units of the simulation volume (that is 1.0 is the same height as the simulation volume). The cloud volume is always vertically centered on the top of the simulation volume, so the part of the volume that enters into the simulation volume is half of the *Cloud height* setting. All types and settings can be changed at run-time from the main menu in the running simulator.



Figure I.5: The selection of strategies to use for cloud rendering. The default type is *Vanilla*; purely decorative clouds.

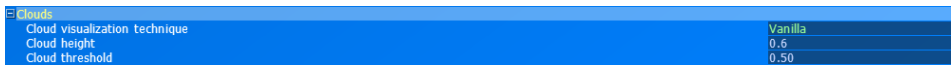


Figure I.6: All cloud rendering settings.

I.4 Terrain

The height-map pre-processing script is included in Appendix G. To see the options, run the script with the `-h` or `--help` option. There are 2 mandatory arguments; the input and output files paths. The third supported parameter is a minimum height scale to use when re-scaling height data. If this is not supplied, the full 16-bit range is used; otherwise that range is set to encompass the larger of the given value and the range of the input data. Missing values in the input data are set to the minimum height, that is to zero.

I.5 3D rendering

To use 3D rendering, the hardware must support OpenGL stereoscopic rendering. This means there must be a NVIDIA Quadro GPU installed, a 120HZ display device, and a NVIDIA 3D vision communication device attached. If the first requirement is not met, enabling the *Stereo rendering (3DVision)* option will fail, as a framebuffer of the correct format cannot be created. Otherwise, the simulator will run, but the content will not be displayed in 3D if the other requirements are not met. The *Inter-pupillary Distance (3DVision)* option sets the distance between the camera positions for each eye in the simulator's rendering coordinate system; the terrain in this system is $64 \times 32 \times 64$ units large, and the default value of 0.015 was found by experimentation for the Mt. St. Helens terrain. 3D rendering can only be toggled at startup, but the IPD can be adjusted at run-time using the hotkeys `K` and `J` to respectively increase and decrease the distance in increments of 0.001.

I.6 OpenCL hardware sampling

This setting is enabled by default. It can be disabled by supplying the pre-processor define `CLMANUAL_SAMPLING` when compiling the snow simulator.

I.7 Performance analysis

The profiling system is used through the `Python` script included in Appendix F. To run a profile, simply run the profiling script with the name of the profile to run. To add a profile, the script must be altered in several places:

- The profile must be assigned a bit for the bit-mask used internally. Simply use the next available bit after the last entry of the type `PROFILE_XX` (in the Listing F.1 this would be the 3rd bit – `0x4` – and the new entry would be on line 19.
- The profile’s description and name must be added to the help text and the `parse_args` function must add it’s bit-mask value to the `profiles` variable if encountered.
- All option permutations must be enumerated or generated. An example of enumerated values is the `tex_perms` list in the Listing F.1.
- The branch on profile bit-masks must call `run_profile` with the option permutations if the bit is masked. See lines 42-43 in Listing F.1.

The option permutations include which binary to use (`CUDA_BIN` and `CL_BIN`), a list of pre-processor directives (these are passed to the `CMAKE`, `C++`, and the `OpenCL` or `CUDA` compilers), option settings for the configuration file, and a list of commands to run after the profiling run is complete.

The `TimingEventType` enum in Listing F.1 contains all the supported timing sections. To add additional timing sections, add an entry to this array, and give it a descriptive name in the corresponding position in the `TimingEventNames` array. Then call `SetElapsedTime` or `CalculateSetElapsedTime` on the `TimingSystem` class instance with the elapsed time or start and end times respectively every frame. The system is intended for timing per-frame events, and it expects a timing value every frame while the system is running.

Finally, the number of frames for which the timing system runs is set with the `FRAME_COUNT` variable of the profiling script. The system will always run 10% of `FRAME_COUNT` frames before starting the timing, allowing the GPU driver to perform any profile-guided optimizations it may apply after a invocations of a kernel or shader.

I.8 Bindless textures

The use of bindless textures is enabled by default. To use legacy binding, supply the pre-processor define `LEGACY_TEXTURES` when compiling the CUDA version of the snow simulator. This may be required to run on older hardware.

I.9 Wind simulation stability problem

Neither of the proposed solutions are enabled by default, as it is not desirable to potentially sacrifice simulation correctness for visual quality, and the increase SOR iteration count proved insufficient to alleviate the problem. The code in Listing 3.3 may be manually inserted into the `wind_advect` kernel if stochastic sampling is desired.