Norwegian University of
Science and Technology

GPU-based Real-Time Snow Avalanche Simulations

Øystein Eklund Krog

Master of Science in Computer Science
Submission date: June 2010
Supervisor: Anne Cathrine Elster, IDI

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

Problem Description

Simulating dense volumes of snow is a computationally intensive task. The emergent behavior of a large amount of interacting snow particles is highly complex due to a variety of factors, such as density, volume, temperature and environment. A previous wind field simulation developed at our lab can simulate falling snow with visually pleasing results, but resorts to simple techniques for modeling snow accumulation on a terrain, and has no real support for dense volumes of snow.

This project will employ parallel programming on the Graphics Processing Unit (GPU) to simulate dense snow volumes using a large amount of particles. Interesting effects such as snow avalanches and snow accumulation will be investigated and a model that is suitable for real-time simulation will be developed. A previous implementation of a particle-based single-fluid model will be further developed and used in this project.

Assignment given: 15. January 2010
Supervisor: Anne Cathrine Elster, IDI

Abstract

Snow is a physical phenomenon that is hard to simulate due to the wide range of behaviors that can be found. Snow avalanches are of interest due to their complex physical properties and because they can have a very high cost, both in terms of economic impact and lives lost. In this thesis, we investigate the possibility of using fluid dynamics to model snow avalanches at real-time speeds. Real-time simulations allow for interactivity which again may accelerate the speed of development processes. It also allows for the simulations to be used as an interactive educational tool.

Although modern GPUs (Graphics Processing Units) are primarily designed to accelerate graphics calculations in computer games, their computational power can now also be harnessed by several other applications. This thesis builds on our previous work on a simple Smoothing Particle Hydrodynamics (SPH) fluid dynamics model accelerated by GPUs. We extend this work by developing a more complex SPH model and integrate these two models in our novel framework for GPU simulations. The simple SPH model is suitable for interactive simulations of low-viscosity Newtonian fluids, whereas our more complex SPH model includes support for Non-Newtonian fluids through the use of rheological models. By using a rheological model that describes the Non-Newtonian flow characteristics of snow avalanches, we can reproduce the flowing behavior of dense flowing snow avalanches at interactive speeds.

Our work shows that using the GPU can lead to very large performance improvements that make it possible to do real-time simulations which previously required costly specialized hardware or took minutes or hours to run. Using our highly optimized framework, we demonstrate a large improvement in performance for the simple model, and achieve generally very high performance for both implementations compared to other state-of-the-art implementations. Our recent results on the new NVIDIA GeForce GTX 470 Fermi-based card, achieves 215.4 IPS (iterations per seconds) for 64K particles using our simple model, and 122.2 IPS and 64.9IPS for 128K and 256K particles, respectively. For the complex model, 69.6 IPS at 64K particles, 37.4 IPS at 128K particles and 19.1 IPS at 256K particles were achieved. Real-time simulations enabling interactivity for the simple model is achieved for up to 256K particles, and up to 32/64K particles for the complex model.

Many other features, such as additional snow and fluid effects, could extend this work. A list of this and other possible avenues for future work is also included.

Acknowledgments

I would like to thank Dr. Anne C. Elster for her help and advice during the project. In addition I would like to thank NVIDIA for their generous donations of lab equipment, which made this thesis possible.

I would also like to thank my family and friends for their support and help and last but not least I would like to thank all the members of the HPC group at NTNU.

Contents

Abstract **i**

Acknowledgments **iii**

Contents **v**

List of Figures **ix**

List of Tables **xiii**

List of Algorithms **xv**

1 Introduction **1**

1.1 Motivation 1

1.1.1 Emergence 1

1.1.2 High Performance 2

1.1.3 Interactivity 2

1.1.4 Avalanche Simulations 3

1.2 Contributions 3

1.3 Outline 4

2 Avalanche Simulation **5**

2.1 Snow Avalanches 6

2.1.1 Classification 7

2.2 Modeling Approaches 8

2.3 Depth-integrated models 9

2.4 Fluid-mechanics models 10

2.5 Geophysical Fluid Flows 11

3 Computational Fluid Dynamics **13**

3.1 The Navier-Stokes Equations 14

3.1.1 Eulerian and Lagrangian methods 14

3.1.2 The Navier-Stokes Equations 14

3.2 Newtonian Fluids and Viscosity 17

3.3 Non-Newtonian Fluids 19

3.3.1 Power-Law/Ostwald model 20

3.3.2 Cross model 20

3.3.3 Bingham Plastic model 21

3.3.4 Herschel–Bulkley model 21

4 Smoothed Particle Hydrodynamics **23**
4.1 The Equations of SPH 24
4.1.1 The Smoothing Kernel 25
4.1.2 Derivatives 26
4.2 Lagrangian Fluid Dynamics 27
4.2.1 Density 28
4.2.2 Incompressibility 28
4.2.3 Pressure 29
4.2.4 Viscous Stresses 30
4.2.5 External Forces and Boundary Conditions 32
4.3 Smoothing Kernels 34
4.3.1 General Kernels 34
4.3.2 Specialized Kernels 38
4.3.3 Tensile Instability 39

5 Parallel Computing and the Graphics Processing Unit **41**
5.1 Parallel Computing 41
5.2 General Purpose Computing on Graphics Processing Units (GPGPU) 42
5.3 Compute Unified Device Architecture (CUDA) 44
5.3.1 CUDA Hardware Model 44
5.3.2 CUDA Programming Model 47
5.3.3 CUDA Memory Model 49
5.3.4 Accuracy and Scaling 52

6 Models and Implementation **55**
6.1 The SPH Algorithm 56
6.2 Simple SPH model 56
6.2.1 Model 57
6.2.2 Precalculation 57
6.2.3 Pseudocode 58
6.2.4 Resource usage and Occupancy 60
6.3 Complex SPH model 61
6.3.1 Model 61
6.3.2 Precalculation 63
6.3.3 Pseudocode 63
6.3.4 Rheological Models 65
6.3.5 Resource usage and Occupancy 67
6.4 Boundary Conditions 68
6.5 Integration of Forces 70

7 Simulation Framework **73**
7.1 Simulation Library 74
7.1.1 Components 74
7.1.2 Code techniques 81
7.1.3 Performance Optimization 84
7.1.4 Visualization and Rendering 89
7.2 Rendering Application 90
7.2.1 Functionality 91
7.2.2 Terrain Support in Ogre 93
7.2.3 Visualization of Particles 94
7.3 Console Test Application 95

8 Results and Discussion **97**
8.1 Test Setup 97
8.1.1 Software 97
8.1.2 Methodology 99
8.2 Performance Evaluation 101
8.2.1 Performance Scaling 101
8.2.2 Memory Scaling 102
8.2.3 Optimizations 106
8.2.4 Fermi performance 107
8.2.5 Review of other implementations 108
8.2.6 Effects of Texture Cache 110
8.2.7 Rendering Overhead 113
8.2.8 Kernels 115
8.3 Visual 116
8.3.1 Simple SPH 116
8.3.2 Complex SPH and Snow Avalanches 120

9 Conclusions and Future Work **125**
9.1 Performance 125
9.2 Snow Avalanche 126
9.3 Future work 126
9.3.1 Snow Avalanche 126
9.3.2 SPH models 127
9.3.3 Implementation 127

Bibliography **129**

A Smoothing Kernel Derivatives **137**

B Source Code **139**
B.1 Visualization Shader Code 139
B.2 Uniform Grid Framework Code 142
B.3 Color Calculation Framework Code 147
B.4 SPH Boundary Handling Code 150
B.5 SPH Neighbor Iteration Framework Code 157
B.6 SPH Smoothing Kernels Code 158
B.7 Simple SPH Implementation Code 165
B.8 Complex SPH Implementation Code 172

C Extended abstract for paper submitted to PARA 2010 **183**

D Poster **189**

List of Figures

2.1 A powder snow avalanche in the Himalayas near Mount Everest. 5

3.1 Computational Fluid Dynamics (CFD) Image of Hyper-X research vehicle at Mach 7 with engine operating. 13
3.2 The Eulerian and Lagrangian point of view. 15
3.3 Stress-strain curve and for a Newtonian Fluid with a dynamic viscosity of 1.5. 18
3.4 Viscosity in the two-plate example 19
3.5 Stress-strain curve and effective viscosity for a power-law fluid with a *flow consistency index* of 1.5 and a *flow behavior index* of 0.5(shear-thinning), 1.0(Newtonian) and 1,5(shear-thickening). . 20
3.6 Effective viscosity of a cross fluid, logarithmic scale on x-axis. Parameters $\mu_0 = 200$, $\mu_\infty = 1$, $n = 1$ and $K = 300$ 21
3.7 Stress-strain curve for a bingham fluid with a *dynamic viscosity constant* of 1.5. Note how it does not pass through the origin due to a nonzero yield stress. 22

4.1 Our "Simple" SPH model implementation, 512K particles. Particles are shaded in a hue gradient depending on their velocity. 23
4.2 The smoothing distance h and the surrounding particles within it. 24
4.3 The Gaussian kernel and it's derivatives along one axis in a 3-dimensional space with a smoothing distance h=1. 34
4.4 The Cubic Spline kernel and it's derivatives along one axis in a 3-dimensional space with a smoothing distance h=1. 35
4.5 The Quintic spline kernel and it's derivatives along one axis in a 3-dimensional space with a smoothing distance h=1. 36
4.6 The Quadratic kernel and it's derivatives along one axis in a 3-dimensional space with a smoothing distance h=1. 37
4.7 The Wendland kernel and it's derivatives along one axis in a 3-dimensional space with a smoothing distance h=1. 38
4.8 The Quartic kernel and it's derivatives along one axis in a 3-dimensional space with a smoothing distance h=1. 39
4.9 The three specialized smoothing kernels W_{poly6}, W_{spiky} and $W_{viscosity}$. The thick lines show the kernels, the thin lines their gradients in the direction toward the center, and the dashed lines the Laplacian. The diagrams are differently scaled. They are plotted along one axis in a 3-dimensional space with a smoothing distance h = 1. Reprinted from Müller et al. [1] with permission from Matthias Müller. 40

5.1 The change in memory bandwidth of modern GPUs compared to CPUs. Reprinted from [2], with permission from NVIDIA. 42
5.2 The development in peak Floating Point Operations Per Second (FLOPS) for GPUs and CPUs. Reprinted from [2], with permission from NVIDIA. 43
5.3 Bandwidth available to the device from various sources. Reprinted from [3], with permission from Rob Farber. 44
5.4 The difference between a CPU and GPU in the distribution of transistors. Reprinted from [2], with permission from NVIDIA. . 45
5.5 The NVIDIA GeForce GTX 470 46
5.6 CUDA Hardware Model. Reprinted from [2], with permission from NVIDIA. 47
5.7 CUDA Grid, block and thread model. Reprinted from [2], with permission from NVIDIA. 48
5.8 A simplified view of the the CUDA memory hierarchy. Reprinted from [2], with permission from NVIDIA. 50

6.1 The 3 main parts of the fluid simulation. 56
6.2 Calculation of forces for Complex SPH model 58
6.3 Calculation of forces for Complex SPH model 64
6.4 Terrain heightmap and normal images. 69
6.5 The terrain with face normals. 70
6.6 The leap-frog integrator. 71

7.1 Overview of the SPH simulation framework. 73
7.2 2D uniform grid with a cell size equal to the smoothing distance h. 75
7.3 Class diagrams of code helper classes. 76
7.4 Sequence diagram of Simulation-Settings-GUI interaction. 77
7.5 Class diagram of SimBufferManager class. 77
7.6 Class diagram of SimBufferManager class. 78
7.7 The particles inside the boundary are pushed back by a force that is proportional to the depth into the boundary. 79
7.8 The particles inside the boundary are affected by a friction force which acts in the opposite direction of velocity along the boundary. 80
7.9 Various color gradients and their application to various fluid properties. 90
7.10 Ogre terrain rendering with a pregenerated skybox. 93
7.11 Ogre terrain rendering with seamless level of detail (LOD) for various distances. 94
7.12 Closeup of fluid particle ball shading. 95
7.13 The console test application running a performance scaling measurement test. 96

8.1 Performance scaling of the two SPH implementations, here in a graph with both axes in log2. 102
8.2 Memory usage of the two SPH models. 105
8.3 Performance comparison of new and old implementations of the Simple SPH model. 106
8.4 Performance increase for new and old implementations of the Simple SPH model. 107
8.5 Performance comparison between the GeForce GTX 470 and the GeForce GTX 260. 107
8.6 Performance comparison to several earlier SPH implementations. Here in a graph with linear x-axis and log10 y-axis to make it possible to show the large differences in performance. 109
8.7 Performance effects in absolute values due to texture cache for the Simple SPH model 110
8.8 Performance effects in absolute values due to texture cache for the Complex SPH model 110
8.9 Performance increase in percent due to texture cache for the Simple SPH model 111
8.10 Performance increase in percent due to texture cache for the Complex SPH model 112
8.11 Performance of real-time rendering. 113
8.12 Performance overhead from real-time rendering. 114
8.13 Distribution of execution time among the different kernels (steps in the SPH algorithm) for the Simple SPH model, here for 128K particles. 115
8.14 Distribution of execution time among the different kernels (steps in the SPH algorithm) for the Complex SPH model, here for 128K particles. 116
8.15 The simulation performance test scene. Shown here are several snapshots in time. Using the Simple SPH model, 512K particles and velocity hue shading. 118
8.16 The Simple model with water-like parameters placed on a terrain. Shown here are several snapshots in time. Using 128K particles and velocity hue shading. 119
8.17 The Complex model with a Cross rheological model with the parameters $\mu_\infty = 1.07$, $\mu_0 = 300$ $n = 1$ and $K = 2.1$. We use a kinetic friction coefficient of 0.2. Shown here is the startin condition and the final runout of the avalanche. Using 64K particles, a fluid density of $400\,kgm^{-3}$ and uniform white particle shading. 121
8.18 The Complex model with a Cross rheological model with the parameters $\mu_\infty = 1$, $\mu_0 = 100$ $n = 2$ and $K = 30000$. We use a kinetic friction coefficient of 0.2. Shown here are several snapshots in time. Using 64K particles and uniform white particle shading. 122

8.19 The Complex model with a Cross rheological model with the parameters $\mu_\infty = 1$, $\mu_0 = 100$ $n = 2$ and $K = 30000$. We use a kinetic friction coefficient of 0.2. Using 64K particles and uniform white particle shading. 123

List of Tables

5.1 Important specifications of NVIDIA GPUs. 46
5.2 CUDA memory types. Data from [3]. 51
5.3 CUDA memory performance characteristics. Data from [3]. 51

6.1 Resource usage for Simple SPH kernels for GT200 architectures. . 60
6.2 Ideal block size and resulting occupancy for Simple SPH kernels for GT200 architectures. 60
6.3 Resource usage for Simple SPH kernels for Fermi architectures. . . 60
6.4 Ideal block size and resulting occupancy for Simple SPH kernels for Fermi architectures. 61
6.5 Resource usage for Complex SPH kernels for GT200 architectures. 67
6.6 Ideal block size and resulting occupancy for Complex SPH kernels for GT200 architectures. 67
6.7 Resource usage for Complex SPH kernels for Fermi architectures. . 68
6.8 Ideal block size and resulting occupancy for Complex SPH kernels for Fermi architectures. 68

7.1 Register improvement for SimpleSPH sum kernels using code restructuring to optimize register usage 88
7.2 Register improvement for SimpleSPH sum kernels using the volatile trick to optimize register usage 88

8.1 Test system. 97
8.2 The default simulation parameters used in performance testing. . . 98
8.3 The simulation parameters scaled across different resolutions, from 32K to 512K particles. 99
8.4 Parameter buffers in Simple SPH implementation 103
8.5 Parameter buffers in Simple SPH implementation 104

List of Algorithms

6.1 Pseudocode for Simple SPH density calculation, the SimpleSPH Sum1 kernel. . . . 59
6.2 Pseudocode for Simple SPH forces calculation, the SimpleSPH Sum2 kernel. . . . 59
6.3 Pseudocode for Complex SPH velocity tensor calculation, the Complex SPH Sum2 kernel. . . . 64
6.4 Pseudocode for Complex SPH forces calculation, the Complex SPH Sum3 kernel. . . . 65
6.5 Pseudocode for collision handling repulsion force against a wall boundary in the x direction. . . . 69
6.6 Pseudocode for collision handling friction force against a wall boundary in the x direction. . . . 69
7.1 Example of using the SimBufferManager class 77
7.2 Pseudocode for collision handling repulsion force. . . . 79
7.3 Pseudocode for collision handling friction force. . . . 80
7.4 An example of the implementation of the gradient of a smoothing kernel (the cubic kernel). . . . 81
7.5 Pseudocode explaining the use of the templated uniform grid neighbor iteration 82
7.6 Example of using the new matrix3 CUDA data structures 84
7.7 Register improvement for SimpleSPH sum kernels using correct data types. . . . 87
7.8 Example of suboptimal register usage using the uniform grid iteration loop. . . . 87
7.9 Improved register usage due to code restructuring using the uniform grid iteration loop. . . . 87
7.10 Example of using the volatile trick for register usage, here in the uniform grid iteration loop. . . . 88
7.11 The rendering application configuration file 92
8.1 Using CUDA Timing API for timing kernels. . . . 100
8.2 Using CUDA event API for timing kernels. . . . 101

Chapter 1

Introduction

Snow is a physical phenomenon that is hard to simulate due to the wide range of behavior that can be found. “Snow” can be everything from heavy and wet almost water-like to powder-like grains. There exist several models for modeling snow avalanches, and most contain at least one layer that is described as a fluid.

Current simulations often fake the visual effects without basing it on an underlying physical model. In this thesis, we investigate how snow avalanches can be modeled by use of fluid dynamics models.

Real-time simulations are interesting for several reasons, chief among them being interactivity. It makes it possible to accelerate the development of both the mathematical model as well as the implementation.

A shown and described in our previous work [4], GPUs (Graphics Processing Units) are responsible for a major increase in the parallel computational power available to consumers. Though the GPU is designed to accelerate very specific things such as 3D graphics, it has recently become possible to use it as a general purpose computational accelerator. Using CUDA, NVIDIAs technology for programming GPUs, it has become easier to accelerate parallelizable problems. We use CUDA and the GPU to massively accelerate simulations of fluid dynamics.

1.1 Motivation

The motivations for this thesis are numerous. Following is a description of the most important areas.

1.1.1 Emergence

The primary motivation for this thesis is the inherent and emerging complexity that can be observed in systems with fairly simple rules. This phenomenon is generally referred to as *emergence*, and is a results of the great number of possible interactions between different rules and states in a system. It is very interesting how

defining a system with very few rules can result in behaviors which are completely unpredictable for most observers.

Fluid simulations is one example of such a system, and physics in general are of great interest due to this phenomenon. To explore the behavior of such systems, mathematical models and computer simulations can be very helpful.

1.1.2 High Performance

Simulations of systems with high emerging complexity has, almost by definition, a high computational cost. This restricts the scale and accuracy with which models can be simulated.

To counteract this problem, it is interesting to explore and exploit the computational power of specialized hardware. By using specialized hardware greater performance can be achieved since it is possible to create and use computer hardware which is better suited for a specific problem.

A good example of this is how the Central Processing Unit (CPU) in most computers works on a few pieces of data at a time, while the Graphics Processing Unit (GPU) may work on hundreds and thousands of pieces of data in parallel. This property is of great interest when simulating systems where the interactions between different parts can be separated and hence be computed in parallel, thus achieving much greater performance.

It is also interesting to note that during the last decade the gaming and entertainment industry has been the driving force for the development of high performance hardware. This coincides with a move towards more physically based games and more advanced computer graphics. It is inevitable that this trend will continue for the near future, and as such the research community should try to take advantage of it.

1.1.3 Interactivity

Interactivity and “real-time” performance is of great interest for many reasons. On of the most important is the great advantage the human learning process has from the *feedback-loop*. In learning theory, this concept is used to describe the process with which humans learn. By establishing a loop where a human observer can observe a system, introduce changes in the system and then observe the results, learning and understanding is enhanced.

This means that the development of mathematical models to describe complex systems can be accelerated. An average observer will understand and be able to predict system behavior quicker, and a scientist can more quickly formulate hypotheses, test theories and establish facts. In the context of avalanche, simulations this is very important since many avalanche models rely on calibration against real-world events.

1.1.4 Avalanche Simulations

Avalanches are highly complex and are prime example of systems with high emergent complexity. Due to the great forces that are at play, avalanches can have very high cost, both in terms of economic impact and lives lost. It is thus of great interest to be able to model and simulate avalanches.

In the context of avalanche simulations, interactivity is doubly important because most avalanche models rely on calibration against real-world events. Interactivity also makes it possible to educate people on the behavior of avalanches and make make them fully comprehend how dangerous an avalanche event can be.

Avalanches have several key features. The *formation*, meaning the events which contribute the state where an avalanche can occur. The *release*, which describes the failure event where an avalanche is created. And the *flow*, which describes the motion of the avalanche mass.

In this thesis, we will focus on the flow of the avalanche itself, not formation or release.

1.2 Contributions

In this thesis, we investigate creating an interactive snow avalanche simulation. We build on our earlier [4] implementation of Smoothing Particle Hydrodynamics (SPH), which is very suitable for interactive simulations.

To ease the development of SPH-models on the GPU we create a new simulation framework. The framework uses NVIDIA CUDA to make use of the power inherent in modern GPUs and contains modularized components which ease the developement of GPU-based SPH simulations.

We use our framework to implement two different Smoothed Particle Hydrodynamics (SPH) models, a Simple model and a Complex model.

The Simple SPH model was implemented and evaluated in our previous work [4] and is suitable for simulating low-viscosity incompressible Newtonian fluids. It trades accuracy for high performance and stability.

The Complex SPH model is more accurate and has support for simulating Non-Newtonian fluids with viscosity determined by several *rheological* (the study of flow of matter) models.

We use the Complex model to simulate flowing avalanches on a terrain with interactive performance and the Simple model to simulate water-like fluids.

Both SPH-implementations are highly performance-optimized and run entirely on the GPU.

We perform a performance evaluation of both implementations and show that their performance is much greater than previous SPH implementations.

1.3 Outline

This thesis is structured in the following manner:

2 (Avalanche Simulation) provides an introduction to avalanche modelling, the challenges in modelling snow avalanches and related work.

3 (Computational Fluid Dynamics) includes a brief description of Computational Fluid Dynamics (CFD) and Newtonian and Non-Newtonian fluids.

4 (Smoothed Particle Hydrodynamics) is a description of the SPH method and how it can be used to model Newtonian and Non-Newtonian fluids.

5 (**Parallel Computing and the Graphics Processing Unit**) describes the existing knowledge that exists in the field of parallel computing and the details of the CUDA GPU-programming environment.

6 (Models and Implementation) describes the two SPH models and their implementation.

7 (Simulation Framework) describes our new simulation framework, the components within and and the optimization techniques we employed.

8 (Results and Discussion) presents the results, both in terms of performance and visual results. We analyze the performance and compare it to our previous implementation and other implementations.

9 (Conclusions) present the conclusion and future work that is relevant to this project.

A (Smoothing Kernel Derivatives) include spatial derivatives of the smoothing kernels used in the implementation.

B (Source Code) lists extracts from source code of our implementations.

C (Short Paper) is the extended abstract for a paper, submitted to PARA 2010.

D (Poster) gives a nice overview of our work in fluids simulations on the GPU. It was presented in our groups booth at ISC 2010 and at CCP 2010.

Chapter 2

Avalanche Simulation

Figure 2.1: A powder snow avalanche in the Himalayas near Mount Everest.

An avalanche is a physical phenomenon that is difficult to model and to simulate. Avalanches are inherently complex due to their great emergent complexity, but also because there are many "external" factors at play.

An avalanche can be defined as a rapid gravity-driven slide or release of mass moving down a sloped terrain (Figure 2.1 on page 5 [1]). Beyond this definition an avalanche can vary greatly in both composition of materials and the state of the materials themselves.

A snow avalanche can be everything from an almost water-like flow to a very dry, almost powder-like, air-suspended particle flow. This great variety in behavior also

[1]Public domain image, reprinted from Wikipedia.
http://commons.wikimedia.org/wiki/File:Avalanche_on_Everest.JPG

means that it is exceedingly hard to create a single unified model that can cover all possible types of avalanches.

Avalanches consists of 3 key features [5]:

1. The *formation*, meaning the events which contribute to the state where an avalanche can occur.
2. The *release*, which describes the failure event where an avalanche is created.
3. The *flow*, which describes the motion of the avalanche mass.

The *formation* of an avalanche is complex since it involves a great number of variables. An example of this is the complexities found in a snow. Snow has both complex microstructure and macrostructure. The microstructure of a snow volume consists of snowflakes. Each snowflake is a crystal which forms around a small impurity. The structure of a snow crystal is dependent on factors such as humidity and temperature at the moment of formation. The macrostructure of snow is also complex. When snow accumulates during a longer period of time we get a *snowpack*, where different layers consist of snow with different properties. Over time the properties of each layer can change, which further complicates matters.

The *release* of an avalanche can be due to a variety of conditions, but the general event can be described as the catastrophic failure that happens when the gravitational force applied to the material in a slope exceeds the binding force of the material. The material of the top of the slope can then push or flow over the underlying layer, thus creating an avalanche. Taking the example of a snow avalanche several factors can contribute to such a failure, including new snow, wind, temperature, rain and the snowpack structure itself.

The *flow* or motion of an avalanche describes the manner in which the avalanche matter moves down the sloped terrain. When considering the flow of an avalanche it is fairly obvious that both the materials involved and their composition is important for the resultant behavior. The flow characteristics is thus dependent on both the formation and the release. Last but not least, the basal friction against the terrain is also very important.

Trying to capture all the variables that contribute to the behavior of an avalanche is exceedingly difficult. This may be why there does not yet exist a single constitutive model for avalanches, instead a variety of modeling approach are used to capture different effects of the avalanche.

2.1 Snow Avalanches

As mentioned earlier, snow avalanches are among the most complex avalanches due to the great complexities which arise from the composition and state of the snow matter. In this thesis we choose to focus on flowing snow avalanches, and in these types of avalanches the properties of the flow are affected by many things.

2.1.1 Classification

The classification of snow avalanches is difficult due to the wide variety of factors involved in the behavior of the avalanche. In the literature several forms of classification can be found.

We include some of the classical descriptions since they cover the qualitative behavior of different types of avalanches quite well, it is however important to keep that these are classifications and that most avalanches include factors from several of these.

The first set of classifications consider the macrostructure of the snowpack and the subsequent effect on the avalanche:

Loose snow avalanche This form of avalanche is used to describe a release of “loose snow” down a mountainside. The loose snow is usually a (relatively) small amount of dry and possibly fairly new snow from the top layer of the snowpack, which start from a point and fan outward as they descend. These avalanches are often found in fairly steep terrain, and have relatively low density. This form of avalanche is usually not very dangerous, especially so since they tend to fracture below skiers in a track instead of above.

Slab avalanche A slab avalanche is a type of avalanche where fairly large cohesive pieces of snow are released. Such slabs form from “stiff” layers of snow which can be formed as a result of strong winds or because the snow is old. Slab avalanche are generally considered to be the most dangerous form of avalanches. They occur when the snowpack has inherent instabilities, such that large coherent layers can break free. Wind can play a significant factor in the creation of such instabilities, since surface saltation of snow can create snowpacks with non-uniform structures.

An alternative set of classifications which considers the wetness of snow is also commonly found:

Dry snow avalanche A dry snow or *airborne* avalanche is usually triggered by putting too much stress on the snowpack, and is the most common type of avalanche triggered by people. Possible sources of stress include people(skiers etc.), additional snow or wind.

These avalanches travel very rapidly (50-100m/s) , have a density of 5-50$kg/m^3$and a flow depth of 10-100m [5]. In this type of avalanche a large amount of the snow involved can be suspended in the air, resulting in a highly turbulent suspension layer or “snow cloud”. Due to this turbulent layer the avalanche is not as suspect to the terrain relief and does not necessarily follow the terrain.

Wet snow avalanche A wet snow or *flowing* avalanche is an avalanche that is triggered by decreasing strength of the snowpack, as such they often occur naturally. Due to factors such as rain, prolonged melting by sun exposure or temperature the cohesion of the snowpack is lost, and a flow of snow occurs.

This type of avalanche is fairly slow, with velocities ranging from 5 to 25 m/s. Flow depth rarely exceeds a few meters and density is fairly high ranging from 150 to 500 kg/m^3 [5]. In this type of avalanche the overall movement of the avalanche does follows the terrain.

2.2 Modeling Approaches

Ancey [5] presents a comprehensive summary of the current state of snow avalanche modelling, where he specifies several categories of models, which will be summarized here:

Statistical methods Statistical methods are fairly powerful tools, they are commonly used in land-planning, where it is important to determine safe areas. These methods require either accurate knowledge of past avalanches or accurate methods for computing avalanche boundaries. They do not focus on the topology of an avalanche, but generally try to predict the extension (stopping position) of an avalanche.

Fluid-mechanics approach (avalanche-dynamics models) Snow avalanches usually appear as viscous flows flowing down a slope, and this obvious property has prompted use of fluid-mechanics as a tool for describing their motion. There are however many problems with this approach. Since there is little data available on the rheological processes in avalanche release and flow, all the avalanche-dynamics models proposed so far rely on analogies with other phenomena. Such phenomena include granular flows, and both Newtonian and Non-Newtonian fluids.

Simple models This category include some of the earliest attempts are avalanche modelling, and generally produce very crude estimations of avalanche features. Some of the earliest simple models simply consider the entire avalanche as a single element, and really only model the friction against the terrain.

Intermediate models (depth-averaged models) For flowing avalanches most models use depth-averaged mass and momentum equations to compute flow characteristics. They commonly use the shallow water (Saint-Ventant) equations, and are often used for simulation flowing avalanches. For these types of avalanches several types of constitutive equations have been proposed; Newtonian fluids, Reiner-Ericksen fluid, Bingham fluid, frictional Coulomb fluid and so on. A well-known but

limited model is the Savage-Hutter model, which assumes that flowing avalanches have many similarities with dry granular flows, and that the Coulomb law can be used to describe the bulk behavior or flowing granular matter.

The SATSIE initiative have produced the MN2L, D2FRAM and other models which are fairly advanced depth-averaged models.

For airborne avalanches intermediate models usually consider an airborne avalanche as a one-phase flow. These models usually do so and also usually consider avalanches as turbulent stratified flows, which means that the bulk behavior of an airborne avalanche is well identified, in sharp contrast to flowing avalanches. The largest differences among the models usually concern the boundary conditions, use of the Boussinesq approximation and the closure equations for turbulence.

Three-dimensional computational models A rapid increase in computer power has made it possible to do simulations in 3D. Compared to depth-averaged models the largest problems concern numerical treatments. For airborne avalanches there exist models using finite-volume codes for turbulent flows.

Small-scale models These types of real-world scale models are based on similarities between avalanches and other gravity-driven flows. Models using both fluids (in a water tank) and granular matter (e.g. pingpong balls) have been used. Since gravity can not be scaled, these forms of models can never be entirely correct, but they can nonetheless provide insights into the behavior of flowing materials.

2.3 Depth-integrated models

The current state of the art models for prediction rely on complex depth-integrated models, which use many compensation factors. These models have been under development for a long time. A classical model is the Savage-Hutter model, which is a depth-averaged dynamical model of a fluid-like continuum. This model consists of hyperbolic differential equations for the distribution of the depth and the depth-averaged velocity components. The model consists of cohesionless granules which form the continuum and was designed to prediction motion and deformation from initiation to runout along an avalanche track on a terrain. This model has since been extensively modified and improved. Hutter et al. [6] concludes that the Savage-Hutter model is a valid model for sand avalanches, but that for snow avalanches it may have to be complemented by a second viscous contribution.

There also exist some new models developed under the SATSIE (Avalanche Studies and Model Validation in Europe) initiative such as MN2L and D2FRAM, which are also depth-integrated models. The largest disadvantage of these models is that they do not capture the full geometrical detail of an avalanche.

2.4 Fluid-mechanics models

The use of fluid-mechanics for avalanche modeling has several key advantages. The foremost is that unlike statistical models and other simplified models it carries the promise of fully capturing all the effects of an avalanche. Of particular interest to us is the fact that such a model can more fully capture the geometry of a moving avalanche.

The use of fluid-dynamics for modeling of avalanche flow has not yet been considered accurate enough for prediction of avalanche result. There are several key challenges to using such an approach. Some of the larger challenges include problems due to the wide range of particle sizes found in an avalanche, the fact that composition of the avalanche changes over time and problems due to unknown boundaries and initial conditions. In addition it is difficult to model snow as a fluid because it is hard to fully quantify the *rheometrical* (the flowing properties) of snow.

It is not currently possible to properly determine the constitutive equations for snow due to a lack of measuring equipment [5]. The basic constitutive relationships of most types of avalanches, including moving soil, rock and snow are in fact largely unknown due to complex and varying rheological behaviors [7].

In addition a fully 3-dimensional fluid-dynamics model is extremely computationally heavy, so much so that it has only in fairly recent years become feasible to do such simulations outside the context of large supercomputers.

Since the use of a 3D fluid-dynamics model is an area that is not as well explored as some simplified avalanche models, it is of great interest to create an environment where it is possible to explore the problem space and create models which fully capture the desired effects.

The use of Non-Newtonian fluids as a modeling tool for snow avalanches has been explored. Dent and Lang [8] and Maeno [9] have measured the velocity profile within snow flows and generally deduced that snow generates a Non-Newtonian viscoplastic flow, whose properties depend a great deal on density.

Nishimura and Maeno [10] experimentally tested various fluid models to describe the motion of snow avalanches, including a Newtonian fluid model.

Kern et al. [11] have done extensive work on establishing a correct rheological model for flowing snow. He argues that without a proper constitutive model, it is impossible to establish a correct model. Though a common approach in avalanche modelling is the use of depth-averaged models, he argues that these models can never be truly correct due to a number of necessary assumptions (among them the assumption that snow is a simple fluid). He presents two different rheological models (a Herschel-Bulkley and a Cross-model) which can predict the velocity profile of flowing snow in a large chute.

Ancey and Meunier [12] show how avalanche-velocity records can be used to determine the bulk frictional force, where a striking result is that the bulk behavior of most snow avalanches can be approximated using a Coulomb frictional model.

Platzer et al. [13] through a series of experiments in a snow chute determine the coefficient of sliding friction for snow avalanches ranging from wet to dry. They did experiments to establish the basal friction coefficient for dense flowing avalanches. This parameter is crucial for determining the runout distance of snow avalanches. They found that a Mohr-Coulomb relation of the form $S = c + bN$ (where S is the ratio of shear and N is the normal stress) accurately describes the measurements they made in a large chute and show that basal shearing is the primary frictional mechanism retarding snow flows. In contrary to many postulated constitutive relations for basal shearing there is no velocity dependence.

Bovet et al. [14] use Non-Newtonian model with a Bingham and Cross fluid to simulate snow avalanches.

McClung and Schaerer [15] conclude that rheological parameters are affected by shear rate and temperature. Increased amount of water cause reduced friction at the snow/ground interface. The snow stiffness/viscosity decreases with water content. Once excess water drains away the glide rate slows.

Ancey [5] says that the rheology of snow avalanches is extremely complex because snow is thermodynamically very sensitive. Snow can hover around the triple-point of water at $0°C$, leading to large variations in composition.

2.5 Geophysical Fluid Flows

Geophysics is the study of the whole Earth, and includes such areas as tectonic plate movement, the internal structure of the earth, earthquakes, but also general geomorphological flows such as avalanches. Flows of mud, soil, rock and snow a

Rheology is the study of the flow of matter. This includes fluids, but also solids or granular matter which under certain conditions exhibit flowing behavior rather than (elastic) deformation.

Rheology is the study of flow of matter and is concerned with not just liquids, but also solids. Complex substances such as muds, sludges and suspensions are some of the things that the science of rheology tries to understand. The flow of these complex substances can not be captured using traditional Newtonian fluids, and relies on the science of both *Non-Newtonian* fluids and the science of *Plasticity* (of solid objects) to quantify, model and simulate how the flow behaves.

The key feature of rheology is that viscosity is not a constant, and can depend on a variety of factors.

Avalanches of mud, soil, rock and snow are all types of geomorphological flows, even though this term and geophysics in general is usually more concerned with flows on large timescales, such as continental drift.

There exist several SPH-based models for simulation of geophysical flows.

McDougall and Hungr [16, 17] have developed a depth-integrated SPH-based model for dynamic analysis of rapid flow slides, debris flows and avalanches. The model is

capable of accurately predicting the margins of various curving flows using a single set of input parameters.

Hungr [18] use the DAN and DAN3D models to model avalanches. The DAN3D model is a SPH-based model based on the shallow-flow equations and model Non-Newtonian fluids in a Lagrangian framework. They have open rheological kernels, allowing for frictional, viscous or turbulent resistance acting on the base of an internally frictional flow.

B. Ataie-Ashtiani [19] descrive a simulation of landslide impulse waves by incompressible SPH, using a rheological model implemented as a combination of the Bingham and general Cross models.

Laigle et al. [20] use a SPH method for a 2-dimensional numerical investigation of mudflow and other fluid flow interactions with structures.

Ferrari et al. [21] use a SPH method to recreate a catastrophic dam break and the resultant mudflow.

Bovet et al. [14] create a model for snow avalanches using Non-Newtonian fluids with shear-thinning and Bingham-like constitutive behaviors.

Paiva et al. [22] implements a SPH-based simulation of viscoplastic materials where fluids with high viscosity interact with solids. They demonstrate effects such as creeping, melting, hardening and flowing and simulate materials such as jelly and lava.

Hosseini et al. [23] implements a SPH-based model for generalized Non-Newtonian flows using several rheological models, including Power-law, Bingham-plastic and Herschel-Bulkley. They validate the model against explicit solutions and show generally high accuracy.

Chapter 3

Computational Fluid Dynamics

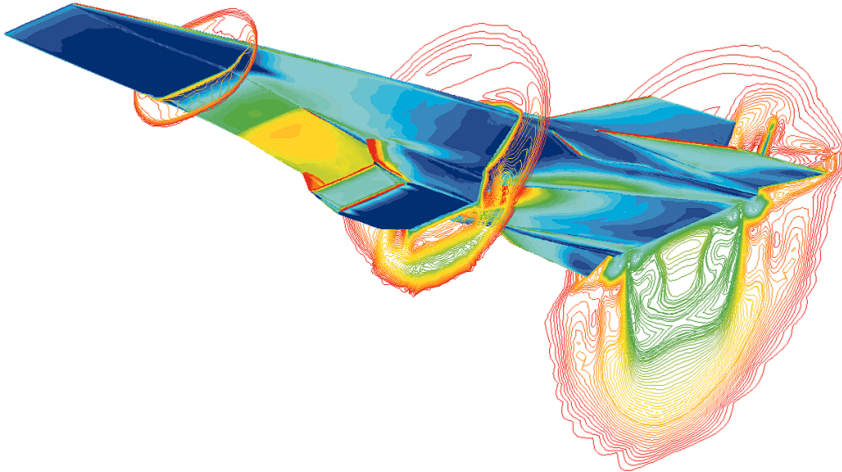


Figure 3.1: Computational Fluid Dynamics (CFD) Image of Hyper-X research vehicle at Mach 7 with engine operating.

This chapter builds on our earlier work [4].

Fluid dynamics deal with the science of fluids (liquids and gases) in motion. This is called fluid flow and can be used for modelling a variety for systems, with some examples being aerodynamics, aeronautics(study of flight capable machines), hydrology(the study of movement, distribution and quality of water on Earth i.e. weather), and computational fluid dynamics (Figure 3.1 on page 13[1]).

[1]Reprinted with permission from NASA.
http://www.dfrc.nasa.gov/Gallery/Photo/X-43A/HTML/ED97-43968-1.html

Computational fluid dynamics (CFD) is a sub-discipline of fluid dynamics, where numerical methods and algorithms are employed to solve and analyze problems involving fluid flows.

The history of CFD began with Bernoulli(1700-1782), who derived the famous Bernoulli equations, followed by Euler(1707-1783) who proposed the Euler equations. The Euler equations describe the conservation of momentum for an inviscid fluid, as well as conservation of mass. Then came Navier(1785-1836) and Stokes(1819-1903) who added support for viscous flows, thus giving us the *Navier-Stokes Equations* (NSE). Since then there have been many additions to the field, but the Navier-Stokes equations still remain as the basis of CFD, and are now well-established as a good model for viscous flows.

3.1 The Navier-Stokes Equations

The Navier-Stokes equations describe the motion of a fluid substance and are named after Claude-Louis Navier and George Gabriel Stokes. Navier and Stokes formulated the equations in the 19th century and the equations can be found in many formulations.

The most important assumption of these equations is that a fluid is a continuum, not a sum of discrete elements but rather a continuous substance. When solved the equations produce a velocity field, where each position in space is a vector of the velocity in that position.

3.1.1 Eulerian and Lagrangian methods

The Navier-Stokes equations can be very time consuming to solve, especially for 3-dimensional volumes. This is because a numerical solution usually results in non-linear partial differential equations. These can be solved by partitioning the volume into a grid and solving using the finite difference or finite volume methods. These methods are usually referred to as Eulerian methods.

An alternative to the Eulerian methods is Lagrangian(or particle-based) methods. These methods work by dividing the fluid itself into discrete particles and then applying the fluid equations to the particle mechanics.

Essentially the Eulerian method focuses on a spatially fixed volume, through which fluid flows. The Lagrangian formulation follows an individual fluid parcel as it moves through time and space (Figure 3.2 on page 15).

3.1.2 The Navier-Stokes Equations

The Navier-Stokes [24] equations in the Eulerian(grid based) formulation describe a fluid flow in terms of velocities, not positions, in contrast to classical mechanics where the goal is usually to find the position.

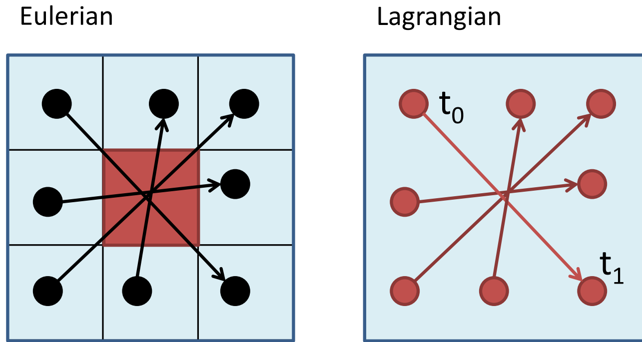


Figure 3.2: The Eulerian and Lagrangian point of view.

There are two primary principles in play when considering a fluid, the conservation of mass and the conservation of momentum.

3.1.2.1 Eularian formulation

The conservation of mass / continuity equation is given by:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0 \tag{3.1}$$

The conservation of momentum is given by:

$$\rho \left(\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} + v \cdot \nabla \mathbf{v} \right) = -\nabla P + \nabla \cdot \tau + \mathbf{f} \tag{3.2}$$

Where$\mathbf{v}$ is the *velocity field*,ρ the *density*, P the *pressure* and τ is the *viscous stress*.

Advection The advection term $(v \cdot \nabla v)$ in the Navier-Stokes can be described as the time independent acceleration of a fluid with respect to space. Essentially the fluid is transported along its own flow.

3.1.2.2 Lagrangian formulation

The conservation of mass / continuity equation is given by:

$$\frac{d\rho}{dt} = -\rho \nabla \cdot \mathbf{v} \tag{3.3}$$

Using the substantive derivative, which specifies: $\frac{d\mathbf{v}}{dt} = \frac{\partial \mathbf{v}}{\partial \mathbf{t}} + v \cdot \nabla v$, we get the Lagrangian formulation of the conservation of momentum:

$$\frac{d\mathbf{v}}{dt} = -\frac{1}{\rho}\nabla P + \frac{1}{\rho}\nabla \cdot \tau + \mathbf{f} \tag{3.4}$$

Essentially the advection term (3.1.2.1) is removed because fluid advection is implicit through particle movement.

We can ignore the mass conservation (Equation 3.1) if we assume that all particles have the same mass, and we have a constant number of particles.

We end up with the expression:

$$\mathbf{a_i} = -\frac{1}{\rho_i}\nabla P_i + \frac{1}{\rho_i}\nabla \cdot \tau_i + \mathbf{f}_i = f_i^{pressure} + f_i^{stress} + f_i^{external} \tag{3.5}$$

where $\mathbf{a}_i$ is the acceleration for a particle, $f_i^{pressure}$ is the force contribution from isotropic stress (pressure), f_i^{stress} is the force contribution from deviatoric stress (viscosity) and $f_i^{external}$ is the force contribution from a external forces (e.g. gravity and boundaries).

Isotropic stress (Pressure) The term $-\nabla P$ in the Navier Stokes Equations ((3.2)) is the *pressure gradient.* The pressure gradient arises from the isotropic(invariant with respect to direction) normal stresses which exist for almost all situations. The pressure constrains the fluid in such a way that the volume of the fluid is constant. This term moves the velocity field along the gradient of the pressure field, essentially moving fluid from high pressure to low pressure areas.

Deviatoric stress (Viscosity/Shear forces) The term $\nabla \cdot \tau$ in the Navier Stokes Equations (Equation 3.2) is the *viscous stress tensor* and is the deviatoric/shear stress of the fluid. For a Newtonian fluid the shear stress is proportional to the shear strain rate, but for a Non-Newtonian fluid it can be dependent on a range of factors (Section 3.2).

External forces The term f in the Navier Stokes Equations (Equation 3.6) is an external force field working on the fluid, often the force of gravity.

3.1.2.3 Incompressible flow of Newtonian fluids

For incompressible flow of Newtonian fluids the formulation of the momentum equation reduces to:

$$\frac{d\mathbf{v}}{dt} = -\frac{1}{\rho}\nabla p + \frac{\mu}{\rho}\nabla^2 \mathbf{v} + \mathbf{f} \tag{3.6}$$

The shear stress term $\nabla \cdot \tau$ becomes $\mu\nabla^2 v$, when the fluid is assumed incompressible and Newtonian. The term μ is the dynamic viscosity and is the proportional

constant between the shear stress and the shear strain rate. Due to the fact that this proportion is constant for a Newtonian fluid it is simply denoted by the scalar term μ.

This gives us the acceleration of a particle:

$$\mathbf{a_i} = -\frac{1}{\rho_i}\nabla P_i + \frac{\mu}{\rho_i}\nabla^2 \mathbf{v}_i + \mathbf{f}_i = f_i^{pressure} + f_i^{viscosity} + f_i^{external} \tag{3.7}$$

This formulation of the Navier-Stokes equations rules out the possibility of shock waves, but holds true even when dealing with a "compressible" fluid under certain situation.

Diffusion The term $\mu\nabla^2 v$ is essentially the deviatoric stress in an incompressible fluid, and is often referred to as the *diffusion of momentum*. The term $\nabla^2 v$ is the vector Laplacian of the velocity field and can be interpreted as the difference between the velocity at a point and the mean velocity in a small volume surrounding it. The term μ is the *volume viscosity coefficient*. This term can be explained as the fluids tendency to resist flow changes depending on the viscosity of the fluid.

3.2 Newtonian Fluids and Viscosity

Viscosity is the measure of internal friction in a fluid. When an external force is imposed on the fluid, the viscosity is the resistance to deformation.

Most people intuitively understand viscosity to be how "thick" a fluid is. Pure water is then understood to be "thin", and syrup is seen to be "thick". Thus water has a low viscosity, and syrup has a high viscosity. For everyday life this understanding is usually sufficient. In fact most of the fluids that people see as fluids (water-like substances) can usually be considered Newtonian fluids, because they are either purely Newtonian, or their behavior can be approximated with a Newtonian model.

Sir Isaac Newton proposed the following Generalized Newtonian model [25]:

$$\tau = \mu(|\dot{\gamma}|)\dot{\gamma} \tag{3.8}$$

where τ is the *shear stress rate*, $\dot{\gamma}$ is the *shear strain rate* and μ is the *dynamic viscosity*.

The term $|\dot{\gamma}|$ is the second invariant of the strain rate, and is a measure of the magnitude of the deformation rate. Similarly we use $|\tau|$ to denote the second invariant of the shear stress rate.

The shear stress τ is usually called the *viscous stress tensor* and the shear rate $\dot{\gamma}$ is usually called the *rate-of-strain tensor*. In one dimension $\dot{\gamma} = \frac{du}{dy}$ where $\frac{du}{dy}$ is the velocity gradient in the direction of shear.

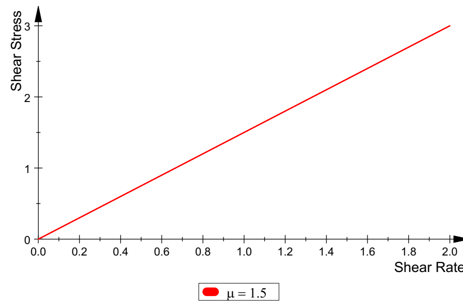


Figure 3.3: Stress-strain curve and for a Newtonian Fluid with a dynamic viscosity of 1.5.

Effective or (apparent) viscosity is thus the ratio between shear stress and the shear rate:

$$\mu_{eff} = \frac{\tau}{\dot{\gamma}} \tag{3.9}$$

A fluid modelled by this equation is known as a Generalized Newtonian fluid, and has a stress versus strain curve that is linear and passes through the origin (Figure 3.3 on page 18). Some examples of Newtonian fluids include oils, syrup, air and other gases.

Viscosity can be decomposed into two components; the shear viscosity describes the reaction in a fluid when shear stress is applied and bulk viscosity (also known as volume viscosity) which describes the reaction to compression. Bulk viscosity is important for modelling shocks in the fluid.

The SI unit of the *dynamic viscosity* μ (also known as η) is the pascal-second ($Pa \cdot s$), which is identical to $N \cdot m^{-2} \cdot s$.

Newton defined the measure of viscosity using a theoretical experiment where two plates are arranged with a fluid layer held between them (Figure 3.4 on page 19). Assume the fluid has a *dynamic viscosity* of one pascal-second and that the plates are very large such that there is a no-slip condition (absolute friction) between the plates and the fluid. Hold the bottom plate fixed and push the top plate sideways with a shear stress equal to one pascal. During one second the plate will then move a distance equal to the thickness of the layer between the plates.

It is also important to remember that temperature can be a large factor in the dynamic viscosity of a fluid as a results of changes in pressure. As an example water viscosity goes from 1.79 $10^{-3} Pa \cdot s$ to 0.28 $10^{-3} Pa \cdot s$ in the temperature range from 0 °C to 100 °C.

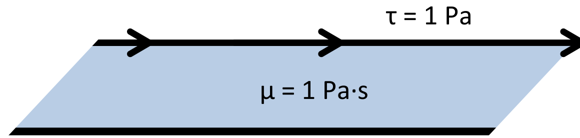


Figure 3.4: Viscosity in the two-plate example

3.3 Non-Newtonian Fluids

Non-Newtonian fluids are different from Newtonian fluids in that their effective viscosity is not just linearly dependent on shear stress. The effective viscosity of a Non-Newtonian fluid can in fact be non-linearly dependent on the shear stress, as well as external factors such as time and temperature. Measuring (and quantifying) the viscosity of these fluids can be difficult since fully isolating these factors is hard, e.g. shearing in a fluid will itself generate heat, and thus change the viscosity.

Non-Newtonian fluids can be classified by what factors determine their behavior, as well as what effect the external factors have on the fluid.

Fluids which change their behavior depending on time and prior stresses are said to have *memory*.

Time-dependent viscosity Many fluids can be considered to have time-dependent viscosity, where the viscosity changes depending on the time a given stress has been applied.

Rheoptic fluids have an effective viscosity that increases as a function of the time a given stress has been imposed. Rheoptic fluids are fairly rare, a good example is whipped cream.

Thixotropic fluids have an effective viscosity that decreases as a function of the time a given stress has been imposed. Examples of thixotropic fluids include certain types of mud and clay

Shear-stress dependent viscosity Fluids which are shear-stress dependent are categorized as either dilatant (viscosity increases with increased stress), or shear-thinning (viscosity decreases with increased stress).

Dilatant (or shear-thickening) fluids are highly counter-intuitive in that their effective viscosity increases when stress is applied. This phenomenon is a hot topic due to potential applications for armor technology and protective clothing, since it would allow the wearer to move with less restriction than a stiff armor, while still being

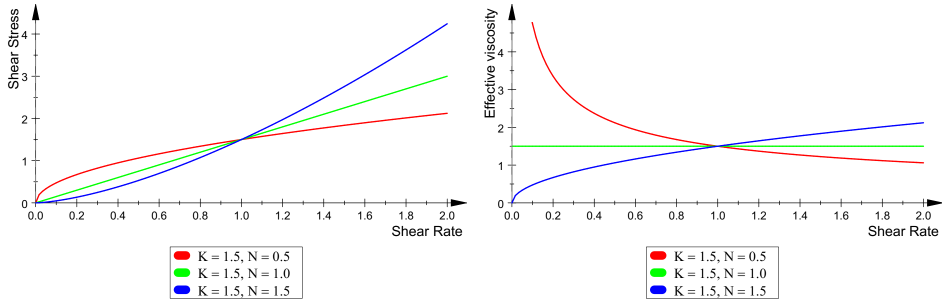


Figure 3.5: Stress-strain curve and effective viscosity for a power-law fluid with a *flow consistency index* of 1.5 and a *flow behavior index* of 0.5(shear-thinning), 1.0(Newtonian) and 1,5(shear-thickening).

protective against high-velocity impacts. The effect can be readily observed in a mix of cornstarch and water, and has in fact been popularized to the degree where TV shows have demonstrated the effect by filling entire swimming pools with dilatant fluids, and then demonstrating the effect by walking across without sinking, and then sinking when at rest. Other more everday dilatant fluids include,

Shear-thinning fluids are often referred to as pseudoplastic, and can be found in a range of materials such as lava, paint, blood and ketchup.

3.3.1 Power-Law/Ostwald model

A power law fluid is a fairly simple Non-Newtonian model that can be used to approximate the behavior of a real fluid. It is also known as the Ostwald-de Waele law [25]:

$$\mu(|\dot{\gamma}|) = K\,|\dot{\gamma}|^{n-1} \tag{3.10}$$

Where K is the *flow consistency index*, and n is the *flow behavior index*.

This model can be used to simulate both shear-thinning ($n < 1$) and shear-thickening ($n > 1$) fluids. When $n = 1$ the model will obviously reduce to a simple Newtonian fluid.

3.3.2 Cross model

The cross model [26] is similar to the power model, but includes a lower and upper bound on the viscosity, μ_0 and μ_∞. These bounds ensure that the fluid behaves like a Newtonian fluid at low shear rates, and like a power-law fluid at high shear rates.

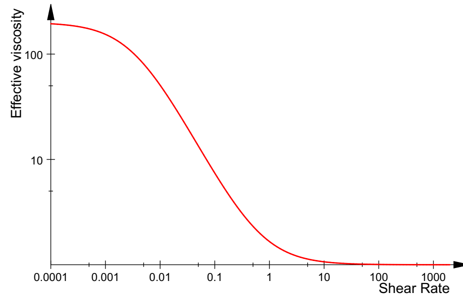


Figure 3.6: Effective viscosity of a cross fluid, logarithmic scale on x-axis. Parameters $\mu_0 = 200$, $\mu_\infty = 1$, $n = 1$ and $K = 300$

$$\mu(|\dot{\gamma}|) = \mu_\infty + \frac{\mu_0 - \mu_\infty}{1 + K(|\dot{\gamma}|)^n}\, 0 \leq \mu_0 \leq \mu_\infty \tag{3.11}$$

3.3.3 Bingham Plastic model

A Bingham plastic (fluid) is a viscoplastic material that flows like a fluid at high stresses, but behaves like a solid material at low stresses (Figure 3.7 on page 22). It has been used to model mud in offshore engineering and other slurries (a suspension of particulate matter in a fluid).

$$\tau = \tau_p + K\dot{\gamma}\ for\ |\tau| \geq \tau_p \tag{3.12}$$

$$\dot{\gamma} = 0\ for\ |\tau| \leq \tau_p \tag{3.13}$$

Where K is the *dynamic viscosity constant* and τ_p is yield stress.

3.3.4 Herschel–Bulkley model

The Herschel-Bulkley is very popular model since it covers both shear-thinning and thickening as well as a yield stress[25].

$$\tau = (K\,|\dot{\gamma}|^{n-1} + \frac{\tau_p}{|\dot{\gamma}|})\dot{\gamma}\ for\ |\tau| \geq \tau_p \tag{3.14}$$

$$\dot{\gamma} = 0\ for\ |\tau| \leq \tau_p \tag{3.15}$$

Where the term $\tau_p$is the yield stress.

Both the power-law and the bingham model are essentially edge cases of this model. If $n = 1$ we get the Bingham model and if $\tau_p = 0$ we get a power-law model. If both $n = 1$ and τ_p =0 the model is reduced to a Newtonian fluid.

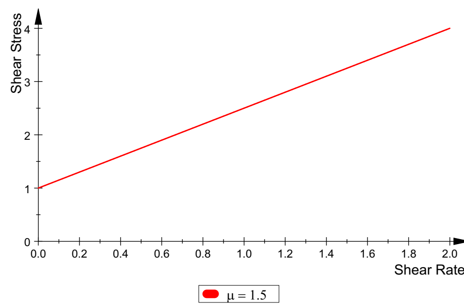


Figure 3.7: Stress-strain curve for a bingham fluid with a *dynamic viscosity constant* of 1.5. Note how it does not pass through the origin due to a nonzero yield stress.

Chapter 4

Smoothed Particle Hydrodynamics

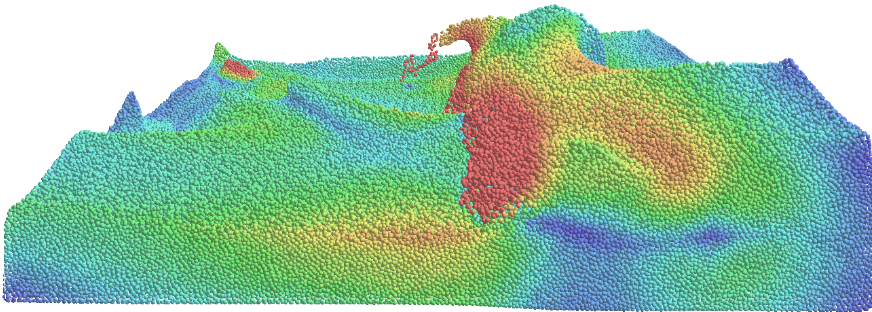


Figure 4.1: Our “Simple” SPH model implementation, 512K particles. Particles are shaded in a hue gradient depending on their velocity.

Smoothed Particle Hydrodynamics is a method for approximating the solution of numerical solutions to the equations of fluid dynamics. The basic idea is to represent a continuous field A(x) by a Monte Carlo sampling of interacting smoothed volumetric particles. It does this by representing the volume as a set of elements(particles) and using these particles as interpolation points for which the properties of the fluid can be calculated [27].

SPH does not use a grid to calculate spatial derivatives, instead these derivatives are found by analytical differentiation of interpolation formulae [28]. How this can be done is not intuitively obvious, but nevertheless a way of doing it was independently proposed by Lucy [29] and Gingold and Monaghan [30].

Smoothed Particle Hydrodynamics was initially developed for use in astrophysics, however it has since been used for modeling a variety of problems in fluid dynamics. Because SPH is basically a method for approximation the continuum equations, it can be used for a wide range of fluid dynamics problems. In particular it can be used

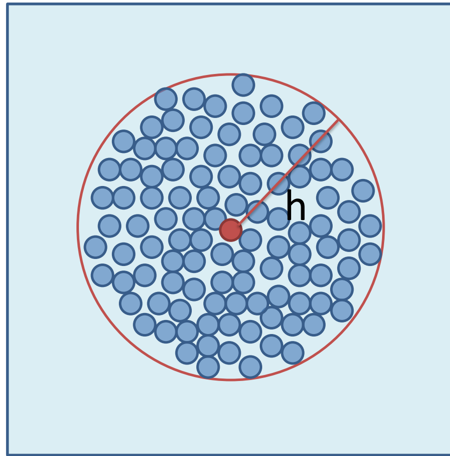


Figure 4.2: The smoothing distance h and the surrounding particles within it.

for problems in incompressible flow by treating the flow as slightly compressible with an appropriate equation of state [31].

By deriving the viscosity and pressure force fields directly from the Navier-Stokes equations it is possible to very efficiently simulate the behavior of fluids [1]. It is also possible to efficiently model the interaction between fluids with different densities [32], which may be important for modelling snow avalanches.

The method has several attractive features, in the context of real-time simulation the ease with which the method can be parallelized is perhaps chief among them. Other advantages include the trivial conservation of mass, high efficiency and easily trackable free surfaces [33]. The handling of complex boundaries of solids is also much easier than with traditional Eulerian simulations, since there is no need for computing complex grids.

Perhaps the biggest disadvantage is that it can be hard or computationally expensive to extract a smooth surface from the particles, the method can also require a fairly large number of particles for realistic results [33].

A good article on the use of SPH in the context of the environmental sciences is the one by Cleary and Prakash [34].

4.1 The Equations of SPH

In SPH the different effects of Navier-Stokes are simulated by a set of forces that act on each particle. These forces are given by scalar quantities that are interpolated at a location $\mathbf{r}$ by a weighted sum of contributions from all surrounding particles within a cutoff distance h in the space Ω [35] (Figure 4.2 on page 24).

In integral form this can be expressed as follows [35]:

$$A_i = \int_\Omega A(\mathbf{r}')W(\mathbf{r}-\mathbf{r}', h)d\mathbf{r}' \quad (4.1)$$

The numerical equivalent to 4.1 is obtained by approximating the integral interpolant by a summation interpolant [35]:

$$A_i = \sum_j A_j V_j W(\mathbf{r}_{ij}, h) \quad (4.2)$$

where j is iterated over all particles, V_j the volume attributed implicitly to the particle j, $\mathbf{r}_{ij}= \mathbf{r}_i - \mathbf{r}_j$ where $\mathbf{r}$ is the position of a particle, and finally A is the scalar quantity that is being interpolated. The summation is over particles which lie within the radius of a circle/sphere centered at $\mathbf{r}_i$.

The following relation between volume, mass and mass-density applies [35]:

$$V = \frac{m}{\rho}$$

where m is the mass and ρ is the mass-density.

Combining this we get the basis formulation of the SPH interpolation function [27]:

$$A_i = \sum_j A_j \frac{m_j}{\rho_j} W(\mathbf{r}_{ij}, h) \quad (4.3)$$

where j iterates over all particles, m_j is the mass of particle j, $\mathbf{r}_j$ its position, ρ_j the mass-density and A_j the scalar quantity at position$\mathbf{r}_j$.

This function can be used to approximate any continuous quantity field, and can be evaluated everywhere in the underlying space [35].

4.1.1 The Smoothing Kernel

The function $W(\mathbf{r}_{ij}, h)$ is the *smoothing kernel*, which is a scalar weighted function. The smoothing kernel can be seen as an analogue to using different difference schemes in finite difference methods. The function uses a position $\mathbf{r}$ and a smoothing length h. This radius can be seen as a cutoff for how many particles will be considered in the interpolation (Figure 4.2 on page 24). This cutoff radius sets $W = 0$ for $|\mathbf{r}_{ij}| > h$.

For a kernel several properties must hold [35], the first one is the normalization condition:

$$\int_r W(\mathbf{r}, h)dr = 1 \quad (4.4)$$

the second condition is the Delta function property which can be observed when the smoothing length approaches zero:

$$\lim_{h \to 0} W(\mathbf{r}_{ij}, h) = \delta(\mathbf{r}_{ij}) \tag{4.5}$$

where δ is Dirac's delta function.

The third condition is the compact condition which ensures that only particles inside the smoothing length are considered:

$$W = 0 \ when \, |\mathbf{r}_{ij}| > h \tag{4.6}$$

In addition a smoothing kernel should also be positive. If the kernel is both even ($W(\mathbf{r}, h) = W(-\mathbf{r}, h)$) and normalized(4.4) it is of second order accuracy [1].

If W is the delta function, the interpolation function will reproduce A exactly. However in practice this is not very useful since it requires a large smoothing length and thus many particle neighbors. To improve computational performance W is often a function that aims to provide as accurate an interpolation as possible while maintaining a reasonable smoothing length.

In Müller et al. [1] and Desbrun and Gascuel [36] several kernels are introduced for the different fields effects, such as viscosity and pressure. By using different kernels for different effects, each kernel can be optimized for the specific demands of the force that is being interpolated.

4.1.2 Derivatives

In SPH, the derivatives of a function can be obtained by using the derivatives of the smoothing kernel, which results in the Basic Gradient Approximation Formula (BGAF) Colin et al. [37]:

$$\nabla A_i = \sum_j A_j \frac{m_j}{\rho_j} \nabla W(\mathbf{r}_{ij}, h) \tag{4.7}$$

$$\nabla^2 A_i = \sum_j A_j \frac{m_j}{\rho_j} \nabla^2 W(\mathbf{r}_{ij}, h) \tag{4.8}$$

These formulations can unfortunately produce spurious results, and several corrected formulations have been developed. One of these is the Difference Gradient Approximation Formula (DGAF) Colin et al. [37]:

$$\nabla A_i = \frac{1}{\rho_i} \sum_j m_j (A_j - A_i) \nabla W(\mathbf{r}_{ij}, h) \tag{4.9}$$

Which has the advantage that the force vanishes exactly when the pressure is constant [28].

Gradient Symmetrization The forces between two particles must observe Newtons Third Law; for every action, there is an equal and opposite reaction. Pairwise forces must be equal in size with opposite sign ($f_i = -f_j$). This means that the differentials in the Navier-Stokes equations that create these forces must be symmetrized.

Monaghan [38] developed a symmetrization, referred to as the Symmetric Gradient Approximation Formula (SGAF). This formulation conserves linear and angular momentum exactly but is not as accurate as DGAF [37]. It is commonly used for the pressure gradient.

$$\nabla A_i = \rho_i \sum_j m_j (\frac{A_i}{\rho_i^2} + \frac{A_j}{\rho_j^2}) \nabla W(\mathbf{r}_{ij}, h) \tag{4.10}$$

For a comparison of the BGAF, DGAF and SGAF formulations please see [37].

An alternative symmetrical formulation [39]:

$$\nabla A_i = \sum_j \frac{m_j}{\rho_j} (A_i + A_j) \nabla W(\mathbf{r}_{ij}, h) \tag{4.11}$$

For a complete derivation as well as equivalent formulations for gradient of a vector, dot product(divergence), tensor product and cross product please refer to [40].

Laplacian Correction The basic formulation of the Laplacian has been found to be somewhat unstable under certain conditions, and there exist a wide range of possible corrections.

Shao and Lo [41] developed a correction well suited for the correction of the Laplacian in the viscous force:

$$\nabla \cdot \left(\frac{1}{\rho} \nabla A \right) = \sum m_j \frac{8}{(\rho_i + \rho_j)^2} \frac{(A_i - A_j) \cdot \nabla W(\mathbf{r}_{ij}, h)}{|\mathbf{r}_{ij}|^2 + \eta^2} \tag{4.12}$$

Where η is a small number introduced to keep the denominator non-zero, usually equal to $0.1h$ where h is the smoothing length.

For a more comprehensive review of SPH please refer to [40, 39].

4.2 Lagrangian Fluid Dynamics

The terms of the Lagrangian formulation (3.1.2.2) of Navier-Stokes can be modelled using SPH by application of the SPH interpolation function (Equation 4.1).

4.2.1 Density

The SPH formulations for the Navier-Stokes forces depends on the mass-density for each particle, and as such the calculation of the density is the first step in the SPH algorithm.

By setting the density (ρ) in the Navier-Stokes equations to be the interpolated value (A) in the SPH interpolation function (Equation 4.3) we get the following equation[1]:

$$\rho_i = \sum_j m_j W(\mathbf{r}_{ij}, h) \tag{4.13}$$

There exist alternative formulations of the mass density approximation, one alternative is to use the SPH version of the continuity equation:

$$\frac{d\rho_i}{dt} = \rho_i \sum_j \frac{m_j}{\rho_j} (\mathbf{v}_i - \mathbf{v}_j) \cdot \nabla W(\mathbf{r}_{ij}, h) \tag{4.14}$$

where $\mathbf{v}_i$ and $\mathbf{v}_j$ is the velocity at particles i and j respectively.

4.2.2 Incompressibility

Using SPH we can find the mass-density ρ, but we also need to calculate the pressure of the fluid. The SPH formulation is inherently compressible. When simulation an incompressible fluid it is necessary to enforce the incompressibility by additional calculations.

Poisson equation It is possible to directly solve the Poisson pressure equation $\nabla^2 P = \rho \frac{\nabla \mathbf{v}}{\Delta t}$ directly [42, 43, 44, 45, 23], which means it is possible to achieve (near) incompressibility . This is unfortunately computationally expensive and though this method makes it possible to use large time steps, it also carries a higher cost per time step.

Equation of State In this thesis, we use an equation of state to achieve weak compressibility [27, 46, 47], which involves a stiff equation of state. This has the disadvantage that we need to use small time steps, fortunately the cost of each time step is also low.

An Equation Of State (EOS),is a relation between state variables, simply put it defined a physical state of matter as a relation to another set of physical conditions.

By using an EOS we can define pressure P as a function of the mass-density ρ.

There exist several possible EOS-formulations for incompressibility, Müller et al. [1] uses the ideal gas state equation:

$$P = k\rho \tag{4.15}$$

Where k is a constant that depends on the temperature. Adding a rest density ρ_0 that increases the numerical stability of the system:

$$P = k(\rho - \rho_0) \tag{4.16}$$

Harada et al. [48] add yet another compensating parameter, the rest pressure P_0:

$$P = P_0 + k(\rho - P_0) \tag{4.17}$$

An alternative equation is the Tait equation [47]:

$$P = B\left(\left(\frac{\rho}{\rho_0}\right)^{\gamma} - 1\right) \tag{4.18}$$

Where B is a pressure constant related to the bulk modulus of elasticity of the fluid, ρ_0 the reference/rest density, usually taken as the density of the fluid at the free surface, γ is the polytrophic constant, usually between 1 and 7.

Enforcing compressibility through an EOS is not ideal since the EOS is essentially a stiff equation, which can introduce instability in the system. In addition using an EOS will cause compressibility, the amount dependent on the stiffness (values of the constants).

When using an EOS it is important to be aware of the above, and to carefully balance the timestep and the EOS constants such that the desired behavior is achieved. For an interactive simulation compressibility may not be a problem, and a large timestep highly desirable.

4.2.3 Pressure

The application of the SPH interpolation function 4.1 to the pressure term $-\frac{1}{\rho}\nabla p$ of the Navier-Stokes equations (3.6) results in the following equation [1]:

$$\mathbf{f}_i^{pressure} = -\frac{1}{\rho}\nabla P = -\frac{1}{\rho_i}\sum_{j\neq i} m_j \frac{P_j}{\rho_j}\nabla W(\mathbf{r}_{ij}, h) \tag{4.19}$$

However this force is not symmetric, so Müller et al. [1] presents the following symmetrization which is well suited for the purposes for speed and stability:

$$\mathbf{f}_i^{pressure} = -\frac{1}{\rho}\nabla P = -\frac{1}{\rho_i}\sum_{j\neq i} m_j \frac{P_i + P_j}{2\rho_j}\nabla W(\mathbf{r}_{ij}, h) \tag{4.20}$$

Another well-known symmetrization that is well suited for accuracy, since it preserves linear and angular momentum exactly [38, 49]:

$$\mathbf{f}_i^{pressure} = -\frac{1}{\rho}\nabla P = -\sum_{j\neq i} m_j (\frac{P_i}{\rho_i^2} + \frac{P_j}{\rho_j^2})\nabla W(\mathbf{r}_{ij}, h) \tag{4.21}$$

4.2.4 Viscous Stresses

SPH was originally formulated for astrophysics simulations, and it was in that context an *artificial viscosity* Π was first added to the momentum equation:

$$\frac{d\mathbf{v}_i}{dt} = -\sum m_j \left(\frac{P_i}{\rho_i^2} + \frac{P_j}{\rho_j^2} + \Pi_{ij}\right)\nabla W(\mathbf{r}_{ij}, h)$$

For astrophysics it has been used for modelling strong shocks, but this *artificial viscosity* has also been used to model real viscous terms. This approach to viscosity modelling has however been shown to produce incorrect velocity profiles in some situations [23] .

There now exist alternative ways of modelling viscosity.

4.2.4.1 Newtonian Stresses

Using the Newtonian incompressible formulation of the Navier-Stakes conservation-of-momentum equation (3.6) it is fairly easy to formulate an exact SPH approximation to the viscosity force.

The application of the SPH interpolation function (Equation 4.1) to the viscosity term $\frac{\mu}{\rho_i}\nabla^2 v$ of the Navier-Stokes equations for incompressible flow (Equation 3.1.2.3) results in the following (symmetrized) equation [1]:

$$\mathbf{f}_i^{viscosity} = \frac{\mu}{\rho_i}\nabla^2\mathbf{v} = \frac{\mu}{\rho_i}\sum_{i\neq j}\frac{m_j}{\rho_j}{}_j\,(\mathbf{v}_j - \mathbf{v}_i)\nabla^2 W(\mathbf{r}_{ij}, h) \tag{4.22}$$

This exact solution relies on a correct solution to the Laplacian of the smoothing kernel, which is currently a weak point of SPH because this second derivative is very sensitive to particle disorder.

Müller et al. [1] solves this problem by using a specialized smoothing kernel (4.3.2), designed to prevent particle instability, which is sufficient for non-rigorous simulations.

Using a a corrected calculation for the Laplacian can more properly correct this problem (Equation 4.12):

$$\mathbf{f}_i^{viscosity} = \frac{\mu}{\rho_i}\nabla^2\mathbf{v} = \sum m_j \frac{4(\mu_i + \mu_b)\mathbf{r}_{ij} \cdot \nabla W(\mathbf{r}_{ij}, h)}{(\rho_i + \rho_j)^2(|\mathbf{r}_{ij}|^2 + \eta^2)}$$

4.2.4.2 Non-Newtonian Stresses

For a Non-Newtonian fluid the calculation of viscosity is more complex. In order to properly calculate both isotropic and deviatoric stresses a much more advanced calculation of shear stresses in the fluid is necessary.

Hosseini et al. [23] demonstrates a SPH formulation which incorporates such a deviatoric stress.

As we recall from Section 3.2 the classical constitutive law for Generalized Newtonian fluids is given by Equation 3.8:

$$\tau = \mu(|\dot{\gamma}|)\dot{\gamma}$$

where τ is the *viscous stress tensor* and $\dot{\gamma}$ the *shear strain rate* (deformation). The term $|\dot{\gamma}|$ is the second invariant of the stress tensor, which is the magnitude of the tensor and thus a measure of the intensity of deformation [23]:

$$|\dot{\gamma}| = \sqrt{trace(\dot{\gamma})^2}$$

Where *trace* is the matrix operation ($trace(A) = \sum_{i=1}^{n} a_{ii}$).

It is possible to express the *shear stress* $\dot{\gamma}$ using the *velocity tensor field:*

$$\nabla\mathbf{v} = \begin{bmatrix} \frac{\partial v_x}{\partial x} & \frac{\partial v_x}{\partial y} & \frac{\partial v_x}{\partial z} \\ \frac{\partial v_y}{\partial x} & \frac{\partial v_y}{\partial y} & \frac{\partial v_y}{\partial z} \\ \frac{\partial v_z}{\partial x} & \frac{\partial v_z}{\partial y} & \frac{\partial v_z}{\partial z} \end{bmatrix}$$

The velocity tensor field contains both a symmetric and an nonsymmetric part. The nonsymmetric part is the rotation velocity tensor and should not affect the viscosity since it does not contribute to fluid deformation. The symmetric part (deformation) is defined as follows [25]:

$$\dot{\gamma} = \frac{1}{2}\left(\nabla\mathbf{v} + (\nabla\mathbf{v})^T\right) \tag{4.23}$$

Where $\nabla\mathbf{v}$ is the *velocity stress tensor* and $(\nabla\mathbf{v})^T$ is simply the tensor transposed. This *rate-of-strain* tensor is known as Cauchy's strain tensor, the linear strain tensor, or the small strain tensor.

$$\dot{\gamma} = \frac{1}{2}\left(\nabla\mathbf{v} + (\nabla\mathbf{v})^T\right) = \begin{bmatrix} \frac{\partial v_x}{\partial x} & \frac{1}{2}\left(\frac{\partial v_x}{\partial y} + \frac{\partial v_y}{\partial x}\right) & \frac{1}{2}\left(\frac{\partial v_x}{\partial z} + \frac{\partial v_z}{\partial x}\right) \\ \frac{1}{2}\left(\frac{\partial v_y}{\partial x} + \frac{\partial v_x}{\partial y}\right) & \frac{\partial v_y}{\partial y} & \frac{1}{2}\left(\frac{\partial v_y}{\partial z} + \frac{\partial v_z}{\partial y}\right) \\ \frac{1}{2}\left(\frac{\partial v_z}{\partial x} + \frac{\partial v_x}{\partial z}\right) & \frac{1}{2}\left(\frac{\partial v_z}{\partial y} + \frac{\partial v_y}{\partial z}\right) & \frac{\partial v_z}{\partial z} \end{bmatrix}$$

The spatial derivatives in the *velocity stress tensor* $\nabla\mathbf{v}$ can be either computed directly using SPH [23] or more concisely by using [22] :

$$\nabla\mathbf{v}_i = \sum_j \frac{m_j}{\rho_j}(\mathbf{v}_j - \mathbf{v}_i) \otimes \nabla W(\mathbf{r}_{ij}, h) \tag{4.24}$$

Where $\otimes$ is the outer product and $\nabla\mathbf{v}$ is the *velocity stress tensor* (a 3x3 matrix).

As we recall the constitutive equation for shear stress for a Generalized Newtonian fluid is $\tau = \mu(|\dot{\gamma}|)\dot{\gamma}$. (Equation 3.8).

For Newtonian fluids the following form is recovered [23]:

$$\tau = 2\mu\dot{\gamma}$$

Where μ is simply the constant of *dynamic viscosity,* described in more detail in 3.2 on page 17.

Using these definitions it is also possible to model Non-Newtonian fluids. A fluid with shear-stress dependent viscosity can be modelled by making μ dependent on the *shear strain rate* $\dot{\gamma}$, through a rheological model such as those presented in Section 3.3.

Now that we have a formulation of the *shear stress* τ a SPH approximation to the *viscous stress force* follows naturally from Equation 4.11:

$$\mathbf{f}_i^{stress} = \frac{1}{\rho_i}\nabla \cdot \tau_i = \frac{1}{\rho_i}\sum_{j \neq i} \frac{m_j}{\rho_j}(\tau_i + \tau_j) \cdot \nabla W(\mathbf{r}_{ij}, h) \tag{4.25}$$

4.2.5 External Forces and Boundary Conditions

Various SPH methods can apply several types of external force. Beyond the common forces from gravity and solid boundaries, effects such as buoyancy, surface tension and artificial constructs such as waves can readily be applied.

4.2.5.1 Gravity

Gravity is applied directly to the SPH particles using:

$$f_i^{gravity} = \mathbf{g} \tag{4.26}$$

where $\mathbf{g}$ is the gravitational acceleration.

4.2.5.2 Boundary conditions

There are several ways to handle boundaries.

Perfect reflection This is the simplest way to handle boundaries, where each particle undergoes a (perfect) collision with a boundary and is reflected away. This is a very naive solution and is not very accurate since it does not affect the SPH solution itself, but the particles directly.

Repulsive forces This solution adds a “repulsive force” to each boundary, which pushes particles away from the boundary, this is more accurate than a naive reflection.

Müller et al. [1] implements collisions with the particle positions directly. If a particle collides with a solid object, they are simply pushed out of the object and the velocity that is perpendicular to the object is reflected.

Harada et al. [48] implement a collisions by affecting the pressure and density of the particles. The pressure correction will essentially “push” particles back from the surface. Because a surface will also be an “empty” volume, the density of nearby particles is increased with a “wall weight function” that simulates the affect on density that should result from contact.

Virtual particles This solution adds “virtual” or “ghost” particles Takeda et al. [50], Morris et al. [51], which lie along the boundary and are fixed in space. This solution is highly dependent on the placement of the virtual particles, but can be more accurate since it affects the internal properties of the SPH continuum instead of the positions of the particles themselves. The big disadvantage with this method is that one has to ensure that all boundaries have correctly placed particles, but the advantage is that one can use SPH forces to simulate boundary effects such as friction and no-penetration.

Kernel Sum Deficiency Boundary conditions are problematic in SPH due the kernel sum deficiency which occurs for particles at the boundary. The kernel sum can become deficient when there are no (or few) neighboring particles in the direction of the boundary. This problem is known as the “boundary deficiency problem”. Various corrective solutions to this problem have been proposed [52], such as compensating for the sum deficiency directly for boundary particles only.

For a more comprehensive review of SPH and how boundaries can be handled please refer to Liu and Liu [39].

4.3 Smoothing Kernels

The choice of smoothing kernel can be important for several aspects of a simulation. Obviously the numerical accuracy is highly dependent on the smoothing kernel, and research has shown that certain kernels offer better results than others [53]. The computational efficiency of a kernel can also be significant, higher-order kernels can carry a significantly higher computational cost, which may impose a limit on other parameters of the simulation and thus negating the advantages. In the context of GPU simulations piecewise kernels carry an additional performance impact due to the high cost of branching (Chapter 5).

4.3.1 General Kernels

Several generalized smoothing kernel have been developed throughout the history of SPH.

4.3.1.1 Gaussian

The *Gaussian function* is an obvious choice for a smoothing kernel, it is sufficiently smooth even for high orders of derivatives. It does however carry a large computational cost. It also does not have *compact support* since it does not reach zero for $q > 2$, which means that one would theoretically have to evaluate all the particles in the simulation.

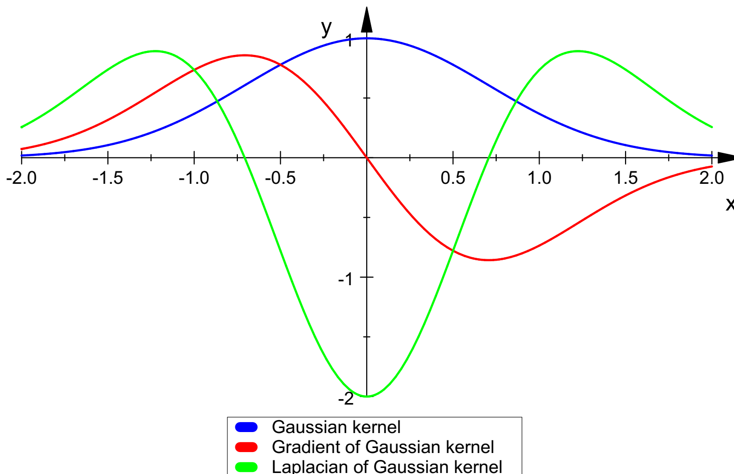


Figure 4.3: The Gaussian kernel and it's derivatives along one axis in a 3-dimensional space with a smoothing distance h=1.

$$W_{gaussian}(\mathbf{r}_{ij},h) = \alpha_D \exp^{-q^2} \quad 0 \leq q \leq 2 \tag{4.27}$$

where $q = \frac{|\mathbf{r}_{ij}|}{h}$ and $\mathbf{r}_{ij}$ is the distance between particles i and j, and α_D is a dimensional factor which is $\frac{120}{\pi^{0.5}}$ in 1D ,$\frac{1}{\pi h^2}$ in 2D and $\frac{1}{\pi^{3/2}h^3}$ in 3D.

4.3.1.2 Piecewise Cubic Spline

The *Cubic Spline* kernel is the most widely used smoothing kernel. It was initially introduced by Monaghan and Lattanzio [49] and offers a reasonable compromise between computational cost and accuracy. In addition it has *compact support.* This kernel is also the one that is most commonly used in SPH literature. The biggest problem with this problem is that the second derivative is not smooth, thus rendering it unsuitable for use in the calculation of the Laplacian.

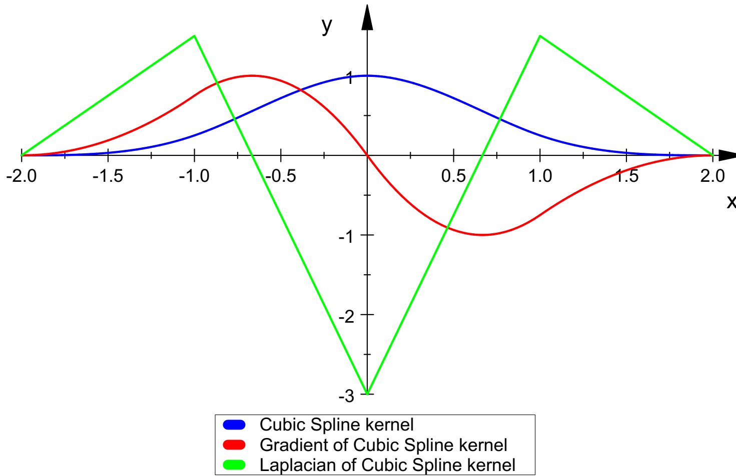


Figure 4.4: The Cubic Spline kernel and it's derivatives along one axis in a 3-dimensional space with a smoothing distance h=1.

$$W_{cubic}(\mathbf{r}_{ij},h) = \alpha_D \begin{cases} 1 - \frac{3}{2}q^2 + \frac{3}{4}q^3 & 0 \leq q \leq 1 \\ \frac{1}{4}(2-q)^3 & 1 \leq q \leq 2 \\ 0 & q \geq 2 \end{cases} \tag{4.28}$$

where $q = \frac{|\mathbf{r}_{ij}|}{h}$ and $\mathbf{r}_{ij}$ is the distance between particles i and j, and α_D is a dimensional factor which is $\frac{10}{7\pi h^2}$ in 2D and $\frac{1}{\pi h^3}$ in 3D.

4.3.1.3 Piecewise Quintic Spline

The quintic spline was used by Morris et al. [51] to simulate low Reynolds number incompressible flow. Unfortunately this kernel is very computationally costly.

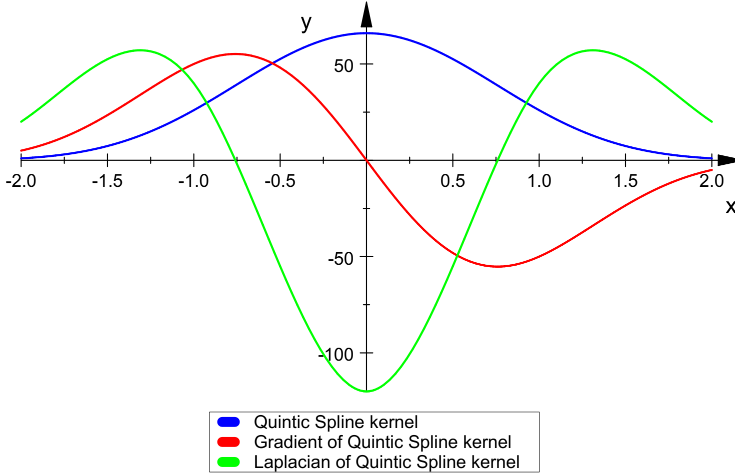


Figure 4.5: The Quintic spline kernel and it's derivatives along one axis in a 3-dimensional space with a smoothing distance h=1.

$$W_{quintic}(\mathbf{r}_{ij,}h) = \alpha_D \begin{cases} (3-q)^5 - 6(2-q)^5 + 15(1-q)^5 & 0 \leq q \leq 1 \\ (3-q)^5 - 6(2-q)^5 & 1 \leq q \leq 2 \\ (3-q)^5 & 2 \leq q \leq 3 \\ 0 & q \geq 3 \end{cases} \tag{4.29}$$

where $q = \frac{|\mathbf{r}_{ij}|}{h}$ and $\mathbf{r}_{ij}$ is the distance between particles i and j, and α_D is a dimensional factor which is $\frac{120}{h}$ in 1D, $\frac{7}{478\pi h^2}$ in 2D and $\frac{3}{359\pi h^3}$ in 3D.

4.3.1.4 Quadratric

This kernel prevents particle clustering in compression problems (e.g no tensile correction is needed). This is because the derivative always increases as particles move closer, and always decreases as they move apart Crespo [53].

$$W_{quadratic}(\mathbf{r}_{ij,}h) = \alpha_D \left\{ \frac{3}{16}q^2 - \frac{3}{4}q + \frac{3}{4} \quad 0 \leq q \leq 2 \right. \tag{4.30}$$

where $q = \frac{|\mathbf{r}_{ij}|}{h}$ and $\mathbf{r}_{ij}$ is the distance between particles i and j, and α_D is a dimensional factor which is $\frac{2}{\pi h^2}$ in 2D and $\frac{51}{4\pi h^3}$ in 3D.

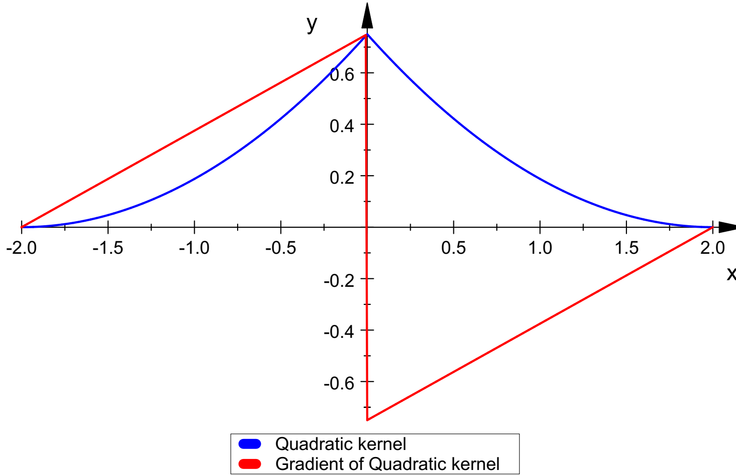


Figure 4.6: The Quadratic kernel and it's derivatives along one axis in a 3-dimensional space with a smoothing distance h=1.

4.3.1.5 Wendland

This kernel Wendland [54, 55] has been found to give the best compromise between accuracy and computational cost Crespo [53].

$$W_{quadratic}(\mathbf{r}_{ij},h) = \alpha_D \begin{cases} \frac{3}{16}q^2 - \frac{3}{4}q + \frac{3}{4} & 0 \leq q \leq 2 \end{cases} \tag{4.31}$$

4.3.1.6 Quartic

Liu et al. [56] has constructed a new smoothing kernel. This kernel satisfies the normalization condition and both the function and the first derivative have compact support. It is very close to the commonly used cubic spline, but has several advantages over it. It is more stable [56] and it is not piecewise, which means there is no need for branching in the evaluation of it.

$$W_{quartic}(\mathbf{r}_{ij},h) = \alpha_D \begin{cases} (\frac{2}{3} - \frac{9}{8}q^2 + \frac{19}{24}q^3 - \frac{5}{32}q^4 & 0 \leq q \leq 2 \\ 0 & q \geq 2 \end{cases} \tag{4.32}$$

where $q = \frac{|\mathbf{r}_{ij}|}{h}$ and $\mathbf{r}_{ij}$ is the distance between particles i and j, and α_D is a dimensional factor which is $\frac{1}{h}$ in 1D, $\frac{15}{7\pi h^2}$ in 2D and $\frac{315}{208\pi h^3}$ in 3D.

The relevant gradient and Laplacian versions of these kernels can be found in Appendix Chapter A.

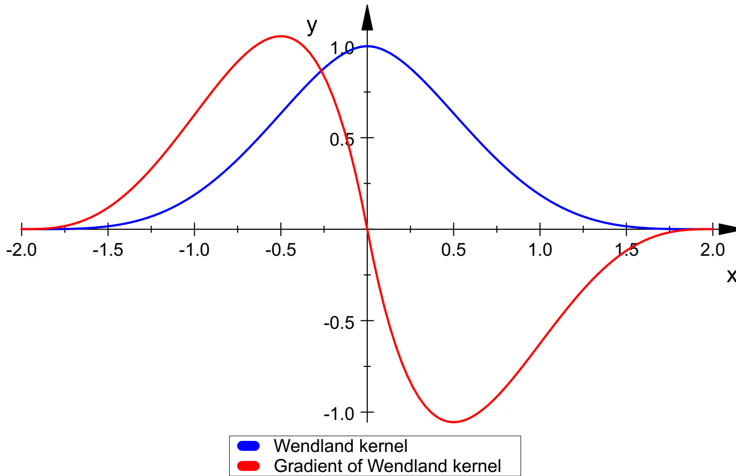


Figure 4.7: The Wendland kernel and it's derivatives along one axis in a 3-dimensional space with a smoothing distance h=1.

4.3.2 Specialized Kernels

It is possible to create specialize smoothing kernels in order to increase performance and stability. These kernels are generally not as accurate, and may introduce errors in the simulation, but can significantly improve the overall stability of the system. These kernel can also be less costly to compute.

4.3.2.1 Mass-Density

In Müller et al. [1] the W_{poly6} kernel is used for all interpolation except viscosity and pressure. This kernel is highly performance efficient, since it is not piecewise(no need for conditionals) and because $|\mathbf{r}|$ only appears squared.

$$W_{poly6}(\mathbf{r}, h) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - |\mathbf{r}|^2)^3 & 0 \leq |\mathbf{r}| \leq h \\ 0 & otherwise \end{cases} \tag{4.33}$$

Where $\mathbf{r}$ is the distance between two particles, $|\mathbf{r}|$ is the length of $\mathbf{r}$ and h is the smoothing distance.

4.3.2.2 Pressure

The reason W_{poly6} is not used for pressure is that it has a vanishing gradient close to zero. This means that particles in high pressure areas will not repulse each other, which can lead to clustering. For this reason Müller et al. [1] uses a different

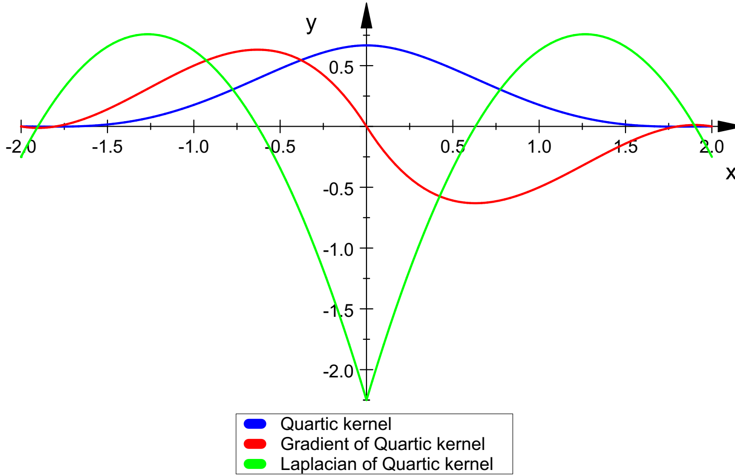


Figure 4.8: The Quartic kernel and it's derivatives along one axis in a 3-dimensional space with a smoothing distance h=1.

smoothing kernel, from Desbrun and Gascuel [36]. The W_{spiky} (4.34) kernel solves the problem by ensure that gradient does not vanish near zero.

$$W_{spiky}(\mathbf{r}, h) = \frac{15}{\pi h^6} \begin{cases} (h - |\mathbf{r}|)^3 & 0 \leq |\mathbf{r}| \leq h \\ 0 & otherwise \end{cases} \tag{4.34}$$

4.3.2.3 Viscosity

Viscosity can be thought of as smoothing(diffusing) the velocity field. In the context of fluid simulation the smoothing of two velocities should lead to a reduction in their relative velocity by conversion to heat. However for particles that are close together the Laplacian of W_{poly6}(4.33) is negative, leading to a relative increase in their velocities[1]. This is a problem in a fluid simulation and is why a special kernel is introduced for the interpolation of viscosity:

$$W_{viscosity}(\mathbf{r}, h) = \frac{15}{2\pi h^3} \begin{cases} -\frac{|\mathbf{r}|^3}{2h^3} + \frac{|\mathbf{r}|^2}{h^2} + \frac{h}{2|\mathbf{r}|} - 1 & 0 \leq |\mathbf{r}| \leq h \\ 0 & otherwise \end{cases} \tag{4.35}$$

The relevant gradient and Laplacian versions for some of these kernels can be found in Appendix A.

4.3.3 Tensile Instability

In SPH *tensile instability* refers to an inherent instability in SPH which occurs under certain conditions. This condition can be compensated for and corrected

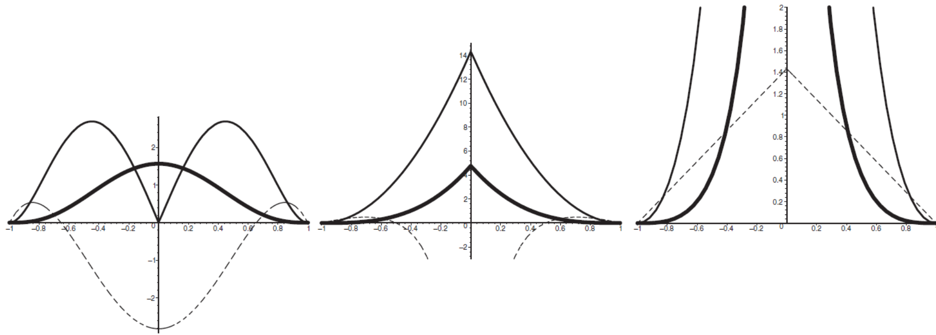


Figure 4.9: The three specialized smoothing kernels W_{poly6}, W_{spiky} and $W_{viscosity}$. The thick lines show the kernels, the thin lines their gradients in the direction toward the center, and the dashed lines the Laplacian. The diagrams are differently scaled. They are plotted along one axis in a 3-dimensional space with a smoothing distance h = 1. Reprinted from Müller et al. [1] with permission from Matthias Müller.

using a tensile correction term Crespo [53].

Chapter 5

Parallel Computing and the Graphics Processing Unit

This chapter gives an overview of different parallel computing models, the emergence of the GPU as a computing platform and the NVIDIA CUDA computing architecture for GPUs. Most of the material in this chapter was covered in our earlier work [4].

5.1 Parallel Computing

Parallel computing is a form of computation where many calculations are performed simultaneously. There exist several different forms of this principle, ranging from parallelism on the bit-level (operating on multiple bits at the same time, e.g. all modern computers), instruction-level parallelism (e.g. pipelining in modern CPUs), data parallelism (same calculations on different data) and task parallelism (different calculations on the same data). These different forms were first classified by Michael J. Flynn, who created what is now known as Flynn's taxonomy:

- **SISD**: Single Instruction, Single Data stream
- **SIMD**: Single Instruction, Multiple Data streams
- **MISD**: Multiple Instruction, Single Data stream
- **MIMD**: Multiple Instruction, Multiple Data streams

The *Graphics Processing Unit* (GPU) is a specialized accelerator, used for the acceleration of graphics. Modern GPUs can be found in embedded systems, mobile phones, personal computers, game consoles and recently in large clusters. Their parallel nature is designed to efficiently handle large amounts of floating-point operations.

A recent development is to use these accelerators as general purpose computing devices. General-Purpose Computing on Graphics Processing Units (GPGPU) has led to a small revolution in the parallel power available to consumers and researchers alike. The Floating Point Operations Per Second (FLOPS) on the

GPU has quickly outpaced the CPU (Figure 5.2 on page 43). Another advantage of the GPU is the massive amount of memory bandwidth available (Figure 5.1 on page 42). These two factors combine to make the GPU extremely interesting for the simulation of computationally intense models. A modern GPU allows for real-time implementations of physical models that have previously been considered the domain of large clusters.

This project will focus on parallelism in the form of SIMD (or SIMT). SIMT is a variation on SIMD, introduced by NVIDIA. In an NVIDIA GPUs threads are grouped together in SIMT *warps*. SIMT allows for all threads to diverge, but threads within the same warp must converge after branching. The SIMT model allows for control-flow on a thread-level, a significant addition in terms of programmability.

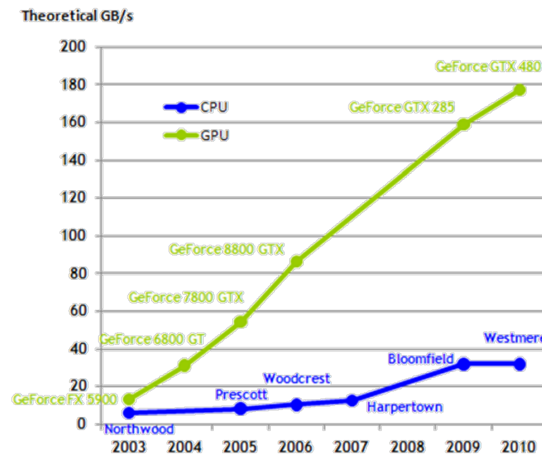


Figure 5.1: The change in memory bandwidth of modern GPUs compared to CPUs. Reprinted from [2], with permission from NVIDIA.

5.2 General Purpose Computing on Graphics Processing Units (GPGPU)

With GPUs now commonly found in most personal computers, a small revolution is taking place due to the sudden increase in parallelism that is possible for most personal computers. Modern GPUs can provide a massive boost in computing power for problems that are parallelizable, for the general consumer problems such as video encoding, image recognition and physics in games are currently seeing a large jump in performance.

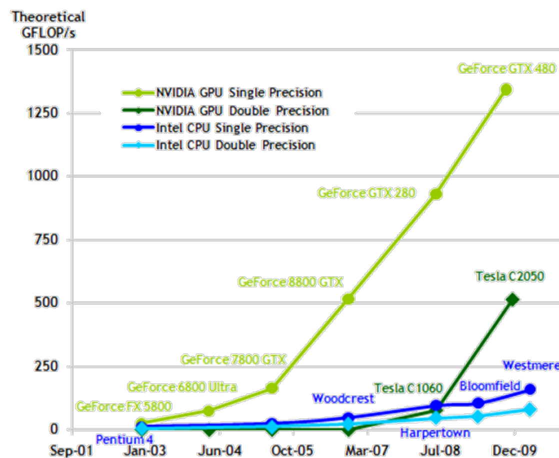


Figure 5.2: The development in peak Floating Point Operations Per Second (FLOPS) for GPUs and CPUs. Reprinted from [2], with permission from NVIDIA.

For the scientific community the recent developments in GPGPU is also creating new possibilities, specifically for the *High Performance Computing* (HPC) community. Problems such as molecular dynamics, fluid simulation, medical imaging, cosmological simulations and many others are seeing large improvements in performance. Such improvements often mean that scientists can quickly and more easily visualize their problems. Having to wait for simulation results is often a tedious and frustrating part of running simulations on large clusters.

Certain problems can see improvements on an order of a magnitude and above [57]. However it is important to keep in mind that the GPU is not a magical cure-all for performance limitations. Modern GPUs can provide massive improvements in parallelism and memory bandwidth, but there are also many limitations to what a GPU can do efficiently.

The biggest bottleneck for the GPU is due to the limitations of the CPU-GPU PCI Express bus. While the internal memory bandwidth of most modern GPUs is more than 100 GB/s, the PCI-E 2.0 x16 bus is less than 10 GB/s. For problems where it is not possible to fit everything into the GPU memory, this can be devastating for performance. The NVIDIA GeForce GTX 470 (Figure 5.5 on page 46) has 133.9 GB/s internal memory bandwidth, while the NVIDIA Tesla S1070 can boast a massive 410 GB/s of internal memory bandwidth.

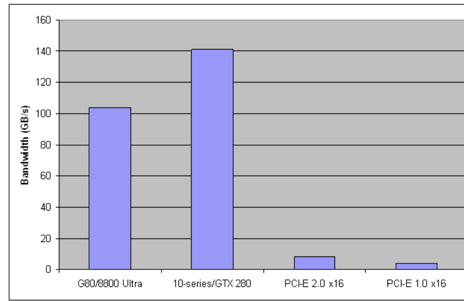


Figure 5.3: Bandwidth available to the device from various sources. Reprinted from [3], with permission from Rob Farber.

5.3 Compute Unified Device Architecture (CUDA)

Using programming languages such as C for CUDA, OpenCL and DirectCompute, it is possible to run general purpose code on the GPU. While it was previously possible to use the old graphics-centered paradigm of shaders to perform computations, it was very difficult and limited.

In this thesis, we will focus on the Compute Unified Device Architecture (CUDA) by NVIDIA since it is the most mature of all the GPU-computing technologies.

CUDA is a parallel computing architecture developed by NVIDIA. Compared to the earlier GPGPU programming models, where the graphics API was used, CUDA offers a number of advantages. CUDA allows developers to use shared memory, scattered reads and integer and bitwise operations. It is also much easier to program, since the entire API has been designed for computation, not graphics.

Much of the information in this section is from the NVIDIA CUDA Programming Guide [58].

5.3.1 CUDA Hardware Model

On March 26, 2010 NVIDIA launched the new GF100 architecture. This architecture was codenamed Fermi and features a number of novel features for GPUs. In particular it has a number of features which makes it more powerful for general computations.

The old GT200-series was the 10th generation of GPUs from NVIDIA, it consists of 1.4 billion transistors, in sharp contrast to the new Fermi chip. The Fermi chip consists of 3.0 billion transistors, more than double that of the previous architecture.

To describe the general layout of a CUDA-enabled GPU we will describe the Tesla C1060. The Tesla C1060 is built on the GT200 architecture and consists of 240 cores. These cores are organized into 30 streaming multiprocessors (SMs), where

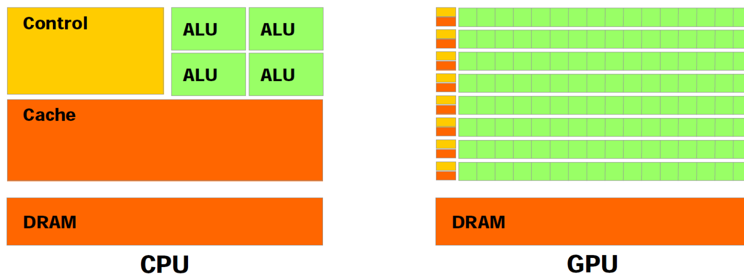


Figure 5.4: The difference between a CPU and GPU in the distribution of transistors. Reprinted from [2], with permission from NVIDIA.

each of them contain 8 single precision (SP) and one double precision (DP) core. The Tesla C1060 can at peak provide 933 SP FLOPS and 78 DP FLOPS.

Each of the SMs have 2 special function units (SFUs) that contain hardware for accelerating often-used mathematical functions, as well as a multithreaded function unit responsible for creating, managing and executing concurrent threads in hardware with zero scheduling overhead [58].

The CUDA hardware model can be summarized as a set of SIMT multiprocessors with on-chip shared memory. A thread is mapped directly to a scalar processor core, and the multiprocessor unit work with 32 threads at a time in a grouping called a *warp*. The SIMT model allows for threads inside a warp to diverge through data-dependent conditional branches (control-flow). When a branch diverges performance can suffer because the warp must wait for the threads to converge again. For this reason control-flow should be kept at an absolute minimum. As long as all the threads in a warp do not diverge performance is not impacted significantly.

The distribution of transistors on the GPU is significantly different from the CPU, with the GPU allocating a significantly large percentage to arithmetic logic units (ALUs) (Figure 5.4 on page 45).

5.6

Figure 5.5: The NVIDIA GeForce GTX 470

	Tesla C1060	Tesla C2050(Fermi)
Streaming Processor Cores	240	448
Frequency of processor cores (MHz)	1300 GHz	1150
Single Precision GFLOPS	933	1030
Double Precision GFLOPS	78	515
Total Dedicated Memory (MB)	4GB (GDDR3)	3GB (GDDR5)
Memory Clock (MHz)	800	1500
Memory Interface	512-bit	384-bit
Memory Bandwidth (GB/sec)	102	144
Max Power Consumption	187.8 W	247 W
Number of Transistors	1.4 billion	3 billion
Generation	2008	2010

	GTX 260	GTX 470(Fermi)
Streaming Processor Cores	192	448
Frequency of processor cores (MHz)	1242	1215
Single Precision GFLOPS	715	933
Double Precision GFLOPS	78	78
Total Dedicated Memory (MB)	896MB (GDDR3)	1280MB (GDDR5)
Memory Clock (MHz)	999	1674
Memory Interface	448-bit	320-bit
Memory Bandwidth (GB/sec)	111.9	133.9
Max Power Consumption	182 W	187.8 W
Number of Transistors	1.4 billion	3 billion
Generation	2008	2010

Table 5.1: Important specifications of NVIDIA GPUs.

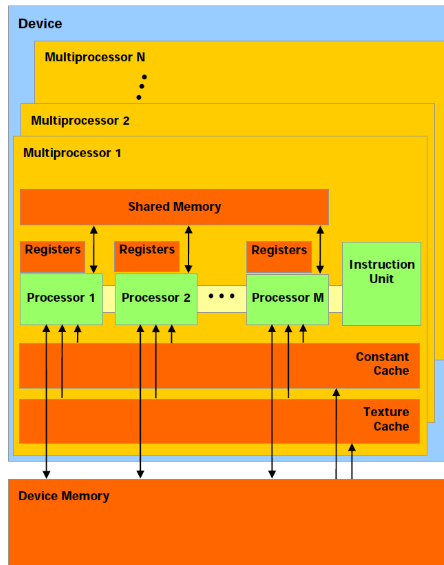


Figure 5.6: CUDA Hardware Model. Reprinted from [2], with permission from NVIDIA.

5.3.2 CUDA Programming Model

The CUDA programming model is summarized fairly quickly;

- Parallel portions of an application is executed on the device as kernels.
- One function at a time([1])
- Many threads run the same function/code in parallel acting on different pieces of data.

However the devil is in the details, and there are certainly a lot of details to keep in mind.

The core concept in CUDA is the *kernel*, a function that is executed by many threads concurrently, where each thread is given an unique index. This index can be used compute memory addresses and make control decisions.

To facilitate cooperation and dependencies between threads, CUDA introduces the concept of *thread blocks*. When launching a CUDA kernel, one does not just specify the number of threads (the *block size)*, but also the number of thread blocks (the *grid size*) (Figure 5.7 on page 48).

[1] Fermi has support for concurrent execution of kernels

Threads within a block can cooperate via *shared memory*. Shared memory is much faster than the device memory, and can be seen as a user-managed cache.

The hardware is free to schedule thread blocks on any multiprocessor, and it is this relaxation between the hardware and software that makes it possible to efficiently and seamlessly utilize the GPU as a general purpose stream processor.

Threads within a block can also synchronize access to global memory by fulfilling certain *coalescing* rules. If memory accesses are not coalesced, a significant performance hit is usually incurred.

Occupancy When launching kernels on the GPU one must specify the block and grid size. These two parameters decide the *occupancy* of the GPU. The multiprocessor occupancy is the ratio of active warps to the maximum number of warps supported on a multiprocessor of the GPU, essentially a measure of how much of the GPU is utilized.

Depending on the resource usage of a kernel (registers, shared and constant memory) it may not be possible to fully utilize all the multiprocessors on the GPU.

As long as the kernel is bandwidth bound, improving occupancy will generally lead to higher performance.

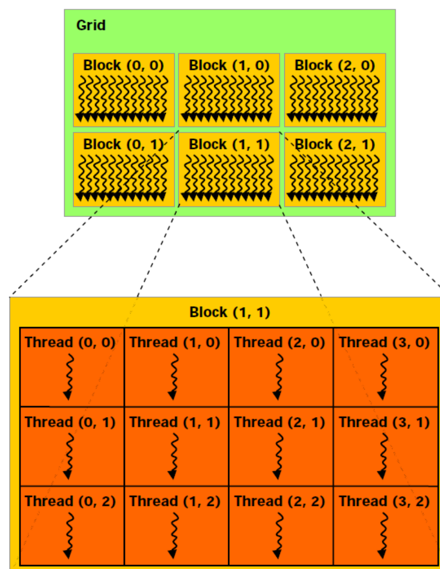


Figure 5.7: CUDA Grid, block and thread model. Reprinted from [2], with permission from NVIDIA.

5.3.3 CUDA Memory Model

In the CUDA memory model there are several distinct types of memory. Memory access is often a major bottleneck for CUDA applications and as such it is important to fully understand the various limitations and advantages of the different memory types.

There are 3 major separate hardware memories:

1. The host (CPU) memory
2. The global device (GPU) memory
3. The on-chip device (GPU) memory.

These different hardware memories are separated into several distinct memories in the memory model, where memories using the same hardware can have different characteristics due to effects from *coalescing, caching and bank conflicts*.

Coalescing is the effect that happens when the CUDA hardware can collect memory request from threads in a wrap and handle them as a single group, enabling a massive boost to the memory bandwidth.

Bank conflicts apply to shared memory, which is divided into banks.

5.3.3.1 Coalescing

In order to get coalescing memory reads from the global memory, a number of restrictions apply.

- Data must be read in 4,8 or 16-byte words.
- Structures larger than 16 bytes are automatically broken up into several load instruction, the user can specify alignment and padding by using alignment specifiers.
- Threads in a half-warp should access memory simultaneously.

For compute capability 1.0/1.1 devices the following restrictions also apply:

- Threads must access the words in sequence.
- All words must lie in the same segment of size equal to the memory transaction size.

For compute capability 1.2(and above) these restrictions are relaxed, threads can access memory in any order (and can access the same address).

For more details on coalescing please refer to the NVIDIA CUDA Programming Guide [58].

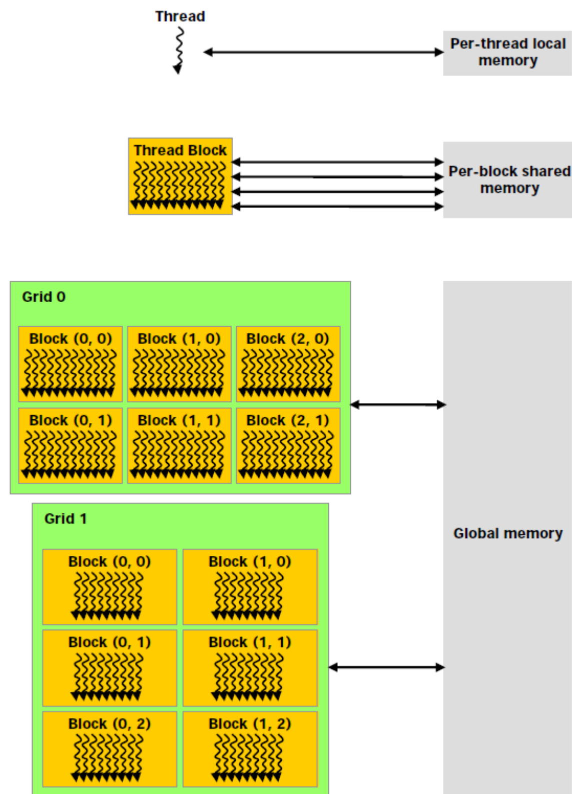


Figure 5.8: A simplified view of the the CUDA memory hierarchy. Reprinted from [2], with permission from NVIDIA.

5.3.3.2 Memory Types

In the following table the properties and relative performance of the different memory types are enumerated. It is important to note that most of these memories are merely abstractions, not real hardware.

Type	Access	Visibility	Lifetime	HW placement
Registers	RW	Per thread	Thread	On-chip
Shared memory	RW	Per block	Block	On-chip
Local memory	RW	Per thread	Thread	Device memory
Global (device) memory	RW	Global	Manual	Device memory
Constant memory	R	Global	Manual	Device memory
Texture memory	R	Global	Manual	Device memory
Host (CPU) memory	RW	NA[a]	Manual	Host memory

[a]Host memory can only be accessed directly from the GPU by utilizing the new *zero-copy* feature introduced in CUDA 2.2

Table 5.2: CUDA memory types. Data from [3].

Type	Cached	Performance	Cycles
Registers	no	fast	Zero to one
Shared memory	no	fast, coalescing, banks	Single[a]
Local memory	no[b]	slow	Hundreds
Global (device) memory	no[c]	slow, coalescing	Hundreds
Constant memory	yes	fast	Single[d]
Texture memory	yes	slow, coalescing	hundreds
Host (CPU) memory	no	slow	hundreds

[a]If no bank conflicts are incurred
[b]Fermi includes an L1 cache that can cache local memory
[c]Fermi includes an L1/L2 cache that can cache global memory
[d]The first cache miss will incur a global memory read, subsequent hits will be a single cycle.

Table 5.3: CUDA memory performance characteristics. Data from [3].

Registers Accessing a register will generally add no extra cycles per instruction. In the current generation of cards (GT200) each MP on the device will have 8192 registers divided among the thread blocks that execute on the MP. Thus the number of registers available in each thread depends on the number of blocks as well as the block size.

Local Memory Access to local memory is as expensive as access to the global memory, but are always coalesced since they are by definition on a per-thread basis. Local memory access willoccur if insufficient register space is available in a function. Use of this memory should obviously be avoided if at all possible.

Constant Memory Constant memory is a cached part of global memory. The first access to this memory will create a cache miss and a read from global memory, but subsequent access will hit the cache and be 1 cycle. The GT200 is equipped with 8KB of constant memory per SM (64KB total), which the new Fermi cards do not improve upon.

Global Memory Global memory is the largest memory available on the device itself. The Tesla C1060 has 4GB of GDDR3, while consumers cards are generally below or around 1GB. Access to the global memory is generally very costly, so care should be taken to optimize such access according to the rules of coalescing (5.3.3.1).

It is possible to apply a cache to global memory called the *texture cache*, so named because it utilizes GPU hardware that is used for accessing textures in memory. The texture cache can generally improve performance, but more so if the memory request are not coalescing, but still have some locality.

Shared Memory The shared memory is an on-chip memory that can be seen as a user-managed cache. The current generation (GT200) is equipped with 64KB, which is not a lot and can be a restriction for several problems. Shared memory is divided into banks, access to banks is governed by specific rules. Provided these rules are followed no bank conflicts will occur and an access will be as fast as a register

Cache In the new GF100 architecture (Fermi) from NVIDIA there are two new caches. The L1 (16KB or 48KB) and the L2 (768KB) cache.

The L2 cache works for global memory reads and the L1 cache works for both local memory (register spills) and global memory reads.

The L1 cache has configurable size for the global and the local memory, either 16/48 or 48/16 for L1 cache or shared memory.

One interesting fact is that the L1 cache has a higher bandwidth than the old texture cache, which means that it is no longer safe to assume that using the texture cache will be always be beneficial. In fact we show that using the texture cache is still important when your memory accesses have spatial locality (8.2.6 on page 110).

5.3.4 Accuracy and Scaling

There has been much controversy surrounding the issue of double precision floating point accuracy and the GPU. The old GT200-architecture has very low double precision architecture, and many felt that the GPU is not suitable for real scientific work due to this fact. Fortunately NVIDIA has realized that there is a real desire for double precision, and with the new Fermi architecture there is support for much

higher double precision performance. Unfortunately the increased double precision performance is only available on the Tesla-series GPUs.

Another feature that is exclusive to the Teslas is Electronic Error Correction (ECC) for the memory. The issue of errors has become increasingly important, especially if the GPU is to be used in large clusters. By using some of the memory on the GPU for ECC, the failure-rate for memory reads drops dramatically. The Tesla cards also provide much more memory than the GeForce-line, and as such they are generally much more suited for GPU computing.

The Fermi architecture provide a more accurate implementation of floating point, more specifically the new IEEE 754-2008 floating-point standard. This change also means that some old CUDA-code may not behave in the exact same manner, more details on this is available in the NVIDIA CUDA Programming Guide [58].

Chapter 6

Models and Implementation

We have thus far investigated the possibilities for creating an interactive avalanche simulation, and have found that an SPH-based fluid simulation with support for Non-Newtonian viscosity is the best choice for simulating geomorphological flows interactively. SPH is fairly well suited for a GPU-based implementation, and we have shown in a previous work Krog [4] that SPH on the GPU can give good performance.

In this thesis, we have developed a new framework for GPU-based SPH simulations described in Chapter 7. We use this framework to implement two different SPH models.

A large part of the goal for our thesis is to implement fluid models suitable for interactivity or "real-time" performance. The focus of our work is thus on the efficiency and computational performance of the simulation, not accuracy.

To achieve high performance we make use of an acceleration data structure (7.1.1.2) which makes it possible to exploit the power the GPU, and also extensive optimizations of the code itself (7.1.3).

Our SPH models are:

1. Simple SPH model

   This is a reimplementation of our previous implementation [4], in the context of our new framework.

   The model itself is based on the one by Müller et al. [1] and is designed for interactivity.

2. Complex SPH model

   This model is a new and more "complex" model. We combine some of the techniques used in Müller et al. [1] with models from Hosseini et al. [23] and Paiva et al. [22]. Compared to the Simple model we use a more accurate smoothing kernel, we correctly calculate shear forces and we support a range of rheological models which enable us to simulate Non-Newtonian fluids.

6.1 The SPH Algorithm

Both our SPH models use the same basics steps in the algorithms. One iteration of the algorithm is one timestep, so all the steps of the algorithm must be executed every time.

The first step is to update the acceleration data structure (7.1.1.2).

Secondly the *SPH* force calculations must be done, these are essentially the SPH sums. The order in which these sums are calculated is driven by data dependencies between the calculations. In SPH, the force interpolation function 4.3 is dependent on the density, so before any of the force density functions can be calculated the density for each particle has to be calculated.

Finally the integration step does time integration of the acceleration from the SPH force, applies external forces and does color calculation and other minor steps.

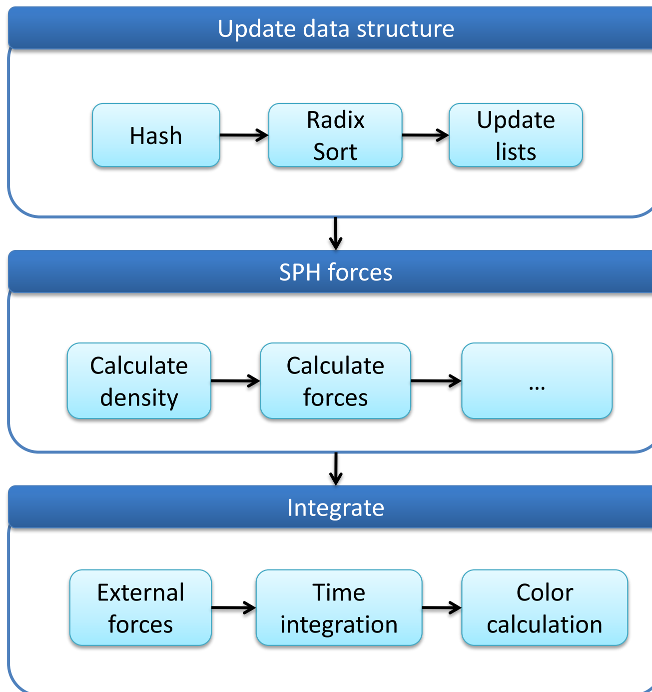


Figure 6.1: The 3 main parts of the fluid simulation.

6.2 Simple SPH model

The “simple” SPH model is based on the model by Müller et al. [1]. This is a weakly compressible Newtonian formulation for interactive use, meaning it trades accuracy

for performance.

The implementation of this model is described in greater detail in Krog [4], but we will include a brief description here since there are significant changes in both implementation and performance.

6.2.1 Model

This model uses the specialized smoothing kernels described in 4.3.2, and there are 3 main equations.

The SPH density (4.2.1):

$$\rho_i = \sum_{j \neq i} m_j W_{poly6} \tag{6.1}$$

The SPH pressure force (4.2.3):

$$\mathbf{f}_i^{pressure} = -\frac{1}{\rho_i} \sum_{j \neq i} m_j \frac{P_i + P_j}{2\rho_j} \nabla W_{spiky} \tag{6.2}$$

Where the pressure P is found using an equation of state based on the ideal gas law (4.2.2):

$$P = k(\rho - \rho_0)$$

And finally the SPH viscosity force (4.2.4.1):

$$\mathbf{f}_i^{viscosity} = \frac{\mu}{\rho_i} \sum_{j \neq i} m_j \frac{\mathbf{v}_j - \mathbf{v}_i}{\rho_j} \nabla^2 W_{viscosity} \tag{6.3}$$

6.2.2 Precalculation

As in the previous implementation, we move all constants outside of the summation. In addition we simulate a single phase fluid with constant mass for each particle, which means that the mass term becomes a constant and can also be moved outside.

$$\rho_i = m * W_{poly6}^{coeffs} \sum_{j \neq i} W_{poly6}^{variable} \tag{6.4}$$

$$\mathbf{f}_i^{pressure} = -m * \nabla W_{spiky}^{coeffs} * \frac{1}{2} * \frac{1}{\rho_i} \sum_{j \neq i} \frac{P_i + P_j}{\rho_j} \nabla W_{spiky}^{variable} \tag{6.5}$$

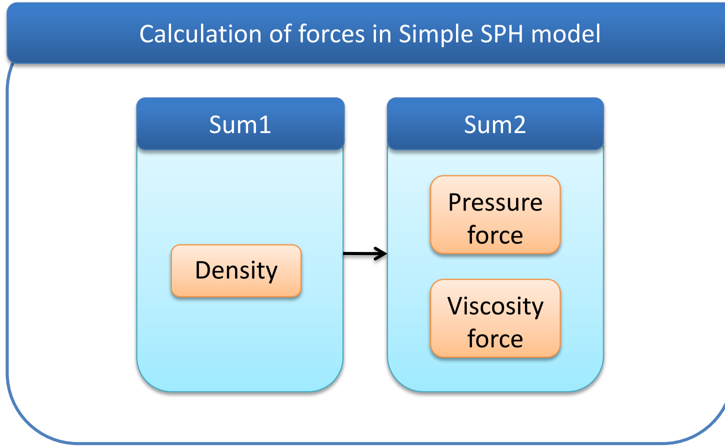


Figure 6.2: Calculation of forces for Complex SPH model

$$\mathbf{f}_i^{viscosity} = \mu * m * \frac{1}{\rho_i} * \nabla W_{spiky}^{coeffs} \sum_{j \neq i} \frac{\mathbf{v}_j - \mathbf{v}_i}{\rho_j} \nabla^2 W_{viscosity}^{variable} \tag{6.6}$$

Where we use the syntax $W^{variable}$ to denote the variables in the smoothing kernel and W^{coeffs} the coefficients that can be precalculated. These coefficients can be found in Chapter A, and will not be included in the following explanations for reasons of brevity.

The data dependencies for this simple model means that we need 2 summation steps. Both the pressure and the viscosity force depends on the density calculations, so we need a separate summation step for the density, and then we can combine the calculation of the pressure and the viscosity in a single summation step.

All the coefficients outside the summation are combined in the actual code to limit memory access, but are kept separate here for clarity.

6.2.3 Pseudocode

Here we present pseudocode for the SPH calculations, the actual code can be found in Section B.7.

The calculation of the density is one of the necessary summations in the SPH algorithm (6.1), this is our Sum1 kernel.

Since the two forces, pressure and viscosity, are not data-dependent on each other their calculation can be combined in the same iteration loop, this greatly improves the performance since it is limits the amount of global memory reads (6.2). This is our Sum2 kernel.

Algorithm 6.1 Pseudocode for Simple SPH density calculation, the SimpleSPH Sum1 kernel.

```
1 for all particles i:
2   density_i = 0
3   for all neighboring cells c:
4     for all particles j in cell c:
5       vector r = particle_pos_i - particle_pos_j
6       scalar rlen = length(r)
7       if(rlen <= smoothing_length)
8         density_i += Wpoly6variable::Kernel(smoothing_length, r, rlen)
9   density_i *= particle_mass * Wpoly6coeff
```

Algorithm 6.2 Pseudocode for Simple SPH forces calculation, the SimpleSPH Sum2 kernel.

```
1 for all particles i:
2   force_i = 0
3   vector f_pressure = 0
4   vector f_viscosity = 0
5
6   for all neighboring cells c:
7     for all particles j in cell c:
8       vector r = particle_pos_i - particle_pos_j
9       scalar rlen = length(r)
10      if(rlen <= smoothing_length)
11        scalar h_rlen = smoothing_length - rlen
12
13        f_pressure  += ( (pressure_i + pressure_j) / density_i * density_j
              ) * Wspikyvariable(smoothing_length, r, rlen)
14        f_viscosity += ( (veleval_j - veleval_i) / density_i * density_j )
              * Wviscosityvariable(smoothing_length, r, rlen)
15
16  force_i += particle_mass * Wspikycoeff * f_pressure
17  force_i += particle_mass * Wviscositycoeff * f_viscosity
```

6.2.4 Resource usage and Occupancy

Using the kernel resource usage we have collected from ptxas during the compilation (7.1.3.1), we have optimized the kernel block size launch parameters for optimal occupancy on the device.

We need to compile to separate versions, one for sm_13 (the old GT200 architecture) and one of sm_20 (the new GF100/Fermi architecture).

Kernel	Registers	Shared Memory	Constant Memory
Grid_Hash	14	48+16	60
Grid_Update	8	160+16	140+4
Sum1	22	128+16	140+4
Sum2	35	96+16	140+4
Integrate	24	224+16	248+24+8+28

Table 6.1: Resource usage for Simple SPH kernels for GT200 architectures.

GT200 architecture Giving us the following ideal block size and occupancy:

Kernel	Block size	Occupancy
Grid_Hash	128	100%
Grid_Update	256	100%
Sum1	128	63%
Sum2	448	44%
Integrate	128	63%

Table 6.2: Ideal block size and resulting occupancy for Simple SPH kernels for GT200 architectures.

Fermi architecture We achieve better occupancy on the Fermi architecture due to the architectural changes [58].

Kernel	Registers	Shared Memory	Constant Memory
Grid_Hash	16	0	80+60
Grid_Update	13	0	216+36+8
Sum1	24	0	152+24+8
Sum2	32	0	128+24+8
Integrate	26	0	248+24+8+28

Table 6.3: Resource usage for Simple SPH kernels for Fermi architectures.

Giving us the following ideal block size and occupancy:

Kernel	Block size	Occupancy
Grid_Hash	128	100%
Grid_Update	256	100%
Sum1	224	88%
Sum2	128	67%
Integrate	352	81%

Table 6.4: Ideal block size and resulting occupancy for Simple SPH kernels for Fermi architectures.

It is interesting to note here that due to changes in the Fermi architecture, shared memory us no longer used for storing the parameters to the kernel functions, meaning that we can clearly see that none of the kernels actually use any shared memory.

6.3 Complex SPH model

The "Complex" Non-Newtonian fluid model is based on the previously presented Newtonian fluid model. It differs most significantly in the calculation of fluids stresses and viscosity. We use the shear stress calculations from Hosseini et al. [23] in combination with the pressure calculations from Paiva et al. [22] and Müller et al. [1] creating a model that includes correct calculation of shear stresses while remaining suitable for interactivity. This is a critical point since the original model in Hosseini et al. [23] includes an explicit calculation of the Poisson equation for the pressure, a computationally very costly calculation.

We have not validated the model and it should be noted that our model is has been designed to allow for interactive or near-interactive simulation performance, not physical accuracy.

We use the model to simulate snow avalanches, but it is equally well or perhaps more suited to other geomorphological flows that are not as complex as a snow avalanche. Hosseini et al. [23] uses their model to simulate mud slides, Paiva et al. [22] simulate lava flows. Using our implementation for these kinds of geomorphological flows is a possible future work.

6.3.1 Model

As with the Simple SPH model there are a few equations that are most important:

The SPH density:

$$\rho_i = \sum_{j \neq i} m_j W_{poly6} \tag{6.7}$$

For the SPH pressure force (4.2.3), we employ the alternative, more accurate version which conserves linear and angular momentum (Equation 4.21):

$$\mathbf{f}_i^{pressure} = -\sum_{j \neq i} m_j \left(\frac{P_i}{\rho_i^2} + \frac{P_j}{\rho_j^2} \right) \nabla W_{spiky} \tag{6.8}$$

Where the pressure P is found using an equation of state based on the ideal gas law (4.2.2):

$$P = k(\rho - \rho_0)$$

We have found that using the artificial viscosity from the Simple model with a low viscosity coefficient helps with stability, and makes it possible to simulate with larger timesteps:

$$\mathbf{f}_i^{viscosity} = \frac{\mu}{\rho_i} \sum_{j \neq i} m_j \frac{\mathbf{v}_j - \mathbf{v}_i}{\rho_j} \nabla^2 W_{viscosity} \tag{6.9}$$

We also use XSPH (6.5) to maintain a more ordered movement of particles:

$$\mathbf{f}_i^{xsph} = \sum_{j \neq i} 2 m_j \frac{(\mathbf{v}_j - \mathbf{v}_i)}{(\rho_i + \rho_j)} W_{poly6} \tag{6.10}$$

Using the velocity stress tensor calculation (Equation 4.24):

$$\nabla \mathbf{v}_i = \sum_j \frac{m_j}{\rho_j} (\mathbf{v}_j - \mathbf{v}_i) \otimes \nabla W_{cubic}$$

We can compute the SPH stress force (4.2.4.2):

$$\mathbf{f}_i^{stress} = \frac{1}{\rho_i} \sum_{j \neq i} m_j \frac{\tau_i + \tau_j}{\rho_j} \cdot \nabla W_{cubic} \tag{6.11}$$

Where $\dot{\gamma} = \frac{1}{2} \left(\nabla \mathbf{v} + (\nabla \mathbf{v})^T \right)$ (Equation 4.23) and τ is defined with a rheological function. We support several rheological models, which are described in more detail in Section 3.3.

Smoothing Kernels We choose to use the cubic spline smoothing kernel for the calculation of the fluids stresses. This is done for two reasons; the first being that this leads to a more accurate simulation and secondly because we wish to have a model which rely on a large amount of branching. From a performance evaluation perspective this is interesting because it is more fair to include calculations that other SPH models are likely to require. If further performance is desired the Quartic smoothing kernel (4.3.1.6) is worth investigating since it is not piecewise, and is thus better suited for the GPU.

6.3.2 Precalculation

As with the simple SPH model we move all applicable constants outside the summation loop as well as make the mass term a constant:

$$\rho_i = m * W_{poly6}^{coeffs} \sum_{j \neq i} W_{poly6}^{variable} \tag{6.12}$$

$$\mathbf{f}_i^{pressure} = -m * \nabla W_{spiky}^{coeffs} \sum_{j \neq i} \left(\frac{P_i}{\rho_i^2} + \frac{P_j}{\rho_j^2} \right) \nabla W_{spiky}^{variable} \tag{6.13}$$

$$\mathbf{f}_i^{xsph} = 2m W_{poly6}^{coeffs} \sum_{j \neq i} \frac{(\mathbf{v}_j - \mathbf{v}_i)}{(\rho_i + \rho_j)} W_{poly6}^{variable} \tag{6.14}$$

$$\mathbf{f}_i^{viscosity} = \mu * m * \frac{1}{\rho_i} * \nabla W_{spiky}^{coeffs} \sum_{j \neq i} \frac{\mathbf{v}_j - \mathbf{v}_i}{\rho_j} \nabla^2 W_{viscosity}^{variable} \tag{6.15}$$

$$\nabla \mathbf{v}_i = m \sum_j \frac{1}{\rho_j} (\mathbf{v}_j - \mathbf{v}_i) \otimes \nabla W_{cubic} \tag{6.16}$$

$$\tau = rheological\,function()$$

$$\mathbf{f}_i^{stress} = \frac{m}{\rho_i} \sum_{j \neq i} \frac{\tau_i + \tau_j}{\rho_j} \cdot \nabla W_{cubic} \tag{6.17}$$

This complex model is very different as far as computational cost, since it requires 3 summation loops, instead of just 2. This is because the calculation of the stress force depends on the velocity tensor (6.16) which again depends on the density calculation (6.12).

These dependencies can naturally not be combined, and must be calculated separately. In addition these summation steps involves calculations on tensor matrices (3x3) which greatly increase the amount of memory usage, which is critical for the performance. Finally we also employ a more accurate smoothing kernel (cubic spline) which introduces additional branching.

6.3.3 Pseudocode

Here we present pseudocode for the SPH calculations, the actual code can be found in Section B.7.

The calculation of the density is one of the necessary summations in the SPH algorithm, this step is identical to that in the Simple SPH model (Algorithm 6.1), the Sum1 kernel.

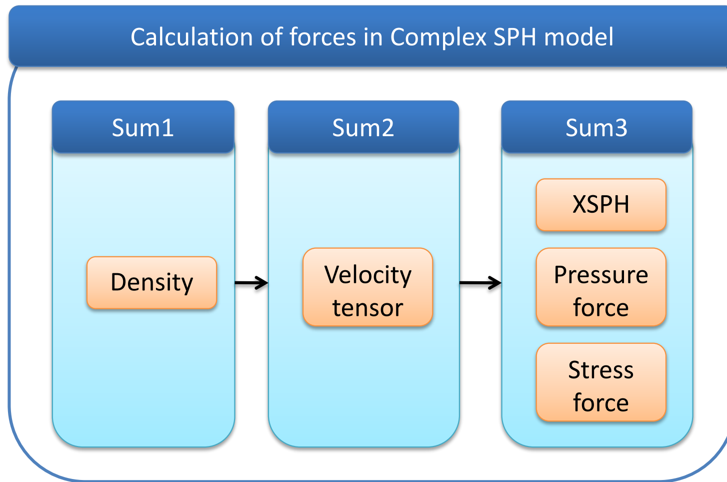


Figure 6.3: Calculation of forces for Complex SPH model

Algorithm 6.3 Pseudocode for Complex SPH velocity tensor calculation, the Complex SPH Sum2 kernel.

```
for all particles i:
  matrix3 sum_velocity_tensor = 0

  for all neighboring cells c:
    for all particles j in cell c:
      vector r = particle_pos_i - particle_pos_j
      scalar rlen = length(r)
      if(rlen <= smoothing_length)
        scalar h_rlen = smoothing_length - rlen
        sum_velocity_tensor = outerproduct((veleval_j  - veleval_i)/(
            density_j), Wcubic::Gradient(...));

  matrix3 velocity_tensor_i = cFluidParams.particle_mass *
      sum_velocity_tensor;
  // calculation of rheological model
  ...
```

Algorithm 6.4 Pseudocode for Complex SPH forces calculation, the Complex SPH Sum3 kernel.

```
for all particles i:
  vector force_i = 0;
  vector f_xsph = 0;
  vector f_pressure = 0;
  vector f_viscosity = 0;
  vector f_stress = 0;

  for all neighboring cells c:
    for all particles j in cell c:
      vector r = particle_pos_i - particle_pos_j
      scalar rlen = length(r)
      if(rlen <= smoothing_length)
        scalar h_rlen = smoothing_length - rlen
        f_xsph = ( (veleval_j - veleval_i) / (density_i+density_j) ) *
            Wpoly6variable::Kernel(smoothing_length, r, rlen)
        f_pressure  += ( (pressure_i + pressure_j) / (density_i*density_j)
             ) * Wspikyvariable(smoothing_length, r, rlen)
        f_viscosity += ( (veleval_j - veleval_i) / (density_i*density_j) )
             * Wviscosityvariable(smoothing_length, r, rlen)
        f_stress += dotproduct((stress_tensor_i+stress_tensor_j)/(
            density_j), Wcubic::Gradient(...))

  f_xsph *= 2 * particle_mass;
  force_i += particle_mass * Wspikycoeff * f_pressure
  force_i += particle_mass * Wviscositycoeff * f_viscosity
  force_i += particle_mass * f_viscosity / density_i
```

In the Complex model we need to calculate the velocity tensor, and a rheological model, this we call the Sum2 kernel (Algorithm 6.3).

Finally we need to do calculation of the various forces in the Complex model. This we do in a kernel we call Sum3 (6.4).

6.3.4 Rheological Models

In our implementation we support several rheological models. In 3.3 we describe these models, here we will show how we implement them.

All these implementations depend on the calculation of the deformation tensor $\dot{\gamma}$ and the "deformation amount" $|\dot{\gamma}|$:

```
matrix3 deformation_tensor_i = 0.5*(velocity_tensor_i + transpose(
    velocity_tensor_i));
float t = trace(deformation_tensor_i);
float deformation_amount = sqrtf(t*t);
```

Newtonian Rheology The implementation of a Newtonian rheology is very simple, since the viscosity is linear.

```
stress_tensor = viscosity*deformation_amount*deformation_tensor_i;
```

Power-Law Rheology The implementation of the Power-Law rheology follows Equation (3.10) very closely, with one large exception. In the Power-Law model there is no maximum or minimum for the fluid viscosity. This does not work very well in a simulation, since the simulation is only stable for viscosities in a certain range (determined in part by the size of the time-step). To prevent instability we add a naive clamping to the viscosity value so as to ensure it does not exceed or fall below some safe values.

```
float viscosity = K*pow(deformation_amount,n-1.0f);
viscosity = clamp(viscosity, 1.0f,300.0f);
stress_tensor = viscosity*deformation_tensor_i;
```

Cross Rheology The implementation of the Cross rheology also follows it's equation very closely (Equation3.11). As for the Power-Law model we add clamping to ensure stability.

```
float viscosity = K*pow(deformation_amount,n-1.0f);
viscosity = clamp(viscosity, 1.0f,300.0f);
stress_tensor = viscosity*deformation_tensor_i;
```

Bingham Rheology The Bingham rheology differentiates itself from the previously described rheological models in that it has a "solid" zone below a given yield stress value. As in [23] we choose to emulate this solid zone by giving the fluid a high viscosity value.

```
stress_tensor = yield_stress + K*deformation_tensor_i;
float s = trace(stress_tensor);
float stress_amount = sqrtf(s*s);
if(stress_amount <= yield_stress) {
  stress_tensor = 500*deformation_amount*deformation_tensor_i;
}
```

Herschel-Bulkley Rheology The implementation of the Herschel-Bulkley rheology (Equation (3.14)) follows that of the Bingham-rheology closely.

```
stress_tensor = (K*pow(deformation_amount,n-1.0f) +yield_stress/
    deformation_amount)*deformation_tensor_i;
float s = trace(stress_tensor);
float stress_amount = sqrtf(s*s);
if(stress_amount < yield_stress) {
```

```
5   stress_tensor = 500*deformation_amount*deformation_tensor_i ;
6 }
```

6.3.5 Resource usage and Occupancy

Using the kernel resource usage we have collected from ptxas during the compilation (7.1.3.1), we have optimized the kernel block size launch parameters for optimal occupancy on the device.

We need to compile to separate versions, one for sm_13 (the old GT200 architecture) and one of sm_20 (the new GF100/Fermi architecture).

GT200 architecture The occupancy of the Complex model is much worse than that of the Simple model due to the higher register usage.

Kernel	Registers	Shared Memory	Constant Memory
Grid_Hash	14	48+16	60
Grid_Update	7	192+16	156+4
Sum1	22	144+16	156+16
Sum2	36	144+16	156+16
Sum3	58	144+16	156+16
Integrate	27	256+16	156+12

Table 6.5: Resource usage for Complex SPH kernels for GT200 architectures.

Kernel	Block size	Occupancy
Grid_Hash	256	100%
Grid_Update	256	100%
Sum1	128	63%
Sum2	256	50%
Sum3	256	25%
Integrate	256	50%

Table 6.6: Ideal block size and resulting occupancy for Complex SPH kernels for GT200 architectures.

Fermi architecture As with the Simple model we we achieve better occupancy on the Fermi architecture due to the architectural changes [58].

Kernel	Registers	Shared Memory	Constant Memory
Grid_Hash	16	0	80+60
Grid_Update	13	0	216+36+8
Sum1	24	0	168+36+8
Sum2	38	0	168+36+8+4
Sum3	60	0	168+36+8+4
Integrate	29	0	288+36+8+4

Table 6.7: Resource usage for Complex SPH kernels for Fermi architectures.

Giving us the following ideal block size and occupancy:

Kernel	Block size	Occupancy
Grid_Hash	256	100%
Grid_Update	256	100%
Sum1	192	88%
Sum2	416	54%
Sum3	64	33%
Integrate	256	67%

Table 6.8: Ideal block size and resulting occupancy for Complex SPH kernels for Fermi architectures.

As seen for the Simple model, shared memory us no longer used for Fermi-cards for kernel parameters.

6.4 Boundary Conditions

We choose to employ a fairly standardized "repulsion" force for the boundaries (4.2.5.2). This method was chosen for the ease with which multiple types of boundaries can be implemented. The repulsion force is fairly easily implemented for both "wall boundaries" as well as "terrain".

As in our previous implementation we choose to use a repulsion force to prevent a particle from penetrating a boundary (7.1.1.4).

For wall/grid boundaries this is simply implemented by using a check for each of the walls in the grid (6.5).

Algorithm 6.5 Pseudocode for collision handling repulsion force against a wall boundary in the x direction.

```
for particle i in all particles:
  d = boundary_distance - (particle_position_x - grid_min_x)
  if(d > EPSILON)
    f_repulsion += calculateRepulsionForce(...)
```

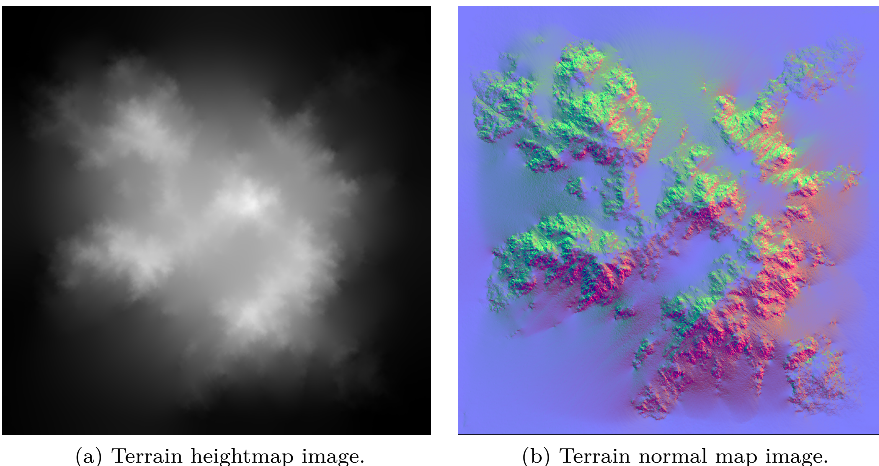
In addition we add a friction force (7.1.1.4). For an avalanche the terrain friction is very important. The basal stresses and friction determines a large part of the overall behavior of the avalanche.

Algorithm 6.6 Pseudocode for collision handling friction force against a wall boundary in the x direction.

```
for particle i in all particles:
  d = boundary_distance - (particle_position_x - grid_min_x)
  if(d > EPSILON)
    f_friction += calculateNoSlipForce(...)
```

Terrain collision detection and handling To add support for terrain collision we add calculations for determining the height of the terrain at a given position.

The terrain data is supplied in a heightmap (Figure 6.4a on page 69), essentially an image where each pixel is a color value which determines the height at that position.



(a) Terrain heightmap image. (b) Terrain normal map image.

Figure 6.4: Terrain heightmap and normal images.

Each particle is checked for a collision against the terrain by finding the terrain height at the position of the particle and then checking if the particle is beneath the terrain.

This makes it possible to check if a particle is colliding with the terrain, but it does not take into account the slope of the terrain at that position. Using a normal map (Figure 6.4b on page 69) we find the normal of each face in the terrain. Using this normal we simply apply the repulsion and friction forces as for the wall boundaries.

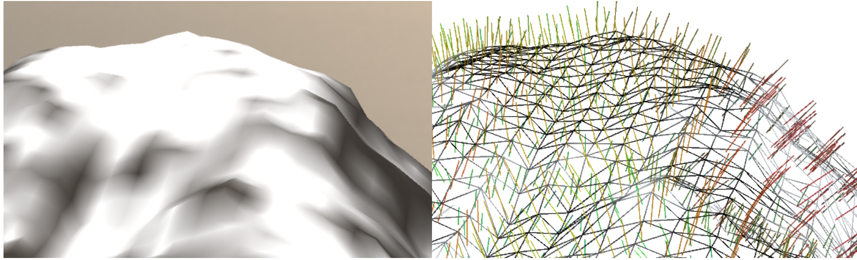


Figure 6.5: The terrain with face normals.

6.5 Integration of Forces

The final step in the SPH algorithm is the integration of the various forces. In this step we also do various other calculations, such as the calculation of the colors for the particles.

Time Integration The integration scheme chosen is the "Leap-Frog" integrator [59], an integrator that is accurate to second-order:

$$x_{i+1} = x_i + v_{i+1/2}dt \tag{6.18}$$

$$v_{i+1/2} = v_{i-1/2} + a_i dt \tag{6.19}$$

where i is time steps.

The name of this integrator is a result of the above formulation of it; the velocities "leap over" the positions.

This scheme can also be formulate in a form where all quantities are defined at integer times only:

$$x_{i+1} = x_i + v_i dt + \frac{a_i}{2}{}_i dt^2 \tag{6.20}$$

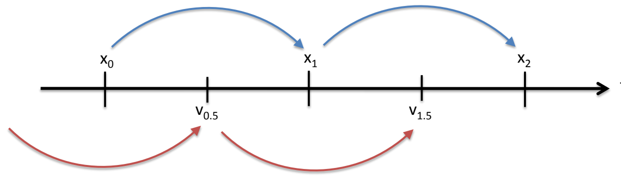


Figure 6.6: The leap-frog integrator.

$$v_{i+1} = v_i + \frac{(a_i + a_{i+1})}{2} dt \tag{6.21}$$

Leap-Frog is a fairly good compromise between the somewhat naive Euler method and more advanced methods that require more than a single evaluation for each force. Such advanced methods include the third-order Runge-Kutta methods, which has been used for SPH models [21].

XSPH An additional smoothing of the velocity integration is used for the Complex SPH model. This technique is called XSPH [60]:

$$v_i = v_i + \epsilon \sum_{j \neq i} 2m_j \frac{(\mathbf{v}_j - \mathbf{v}_i)}{(\rho_i + \rho_j)} W(\mathbf{r}_{ij}, h)$$

Where ϵ is a parameter in the range $[0, 1]$, typically 0.5.

XSPH essentially computes an average velocity from the velocities of neighboring particles

With SPH it is also possible to use adaptive time integration. Desbrun and Gascuel [36] employ adaptive time integration based on the Courant-Friedrichs-Lewy [36] criterion. This criterion intuitively means that if a phenomenon propagates with a maximum velocity v, it must not be integrated with a too large time step, or some grid points will be leaped [36].

Chapter 7

Simulation Framework

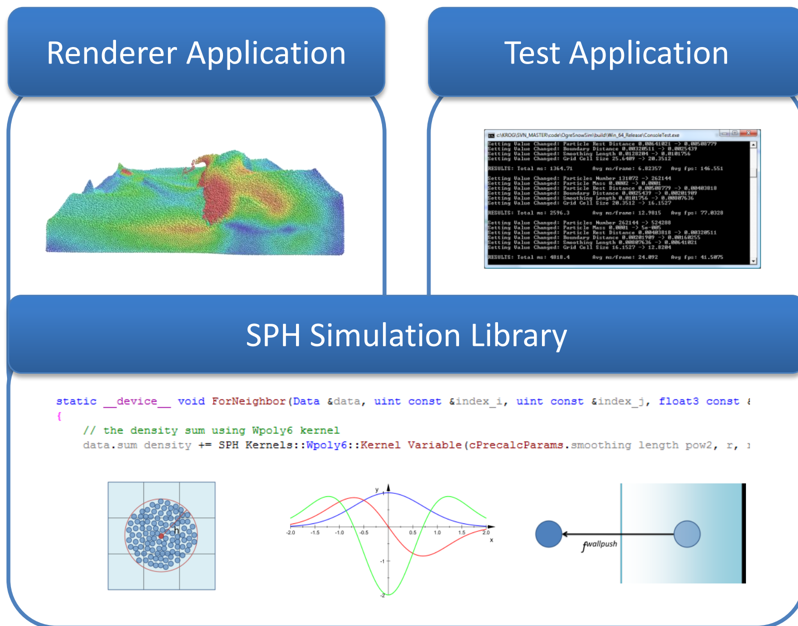


Figure 7.1: Overview of the SPH simulation framework.

As development of our SPH implementations progressed it became clear that it would be a good idea to create a general framework for doing SPH computations on GPUs. By creating separate modules that encapsulate certain functionality, the code base is overall much cleaner and it is possible to more easily create new simulations.

In our framework we include functions for calculating many common SPH operations and also include support for both loose and tight integration with external applications.

7.1 Simulation Library

To ensure that the simulation code is not dependent on any external applications or application-specific code it was decided to move all the simulation code into a separate library. Doing so makes it possible to use the simulation library from any number of external applications. In our framework we have made both a console test application and a 3D rendering application which uses the simulation library and the API it provides.

The simulation library includes several components, including specialized code for accelerating nearest-neighbor particle simulations on the GPU, SPH calculation functions and helper components such as settings and memory buffer management.

7.1.1 Components

As far as possible most of these components have been implemented in the reduced subset of C++ which is supported in the NVIDIA CUDA compiler. Use of templates and subclassing substantially reduces the amount of code and makes it possible to modularize at a highly granular level.

7.1.1.1 Parallelization of SPH on GPUs

Our simulation library has been created with the express purpose of doing SPH calculations on the GPU. To this end it is important the SPH calculations can utilize the GPU, and that it can do so as efficiently as possible.

The GPU is very powerful, it has high memory bandwidth and can also do a lot of calculations due to the massive parallelism. There are however a large number of constraints that apply to GPU code and it is important to create an efficient mapping between the algorithm and the GPU code in order to achieve high efficiency.

When doing calculations on the GPU the issue of parallelization is first and foremost. We choose to use one thread on the GPU for every particle in the simulation, which is the natural choice, since we want to parallelize as much as possible in order to increase GPU utilization.

Unfortunately SPH is not perfectly parallelizable since there are data-dependencies between particles in the domain of the smoothing length. Fortunately this smoothing length is fairly small, which means that each particle/thread need only read values from a small number of nearby particles, not all the particles in the simulation.

When doing these neighbor reads there are two large issues. How to find and read the properties of nearby values, and how to order the pattern with which these reads occur. Doing a naive brute-force $O(N^2)$ search for neighbors, is very inefficient, so a better solution is needed.

On the GPU it is very important to achieve *coalescing* memory accesses and ordering the neighbor reads according to the coalescing rules is a very hard problem.

7.1.1.2 Nearest-Neighbor Search using a Uniform Grid

To efficiently find and read neighbor values in the SPH calculations we use the algorithm which we presented in [4]. This algorithm is originally from Green and NVIDIA [61] and previously implemented in NVIDIA [62]; a hashed, radix sorted, uniform grid.

By dividing the simulation domain into equally sized cells, where the cell size is equal to the SPH smoothing length it is only necessary to check the particles in the nearby cells. For 2D this is 9 cells to check, and in 3D 29 cells.

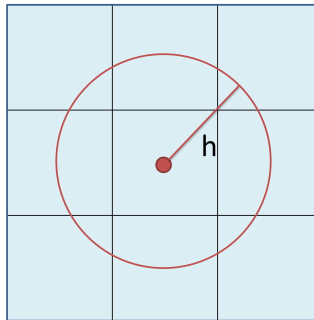


Figure 7.2: 2D uniform grid with a cell size equal to the smoothing distance h.

The algorithm steps are as follows:

1. Divide the simulation domain into a uniform grid.
2. Use to the spatial position of each particle to find the cell it belongs to.
3. Use a spatial *hash* function on the particle cell position.
4. *Sort* the particles according to their spatial hash.
5. *Reorder* all the particle parameters in sorted buffers.
6. Create *cell indice buffers* which track the start and end indices for all cells in the sorted buffers.

Particles in the same cell will then lie consecutively in the linear buffer. Finding the indices for each cell is simply a look-up in the cell indice buffers, and finding particles in neighboring cells is simply a matter of iterating over the correct indices in the buffer.

For the sorting we used the fastest radix sort available for the GPU at the time of implementation [63]. A new sorting method is available [64] which improves the sorting performance by 2x to 3.7x, but as of the time of this thesis, an implementation had not yet been publicized.

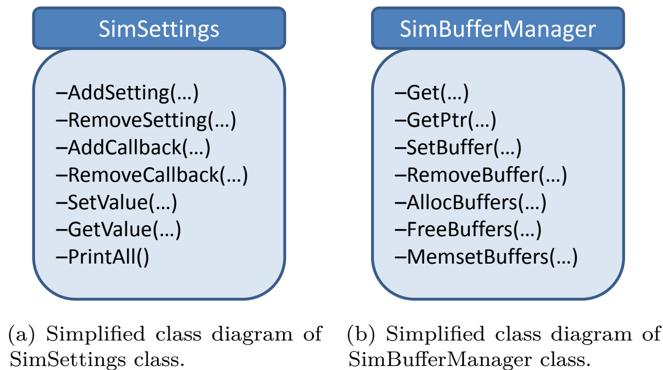
The details of the implementation can be found in [4] and can be summarized by these steps:

1. **Hash** the particles by their wrapped cell position in the uniform grid
2. **Radix sort** the particles by their spatial hash
3. **Updating** all particle parameter arrays so as to be sorted

We have reimplemented this algorithm in our own framework, and we provide modularized functions for doing the calculations and lookups. The source code for our implementation can be found in Section B.2.

7.1.1.3 Code Helpers

Settings and Parameters We have included support classes for handling settings and parameters to the simulation. These classes include event-based feedback for when settings are changed. A simulation implementation adds settings (with sane defaults) and can add itself to the event callback. Whenever a setting is changed, the callback will then be used to notify the event listeners. This is very useful since the settings classes can be exposed to an external user of the simulation.



(a) Simplified class diagram of SimSettings class.

(b) Simplified class diagram of SimBufferManager class.

Figure 7.3: Class diagrams of code helper classes.

This design makes it possible to add support for instantaneous changes. One could for example imagine a Graphical User Interface (GUI) that allowed a user to change parameters in real-time.

Memory Allocation and Buffer management When implementing a simulation memory management can be a significant overhead. To ease the use of memory

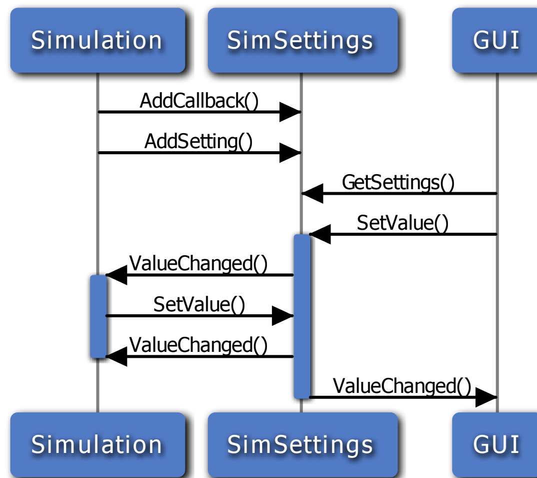


Figure 7.4: Sequence diagram of Simulation-Settings-GUI interaction.

Figure 7.5: Class diagram of SimBufferManager class.

buffers we have created a number of classes that wrap the memory buffers. Using the SimBufferManager class utilization of memory buffers is essentially abstracted away from the implementation itself.

Using the SimBufferManager an implementation can specify what buffers it needs and easily allocate and free them:

Algorithm 7.1 Example of using the SimBufferManager class

```
mSPHBuffers->SetBuffer(BufferSphForceSorted,    new SimBufferCuda(
    mCudaAllocator, Device, sizeof(float_vec)));
mSPHBuffers->SetBuffer(BufferSphPressureSorted,   new SimBufferCuda(
    mCudaAllocator, Device, sizeof(float)));
mSPHBuffers->SetBuffer(BufferSphDensitySorted,    new SimBufferCuda(
    mCudaAllocator, Device, sizeof(float)));

mSPHBuffers->AllocBuffers(numParticles);
mSPHBuffers->FreeBuffers();
```

A typical use case here is that the implementation specifies what buffers it needs in the constructor, and allows external entities to overrule these buffers later. This scenario is used to implement the rendering API used for seamless enabling of interactive rendering.

In addition to the two SimBufferManager class we have also added a SimBuffer

class, which is simply an abstract class that represents a memory allocation.

The advantage of using a class for this is that the entire system can allow use of specialized buffers, this is especially important since it makes it possible to hide the complexity that arises from our real-time rendering integration.

We implement two different subclasses for this class, a SimBufferCuda class which is essentially just a linear GPU buffer and a specialized buffer type called OgreSimBuffer, which is an OpenGL or Direct3D9 vertex buffer that is rendered in the rendering application. Since the entire system is designed to use the abstract baseclass SimBuffer this specialized buffer class requires no additional code in the general framework and is used seamlessly by the simulation library.

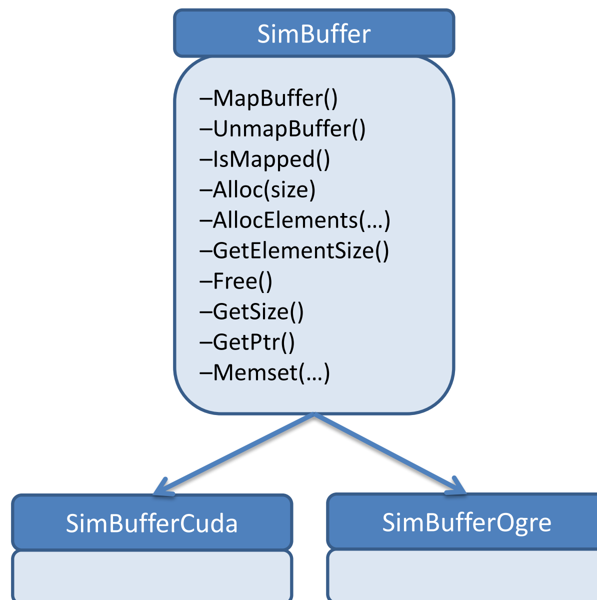


Figure 7.6: Class diagram of SimBufferManager class.

Finally we have added a SimCudaAllocator which is simply a wrapper around the CUDA allocation functions. This wrapper is very useful since it allows use to keep track of the amount of memory that is allocated.

7.1.1.4 SPH calculations

We have added functions for calculating many common SPH operations.

On of the biggest advantages of using SPH is how easily it can handle interaction with complex boundary conditions. Unlike Eulerian methods, there is no need to generate a mesh, which means that there is also no need for a computationally costly operation. This is particularly important when dealing with simulations where the boundaries within the domain is animated.

Boundary Repulsion Force In our framework we include implementations of two different boundary forces, a repulsion force and a friction force.

We use the repulsion force described by Amada [65] to repel particles from boundaries (Figure 7.7 on page 79):

$$f_i^{repulse} = \begin{cases} K_s d - (\mathbf{v}_i \cdot \mathbf{n}) K_d)\mathbf{n} & d > \epsilon \\ 0 & otherwise \end{cases} \tag{7.1}$$

where d is the particle distance to the boundary, ε is the collision accuracy (a small number), v_i is the velocity of particle i, n is the surface normal of the wall, K_{stiff} is a stiffness parameter and K_{damp} is a dampening parameter. This force acts as a spring, the more a particle penetrates a boundary, the more it is pushed away from the boundary.

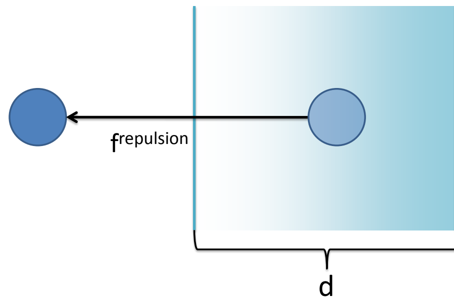


Figure 7.7: The particles inside the boundary are pushed back by a force that is proportional to the depth into the boundary.

Algorithm 7.2 Pseudocode for collision handling repulsion force.

```
1 vector calculateRepulsionForce(...)
2   return (boundary_stiffness * boundary_distance - boundary_dampening *
        dot(normal, vel)) * normal;
```

The full source code can be found in Section B.4.

Boundary Friction Force For the friction force we use a fairly simple kinetic friction coefficient. The friction coefficient simply reduces the velocity along the boundary (Figure 7.8 on page 80).

$$f_i^{friction} = -K_{kinetic}(\mathbf{v}_i - \mathbf{v}_i(\mathbf{v}_i \cdot \mathbf{n})) \tag{7.2}$$

where $\mathbf{v}_i$ is the velocity of the particle, $\mathbf{n}$ is the normal of the boundary and $K_{kinetic}$ is the kinetic friction coefficient.

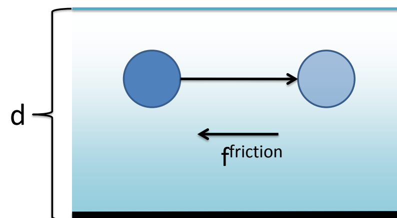


Figure 7.8: The particles inside the boundary are affected by a friction force which acts in the opposite direction of velocity along the boundary.

Algorithm 7.3 Pseudocode for collision handling friction force.

```
vector calculateNoSlipForce(...)
  // the normal part of the velocity vector (ie, the part that is going "
      towards" the boundary
  float3 v_n = vel * dot(normal, vel);
  // tangent on the terrain along the velocity direction (unit vector of
      tangential velocity)
  float3 v_t = vel - v_n;
  return friction_kinetic * -v_t;
```

The full source code can be found in Section B.4.

Smoothing Kernels One of the central calculations in SPH is the smoothing kernel, in our previous implementation these were hardcoded. In our new framework we include implementations of several smoothing kernels, most of which are described in Section 4.3.

We have implemented the following smoothing kernels in our framework:

- Cubic
- Gaussian

- Poly6
- Quadratic
- Quintic
- Spiky
- Viscosity

The full source code can be found in Section B.6.

Here we show the source code for the implementation of the first derivative of the cubic spline smoothing kernel (4.3.1.2)

Algorithm 7.4 An example of the implementation of the gradient of a smoothing kernel (the cubic kernel).

```
//third order B-spline
class Wcubic {
public:
static __device__ __host__ float3 Gradient(float smoothing_length, float
    smoothing_length_pow2, float smoothing_length_pow3, float
    smoothing_length_pow4, float3 r, float rlen, float rlen_sq)
{
  float Q = rlen / smoothing_length;
  if(Q <= 1)
  {
    float c =  1 / (M_PI *(smoothing_length_pow3));
    return - r * c * ( 3/(smoothing_length_pow2) - (9*rlen)/(4*
        smoothing_length_pow3) );
  }
  else if(Q <= 2)
  {
    float c = 3 / ( 4* M_PI * (smoothing_length_pow4));
    float dif = Q-2;
    return - r * (c * dif * dif) / rlen;
  }
  return make_float3(0.0f);
}
};
```

7.1.2 Code techniques

In our framework we make use of several code techniques. One of the principal challenges with developing for CUDA is that there is no linking for device code. In practice this means that you need to keep all your functions inside a single file, or include files manually into a one big file.

We choose the second solution, each of the components in the framework is split into it's own file which can be included when needed.

Another challenge is the lack of function pointers on the GPU (though the new Fermi architectures has support for this). Without function pointers it is difficult to create generalized functions that can be used for several data types. One example of this is the uniform grid. Ideally one would want the implementation of the uniform grid to be completely modular, such that it could be used with any number of data structures.

We achieve this by using c++ templating.

7.1.2.1 Templating

Templates in c++ is simply put a very powerful macro-language. We exploit the limited subset of templating that is available in CUDA to achieve greater modularization of the framework components.

Below is an example of how the iteration over neighbors in the SPH calculation is removed from the specific implementation of the SPH model:

Algorithm 7.5 Pseudocode explaining the use of the templated uniform grid neighbor iteration .

```
1  class Sum1 {
2  public:
3    struct Data
4    {...}
5    class Calc
6    {
7      public:
8      void PreCalc(...)
9      {...}
10     void ForNeighbor(...)
11     {...}
12     void PostCalc(...)
13     {...}
14   }
15 }
16 void Kernel_Sum1(...)
17 {
18 // call the uniform grid iteration function on our SPH sum1 class/function
19   UniformGridUtils::IterateParticlesInNearbyCells<SPHNeighborCalc<Step1::
         Calc, Step1::Data>, Step1::Data>(....);
20 }
```

In this example we define a SPH calculation class that has a “Data” struct, which contains all the variables that are necessary. There is also an internal class called “Calc” which contains the three functions that are necessary for the neighbor iteration. A “Precalc” function which is run before the iteration over neighbor, a “ForNeighbor” function which is run for each neighbor and a “PostCalc” function which is run after the neighbor iteration.

Though this solution is not the most elegant it provides a very large benefit; there is no need to reimplement the neighbor iteration algorithm for each and every SPH summation. We have implemented two different SPH models, with a total of 5 summations. By splitting the uniform grid neighbor iteration functionality into a separate component we have been able to optimize that single function much more easily, since there is no need to change source code i 5 different locations.

We also use templating to avoid expensive branching due to if-testing. An example of how we use this is the implementation of the hue-gradient calculations in our framework. By using templating to specify the desired gradient, the CUDA compiler creates several separate versions of the function that calculates the hue gradient. By passing the gradient type parameter as a template argument instead of a function argument, the CUDA framework compiles several versions of the function and chooses the correct function at runtime. This removes the need for expensive branching inside the function.

7.1.2.2 Code quality

In doing research for this thesis we studied the code of several simulation libraries and found that it is very common that the readability of the core algorithms are often very low. It is a common practice to maintain very shortand variable names and there are often very few comments. This can make the code easier to work for the principal author, but for new developers it is very hard to read and understand.

To avoid this problem we resolved to focus on the quality of the code, to include adequate comments and to ensure that the code itself is clear enough that it is understandable for new developers.

As part of this strategy we have tried to make the code both as concise and performant as possible, and a large part of this is made possible by the use of fairly high-level data structures and functions.

C for CUDA support a number of default datatypes such as float3 (a 3-dimensional vector of float values), but has no support for higher-level constructs such as matrices. The tensors in the complex SPH model are 3x3 matrices, and one of the new data structures we have added is the matrix3. This new datatype is implemented using 3 float4 vectors and we have also included operations such as the outer product (used in 7.6). We also added support for high-performance operations such as texture fetches, which is seamlessly handled in the same manner as with other texture fetches using a new function called "tex1DfetchMatrix3".

The end result is that code that operates on the SPH tensors is much simplified. Instead of manually keeping track of all the float values in each matrix and manually doing matrix operations, clear and concise functions can be used and both high maintainability and readability is ensured.

In the following example the velocity tensor in the shear stress calculations of the complex SPH model is updated.. Note how we employ the "outer" function

to calculate the tensor (a matrix3) and use operator overloading to ensure that multiplication is performed correctly for all elements in the matrix:

Algorithm 7.6 Example of using the new matrix3 CUDA data structures

```
float3 gradWcubic = SPH_Kernels::Wcubic::Gradient(cFluidParams.
    smoothing_length, r, rlen, rlen_sq);
// calculate the velocity tensor sum
data.sum_velocity_tensor += outer(
 (veleval_j  - data.veleval_i)/(density_j),
 gradWcubic
);
...
matrix3 velocity_tensor_i = cFluidParams.particle_mass * data.
    sum_velocity_tensor;
```

7.1.3 Performance Optimization

Performance optimization in CUDA is no easy task due to several factors. The foremost is difficulties due to the specialized hardware architecture of the GPU. Ensuring that the code is well-written and correct insofar as making use of the full power of the GPU is no easy task. Going beyond the basic facts such as ensuring good memory access patterns (coalescing), using the texture cache and correct block sizes for kernel launches, optimizing becomes harder because the NVIDIA CUDA C Compiler is not very well documented. For some kernels the number of registers that are used can be critical, since it affects the occupancy (5.3.2). By reducing the number of registers used it is possible to increase occupancy and this can in some cases give very significant performance improvements.

When compiling it is possible to output an intermediary "assembly" form, where the C code has been translated to PTX, the CUDA assembly language. Unfortunately this assembly language is not the final form, in fact most of the optimizations are done after the PTX stage, where developers can no longer do any optimizations.

In the PTX assembly language the registers are in Static Single Assignment (SSA) form, meaning that each "register" is never reused. This makes register optimizations easier for the compiler, but hard for humans, since the register usage in the assembly file does not match that which is used in the final GPU execution or the variables in the C code. Nonetheless the PTX assembly can be useful, since it is possible to see where new registers are used, meaning it is possible to analyze and find hotspots in the C code which are register intensive. Sometimes it is possible to find code which can be restructured so as to use less registers.

What makes performance optimizations even more complicated is that new GPU architectures, such as the new GF100-series GPUs from NVIDIA (Fermi), have a different hardware architecture from the GT200 line of GPUs.

A good example of how this affects performance is how the performance of our SPH implementations varies on these two architectures depending on register limiting.

Register limiting is an option in the NVIDA CUDA Compiler (NVCC) which makes it possible to "force" the compiler to use no more than a given number of registers. If a function requires more registers than that which is available, it will "spill" registers to local memory. Spilling registers is usually very costly, since it means that instead of reading from a register (1 cycle) the GPU has to read from global memory (hundreds of cycles) instead.

For the GT200-line of architectures this means that register limiting reduces performance, however for GT400-architectures (Fermi) it can actually improve performance. This is most likely due to the fact that the GT400-architecture includes a L1 cache which can cache these register spills. The combination of the cache and increased occupancy then means that for Fermi it may be desirable to limit register usages, making register optimizations less critical.

7.1.3.1 CUDA Toolchain

The NVIDIA toolchain for CUDA has several steps for the transformation from C for CUDA code to the final GPU execution. These steps can be roughly summarized as follows:

1. Compile C for CUDA code to PTX files by the NVIDIA (R) CUDA compiler driver (NVCC)
2. Assemble PTX files to Cubin files by the NVIDIA (R) PTX optimizing assembler (PTXAS)
3. Execute Cubin file on the GPU

Compilation During compilation it is possible to enable detailed feedback from the NVIDIA CUDA Compiler (NVCC), as well as enable various options which deal with the compiler. These options are fully described in NVIDIA [58].

The command line option "–ptax-options=-v" enables verbose output for the PTX Assembler, which is critical for evaluating the performance characteristics of a kernel. The following is an example of the output for one of the summation steps in the Simple SPH model.

```
1 ptxas info: Compiling entry function SimLib::Sim::SimpleSPH::SumStep1
2 ptxas info: Used 24 registers, 152 bytes cmem[0], 24 bytes cmem[2], 8
    bytes cmem[14]
```

PTX Files PTX files (".ptx") is an abstract assembly language and is an attempt by NVIDIA at a standardized assembly language for several GPU architectures. Though this format is upgraded as new features become available with new GPU architectures, the old versions are in principle compatible with new GPUs as well.

The assembly language in the PTX files uses operators which are fairly intuitive, however the big difference is that the registers are used in Static Single Assignment form, essentially creating new “virtual” registers for all new values.

By analyzing PTX files it is possible to see where register allocation and thus pressure is greatest, making it much easier to pinpoint places where code restructuring and other techniques can improve register usage.

Cubin Files Cubin files (“.cubin”) is the final GPU code that is actually executed on the GPU. The format of these files are closed and changes for different GPU architectures and platforms. As such these files are not easily optimized, however there does exist open-source projects which aim to open these files to developers. The DECUDA project has created a dissassembler and reassembler for these files, making it possible to hand-optimize these very low-level assembly files. In this thesis, we did not look at optimization of these files, but it may be possible to further optimize the code by manually tuning these files.

A few authors have attempted to do this kind of hand-optimizations [66, 67].

7.1.3.2 Register Optimization

The register usage of kernels is usually a large factor in the occupancy of the kernel. Here we will present some of the techniques we have use to lower register usage.

Register Limiting In our previous implementation we were not aware of the fact that the CUDA compiler default settings enables a register limit of 32. As such we mistakenly believed that our kernels did not use more than 32 register. After disabling this register limiting we have found that the real register usage was 29 for the Sum1(computeDensity) kernel and 50 for the Sum2(computeForce) kernel.

Obviously this also meant there was greater room for improvement.

Correct Data Types We use float4 for all the float3 values in our implementations. This is necessary since the texture cache does not support float3. In our previous work(Krog [4]) we show that the added overhead is in fact beneficial due to the great performance boost the texture cache provides.

One mistake we made in our previous implementation [4], was that we also used float4 for all the computations in our kernels, thus doing many redundant computations. By converting from float4 to float3 immediately upon memory read, using float3 for all computations and converting to float4 when writing back to memory, we were able to reduce the register usage to 26 and 47 for Sum1 and Sum2 kernel.

Algorithm 7.7 Register improvement for SimpleSPH sum kernels using correct data types.

	SimpleSPH Sum1	SimpleSPH Sum2
Without the restructuring	29	50
With the restructuring	27	47

Code Restructuring By analyzing PTX files and simply doing critical analysis of C code it is possible to find code that can benefit from restructuring. This process is unfortunately often very hit-and-miss, and sometimes seemingly random. This makes it both time and effort-intensive to do this kind of optimization. It is however often possible to save many registers this way.

Here we present an example in the context of the implementation of the uniform grid algorithm (7.1.1.2).

Algorithm 7.9 Improved register usage due to code restructuring using the uniform grid iteration loop.

```
1  // get cell in grid for the given position
2  int3 cell = UniformGridUtils::calcGridCell(position_i, cGridParams.
       grid_min, cGridParams.grid_delta);
3  // iterate through the 3^3 cells in and around the given position
4  for(int z=-1; z<1; ++z) {
5    for(int y=-1; y<=1; ++y) {
6      for(int x=1; x<=1; ++x) {
7        IterateParticlesInCell<O,D>(data, cell+make_int3(x,y,z), index_i,
             position_i, dGridData);
8      }
9    }
10 }
```

```
1  // get cell in grid for the given position
2  int3 cell = UniformGridUtils::calcGridCell(position_i, cGridParams.
       grid_min, cGridParams.grid_delta);
3  // iterate through the 3^3 cells in and around the given position
4  for(uint z=cell.z-1; z<=cell.z+1; ++z)  {
5    for(uint y=cell.y-1; y<=cell.y+1; ++y)  {
6      for(uint x=cell.x-1; x<=cell.x+1; ++x)  {
7        IterateParticlesInCell<O,D>(data, make_int3(x,y,z), index_i,
             position_i, dGridData);
8      }
9    }
10 }
```

	SimpleSPH Sum1	SimpleSPH Sum2
Without the restructuring	27	41
With the restructuring	25	39

Table 7.1: Register improvement for SimpleSPH sum kernels using code restructuring to optimize register usage

Volatile keyword In the C programming language the volatile keyword when used on a variable alerts the compiler that the variable may be modified externally. Essentially it forces to compiler to avoid optimizing the variable.

In CUDA the "volatile trick" works in almost the exactly opposite way, it prevents the compiler from using too many registers. By applying the volatile keyword to variables it is possible to force the compiler to put the value into a register immediately, thus reducing register pressure and allocation. The volatile trick makes NVCC allocate less "virtual" registers in the PTX file, which again helps for the real register usage in the Cubin files.

The volatile trick has been applied as a final optimization phase throughout the entire framework. Here we present a specific example of an improvement in register usage

Algorithm 7.10 Example of using the volatile trick for register usage, here in the uniform grid iteration loop.

```
1   // get cell in grid for the given position
2   volatile int3 cell = UniformGridUtils::calcGridCell(position_i,
        cGridParams.grid_min, cGridParams.grid_delta);
3
4   // iterate through the 3^3 cells in and around the given position
5   for(uint z=cell.z-1; z<=cell.z+1; ++z)  {
6     for(uint y=cell.y-1; y<=cell.y+1; ++y)  {
7       for(uint x=cell.x-1; x<=cell.x+1; ++x)  {
8         IterateParticlesInCell<O,D>(data, make_int3(x,y,z), index_i,
              position_i, dGridData);
9       }
10    }
11  }
```

	SimpleSPH Sum1	SimpleSPH Sum2
Without the volatile trick	25	39
With the volatile trick	22	35

Table 7.2: Register improvement for SimpleSPH sum kernels using the volatile trick to optimize register usage

Miscellaneous It is also possible to save registers by a few more tricks:

- Use the *const* keyword for function parameters, letting the compiler do additional optimizations.
- Passing variables by reference in functions (e.g. &variable)
- Unrolling loops and creating loops
- Avoiding conditionals, it is often better to do redundant computations instead.

7.1.3.3 Neighbor List

Since the summation terms in the SPH calculations dominate the performance of the system REF(results), it was hypothesized that it might be desirable to precompute the lists of particle neighbors. The uniform grid structure makes it possible to only check neighboring particles in neighboring cells, but there are still particles in these cells which are outside the smoothing cutoff length. By combining the first SPH summation step with a neighbor list calculation the second (and third) SPH summations will not have to do redundant checks of neighboring particles.

Unfortunately we found that this optimization had very little performance gain, since the additional computations that are necessary more or less outweigh the gain that is achieved.

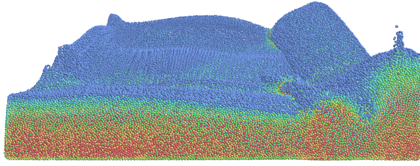
7.1.4 Visualization and Rendering

In order to visualize the fluid properties we do visualization through color gradients. Each particle has a number of properties such as pressure, velocity and deviatoric stress. These values are then converted into a color value for each particle using a gradient calculation. We support combinations of several fluid properties and color gradients.

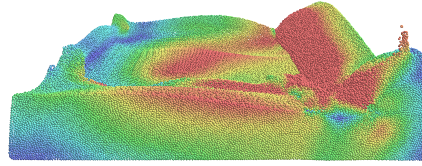
For color we support several gradients, such as Blackish, BlacktoCyan, BlueToWhite and HSVBlueToRed. These gradients are largely self-explaining. The HSVBlue-ToRed gradient is a Hue Saturation Value-model [68].

In addition we also support direct mapping of fluid forces (in 3 spatial dimensions) directly onto the colors in the Red Green Blue (RGB) model. This makes it possible to visualize the internal fluid stresses (Figure 7.9 on page 90), since stress in a spatial direction will produce the color corresponding to that spatial direction.

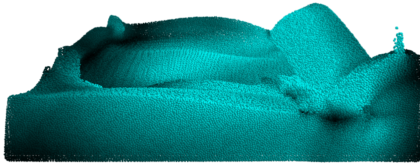
The source code for calculating these gradients can be found in Section B.3.



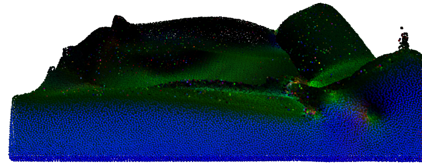
(a) Using the HSV (hue) based color gradient scheme for fluid pressure.



(b) Using the HSV (hue) based color gradient scheme for fluid velocity.



(c) Using the BlackToCyan color gradient scheme for fluid velocity.



(d) Using a direct RGB mapping color gradient scheme for fluid stress forces. Each of the spatial dimensions is mapped to one of the colors in the Red Green Blue (RGB) model.

Figure 7.9: Various color gradients and their application to various fluid properties.

Rendering API Since the simulation framework is now completely separated from the application frontend, an Application Programming Interface (API) is required so as to facilitate interactive rendering.

We choose to implement this API in such a manner that all the internal memory buffers in the simulation can be "overridden" by an external user. This makes it possible to override the particle positions and particle colors buffers. By overriding these buffers with specialized buffers that are in reality Vertex Buffer Objects (VBOs), seamless rendering is possible since CUDA can operate on these buffers as if they were normal memory buffers.

This approach is good for several reasons. By making the simulation framework itself completely rendering agnostic, better organization of the code and architecture is possible. In addition it makes it easier to reuse the simulation framework , since the simulation library has no dependencies on specialized rendering code.

7.2 Rendering Application

When developing the simulation framework it quickly became apparent that it would be desirable to separate the rendering application "frontend" from the simulation library "backend". Our previous implementation of SPH was done using a custom OpenGL application, and while it worked fairly well it was not well suited for further growth of the application.

To promote further growth of the application we have developed a new rendering application based on the open-source 3D graphics engine OGRE.

OGRE has a number of highly desirable features:

- Easy to understand object-oriented interface
- Extensible framework
- Clean and uncluttered design
- Both Direct3D and OpenGL support
- Cross-platform (Windows, Linux, OS X and several others)

In addition the OGRE engine is so stable and well-supported that several large commercial products are now based on it as well.

The advantages of using a well-established engine such as OGRE are many, foremost is the fact that it allows you to build upon the work of others instead of reinventing the wheel. By using an engine that is fairly well-known and standardized the code is also much more understandable for other developers, and also more accessible for others to use.

OGRE provides support for all the features we need, among other things we make use of the terrain component, which allows use to render very large terrains seamlessly in real-time.

7.2.1 Functionality

The rendering application has been used as a test application, and as such there are a large number of features. Interactively exploring the parameter space of the models is very useful both for development and debugging purposes.

Configuration file The rendering application is highly configurable through a configuration file (7.11). Using this configuration file it is possible to specify most relevant parameters of the simulation.

Keyboard shortcuts The rendering application supports a large number of keyboard shortcuts to manipulate the simulation.

Algorithm 7.11 The rendering application configuration file

```
[General]
showOgreConfigDialog = false
showOgreGui = true
cudadevice = 1
logLevel = 1
fluidshader = shader/ParticleBall

[Scene]
cameraRelativeToFluid = true
cameraPosition = 510 500 2000
cameraOrientation = 1 -0.06 0 0
skyBoxMaterial = SkyBoxes/VueSky_Threatening

backgroundColor = 1 1 1
fluidGridColor = 1 0 0

[Fluid]
simpleSPH = false
enabled = true
enableKernelTiming = false
showFluidGrid = true
terrainCollisions = true
gridWallCollisions = false

[FluidParams]
Particles Number = 131072
Timestep = 0.0005
Grid World Size = 1024
Simulation Scale = 0.0005
Rest Density = 1000
Rest Pressure = 0
Ideal Gas Constant = 1.5
Viscosity = 1
Boundary Stiffness = 20000
Boundary Dampening = 256
Static Friction Limit = 0
Kinetic Friction = 0.2

[Terrain]
enabled = true
flat = false
showDebugNormals = false
size = 2049
worldSize = 2250.0
worldScale = 500;
heightDataFile = terrain_2048_alpine3_height_raw32.raw
//normalsDataFile = terrain_2048_alpine3_normal.bmp
textureLayerDiffSpecFile0 =
    terrain_2048_alpine3_shader_base_diffusespecular.png
...
textureLayerNormalHeightFile0 =
    terrain_2048_alpine3_shader_base_normalheight.png
....
textureBlendFile1 = terrain_2048_alpine3_select_thinflowsdeep0.bmp
....
```

Key	Function
Sysrq	Take a screenshot
F	Show advanced frame statistics
R	Use various polygon rendering modes (wireframe etc)
F9	Start screencapture mode (save all rendered frames to disk)
G	Toggle rendering of fluid grid
N	Toggle rendering of terrain normals
1-9	Change the fluid scene (placement of particles)
O	Disable integration step in simulation
Left	Move fluid grid in +x-axis
Right	Move fluid grid in -x-axis
Up	Move fluid grid in +z-axis
Down	Move fluid grid in -z-axis
Shift-Up	Move fluid grid in +y-axis
Shift-Down	Move fluid grid in -y-axis
PageUp	Increase number of particles by 1000
PageDown	Decrease number of particles by 1000
Shift-PageUp	Double the number of particles.
Shift-PageUp	Halve the number of particles
(Shift-)Pluss	Increase the time step by 0.00001 (0.0001)
(Shift-)Minus	Decreasethe time step by 0.00001 (0.0001)

7.2.2 Terrain Support in Ogre

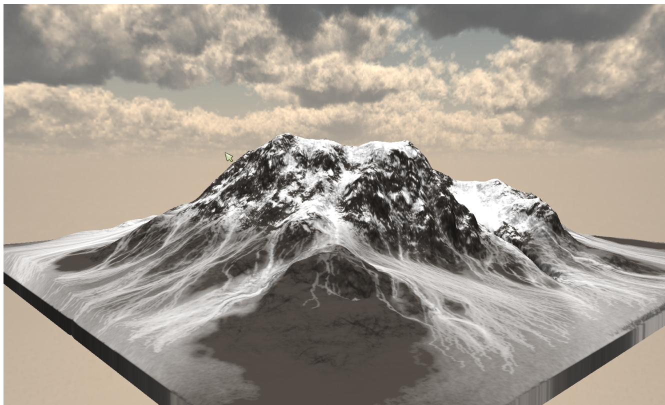


Figure 7.10: Ogre terrain rendering with a pregenerated skybox.

Using the new terrain rendering component that is available in Ogre 1.7 it is possible to render very large terrains seamlessly (Figure 7.10 on page 93). The new terrain rendering uses a seamless level of detail (LOD) process where the rendering detail

is decreased as you move further away from the terrain (Figure 7.11 on page 94). This process makes it possible to render very large and detailed terrains.

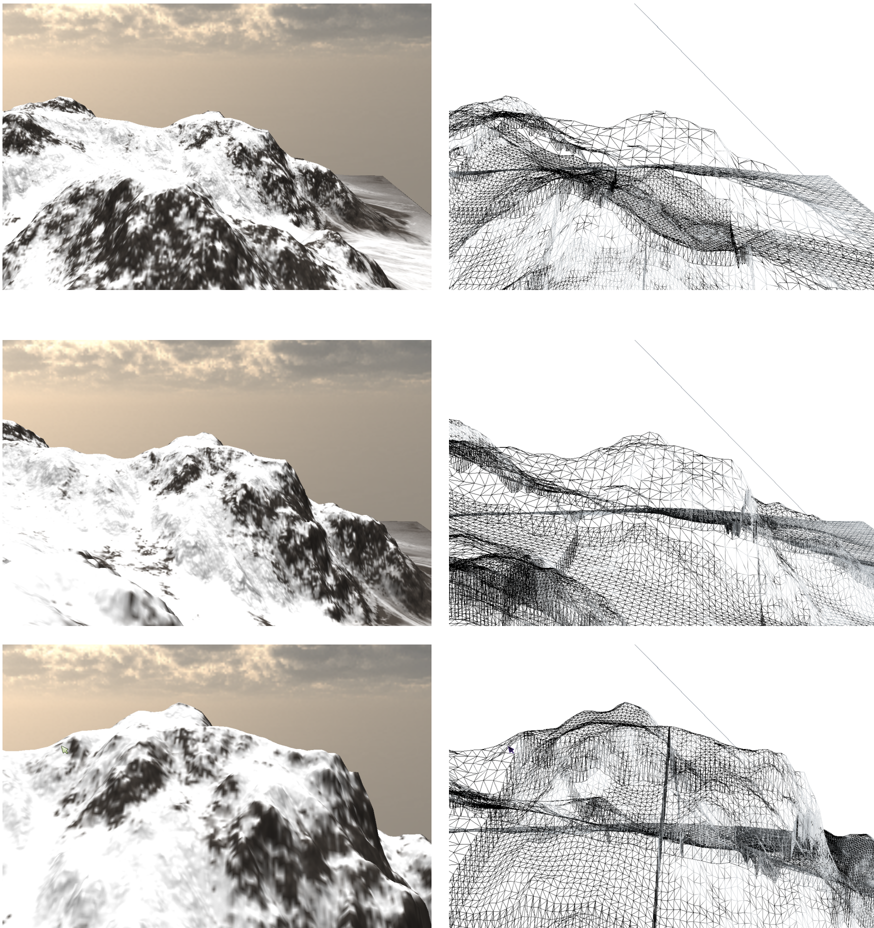


Figure 7.11: Ogre terrain rendering with seamless level of detail (LOD) for various distances.

7.2.3 Visualization of Particles

Due to a lack of time we could not implement a proper surface rendering technique for the fluid particles. Instead we chose to reuse the existing “ball” shader from the previous implementation (Figure 7.12 on page 95).

In order to support both OpenGL and Direct3D, the shader had to be rewritten in C for Graphics (CG). The rewritten shader provides the same visual output as that used in the previous implementation, for more details regarding this, please refer to Krog [4].

The source code for the new CG shader and Ogre material can be found in Section B.1.

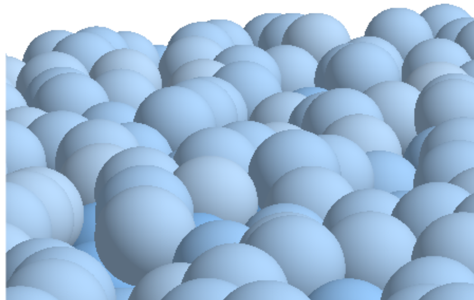


Figure 7.12: Closeup of fluid particle ball shading.

7.3 Console Test Application

In our previous implementation [4] we found that doing performance measurements in an application that also does real-time rendering is not very accurate. The overhead of rendering can be significant and in addition it can fluctuate. As part of the development of the framework it was decided that we would create a small console application that uses the simulation library and does nothing except perform simulations and do performance measurements.

The test application (Figure 7.13 on page 96) has made it much easier to do performance measurements, and has also made it possible to make these measurements more systematic and accurate.

Figure 7.13: The console test application running a performance scaling measurement test.

The console application in itself is very simple, the real power lies in the fact that the SPH framework and SPH implementations have been completely separated into their own library, such that it is very simple to use them from the console application.

Chapter 8

Results and Discussion

In this chapter we will present and discuss our results. The performance of the system will be evaluated and compared to both our earlier implementation and other state-of-the-art implementations. Interesting effects such as the performance impact of real-time rendering and the performance increase from using the new Fermi GPUs will be shown.

The visual results of the implementations will be evaluated in light of the interactivity criteria.

8.1 Test Setup

For all our performance results we a fairly high-end computer equipped a Intel Core2 Quad Q9550 processor (Table 8.1 on page 97). We use three different graphics cards, an NVIDIA GeForce GTX 260, a GeForce GTX 470 and a Tesla C2050. The specifications for these can be found in Table 5.1 on page 46.

8.1.1 Software

For comparing and evaluating the simulation performance of the system, a "realistic" test was designed.

	Test System
CPU	Intel Core2 Q9550
Processor	2.83 GHz
Bus Speed	1333 Mhz
L2 Cache Size	12MB
Memory	4GB
Type	DDR3
Speed	533.3 MHz
MainBoard	EVGA 132-CK-NF79

Table 8.1: Test system.

Parameter	Value for Simple SPH
timestep	0.0005
rest_density	1000.0
rest_pressure	0
external_stiffness	20000.0
external_dampening	256.0
viscosity	1.0
sim_scale	0.0005

Table 8.2: The default simulation parameters used in performance testing.

To ensure consistent results, the simulation is set to a static simulation setup where a cubical volume of fluid is dropped into a shallow pool of fluid (Figure 8.15 on page 118). This setup is good because it contains a wide range of states for the fluid, from low to high pressure. To collect absolute performance numbers, the average of a large number of iterations (1000) is used.

Simulation parameters The simulation parameters (Table 8.2 on page 98) were chosen for their stability and their realistic behavior as a fluid. A set of good starting parameters were collected from several other SPH models and then tweaked until they gave good results.

For the Simple SPH model, the task of finding parameters was an exhaustive process, due to the many combinations. For the Complex SPH model this is an even harder problem, since the viscosity of the fluid changes due to a rheological model. Performance measurements of the Complex SPH model is difficult due to the additional complexity introduced by this rheological model. To improve the consistency and validity of the results we chose to use a simple Newtonian fluid rheology for performance measurements of the Complex SPH model as well. This effectively reduces the fluid behavior to almost the same as the Simple SPH model, but all the computational cost of the Complex model is kept.

Fluid Volume We test across a wide range of particles, it is important to remember that this will also change the amount of fluid that is simulated.

The number of particles in a volume can be found by [35]:

$$n = \rho \frac{V}{m} \tag{8.1}$$

Using this for the above parameters give us a fluid volume of:

$$V = \frac{1}{\rho} m * n = \frac{1}{1000} \frac{kg}{m^3} 0.00002kg * 131027 = 0,00262144m^3 \tag{8.2}$$

It follows that if we double the number of particles the amount of fluid simulated will double as well:

$$V = \frac{1}{\rho} m * n = \frac{1}{1000} \frac{kg}{m^3} 0.00002kg * 2622144 = 0,00524288m^3 \quad (8.3)$$

This is not desirable from performance measurement standpoint, since we want to ensure that the fluid behaves in the same manner for each performance run. We avoid this problem by simply compensating the particle mass (halving mass when doubling number of particles), this also affects other parameters of the fluid. In Table 8.3 on page 99 the various parameters can be seen in the range from 32K to 512K particles.

Parameter/Particles	32K	64K	128K	256K	512K
Particle Mass	0.0008	0.0004	0.0002	0.0001	0.00005
Particle Rest Distance	0.00807636	0.00641021	0.00508779	0.00403818	0.00320511
Boundary Distance	0.00403818	0.00320511	0.0025439	0.00201909	0.00160255
Smoothing Length	0.0161527	0.0128204	0.0101756	0.00807636	0.00641021
Grid Cell Size	32.3055	25.6409	20.3512	16.1527	12.8204

Table 8.3: The simulation parameters scaled across different resolutions, from 32K to 512K particles.

All our performance measurements are in powers of 2, e.g. 32K particles is $32*1024 = 32768$ particles.

8.1.2 Methodology

Throughout development considerations were taken to ensure consistent results when testing. To ensure consistent results, any and all parameters are initialized to the same values.

We initialize the random function with static values in order to ensure the same random values are used every time the program runs. We disable nonessential functionality when gathering performance data, and when measuring the performance of the simulation real-time rendering is not enabled. When rendering is enabled we ensure that artificial limiters, such as vertical refresh synchronization (vsync) is turned off.

In addition to these precautions we test variations in one parameter at a time, if at all possible.

8.1.2.1 Performance Testing Application

In our previous implementation of SPH ([4]) we found that the overhead associated with rendering could be significant. To avoid the overhead rendering imposes on the

Algorithm 8.1 Using CUDA Timing API for timing kernels.

```
#ifdef LOG_TIMINGS
  CUT_SAFE_CALL(cutResetTimer(timer));
  float timer;
  CUT_SAFE_CALL(cutCreateTimer(timer));
  CUT_SAFE_CALL(cutStartTimer(timer));
#endif
  K_Integrate<<<hFluidParams.numBlocks, hFluidParams.numThreads>>>(......)
      ;
  CUT_CHECK_ERROR("Kernel execution failed");
  CUDA_SAFE_CALL(cudaThreadSynchronize());
#ifdef LOG_TIMINGS
  CUT_SAFE_CALL(cutStopTimer(timer));
  time_integrate = cutGetTimerValue(timer);
#endif
```

performance results, we have now developed a separate test application which does no rendering whatsoever. This was only possible due to the fact that all simulation code is now separated into a separate library, with an API for providing rendering buffers. The default operation mode of the simulation framework is in fact to operate without any rendering whatsoever, which means that the test application does nothing special in order to operate without the overhead of rendering.

The performance testing application simply initializes the simulation framework with consistent parameters and does performance timing on the simulation iterations.

8.1.2.2 CUDA Kernel Timing

In a previous implementation we used the CUDA timing API (8.1) to measure the performance of individual kernels. On Windows this API uses the QueryPerformanceFrequency, the most accurate CPU clock that is available. We have since discovered that this API is in fact not very accurate since it does use a CPU clock for measuring things on the GPU. Fortunately this inaccuracy did not impact the results of our previous work ([4]) too much, since we used averages of many measurements. In addition this API requires a synchronization after the kernel to ensure that the GPU is finished. This also impacts the measurements.

A better API for timing is the CUDA event API (8.2), which does not rely on a user-mode CPU clock for the measurements and instead uses GPU streams to do timing. Using this new API the accuracy of kernel measurements is improved and there is no need to synchronize the CPU and the GPU, providing a much more accurate image of the total algorithm performance. For algorithms with several GPU kernels, the synchronization overhead can be significant.

Algorithm 8.2 Using CUDA event API for timing kernels.

```
#ifdef LOG_TIMINGS
  e_start = new cudaEvent_t;
  e_stop = new cudaEvent_t;
  cudaEventCreate((cudaEvent_t *)e_start);
  cudaEventCreate((cudaEvent_t *)e_stop);
  cudaEventRecord(*((cudaEvent_t *)e_start), 0);
#endif
  K_Integrate<<<hFluidParams.numBlocks, hFluidParams.numThreads>>>(......)
      ;
  CUT_CHECK_ERROR("Kernel execution failed");
#ifdef LOG_TIMINGS
  cudaEventRecord(*((cudaEvent_t *)e_stop), 0);
  cudaEventSynchronize(*((cudaEvent_t *)e_stop));
  float time_integrate;
  cudaEventElapsedTime(&time_integrate , *((cudaEvent_t *)e_start), *((
      cudaEvent_t *)e_stop));
#endif
```

8.2 Performance Evaluation

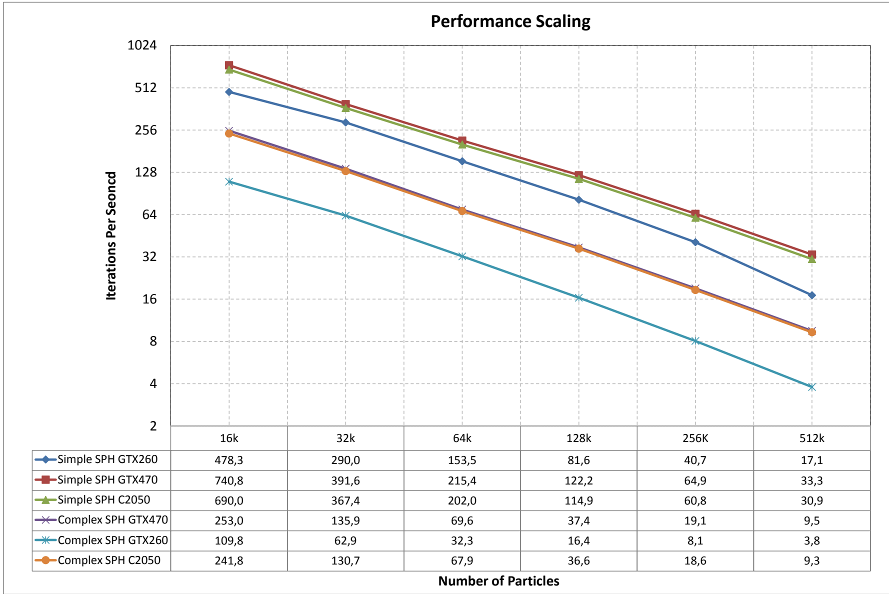
To evaluate our optimizations and the performance of our new SPH model we do a thorough performance evaluation of the two SPH implementations. We would like to note that most of the graphs in this section use log2 for both axes.

8.2.1 Performance Scaling

The performance scaling of the implementations are nearly linear (Figure 8.1 on page 102). It is interesting to note that the two Fermi-cards, the GeForce GTX 470 and the Tesla C2050 seems to scale more linearly than the old GeForce GTX 260. This slight nonlinearity may be contributed to the effects of both texture cache as well as L1 and L2 cache on the GTX470 (Fermi)

We can also see that the Tesla C2050 has performance that is almost identical to the GeForce GTX 470, though a little bit lower. This difference in performance can most likely be attributed to the difference in memory speed (1500 MHz *V.S* 1674 MHz) and clock speed (1150 MHz *V.S* 1215 MHz) (Table 5.1 on page 46).

It is interesting to note that though the Tesla has greater memory bandwidth due to the larger memory bus (384 bit *V.S* 320 bit), this does not seem to help counteract the slightly lower clocks. Our implementation is single precision only, so we can utilize the greatest feature of the Tesla; much better double precision performance. We can however use the much larger memory, which is described in 8.2.2.2.



	16k	32k	64k	128k	256K	512k
Simple SPH GTX260	478,3	290,0	153,5	81,6	40,7	17,1
Simple SPH GTX470	740,8	391,6	215,4	122,2	64,9	33,3
Simple SPH C2050	690,0	367,4	202,0	114,9	60,8	30,9
Complex SPH GTX470	253,0	135,9	69,6	37,4	19,1	9,5
Complex SPH GTX260	109,8	62,9	32,3	16,4	8,1	3,8
Complex SPH C2050	241,8	130,7	67,9	36,6	18,6	9,3

Figure 8.1: Performance scaling of the two SPH implementations, here in a graph with both axes in log2.

8.2.2 Memory Scaling

Since our implementations do not use constant or shared memory to any significant degree, the only memory usage that is of importance is the usage of the global memory on the device.

8.2.2.1 Memory Usage

Data Structure In a previous work [4] we found that the memory usage of the acceleration data structure (7.1.1.2) we employ is very sparse. Had the uniform grid been allocated directly the usage would be much larger, the hashing means that the only significant overhead is the memory used for tracking start/end indices for the cells and for cell hashes. The overhead of the cell indices and the cell hashes can be represented by the following formula:

$$2 * sizeof(uint) * (n_{particles} + n_{cells})\, bytes = 8 * n_{particles} + 8 * n_{cells}\, bytes$$

We have also found that the radix sort (Satish et al. [63]) allocates temporary memory. The amount is determined by the following formula:

$$n_{particles} * (2 + 3 * 8/256) * sizeof(uint)\, bytes = 8.375 * n_{particles}\, bytes$$

Giving us a total formula for the data structure memory usage:

$$8 * n_{particles} + 8 * n_{cells} + 8.375 * n_{particles}\, bytes \tag{8.4}$$

The number of cells for a given simulation depends on the smoothing length, the simulation domain size and also the scale of the simulation. For the parameters we use in our testing, the following number of cells is used:

Particles	16K	32K	64K	128K	256K	512K
Cells	17576	32768	64000	132651	262144	512000

The number of cells is roughly the same as the number of particles, meaning we can approximate the total memory usage as:

$$24.375 * n_{particles}\, bytes \tag{8.5}$$

Simple SPH model In our previous implementation of the Simple SPH model we achieved a memory usage of:

$$2 * sizeof(ParticleParams) * n_{particles} = 176 * n_{particles}$$

for just the SPH parameters (excluding the overhead of the data structure).

In our new implementation we have been able to remove some redundant buffers, thus improving the memory usage of the system.

The size of the particle parameter buffers is found by taking the sum of the following buffers:

Name	Data Type	Bytes
Position	float4	16
Velocity	float4	16
Veleval	float4	16
Color	float4	16
PositionSorted	float4	16
VelocitySorted	float4	16
VelevalSorted	float4	16
ColorSorted	float4	16
ForceSorted	float4	16
PressureSorted	float	4
DensitySorted	float	4
SUM		152

Table 8.4: Parameter buffers in Simple SPH implementation

Giving us a new formula for the memory usage:

$$152 * n_{particles}\, bytes$$

For a total memory usage of

$$176.376 * n_{particles}\, bytes \tag{8.6}$$

Complex SPH model The memory usage for the new Complex SPH model is significantly higher due to the use of tensor matrices (3x3) and additional techniques such as XSPH.

Name	Data Type	Bytes
Position	float4	16
Velocity	float4	16
Veleval	float4	16
Color	float4	16
PositionSorted	float4	16
VelocitySorted	float4	16
VelevalSorted	float4	16
ColorSorted	float4	16
XSPHSorted	float4	16
ForceSorted	float	16
PressureSorted	float	4
DensitySorted	float	4
StressTensorSorted	matrix3(3xfloat4)	48
SUM		216

Table 8.5: Parameter buffers in Simple SPH implementation

Giving us a formula for the memory usage of just the SPH parameter buffers:

$$216 * n_{particles}\, bytes \tag{8.7}$$

For a total memory usage of

$$240.376 * n_{particles}\, bytes \tag{8.8}$$

8.2.2.2 Analysis

Our analysis of the memory consumption of the implementation show that the memory scaling of the implementations are linear (8.2), and scale very well.

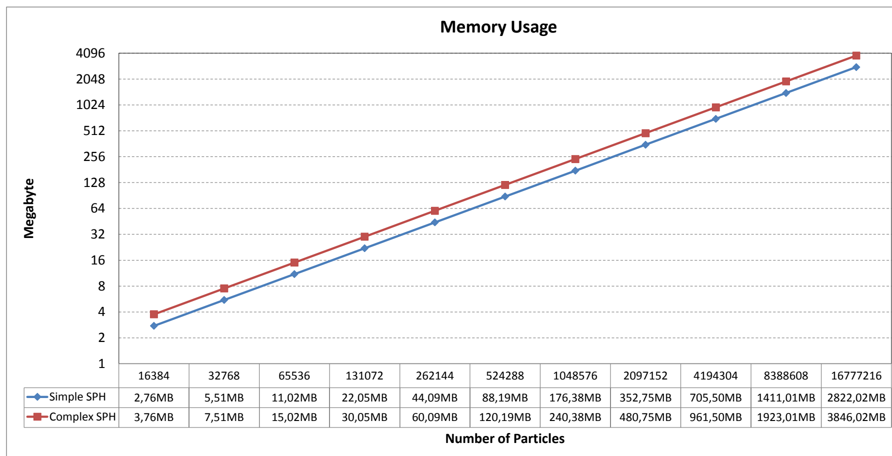


Figure 8.2: Memory usage of the two SPH models.

Using the NVIDIA Tesla C2050 (5.1 on page 46), we have simulated up to 12 millions particles with the Simple SPH model, thus using roughly 2GB of memory.

With this many particles there are in fact a few problems, among them the fact that the kernel grid size becomes larger than 64K, the maximum for current GPU architectures. We mitigate this problem by increasing the block size (the number of threads per block), this does however lead to lower occupancy and thus non-optimal performance.

Though our analysis shows that it should be possible to simulate more than this amount we have found that in practice is is difficult to allocate all the memory on the GPU.

This problem can be explained for the consumer cards such as the GeForce GTX 470 (Table 5.1 on page 46), since they use a large part of the memory for the Operation System (Figure **??** on page **??**).

To mitigate this problem NVIDIA has created special compute-only drivers that makes it possible to allocate more of the GPU memory.

Another problem related to large simulation sizes is the the fact that the Windows WDDM display driver system has a watchdog timer that automaticall restarts the display driver if it is unresponsive for more than a few seconds. When doing large simulations each kernel can take long enough to execute that the watchdog timer is tripped. Again, the solution is to use the specialized compute-only driver. It is also possible to disable the watchdog timer, but this is not recommended since it serves a real purpose.

8.2.3 Optimizations

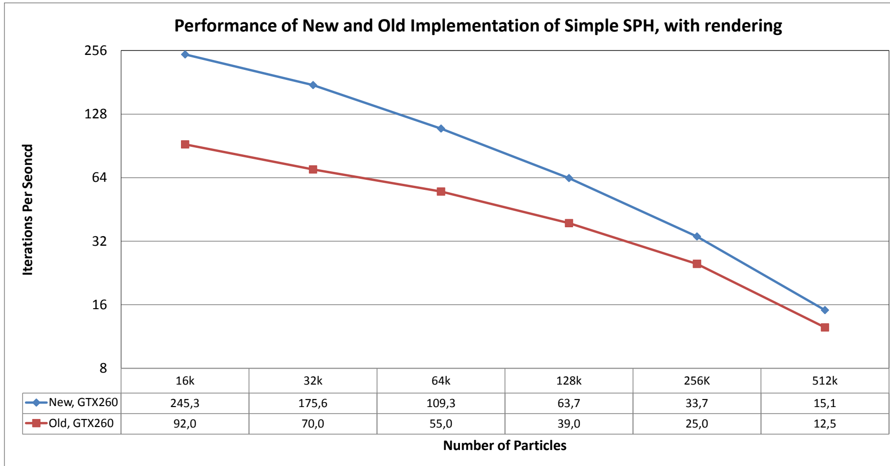


Figure 8.3: Performance comparison of new and old implementations of the Simple SPH model.

We have reimplemented a previous implementation of the Simple SPH model using our new framework.

For this comparison we use performance numbers from the performance measurements with rendering, since all the measurements in the old implementation were done with rendering.

Due to significant optimizations of the framework we observe a significant performance increase (Figure 8.3 on page 106). For a more detailed description of the techniques which provided this improvement please refer to 7.1.3.

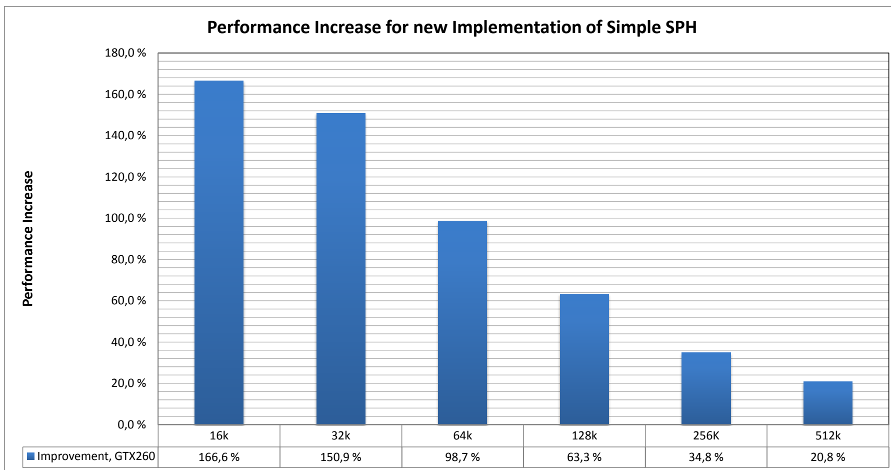


Figure 8.4: Performance increase for new and old implementations of the Simple SPH model.

We achieve as much as a 167% increase in performance for 16K Particles, and decreasing improvements with larger amount of particles (Figure 8.4 on page 107).

8.2.4 Fermi performance

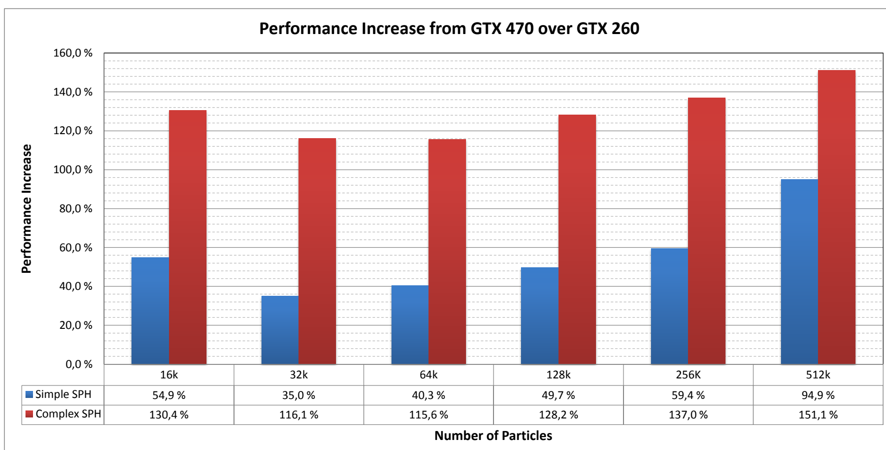


Figure 8.5: Performance comparison between the GeForce GTX 470 and the GeForce GTX 260.

The new GF100/Fermi-architecture provides a large boost in performance compared to the GT200-architecture.

In Figure 8.5 on page 107 we show the performance increase from using a GeForce GTX 470 over a GeForce GTX 260. The Complex SPH model benefits the most, with improvements up to 150%, while the Simple SPH model sees improvements up to 95%. This large increase in performance can be attributed to a combination of the new architectural features in the Fermi architecture and high memory clocks due to the use of GDDR5 instead of GDDR3 (Table 5.1 on page 46).

8.2.5 Review of other implementations

We have compared our implementation performance for the Simple SPH model with that of other implementations. This algorithm has been widely implemented since it is very well suited for interactive or "real-time" simulation and as such it is possible to find comparable implementations.

Unfortunately we have found that it is near impossible to do a review of earlier implementations that is both comprehensive and accurate since most authors do not specify all the parameters they use. In addition there are slight differences in the SPH models and finally also because of different hardware used. Nonetheless we have attempted a comparison, if only to give a rough picture of the performance landscape.

It is worth noting that our comparison includes both earlier GPU as well as CPU implementations (Figure 8.6 on page 109).

GPU Implementations We find that our GPU implementation is significantly faster then earlier GPU implementations, even for implementations using faster graphics cards, such as the one by Yan et al. [69] where they use a NVIDIA GTX 280 and get 66 iterations per second at 16K particles. Comparing their implementation against our implementation running on a GTX260 (without rendering) we see a 6x speedup. It is also interesting to note that our implementation seems to scale better, though the available data is not enough to draw any conclusions.

Harada et al. [48] implemented SPH on the GPU using OpenGL and Cg. Their methods achieves real-time performance (17 frames per second) with 60000 particles on an NVIDIA GeForce 8800GTX. This is an earlier GPU architecture, so we will not do a direct comparison against this implementation.

Zhang et al. [70] also implemented SPH on the GPU (a NVIDIA GeForce 8800GTX) with a performance of 56 frames per second at 60000 particles on the GPU. This is also an earlier GPU architecture, so we will not do a direct comparison against this implementation either.

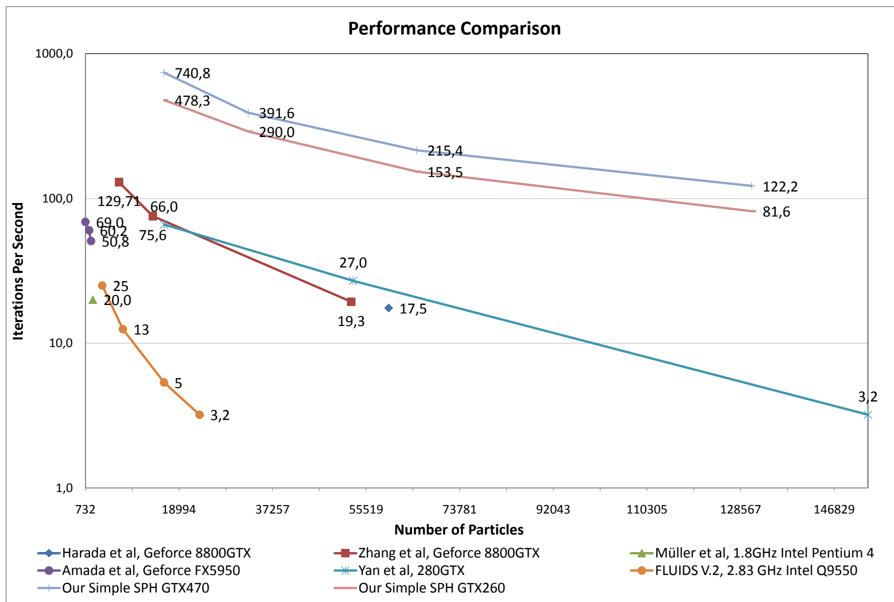


Figure 8.6: Performance comparison to several earlier SPH implementations. Here in a graph with linear x-axis and log10 y-axis to make it possible to show the large differences in performance.

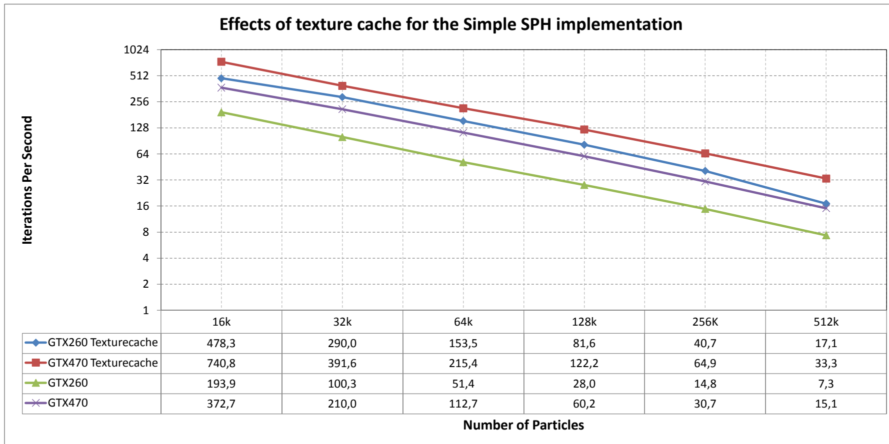
CPU Implementations A very interesting comparison is with the FLUIDS V.2 software, which is a highly optimized SPH implementation for the CPU. We tested the FLUIDS software on test system 2 (Table 8.1 on page 97), a 2.83 GHz Quad-CPU Intel.

Unfortunately FLUIDS can only use on of the cores in this CPU so it should be assumed that the performance could be almost quadrupled using all 4 cores. Event assuming so the performance results would still be very low compared to our GPU implementation. Comparing the FLUIDS software with our GPU implementation (with rendering), we see speedups of 91x for the GeForce GTX 470 and 49x for the GeForce GTX 260 at 16K particles.

Another CPU implementation is the one by Müller et al. [1] , who created the original "Simple" SPH model. This comparison is perhaps only interesting as a measure of how much the computation power of commodity hardware has increase during the last decade, since their measurements is on a 1.8Ghz Pentium 4, not a very fair comparison. Nonetheless they achieve 20FPS with 2200 particles. We measure 950 iterations per second using the NVIDIA GTX 470 at 2200 particles, a very large increase in performance.

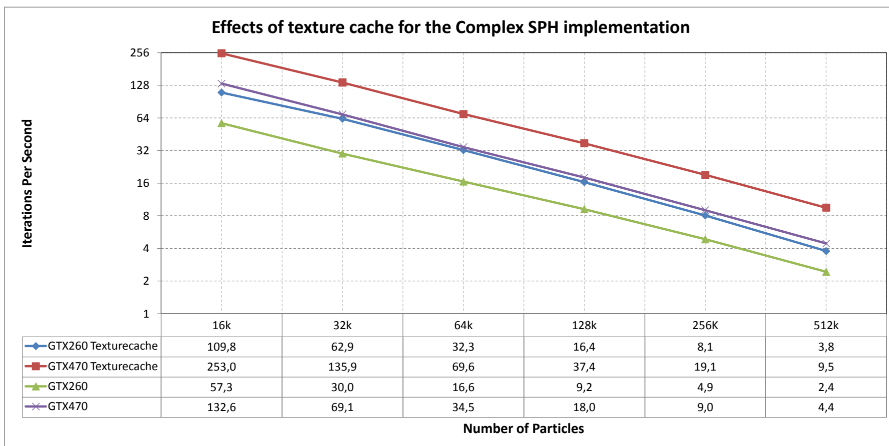
8.2.6 Effects of Texture Cache

As with our previous implementation we find that the texture cache has a very significant impact on performance(Figure 8.7 on page 110). We have also tested our new implementation on the GeForce GTX 470 (Fermi) card.



	16k	32k	64k	128k	256K	512k
GTX260 Texturecache	478,3	290,0	153,5	81,6	40,7	17,1
GTX470 Texturecache	740,8	391,6	215,4	122,2	64,9	33,3
GTX260	193,9	100,3	51,4	28,0	14,8	7,3
GTX470	372,7	210,0	112,7	60,2	30,7	15,1

Figure 8.7: Performance effects in absolute values due to texture cache for the Simple SPH model



	16k	32k	64k	128k	256K	512k
GTX260 Texturecache	109,8	62,9	32,3	16,4	8,1	3,8
GTX470 Texturecache	253,0	135,9	69,6	37,4	19,1	9,5
GTX260	57,3	30,0	16,6	9,2	4,9	2,4
GTX470	132,6	69,1	34,5	18,0	9,0	4,4

Figure 8.8: Performance effects in absolute values due to texture cache for the Complex SPH model

The new Fermi architecture has a new L1 and L2 cache. As described in 5.3.3.2 the L2 cache works for global memory reads and the L1 cache works for both

local memory (register spills) and global memory. In addition the L1 cache has much higher bandwidth than the texture cache, meaning that for the Fermi-line of architectures the use of the texture cache can hurt performance.

These factors may be why the texture cache is not as significant for performance on the Fermi card as it is on the earlier cards (Figure 8.9 on page 111), however we still find that the use of the texture cache is advantageous. This is probably because the texture cache is optimized for spatial locality in memory accesses, something which helps us a lot since the data structure we employ exhibit exactly this spatial locality.

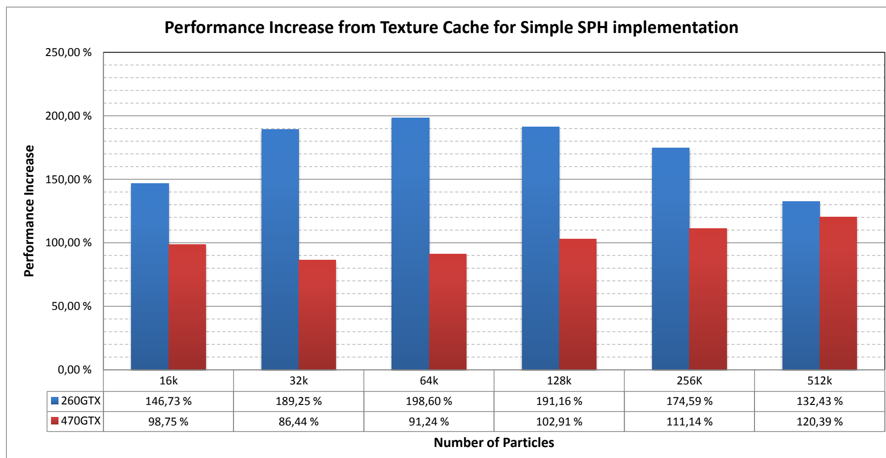


Figure 8.9: Performance increase in percent due to texture cache for the Simple SPH model

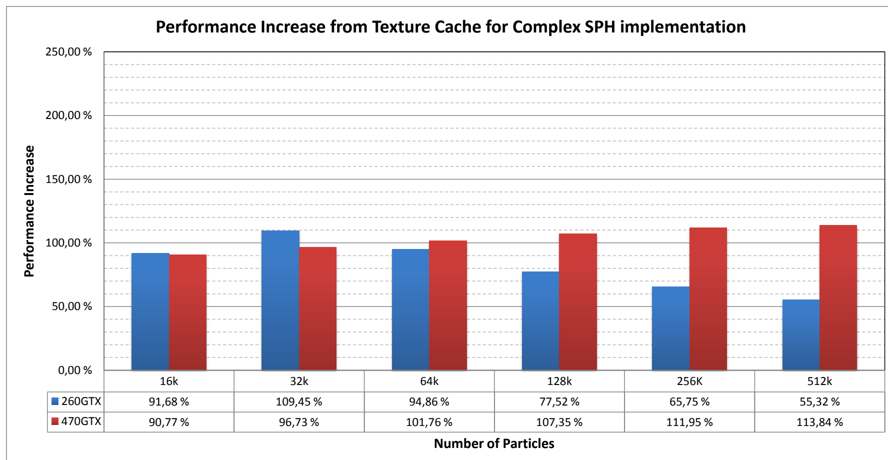
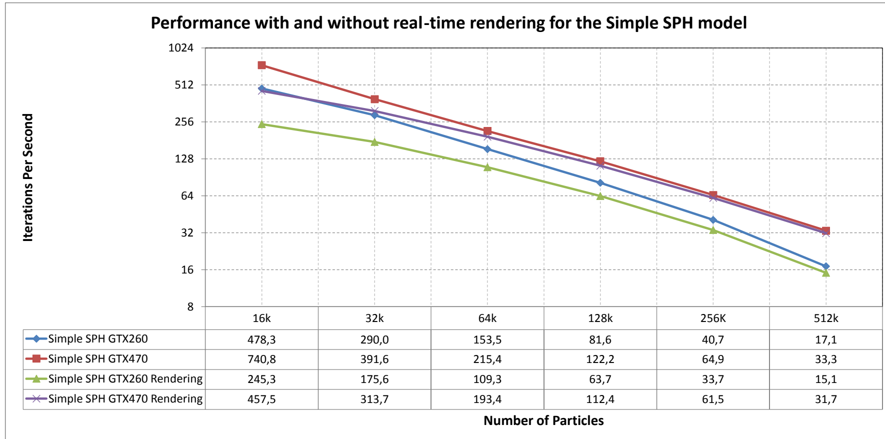


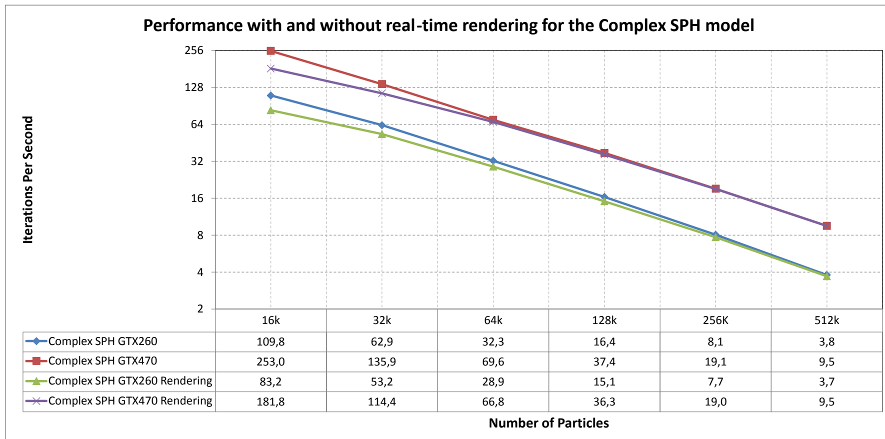
Figure 8.10: Performance increase in percent due to texture cache for the Complex SPH model

8.2.7 Rendering Overhead



	16k	32k	64k	128k	256K	512k
Simple SPH GTX260	478,3	290,0	153,5	81,6	40,7	17,1
Simple SPH GTX470	740,8	391,6	215,4	122,2	64,9	33,3
Simple SPH GTX260 Rendering	245,3	175,6	109,3	63,7	33,7	15,1
Simple SPH GTX470 Rendering	457,5	313,7	193,4	112,4	61,5	31,7

(a) Performance of real-time rendering for the Simple SPH model.

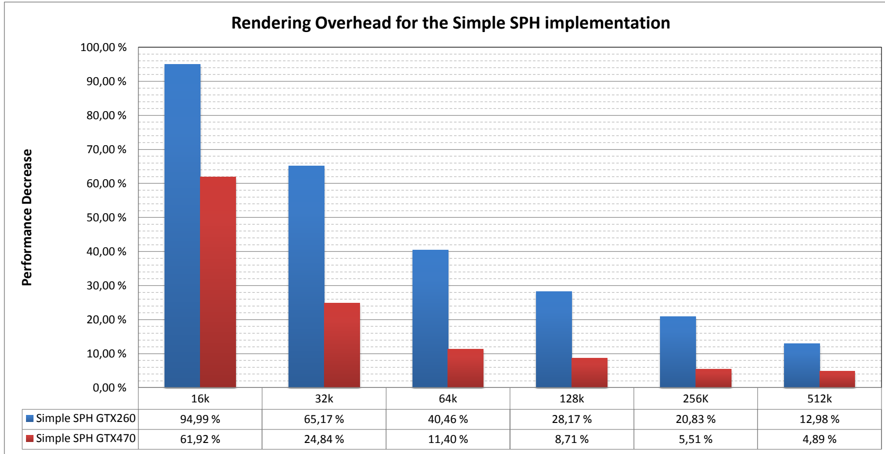


	16k	32k	64k	128k	256K	512k
Complex SPH GTX260	109,8	62,9	32,3	16,4	8,1	3,8
Complex SPH GTX470	253,0	135,9	69,6	37,4	19,1	9,5
Complex SPH GTX260 Rendering	83,2	53,2	28,9	15,1	7,7	3,7
Complex SPH GTX470 Rendering	181,8	114,4	66,8	36,3	19,0	9,5

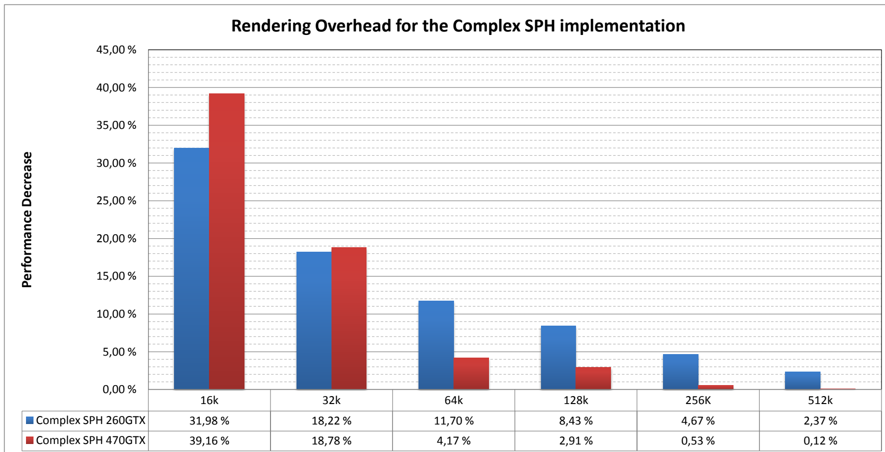
(b) Performance of real-time rendering for the Complex SPH model.

Figure 8.11: Performance of real-time rendering.

Part of the reason for the development of our new framework was that we wanted to measure the performance overhead of the real-time rendering. Our rendering technique is fairly simple, so it was thought that this overhead was negligible, but our findings are in fact the opposite. We find that the overhead of rendering is in fact fairly high (8.11a and 8.12a).



(a) Performance overhead from real-time rendering for the Simple SPH model, in percent of performance decrease.



(b) Performance overhead from real-time rendering for the Complex SPH model, in percent of performance decrease.

Figure 8.12: Performance overhead from real-time rendering.

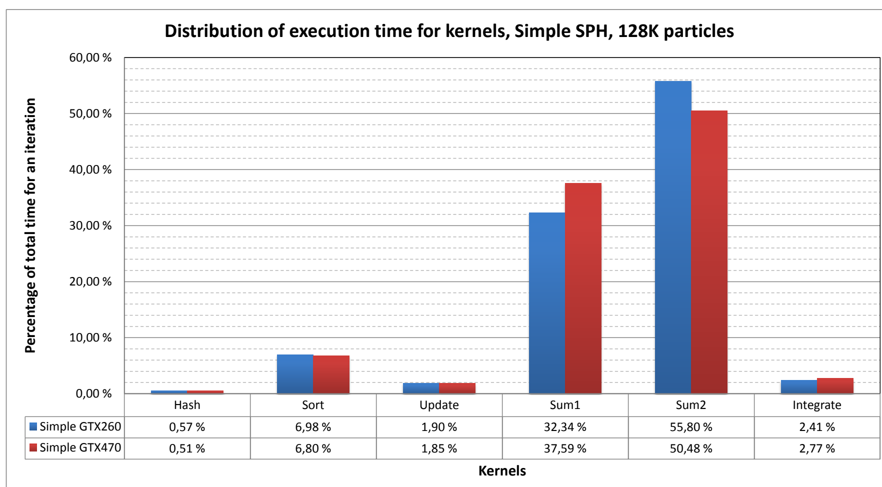
8.2.8 Kernels

We have measured the relative performance of the different kernels in our two SPH implementations. Our findings show that the most performance intensive parts are in the calculation of the SPH summation over neighboring particles.

In the Simple SPH algorithm the Sum2 step is the most demanding step, but close behind is the Sum1 step (Figure 8.13 on page 115).

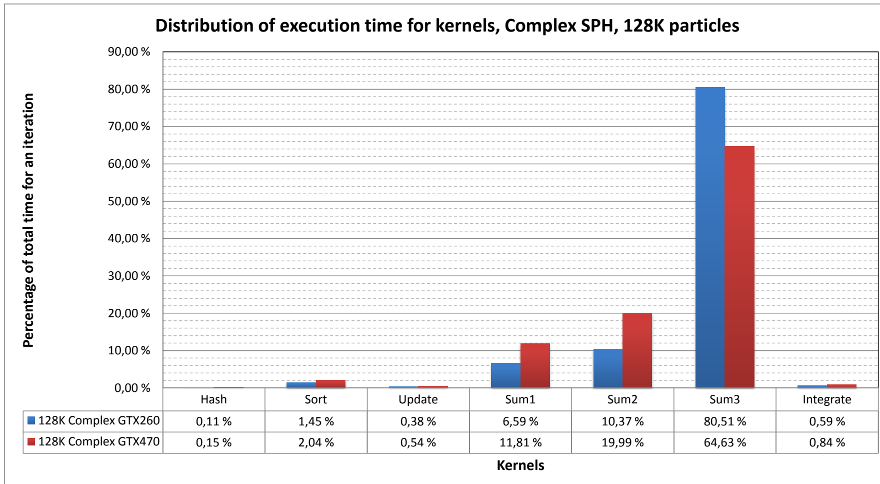
For the Complex SPH algorithm the Sum3 step completely dominates the overall performance 8.14. This is due to the large amount of memory reads that are necessary in this step.

If further optimizations of the implementation are to be carried it, they should focus on improving the coalescing in these steps, and to do so it may be necessary to investigate alternative data structures.



	Hash	Sort	Update	Sum1	Sum2	Integrate
Simple GTX260	0,57 %	6,98 %	1,90 %	32,34 %	55,80 %	2,41 %
Simple GTX470	0,51 %	6,80 %	1,85 %	37,59 %	50,48 %	2,77 %

Figure 8.13: Distribution of execution time among the different kernels (steps in the SPH algorithm) for the Simple SPH model, here for 128K particles.



	Hash	Sort	Update	Sum1	Sum2	Sum3	Integrate
128K Complex GTX260	0,11 %	1,45 %	0,38 %	6,59 %	10,37 %	80,51 %	0,59 %
128K Complex GTX470	0,15 %	2,04 %	0,54 %	11,81 %	19,99 %	64,63 %	0,84 %

Figure 8.14: Distribution of execution time among the different kernels (steps in the SPH algorithm) for the Complex SPH model, here for 128K particles.

8.3 Visual

The visual results of the simulation is important since it allows us to gauge the accuracy with which we reproduce the desired behavior. One of the principal goals in this thesis, is to achieve interactive performance, and we define this not just based on a performance measure such as the frames per second, but also as how the fluid animation "feels". When doing simulation it is often not practical to simulate large amounts of fluid, so most simulations scale both the simulation domain and the time.

Both our SPH implementations use an Equation of State to enforce the incompressibility, which means that we have to balance the timestep carefully. If the timestep is too large the simulation becomes unstable and inaccurate, but if the timestep is too low the perceived speed of the simulation is too slow.

8.3.1 Simple SPH

In our previous implementation we achieved real-time performace (interactivity) with 23 FPS at 256K particles using an NVIDIA GeForce GTX 260. Thanks to the performance optimizations we have performed, it is now possible to achieve 34 FPS. Due to the increase in performance, interactivity is improved, and it is possible use more particles (greater accuracy) with the same performance.

We find that for a viscosity of 1 we can use a time-step of 0.002. This is near the maximum possible while still maintaining stability. With the GeForce GTX 470

we get 60 frames per second (with rendering) at 256K and this produces a very believable fluid behavior.

The visual results from the Simple SPH model is much the same as in our previous implementation, but we include some screenshots (8.15) for reference purposes.

In addition we also include some screenshots of the Simple SPH model with a terrain boundary (8.16). Though we have not had time to work on this aspect of the simulation, it is fairly easy to imagine how the model could be used to simulate water and other low-viscosity fluids flowing on a terrain, for example in a river. This kind of simulation has also seen much use in simulating dam breaks.

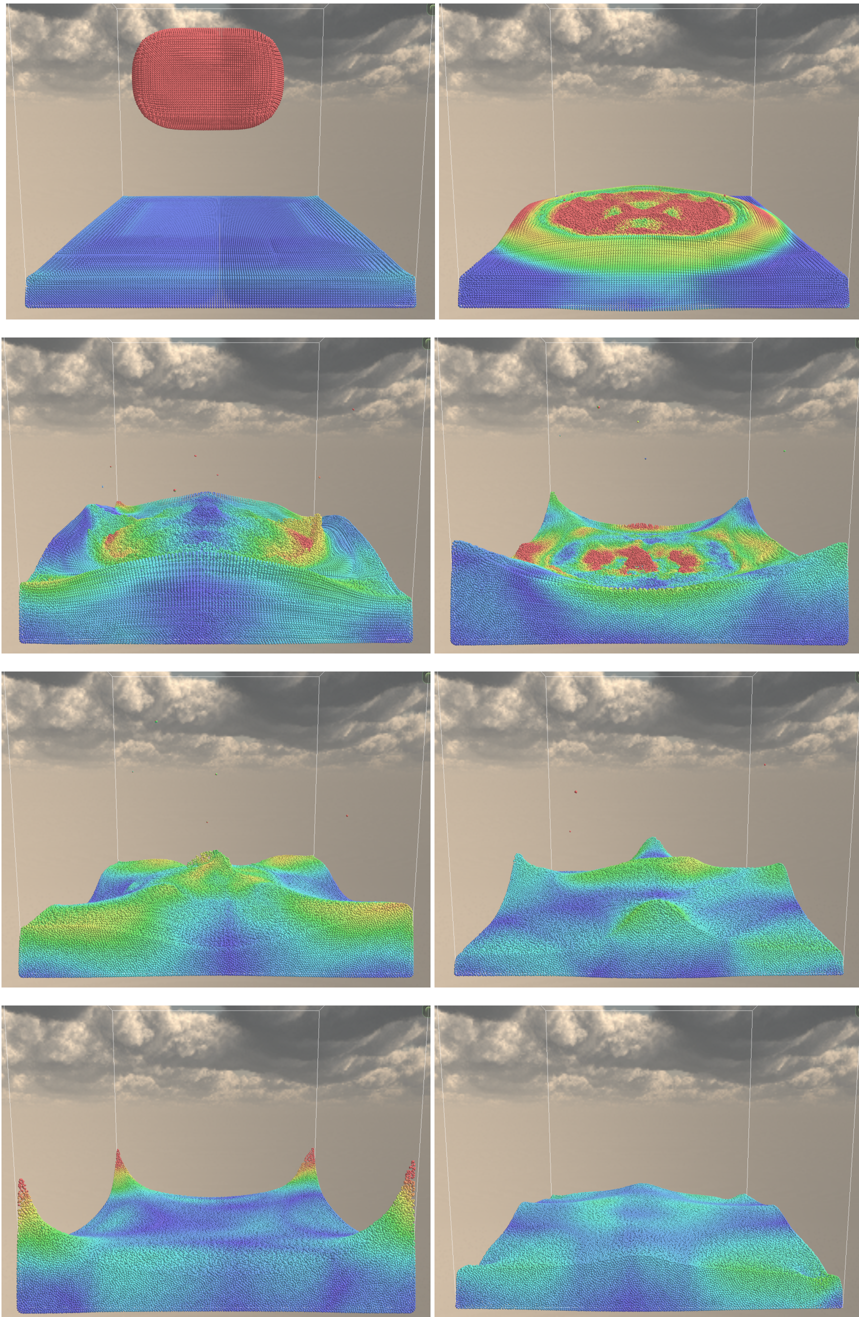


Figure 8.15: The simulation performance test scene. Shown here are several snapshots in time. Using the Simple SPH model, 512K particles and velocity hue shading.

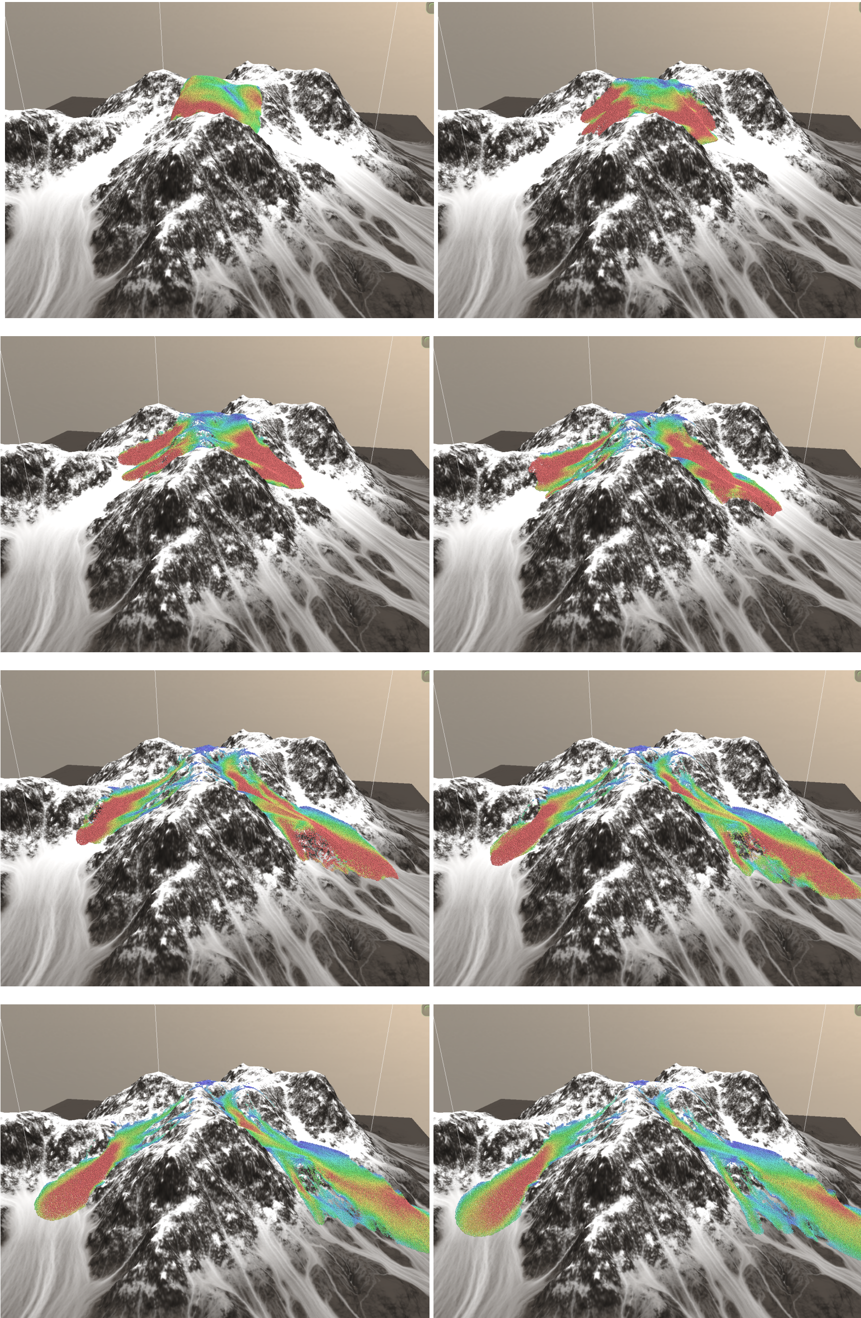


Figure 8.16: The Simple model with water-like parameters placed on a terrain. Shown here are several snapshots in time. Using 128K particles and velocity hue shading.

8.3.2 Complex SPH and Snow Avalanches

To test the fluid model suitability for snow avalanche simulation we have created a test scene. In this scene we use a generated terrain of a mountain with fairly steep slopes. Since our model does not capture the avalanche release we place the fluid along the terrain in a slope of the terrain and allow the fluid to flow downwards.

By using the correct friction parameters as well as the correct rheological model and rheological parameters we have been able to reproduce flowing behavior that is similar to that of a flowing avalanche.

To visualize the snow, we choose a uniform white shading for all the particles, this works surprisingly well to capture the appearance of real snow.

One important finding is that for the Complex SPH model the timestep is critical, higher viscosities require a lowering of the timestep and thus a simulation that is perceived as slower.

We find that for a maximum viscosity of 300 we can use a time-step of 0.0005. This is near the maximum possible while still maintaining stability. With the GeForce GTX 470 we get 60 frames per second (with rendering) at 256K and this produces a very believable fluid behavior.

Balancing the viscosity and the rheological model against the time step and the number of particles is a difficult task. Through trial and error we have found that we can achieve good interactivity around 64K particles. We use a time step of 0.001 and a rheological model that is limited to a maximum viscosity of 100, to ensure stability of the simulation.

Avalanche Parameters Following Ancey [5], Bovet et al. [14] we use a density of 400 kgm^{-3}. Kern et al. [11] demonstrates that the flowing behavior of snow is consistent with that of the Cross and Herschkel-Bulkley mode.

We use the best fit parameters from Kern et al. [11] , but modify the maximum viscosity from 800 to 300 to ensure real-time performance.

Parameter	Value
n	1
K	2.1
μ_∞	1.07
μ_0	300

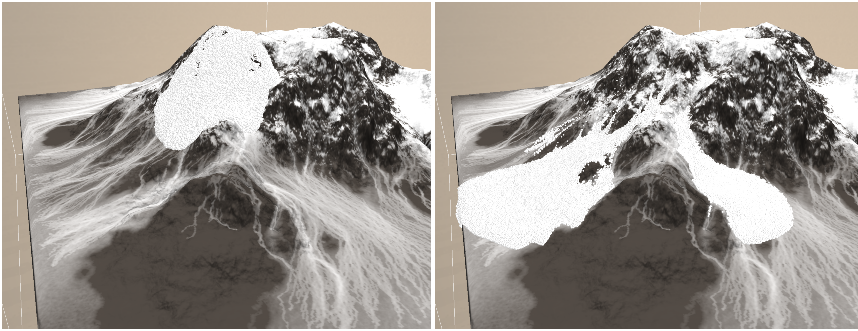


Figure 8.17: The Complex model with a Cross rheological model with the parameters $\mu_\infty = 1.07$, $\mu_0 = 300$ $n = 1$ and $K = 2.1$. We use a kinetic friction coefficient of 0.2. Shown here is the startin condition and the final runout of the avalanche. Using 64K particles, a fluid density of $400\,kgm^{-3}$ and uniform white particle shading.

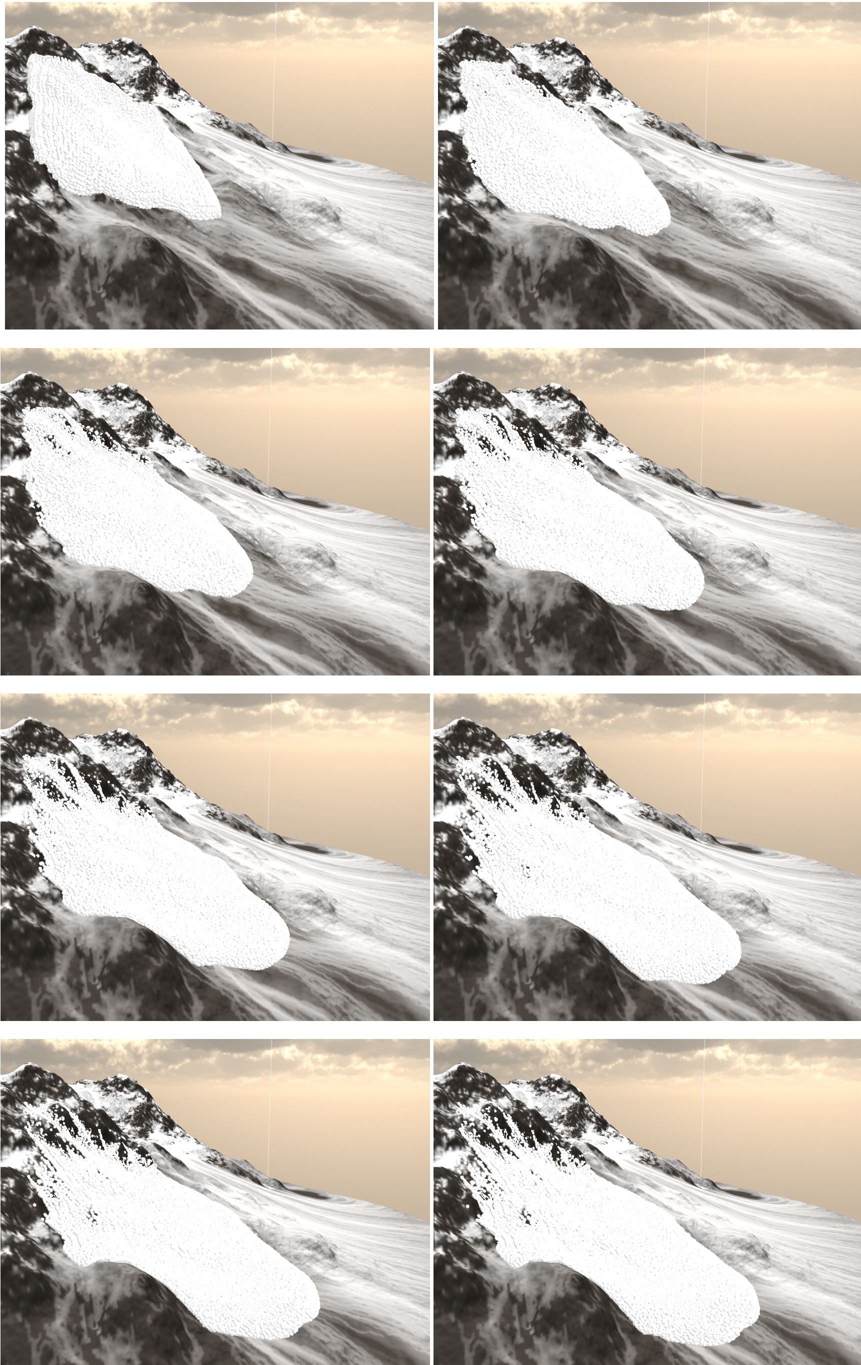


Figure 8.18: The Complex model with a Cross rheological model with the parameters $\mu_\infty = 1$, $\mu_0 = 100$ $n = 2$ and $K = 30000$. We use a kinetic friction coefficient of 0.2. Shown here are several snapshots in time. Using 64K particles and uniform white particle shading.

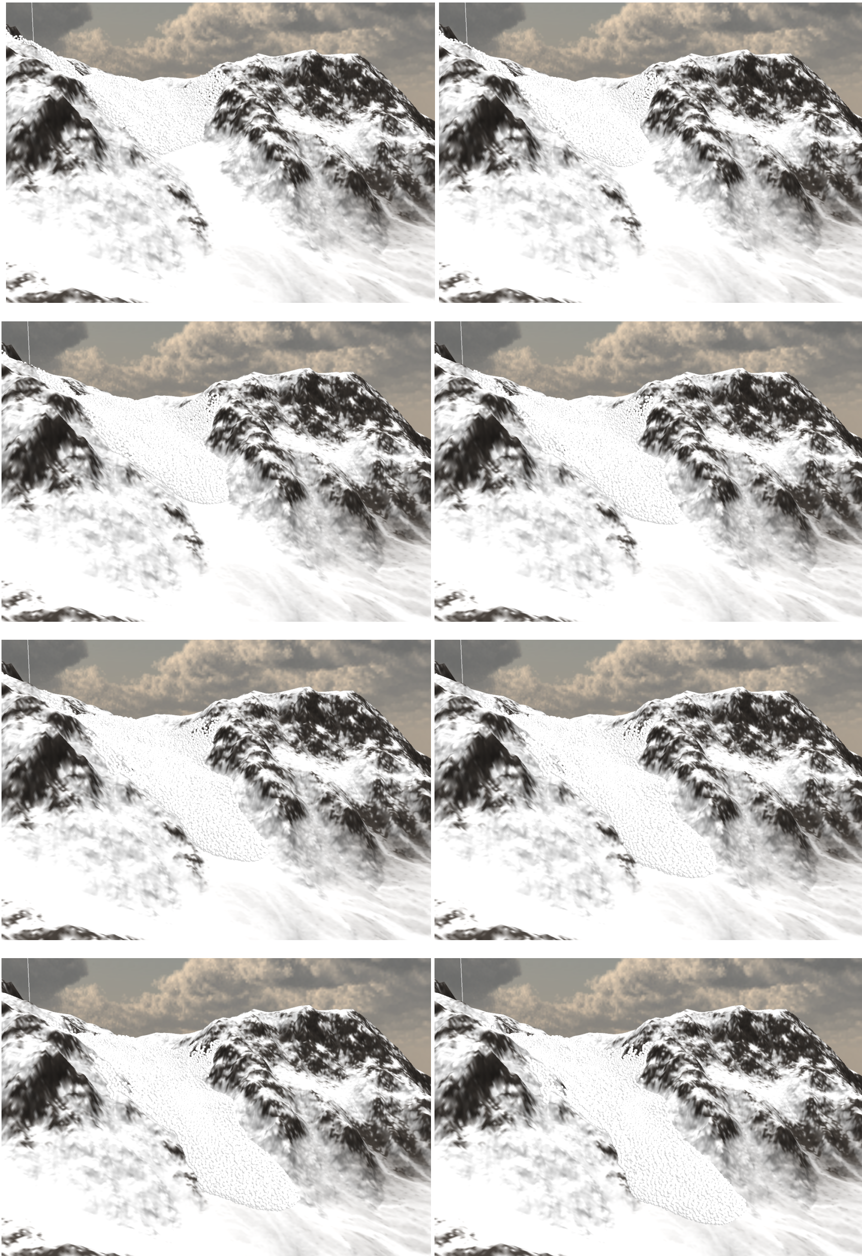


Figure 8.19: The Complex model with a Cross rheological model with the parameters $\mu_\infty = 1$, $\mu_0 = 100$ $n = 2$ and $K = 30000$. We use a kinetic friction coefficient of 0.2. Using 64K particles and uniform white particle shading.

Chapter 9

Conclusions and Future Work

In this thesis, we investigated the possibility of using fluid dynamics for snow avalanche modelling. We have found that there exists much prior research in this area. Using Computational Fluid Dynamics (CFD) and a rheological model for Non-Newtonian fluids it is possible to model the flowing behavior of snow avalanches.

A comprehensive description of Smoothed Particle Hydrodynamics (SPH) has been presented. SPH is ideal for interactive GPU simulations because of the high efficiency and because it is highly parallelizable. We have used several complex techniques to ensure an efficient implementation on the Graphics Processing Unit (GPU).

A novel framework for doing Nearest-Neighbor particle simulations on the GPU has been developed and implemented. Using this framework we implement two different SPH models. A Simple model that is a reimplementation of an SPH model that is suitable for low-viscosity fluids in interactive simulations and a Complex model that includes support for Non-Newtonian fluids. Our two SPH implementations have support for complex terrain boundaries and simple wall boundaries.

By applying extensive optimizations to the framework and implementations very good performance was demonstrated and compared to state-of-the-art implementations our implementations are many times faster.

9.1 Performance

To ensure the greatest possible performance considerable effort was spent on optimizing the SPH implementations. Compared to our previous implementation of the Simple SPH model a further performance increase in the range of 35% to 100% was achieved.

We have also evaluated the overhead of real-time rendering and find that it can carry an overhead of between 15% and 50%.

Benchmarking the two implementations on the new Fermi GPU-architecture from NVIDIA we have found that the GeForce GTX 470 improves performance further by 35% to 95% for the Simple SPH model and 115% to 150% for the Complex model.

9.2 Snow Avalanche

Our Complex SPH model has support for Non-Newtonian fluids through the use of rheological functions. By using a rheological function that has been found to correlate well to the movement of real snow avalanches, we have been able to reproduce a flowing behavior that resembles that of a snow avalanches on a terrain.

We have not validated the model and it should be noted that the model may not be physically accurate. However it does capture the flowing characteristics of dense snow avalanches. Our implementation is very dependent on correct parameters and would benefit from more work into the initial conditions. In addition, improvements to the snow placement on the terrain, the avalanche release and the terrain boundary friction would improve the physical accuracy.

Due to the very high performance of the implementations we have successfully been able to simulate snow avalanches at interactive speeds. To achieve this, the models sacrifice some numerical accuracy, but applying the techniques we use in the framework to models more focused on physical accuracy should yield large gains in performance for these models as well.

We have created interactive simulations that are believable, by adding better visualization the quality in this respect would be further improved.

Though we have only used the implementations for simulations of water-like fluids and snow avalanches, both models could be used for simulations of other fluids. The complex model in particular is well suited for other types of geomorphological flows such as mud slides.

9.3 Future work

Due to the great flexibility of SPH and the complexity of snow avalanches, there are a great many possibilities for future work.

9.3.1 Snow Avalanche

Our model is designed to capture the flowing properties of avalanches, but does not consider other parts of an avalanche. The accumulation on snow on the terrain and the subsequent avalanche release, is completely unexplored on our part. An integration between our model with a model that captures this phenomenon would produce a more completely simulation of the entire avalanche lifecycle.

The entrainment of snow in the avalanche is also unexplored on our part, and including this effect would increase the accuracy of the model greatly.

Finally our model and implementation has not been validated against real avalanches. Using real terrain topology, experimental data and data from known avalanches it would be interesting to quantify the accuracy of the model.

9.3.2 SPH models

In the context of SPH models, the two SPH models we implemented are relatively simple. There exist many new formulations of SPH which aim to correct some of their deficiencies. It would be interesting to extend our implementations or use our framework to implement some of these models on the GPU.

For interactivity, it would be especially interesting to implement an SPH model that does not use an *Equation of State* to enforce incompressibility, but solves the Poisson equation directly. Doing so would enable larger timestep. Although each iteration would then be more computationally costly, it may be worth it in the context of achieving better interactivity.

It would also be interesting to extend the models with support for interactions between multiple fluids.

Adding support for temperature and energy calculations in the SPH models would allow for a range of effects such as freezing and melting. In the context of snow avalanches this is very interesting since it would be possible to explore a more complex snow avalanche rheology model that depends on temperature. In this way the shear forces/friction in the avalanche would affect the temperature and thus also the viscosity.

It would also be interesting to investigate if it would be possible to use the temperature formulation to model the avalanche release. Since the release of an avalanche can be caused by melting snow and heat from the sun, one could model the temperature in the snowpack and use the temperature to calculate the water content in the snowpack. Using this water content it should be possible to model the structural integrity of the snowpack, and thus also the release conditions for an avalanche.

The runout of an avalanche is highly dependent on the basal friction against the terrain. In our thesis, we use a fairly simple friction model. By implementing a more complex friction model increased accuracy would be possible. In addition, one could model interesting effects such as erosion [71].

Adding more complex terrains would be interesting since one could describe different frictions for different parts of the terrain, in effect creating a friction-map. Using this one could model different materials for the ground, such as rock, dirt, snow and vegetation.

9.3.3 Implementation

In our implementation we rendered the SPH particles directly. By doing some form of surface reconstruction one could render a more correct fluid "surface".

Adding support for arbitrary meshes is also a possibility. There would be a need for some kind of spatial index, and it might be possible to reuse the existing uniform grid that we have implemented.

Our implementations use NVIDIA CUDA, but the emerging standard for GPU-computing is OpenCL. Porting our framework to use OpenCL would make it possible to use other GPUs than those by NVIDIA.

Adding support for multi-GPU simulations would make it possible to scale the simulations both in accuracy and performance. Doing so would require some way to split the simulation domain such that load is balanced fairly among the GPUs. Fleissner and Eberhard [72] propose such method for parallel load balancing.

Extending this it would also be interesting to extend our framework to support clusters of GPUs, which would make it possible to do very large simulations.

Bibliography

[1] Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 154–159, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association. ISBN 1-58113-659-5.

[2] NVIDIA. *CUDA Programming Guide 3.1*, 2009. URL http://www.nvidia.com/object/cuda_get.html.

[3] Rob Farber. Dr. Dobb's, CUDA, Supercomputing for the Masses: Part 9, 3 2009. URL http://www.ddj.com/architect/211800683.

[4] Øystein Eklund Krog. GPU-based Real-Time Particle Hydrodynamics. Technical Report 6/2010, Norwegian University of Science and Technology, Faculty of Information Technology, Mathematics and Electrical Engineering, Department of Computer and Information Science, Trondheim, 2010.

[5] C. Ancey. Snow avalanches. In N. J. Balmforth & A. Provenzale, editor, *Geomorphological Fluid Mechanics*, volume 582 of *Lecture Notes in Physics, Berlin Springer Verlag*, pages 319–+, 2001.

[6] Kolumban Hutter, Yongqi Wang, and Shiva P Pudasaini. The Savage-Hutter avalanche model: how far can it be pushed? *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 363(1832):1507–1528, 2005. doi: 10.1098/rsta.2005.1594. URL http://rsta.royalsocietypublishing.org/content/363/1832/1507.abstract.

[7] Bard Glenne. Motion resistance of avalanches on smooth paths. *Cold Regions Science and Technology*, 12(2):115 – 119, 1986. ISSN 0165-232X. doi: DOI:10.1016/0165-232X(86)90026-1. URL http://www.sciencedirect.com/science/article/B6V86-488G7VG-Y/2/b7891644f266afffb3dcd60dba8db721.

[8] J.D. Dent and T.E. Lang. Experiments on mechanics of flowing snow. *Cold Regions Science and Technology*, 5(3):253 – 258, 1982. ISSN 0165-232X. doi: DOI:10.1016/0165-232X(82)90018-0. URL http://www.sciencedirect.com/science/article/B6V86-4894RCY-8W/2/5c1813e1ba5f9e04768b418fa0e994b9.

[9] O Maeno. Rheological characteristics of snow flows. In L. Buisson, editor, *International Workshop on Gravitational Mass Movements*, pages 209–220, 1993.

[10] K. Nishimura and N. Maeno. Experiments on snow-avalanche dynamics. In *Proceedings of Avalanche Formation, Movements and Effects*, volume Avalanche Formation, Movement and Effects (Proceedings of the Davos Symposium, September 1986)., page 395Ű404. IAHS Publication, 1987.

[11] M. A. Kern, F. Tiefenbacher, and J. N. McElwaine. The rheology of snow in large chute flows. *Cold Regions Science and Technology*, 39(2-3): 181 – 192, 2004. ISSN 0165-232X. doi: DOI:10.1016/j.coldregions.2004.03.006. URL http://www.sciencedirect.com/science/article/B6V86-4CS4G7W-1/2/664993b41275bfb273c4b9b1d40cfd52.

[12] C. Ancey and M. Meunier. Estimating bulk rheological properties of flowing snow avalanches from field data. *Journal of Geophysical Research, AGU*, page F01004, 2004. doi: NA.

[13] K. Platzer, P. Bartelt, and C. Jaedicke. Basal shear and normal stresses of dry and wet snow avalanches after a slope deviation. *Cold Regions Science and Technology*, 49(1):11 – 25, 2007. ISSN 0165-232X. doi: DOI:10.1016/j.coldregions.2007.04.003. URL http://www.sciencedirect.com/science/article/B6V86-4NJ7W6G-1/2/b9560552e3826b020d31dd80f66b7485. Selected Papers from the General Assembly of the European Geosciences Union (EGU), Vienna, Austria, 25 April 2005.

[14] Eloise Bovet, Bernardino Chiaia, and Luigi Preziosi. A new model for snow avalanche dynamics based on non-newtonian fluids. *Meccanica*, 2008. ISSN 0025-6455 (Print) 1572-9648 (Online). doi: 10.1007/s11012-009-9278-z. URL http://www.springerlink.com/content/g82xk01766833788.

[15] David McClung and Peter Schaerer. *The Avalanche Handbook*. Mountaineers Books, 3rd edition, 10 2006. ISBN 9780898868098.

[16] Scott McDougall and Oldrich Hungr. A model for the analysis of rapid landslide motion across three-dimensional terrain. *Canadian Geotechnical Journal*, 41(6):1084–1097, 2004. ISSN 12086010. doi: 10.1139/t04-052. URL http://article.pubs.nrc-cnrc.gc.ca/ppv/RPViewDoc?issn=1208-6010&volume=41&issue=6&startPage=1084&ab=y.

[17] Scott McDougall and Oldrich Hungr. Dynamic modelling of entrainment in rapid landslides. *Canadian geotechnical journal*, 42(5):1437–1448, 2005. ISSN 0008-3674. URL http://cat.inist.fr/?aModele=afficheN&cpsidt=17335612.

[18] Oldrich Hungr. Numerical modelling of the motion of rapid, flow-like landslides for hazard assessment. *KSCE Journal of Civil Engineering*, 13(4): 281–287, July 2009. ISSN 1226-7988 (Print) 1976-3808 (Online). doi: 10.1007/s12205-009-0281-7. URL http://www.springerlink.com/content/05n426w5452r32q8.

[19] G. Shobeyri B. Ataie-Ashtiani. Numerical simulation of landslide impulsive waves by incompressible smoothed particle hydrodynamics. *International journal for numerical methods in fluids*, 56(2):209–232, 2008. ISSN 0271-2091. URL http://cat.inist.fr/?aModele=afficheN&cpsidt=19965078.

[20] Dominique Laigle, Philippe Lachamp, and Mohamed Naaim. SPH-based numerical investigation of mudflow and other complex fluid flow interactions with structures. *Computational Geosciences*, 11(4):297–306, December 2007. doi: 10.1007/s10596-007-9053-y. URL http://dx.doi.org/10.1007/s10596-007-9053-y.

[21] Angela Ferrari, Michael Dumbser, Eleuterio F. Toro, and Aronne Armanini. A new 3D parallel SPH scheme for free surface flows. *Computers & Fluids*, 38(6): 1203 – 1217, 2009. ISSN 0045-7930. doi: DOI:10.1016/j.compfluid.2008.11.012. URL http://www.sciencedirect.com/science/article/B6V26-4V2NK8P-1/2/b6874440b900286ed533b6be7ef0a898.

[22] Afonso Paiva, Fabiano Petronetto, Thomas Lewiner, and Geovan Tavares. Particle-based viscoplastic fluid/solid simulation. *Computer-Aided Design*, 41(4):306 – 314, 2009. ISSN 0010-4485. doi: DOI:10.1016/j.cad.2008.10.004. URL http://www.sciencedirect.com/science/article/B6TYR-4TTMNFW-1/2/3e798fdc322f7e878f386d435f80b01b. Point-based Computational Techniques.

[23] S. M. Hosseini, M. T. Manzari, and S. K. Hannani. A fully explicit three-step SPH algorithm for simulation of non-Newtonian fluid flow. *International Journal of Numerical Methods for Heat & Fluid Flow*, 17(7):715–735, 2007. ISSN 0961-5539. doi: 10.1108/09615530710777976. URL http://dx.doi.org/10.1108/09615530710777976.

[24] David Pnueli and Chaim Gutfinger. *Fluid Mechanics*. Cambridge University Press, 1997. ISBN 0521587972. URL http://www.amazon.com/Fluid-Mechanics-David-Pnueli/dp/0521587972%3FSubscriptionId%3D0JYN1NVW651KCA56C102%26tag%3Dtechkie-20%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D0521587972.

[25] N.J. Balmforth and R.V. Craster. *Geophysical aspects of Non-Newtonian Fluid Mechanics*, volume 582. Springer, 2001.

[26] K. Taous J.-M. Sac-Épee. On a wide class of nonlinear models for non-newtonian fluids with mixed boundary conditions in thin domains. *Asymptotic Analysis*, Volume 44(1-2/2005):151–171, 2005.

[27] J. J. Monaghan. Smoothed particle hydrodynamics. *Reports on Progress in Physics*, 68:1703Ű1759, 2005. URL http://www.iop.org/EJ/abstract/0034-4885/68/8/R01/.

[28] J. J. Monaghan. Smoothed particle hydrodynamics. *Annual Review of Astronomy and Astrophysics*, 30:543Ű574, 1992. URL http://arjournals.annualreviews.org/doi/abs/10.1146%2Fannurev.aa.30.090192.002551.

[29] L. B. Lucy. A numerical approach to the testing of the fission hypothesis. *Astronomical Journal*, 82:1013–1024, 1977.

[30] R. A. Gingold and J. J. Monaghan. Smoothed Particle Hydrodynamics - Theory and application to non-spherical stars. *Royal Astronomical Society, Monthly Notices*, 181:375–389, 1977.

[31] J. J. Monaghan. Simulating free surface flows with SPH. *J. Comput. Phys.*, 110(2):399–406, February 1994. ISSN 0021-9991. doi: 10.1006/jcph.1994.1034. URL http://dx.doi.org/10.1006/jcph.1994.1034.

[32] Matthias Müller, Barbara Solenthaler, Richard Keiser, and Markus Gross. Particle-based fluid-fluid interaction. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 237–244, New York, NY, USA, 2005. ACM. ISBN 1-7695-2270-X. doi: http://doi.acm.org/10.1145/1073368.1073402.

[33] Simon Green and NVIDIA. Particle-based fluid simulation. February 2008.

[34] Paul W. Cleary and Mahesh Prakash. Discrete-Element Modelling and Smoothed Particle Hydrodynamics: Potential in the Environmental Sciences. *Philosophical Transactions: Mathematical, Physical and Engineering Sciences*, 362(1822):2003–2030, 2004. ISSN 1364503X. URL http://www.jstor.org/stable/4142473.

[35] Micky Kelager. Lagrangian fluid dynamics using smoothed particle hydrodynamics. Master's thesis, University of Copenhagen, Department of Computer Science, 2006.

[36] Mathieu Desbrun and Marie-Paule Gascuel. Smoothed particles: A new paradigm for animating highly deformable bodies. In *In Computer Animation and Simulation Š96 (Proceedings of EG Workshop on Animation and Simulation*, pages 61–76. Springer-Verlag, 1996.

[37] F. Colin, R. Egli, and F. Y. Lin. Computing a null divergence velocity field using smoothed particle hydrodynamics. *J. Comput. Phys.*, 217(2):680–692, 2006. ISSN 0021-9991. doi: http://dx.doi.org/10.1016/j.jcp.2006.01.021.

[38] J. J. Monaghan. An introduction to SPH. *Computer Physics Communications*, 48(1):89 – 96, 1988. ISSN 0010-4655. doi: DOI:10.1016/0010-4655(88)90026-4. URL http://www.sciencedirect.com/science/article/B6TJ5-46FXB87-4J/2/6a7e03ab627e024618fb13ad48081026.

[39] M. Liu and G. Liu. Smoothed Particle Hydrodynamics (SPH): an Overview and Recent Developments. *Archives of Computational Methods in Engineering*, 17(1):25–76, March 2010. ISSN 1134-3060. doi: 10.1007/s11831-010-9040-7. URL http://dx.doi.org/10.1007/s11831-010-9040-7.

[40] G. R. Liu and M. B. Liu. *Smoothed Particle Hydrodynamics: A Meshfree Particle Method*. World Scientific Publishing Company, 12 2003. ISBN 9789812384560. URL http://amazon.com/o/ASIN/9812384561/.

[41] Songdong Shao and Edmond Y. M. Lo. Incompressible SPH method for simulating Newtonian and non-Newtonian flows with a free surface. *Advances in Water Resources*, 26(7):787 – 800, 2003. ISSN 0309-1708. doi: DOI:10.1016/S0309-1708(03)00030-7. URL http://www.sciencedirect.com/science/article/B6VCF-48HXX4F-1/2/380162b8c29394cdf101a2dc3672598c.

[42] Sharen J. Cummins and Murray Rudman. An SPH Projection Method. *Journal of Computational Physics*, 152(2):584 – 607, 1999. ISSN 0021-9991. doi: DOI:10.1006/jcph.1999.6246. URL http://www.sciencedirect.com/science/article/B6WHY-45GMW83-35/2/b807a8f9e1692de8c4a7dad9e73fdc00.

[43] Songdong Shao, Changming Ji, David I. Graham, Dominic E. Reeve, Philip W. James, and Andrew J. Chadwick. Simulation of wave overtopping by an incompressible SPH model. *Coastal Engineering*, 53(9): 723 – 735, 2006. ISSN 0378-3839. doi: DOI:10.1016/j.coastaleng.2006.02.005. URL http://www.sciencedirect.com/science/article/B6VCX-4JVTCCR-2/2/7870779a7e6119b61d74939031d60e9d.

[44] X. Y. Hu and N. A. Adams. An incompressible multi-phase SPH method. *J. Comput. Phys.*, 227(1):264–278, 2007. ISSN 0021-9991. doi: http://dx.doi.org/10.1016/j.jcp.2007.07.013.

[45] Nick Foster and Ronald Fedkiw. Practical animation of liquids. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 23–30, New York, NY, USA, 2001. ACM. ISBN 1-58113-374-X. doi: http://doi.acm.org/10.1145/383259.383261.

[46] Markus Becker, Hendrik Tessendorf, and Matthias Teschner. Direct forcing for lagrangian rigid-fluid coupling. *IEEE Transactions on Visualization and Computer Graphics*, 15:493–503, 2009. ISSN 1077-2626. doi: http://doi.ieeecomputersociety.org/10.1109/TVCG.2008.107.

[47] Markus Becker and Matthias Teschner. Weakly compressible SPH for free surface flows. In *SCA '07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 209–217, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association. ISBN 978-1-59593-624-4.

[48] Takahiro Harada, Seiichi Koshizuka, and Yoichiro Kawaguchi. Smoothed Particle Hydrodynamics on GPUs. 2007. URL http://www.inf.ufrgs.br/cgi2007/cd_cgi/papers/harada.pdf.

[49] J. J. Monaghan and J. C. Lattanzio. A refined particle method for astrophysical problems. *Astronomy and Astrophysics*, 149:135–143, August 1985.

[50] H. Takeda, S. Miyama, and M. Sekiya. Numerical simulation of viscous flow by smoothed particle hydrodynamics. *Progress of Theoretical Physics*, 92(5): 939–960, 1994.

[51] Joseph P. Morris, Patrick J. Fox, and Yi Zhu. Modeling low Reynolds number incompressible flows using SPH. *J. Comput. Phys.*, 136(1):214–226, 1997. ISSN 0021-9991. doi: http://dx.doi.org/10.1006/jcph.1997.5776.

[52] P. W. Randles and L. D. Libersky. Smoothed particle hydrodynamics: Some recent improvements and applications. *Computer Methods in Applied Mechanics and Engineering*, 139(1-4):375 – 408, 1996. ISSN 0045-7825. doi: DOI:10.1016/S0045-7825(96)01090-0. URL http://www.sciencedirect.com/science/article/B6V29-41FDG4T-F/2/8220aeb1a97726e157007f98e4215d5a.

[53] A.J.C Crespo. *Application of the Smoothed Particle Hydrodynamics model SPHysics to free-surface hydrodynamics*. PhD thesis, University of Vigo, 2008.

[54] Holger Wendland. Computational aspects of radial basis function approximation. In *In IEEE Int. Symp. Circuits Syst. ISCASŠ97*, 1997.

[55] Holger Wendland. Piecewise polynomial, positive definite and compactly supported radial functions of minimal degree. *Advances in Computational Mathematics*, 4(1):389–396, December 1995. doi: 10.1007/BF02123482. URL http://dx.doi.org/10.1007/BF02123482.

[56] M. B. Liu, G. R. Liu, and K. Y. Lam. Constructing smoothing functions in smoothed particle hydrodynamics with applications. *Journal of Computational and Applied Mathematics*, 155(2):263 – 284, 2003. ISSN 0377-0427. doi: DOI:10.1016/S0377-0427(02)00869-5. URL http://www.sciencedirect.com/science/article/B6TYH-48MX4V4-3/2/50fab430c45b54747e1a92ea4bbdd00b.

[57] Robert G. Belleman, Jeroen Bédorf, and Simon F. Portegies Zwart. High performance direct gravitational N-body simulations on graphics processing units II: An implementation in CUDA. *New Astronomy*, 13(2): 103 – 112, 2008. ISSN 1384-1076. doi: DOI:10.1016/j.newast.2007.07.004. URL http://www.sciencedirect.com/science/article/B6TJK-4P940VB-1/2/f0539ed1a144078942ec2c6486808ffa.

[58] NVIDIA. *CUDA Programming Guide 3.1*, 2009. URL http://www.nvidia.com/object/cuda_get.html.

[59] Dave H. Eberly. *Game Physics*. Elsevier Science Inc., New York, NY, USA, 2003. ISBN 1558607404.

[60] J.J. Monaghan. On the problem of penetration in particle methods. *Journal of Computational Physics*, 82(1):1 – 15, 1989. ISSN 0021-9991. doi: DOI:10.1016/0021-9991(89)90032-6. URL `http://www.sciencedirect.com/science/article/B6WHY-4DD1XDC-1M7/2/67de50778df557038d6786dd3408afee`.

[61] Simon Green and NVIDIA. CUDA Particles, Presentation slides. Technical report, NVIDIA, 2008.

[62] NVIDIA. CUDA SDK 2.3 Code Samples, 2009. URL `http://www.nvidia.com/object/cuda_get.html`.

[63] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore GPUs. Technical report, Los Alamitos, CA, USA, 2009. URL `http://dx.doi.org/10.1109/IPDPS.2009.5161005`.

[64] Duane Merrill and Andrew Grimshaw. Revisiting Sorting for GPGPU Stream Architectures. Technical Report CS2010-03, Department of Computer Science, University of Virginia, 2010.

[65] T. Amada. *Real-time particle-based fluid simulation with rigid-body interaction*, pages 189–205. Charles River Media, 2006.

[66] Vasily Volkov and James W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press. ISBN 978-1-4244-2835-9.

[67] Hongliang Gao, Martin Dimitrov, Jingfei Kong, and Huiyang Zhou. Experiencing various massively parallel architectures and programming models for dataintensive. In *ApplicationsŤ, Workshop on Computer Architecture Education (WCAE-08), in conjunction with ISCA-35*, 2008.

[68] Donald D. Hearn and M. Pauline Baker. *Computer Graphics with OpenGL*. Prentice Hall Professional Technical Reference, 2003. ISBN 0130153907.

[69] He Yan, Zhangye Wang, Jian He, Xi Chen, Changbo Wang, and Qunsheng Peng. Real-time fluid simulation with adaptive SPH. *Comput. Animat. Virtual Worlds*, 20(2‐3):417–426, 2009. ISSN 1546-4261. doi: http://dx.doi.org/10.1002/cav.v20:2/3.

[70] Yanci Zhang, Barbara Solenthaler, and Renato Pajarola. GPU accelerated SPH particle simulation and rendering. In *SIGGRAPH '07: ACM SIGGRAPH 2007 posters*, page 9, New York, NY, USA, 2007. ACM. doi: http://doi.acm.org/10.1145/1280720.1280731.

[71] Jaroslav Krivanek Peter Kristof, Bedrich Benes and Ondrej Stava. Hydraulic erosion using smoothed particle hydrodynamics. In *Computer Graphics Forum (Proc. of EUROGRAPHICS ?09)*, number 2, 2009.

[72] Florian Fleissner and Peter Eberhard. Load Balanced Parallel Simulation of Particle-Fluid DEM-SPH Systems with Moving Boundaries. 15:37–44, 2007.

Appendix A

Smoothing Kernel Derivatives

These are the Gradients and Laplacians of the smoothing kernels used in the implementations.

From Müller et al. [1].

$$\nabla W_{poly6}(\mathbf{r},h) = -\frac{945}{32\pi h^9}\mathbf{r}\begin{cases} (h^2 - |\mathbf{r}|^2)^2 & 0 \leq |\mathbf{r}| \leq h \\ 0 & otherwise \end{cases} \tag{A.1}$$

$$\nabla^2 W_{poly6}(\mathbf{r},h) = -\frac{945}{32\pi h^9}\begin{cases} (h^2 - |\mathbf{r}|^2)(3h^2 - 7|\mathbf{r}|^2) & 0 \leq |\mathbf{r}| \leq h \\ 0 & otherwise \end{cases} \tag{A.2}$$

$$\nabla W(\mathbf{r},h)_{viscosity} = \frac{15}{2\pi h^3}\mathbf{r}\begin{cases} \frac{-3|\mathbf{r}|}{2h^3} & + \frac{2}{h^2} - \frac{h}{2|\mathbf{r}|^3} 0 \leq |\mathbf{r}| \leq h \\ 0 & otherwise \end{cases} \tag{A.3}$$

$$\nabla^2 W_{viscosity}(\mathbf{r},h) = \frac{45}{\pi h^6}\begin{cases} h - |\mathbf{r}| & 0 \leq |\mathbf{r}| \leq h \\ 0 & otherwise \end{cases} \tag{A.4}$$

And from Desbrun and Gascuel [36]:

$$\nabla W_{spiky}(\mathbf{r},h) = -\frac{45}{\pi h^6}\frac{\mathbf{r}}{|\mathbf{r}|}\begin{cases} (h - |\mathbf{r}|)^2 & 0 \leq |\mathbf{r}| \leq h \\ 0 & otherwise \end{cases} \tag{A.5}$$

$$\nabla^2 W_{spiky}(\mathbf{r},h) = -\frac{90}{\pi h^6}\frac{1}{|\mathbf{r}|}\begin{cases} (h - |\mathbf{r}|)(h - 2|\mathbf{r}|) & 0 \leq |\mathbf{r}| \leq h \\ 0 & otherwise \end{cases} \tag{A.6}$$

And finally the gradient of the cubic spline smoothing kernel:

$$\nabla W_{cubic}(\mathbf{r},h) = \begin{pmatrix} \begin{cases} 0 & \text{if} \quad 2 < \frac{|\mathbf{r}|}{h} \\ -\frac{\frac{3\,x}{h^2} - \frac{9\,x\,|\mathbf{r}|}{4\,h^3}}{\pi\,h^3} & \text{if} \quad \frac{|\mathbf{r}|}{h} \in (0,1) \\ -\frac{3\,x\left(\frac{|\mathbf{r}|}{h}-2\right)^2}{4\,\pi\,h^4\,|\mathbf{r}|} & \text{if} \quad \frac{|\mathbf{r}|}{h} \in (1,2) \end{cases} \\ \begin{cases} 0 & \text{if} \quad 2 < \frac{|\mathbf{r}|}{h} \\ -\frac{\frac{3\,y}{h^2} - \frac{9\,y\,|\mathbf{r}|}{4\,h^3}}{\pi\,h^3} & \text{if} \quad \frac{|\mathbf{r}|}{h} \in (0,1) \\ -\frac{3\,y\left(\frac{|\mathbf{r}|}{h}-2\right)^2}{4\,\pi\,h^4\,|\mathbf{r}|} & \text{if} \quad \frac{|\mathbf{r}|}{h} \in (1,2) \end{cases} \\ \begin{cases} 0 & \text{if} \quad 2 < \frac{|\mathbf{r}|}{h} \\ -\frac{\frac{3\,z}{h^2} - \frac{9\,z\,|\mathbf{r}|}{4\,h^3}}{\pi\,h^3} & \text{if} \quad \frac{|\mathbf{r}|}{h} \in (0,1) \\ -\frac{3\,z\left(\frac{|\mathbf{r}|}{h}-2\right)^2}{4\,\pi\,h^4\,|\mathbf{r}|} & \text{if} \quad \frac{|\mathbf{r}|}{h} \in (1,2) \end{cases} \end{pmatrix}$$

Appendix B

Source Code

We include some of the most important source from our implementations. Only GPU code is included, all host code is excluded for reasons of brevity.

B.1 Visualization Shader Code

```
vertex_program shader/ParticleBall_VS cg
{
  source ParticleBall.cg
  profiles vs_1_1 arbvp1
  entry_point ParticleBall_VS

  default_params
  {
      param_named_auto lightPosition        light_position_object_space 0
      param_named_auto eyePosition        camera_position_object_space

      param_named_auto worldViewProjMatrix    worldviewproj_matrix
      param_named_auto texWorldViewProjMatrix0
          texture_worldviewproj_matrix 0
      param_named_auto texWorldViewProjMatrix1
          texture_worldviewproj_matrix 1
      param_named_auto texWorldViewProjMatrix2
          texture_worldviewproj_matrix 2

    param_named   pointRadius   float 100
    param_named   pointScale    float 400
  }
}

fragment_program shader/ParticleBall_PS cg
{
  source ParticleBall.cg
  profiles ps_2_0 arbfp1
  entry_point ParticleBall_PS
}

fragment_program shader/ParticleBallSnow_PS cg
{
  source ParticleBall.cg
  profiles ps_2_0 arbfp1
  entry_point ParticleBallSnow_PS
```

```
34 }
35
36 material shader/ParticleBall
37 {
38   technique
39   {
40     pass
41     {
42       point_sprites on
43       point_size 10
44       point_size_attenuation on
45       vertex_program_ref shader/ParticleBall_VS {}
46       fragment_program_ref shader/ParticleBall_PS {}
47     }
48   }
49 }
50 material shader/ParticleBallSnow
51 {
52   technique
53   {
54     pass
55     {
56       point_sprites on
57       point_size 10
58       point_size_attenuation on
59       vertex_program_ref shader/ParticleBall_VS {}
60       fragment_program_ref shader/ParticleBallSnow_PS {}
61     }
62   }
63 }
```

```
1  void ParticleBall_VS(
2    float4 position       : POSITION,
3    //float3 normal       : NORMAL,
4    float4 color        : COLOR,
5    float2 uv          : TEXCOORD0,
6
7    out float4 oPosition     : POSITION,
8    out float4 oColor      : COLOR,
9    out float3 oUv         : TEXCOORD0,
10   out float3 oLightDir     : TEXCOORD1,
11   out float3 oHalfAngle    : TEXCOORD2,
12   out float4 oLightPosition0   : TEXCOORD3,
13   out float4 oLightPosition1   : TEXCOORD4,
14   out float4 oLightPosition2   : TEXCOORD5,
15   out float3 oNormal       : TEXCOORD6,
16   out float oPointSize     : PSIZE,
17
18   uniform float pointRadius,
19   uniform float pointScale,
20
21   uniform float4 lightPosition,        // object space
22   uniform float3 eyePosition,          // object
23
24     uniform float4x4 worldViewProjMatrix,
25
26   uniform float4x4 texWorldViewProjMatrix0,
```

```
27    uniform float4x4 texWorldViewProjMatrix1,
28    uniform float4x4 texWorldViewProjMatrix2
29  )
30  {
31    // calculate output position
32    oPosition = mul(worldViewProjMatrix, vec4(position.xyz, 1.0));
33
34    // pass the main uvs straight through unchanged
35    oUv.xy = uv;
36    oUv.z = oPosition.z;
37
38    // pass color through as well
39    oColor = color;
40
41    // pass through normals
42    //oNormal = normal;
43
44    float dist = length(oPosition);
45    oPointSize = pointRadius * (pointScale / dist);
46
47    // calculate tangent space light vector
48    // Get object space light direction
49    oLightDir = normalize(lightPosition.xyz - (position * lightPosition.w).
          xyz);
50
51    // Calculate half-angle in tangent space
52    float3 eyeDir = normalize(eyePosition - position.xyz);
53    oHalfAngle = normalize(eyeDir + oLightDir);
54
55    // Calculate the position of vertex in light space
56    oLightPosition0 = mul(texWorldViewProjMatrix0, position);
57    oLightPosition1 = mul(texWorldViewProjMatrix1, position);
58    oLightPosition2 = mul(texWorldViewProjMatrix2, position);
59
60      return;
61  }
62
63  /********* pixel shaders ********/
64
65  void ParticleBall_PS(
66      float4 position     : POSITION,
67      float4 color      : COLOR,
68
69    float3 uv        : TEXCOORD0,
70    float3 OSlightDir    : TEXCOORD1,
71    float3 OShalfAngle    : TEXCOORD2,
72    float4 LightPosition0 : TEXCOORD3,
73    float4 LightPosition1 : TEXCOORD4,
74    float4 LightPosition2 : TEXCOORD5,
75    float3 normal      : TEXCOORD6,
76
77    out float4 oColour    : COLOR
78    )
79  {
80      const vec3 lightDir = vec3(0.577, 0.577, 0.577);
81
82      // calculate normal from texture coordinates
```

```
    vec3 N;
    N.xy = uv.xy*vec2(2.0, -2.0) + vec2(-1.0, 1.0);
    float mag = dot(N.xy, N.xy);
    if (mag > 1.0) discard;   // kill pixels outside circle
    N.z = sqrt(1.0-mag);

    // calculate lighting
    float diffuse = 0.5 + 0.5* max(0.0, dot(lightDir, N));

    float alpha = 0.5;
    oColour = float4(color.rgb * diffuse,alpha);
}

void ParticleBallSnow_PS(
    float4 position     : POSITION,
    float4 color       : COLOR,

  float3 uv        : TEXCOORD0,
  float3 OSlightDir    : TEXCOORD1,
  float3 OShalfAngle     : TEXCOORD2,
  float4 LightPosition0 : TEXCOORD3,
  float4 LightPosition1 : TEXCOORD4,
  float4 LightPosition2 : TEXCOORD5,
  float3 normal      : TEXCOORD6,

  out float4 oColour     : COLOR
  )
{
    const vec3 lightDir = vec3(0.577, 0.577, 0.577);

    // calculate normal from texture coordinates
    vec3 N;
    N.xy = uv.xy*vec2(2.0, -2.0) + vec2(-1.0, 1.0);
    float mag = dot(N.xy, N.xy);
    if (mag > 1.0) discard;   // kill pixels outside circle
    N.z = sqrt(1.0-mag);

    // calculate lighting
    float diffuse = 0.8 + 0.8* max(0.0, dot(lightDir, N));

    float alpha = 0.5;
    oColour = float4(color.rgb * diffuse,alpha);
}
```

B.2 Uniform Grid Framework Code

```
// This software contains source code provided by NVIDIA Corporation.
// Specifically code from the CUDA 2.3 SDK "Particles" sample

#ifndef __UniformGrid_cu__
#define __UniformGrid_cu__

#include "K_UniformGrid_Utils.cu"

```

```
 9  // Calculate a grid hash value for each particle
10
11  __global__ void K_Grid_Hash (
12                    uint        numParticles,
13                    float_vec*   dParticlePositions,
14                    GridData     dGridData
15                    )
16  {
17    // particle index
18    uint index = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
19    if (index >= numParticles) return;
20
21    // particle position
22    float4 p = dParticlePositions[index];
23
24    // get address in grid
25    int3 gridPos = UniformGridUtils::calcGridCell(make_float3(p),
          cGridParams.grid_min, cGridParams.grid_delta);
26    uint hash = UniformGridUtils::calcGridHash<true>(gridPos, cGridParams.
          grid_res);
27
28    // store grid hash and particle index
29    dGridData.sort_hashes[index] = hash;
30    dGridData.sort_indexes[index] = index;
31
32  }
33
34
35  #endif
```

```
 1  // This software contains source code provided by NVIDIA Corporation.
 2  // Specifically code from the CUDA 2.3 SDK "Particles" sample
 3
 4  #ifndef __K_UniformGrid_Update_cu__
 5  #define __K_UniformGrid_Update_cu__
 6
 7  // read/write from the unsorted data structure to the sorted one
 8  template <class T, class D>
 9  __global__ void K_Grid_UpdateSorted (
10                    int    numParticles,
11                    D      dParticles,
12                    D      dParticlesSorted,
13                    GridData dGridData
14                    )
15  {
16    // particle index
17    uint index = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
18    if (index >= numParticles) return;
19
20    // blockSize + 1 elements
21    extern __shared__ uint sharedHash[];
22
23    uint hash = dGridData.sort_hashes[index];
24
25    // Load hash data into shared memory so that we can look
26    // at neighboring particle's hash value without loading
27    // two hash values per thread
```

```
28   sharedHash[threadIdx.x+1] = hash;
29   if (index > 0 && threadIdx.x == 0 ) {
30 
31     // first thread in block must load neighbor particle hash
32     sharedHash[0] = dGridData.sort_hashes[index-1];
33   }
34 
35 #ifndef __DEVICE_EMULATION__
36   __syncthreads ();
37 #endif
38 
39   // If this particle has a different cell index to the previous
40   // particle then it must be the first particle in the cell,
41   // so store the index of this particle in the cell.
42   // As it isn't the first particle, it must also be the cell end of
43   // the previous particle's cell
44 
45   if ((index == 0 || hash != sharedHash[threadIdx.x]) )
46   {
47     dGridData.cell_indexes_start[hash] = index;
48     if (index > 0)
49       dGridData.cell_indexes_end[sharedHash[threadIdx.x]] = index;
50   }
51 
52   if (index == numParticles - 1)
53   {
54     dGridData.cell_indexes_end[hash] = index + 1;
55   }
56 
57   uint sortedIndex = dGridData.sort_indexes[index];
58 
59   // Copy data from old unsorted buffer to sorted buffer
60   T::UpdateSortedValues(dParticlesSorted, dParticles, index, sortedIndex);
61 }
62 
63 #endif
```

```
1 #ifndef __K_UniformGrid_Utils_cu__
2 #define __K_UniformGrid_Utils_cu__
3 
4 namespace UniformGridUtils
5 {
6   // find the grid cell from a position in world space
7   static __device__ int3 calcGridCell(float3 const &p, float3 grid_min,
        float3 grid_delta)
8   {
9     // subtract grid_min (cell position) and multiply by delta
10    return make_int3((p-grid_min) * grid_delta);
11  }
12 
13 
14  // calculate hash from grid cell
15  template <bool wrapEdges>
16  static __device__ uint calcGridHash(int3 const &gridPos, float3 grid_res
        )
17  {
18    int gx,gy,gz;
```

```
19      if(wrapEdges)
20      {
21        int gsx = (int)floor(grid_res.x);
22        int gsy = (int)floor(grid_res.y);
23        int gsz = (int)floor(grid_res.z);
24
25  //        //power of 2 wrapping..
26  //        gx = gridPos.x & gsx-1;
27  //        gy = gridPos.y & gsy-1;
28  //        gz = gridPos.z & gsz-1;
29
30        // wrap grid... but since we can not assume size is power of 2 we
              can't use binary AND/& :/
31        gx = gridPos.x % gsx;
32        gy = gridPos.y % gsy;
33        gz = gridPos.z % gsz;
34        if(gx < 0) gx+=gsx;
35        if(gy < 0) gy+=gsy;
36        if(gz < 0) gz+=gsz;
37      }
38      else
39      {
40        gx = gridPos.x;
41        gy = gridPos.y;
42        gz = gridPos.z;
43      }
44
45      //return  __mul24(__mul24(gz, (int) cGridParams.grid_res.y)+gy, (int)
            cGridParams.grid_res.x) + gx;
46
47      //We choose to simply traverse the grid cells along the x, y, and z
            axes, in that order. The inverse of
48      //this space filling curve is then simply:
49      // index = x + y*width + z*width*height
50      //This means that we process the grid structure in "depth slice" order
            , and
51      //each such slice is processed in row-column order.
52      return __mul24(__umul24(gz, grid_res.y), grid_res.x) + __mul24(gy,
            grid_res.x) + gx;
53    }
54
55
56    // Iterate over particles found in the nearby cells (including cell of
          position_i)
57    template<class O, class D>
58    static __device__ void IterateParticlesInCell(
59      D               &data,
60      int3 const      &cellPos,
61      uint const      &index_i,
62      float3 const    &position_i,
63      GridData const    &dGridData
64  )
65    {
66      // get hash (of position) of current cell
67      volatile uint cellHash = UniformGridUtils::calcGridHash<true>(cellPos,
             cGridParams.grid_res);
68
```

```
69      // get start/end positions for this cell/bucket
70      //uint startIndex = FETCH_NOTEX(dGridData,cell_indexes_start,cellHash)
            ;
71      volatile uint startIndex = FETCH(dGridData,cell_indexes_start,cellHash
            );
72
73      // check cell is not empty
74      if (startIndex != 0xffffffff)
75      {
76        //uint endIndex = FETCH_NOTEX(dGridData,cell_indexes_end,cellHash);
77        volatile uint endIndex = FETCH(dGridData, cell_indexes_end, cellHash
            );
78
79        // iterate over particles in this cell
80        for(uint index_j=startIndex; index_j < endIndex; index_j++)
81        {
82          O::ForPossibleNeighbor(data, index_i, index_j, position_i);
83        }
84      }
85    }
86
87    // Iterate over particles found in the nearby cells (including cell of
          position_i)
88    template<class O, class D>
89    static __device__ void IterateParticlesInNearbyCells(
90      D            &data,
91      uint const       &index_i,
92      float3 const     &position_i,
93      GridData const     &dGridData)
94    {
95      O::PreCalc(data, index_i);
96
97      // get cell in grid for the given position
98      volatile int3 cell = UniformGridUtils::calcGridCell(position_i,
            cGridParams.grid_min, cGridParams.grid_delta);
99
100     // iterate through the 3^3 cells in and around the given position
101     // can't unroll these loops, they are not innermost
102     for(int z=cell.z-1; z<=cell.z+1; ++z)
103     {
104       for(int y=cell.y-1; y<=cell.y+1; ++y)
105       {
106         for(int x=cell.x-1; x<=cell.x+1; ++x)
107         {
108           IterateParticlesInCell<O,D>(data, make_int3(x,y,z), index_i,
                  position_i, dGridData);
109         }
110       }
111     }
112
113     O::PostCalc(data, index_i);
114   }
115   // Iterate over particles found in the neighbor list
116   template<class O, class D>
117   static __device__ void IterateParticlesInNearbyCells(
118     D            &data,
119     uint const       &index_i,
```

```
120     float3 const     &position_i,
121     NeighborList const   &dNeighborList
122     )
123   {
124     O::PreCalc(data, index_i);
125
126     // iterate over particles in neighbor list
127     for(uint counter=0; counter < dNeighborList.MAX_NEIGHBORS; counter++)
128     {
129       //const uint index_j = FETCH(dNeighborList,neighbors, index_i*
              dNeighborList.neighbors_pitch+counter);
130       const uint index_j = FETCH_NOTEX(dNeighborList,neighbors, index_i*
              dNeighborList.MAX_NEIGHBORS+counter);
131
132       // no more neighbors for this particle
133       if(index_j == 0xffffffff)
134         break;
135
136       O::ForPossibleNeighbor(data, index_i, index_j, position_i);
137
138     }
139
140     O::PostCalc(data, index_i);
141   }
142
143 };
144
145 #endif
```

B.3 Color Calculation Framework Code

```
1  #ifndef __K_Coloring_cu__
2  #define __K_Coloring_cu__
3
4  #include <cutil_math.h>
5
6  #include "K_Coloring.cuh"
7
8  //from http://www.cs.rit.edu/~ncs/color/t_convert.html
9  //The hue value H runs from 0 to 360ž.
10 //The saturation S is the degree of strength or purity and is from 0 to 1.
11 //Purity is how much white is added to the color, so S=1 makes the purest
       color (no white).
12 //Brightness V also ranges from 0 to 1, where 0 is the black.
13 __device__ float3 HSVtoRGB(float h, float s, float v )
14 {
15   float r=0,g=0,b=0;
16   int i;
17   float f, p, q, t;
18   if( s == 0 ) {
19     // achromatic (grey)
20     r = g = b = v;
21     return make_float3(r,g,b);
22   }
```

```
23    h /= 60;       // sector 0 to 5
24    i = floor( h );
25    f = h - i;       // factorial part of h
26    p = v * ( 1.0f - s );
27    q = v * ( 1.0f - s * f );
28    t = v * ( 1.0f - s * ( 1.0f - f ) );
29    switch( i ) {
30      case 0:
31        r = v;  g = t;  b = p;
32        break;
33      case 1:
34        r = q;  g = v;  b = p;
35        break;
36      case 2:
37        r = p;  g = v;  b = t;
38        break;
39      case 3:
40        r = p;  g = q;  b = v;
41        break;
42      case 4:
43        r = t;  g = p;  b = v;
44        break;
45      default:    // case 5:
46        r = v;  g = p;  b = q;
47        break;
48    }
49
50    return make_float3(r,g,b);
51  }
52
53
54  __device__ float3 calculateColor(ColoringGradient coloringGradient, float
        colorScalar)
55  {
56    float3 color = make_float3(0,0,0);
57    switch(coloringGradient)
58    {
59    case White:
60      // completely white
61      {
62        color = make_float3(1,1,1);
63      }
64      break;
65    case Blackish:
66      // acromatic gradient with V from 0 to 0.5
67      {
68        float h = colorScalar*0.5;
69        color = make_float3(h,h,h);
70      }
71      break;
72    case BlackToCyan:
73      {
74        color = make_float3(0,colorScalar,colorScalar);
75      }
76      break;
77    case BlueToWhite:
78      // blue to white gradient
```

```
79      {
80        color = make_float3(1-colorScalar, 1-0.5f*colorScalar, 1);
81      }
82      break;
83    case HSVBlueToRed:
84      // hsv gradient from blue to red (0 to 245 degrees in hue)
85      {
86        float h = clamp((1-colorScalar)*245.0f,0.0f,245.0f);
87        color = HSVtoRGB(h,0.5f,1);
88      }
89      break;
90    }
91    return color;
92  }
93
94
95  static __device__ float3 CalculateColor(ColoringGradient coloringGradient,
        SPHColoringSource coloringSource, float3 vnext, float pressure,
      float3 force)
96  {
97    float3 color = make_float3(0);
98    switch(coloringSource)
99    {
100   case Velocity:
101     // color given by velocity
102     {
103       float colorScalar = fabs(vnext.x)+fabs(vnext.y)+fabs(vnext.z) /
            11000.0;
104       colorScalar = clamp(colorScalar, 0.0f, 1.0f);
105       color =  calculateColor(coloringGradient, colorScalar);
106     }
107     break;
108   case Pressure:
109     // color given by pressure
110     {
111       float colorScalar = clamp(( (pressure - cFluidParams.rest_pressure)/
             400.0), 0.1f, 1.0f);
112       color =  calculateColor(coloringGradient, colorScalar);
113     }
114     break;
115   case Force:
116     // color given by force
117     {
118       if(coloringGradient == Direct)
119       {
120         //color = clamp(make_float3(0.5)+(force/80.0f),make_float3(0),
              make_float3(1));
121         color = clamp((fabs(force)/80.0f),make_float3(0),make_float3(1));
122       }
123       else
124       {
125         force /= 80.0f;
126         float colorScalar = clamp((force.x+force.y+force.z)/3.0f,0.1f,1.0f
              );
127         color =  calculateColor(coloringGradient, colorScalar);
128       }
129     }
```

```
130     break;
131   }
132   return color;
133 }
134
135
136 #endif
```

B.4 SPH Boundary Handling Code

```
1  #ifndef __K_Boundaries_Common_cu__
2  #define __K_Boundaries_Common_cu__
3
4
5  #define EPSILON     0.00001f      //for collision detection
6
7
8  __device__ float3 calculateRepulsionForce(
9                       float3 const& vel,
10                      float3 const& normal,
11                      float const& boundary_distance,
12                      float const& boundary_dampening,
13                      float const& boundary_stiffness
14                      )
15 {
16
17   // from ama06
18   return (boundary_stiffness * boundary_distance - boundary_dampening *
         dot(normal, vel)) * normal;
19 }
20
21
22 /*
23 COLLISION RESPONSE
24 SIMPLE REFLECTION
25 ................
26 ....^...........
27 Vn..|.../.V.....
28 ....|../........
29 ....|./.........
30 ....|/________>.
31 ........Vt......
32 ................
33
34 Vn = (Vbc * n)Vbc
35 Vt = Vbc Űvn
36 Vbc = velocity before collision
37 Vn = normal component of velocity
38 Vt == tangential component of velocity
39 V = (1-u)Vt Ű eVn
40 u = dynamic friction (affects tangent velocity)
41 e = resilience (affects normal velocity)
42 */
43
```

```
__device__ float3 calculateFrictionForce(
                        float3 const& vel,
                        float3 const& force,
                        float3 const& normal,
                        float const& friction_kinetic,
                        float const& friction_static_limit
                        )
{
  float3 friction_force = make_float3(0,0,0);

  // the normal part of the force vector (ie, the part that is going " towards" the boundary
  float3 f_n = force * dot(normal, force);
  // tangent on the terrain along the force direction (unit vector of tangential force)
  float3 f_t = force - f_n;

  // the normal part of the velocity vector (ie, the part that is going " towards" the boundary
  float3 v_n = vel * dot(normal, vel);
  // tangent on the terrain along the velocity direction (unit vector of tangential velocity)
  float3 v_t = vel - v_n;

  if((v_t.x + v_t.y + v_t.z)/3.0f > friction_static_limit)
    friction_force = -v_t;
  else
    friction_force = friction_kinetic * -v_t;

  // above static friction limit?
//    friction_force.x = f_t.x > friction_static_limit ? friction_kinetic * -v_t.x : -v_t.x;
//    friction_force.y = f_t.y > friction_static_limit ? friction_kinetic * -v_t.y : -v_t.y;
//    friction_force.z = f_t.z > friction_static_limit ? friction_kinetic * -v_t.z : -v_t.z;

  //TODO; friction should cause energy/heat in contact particles!
  friction_force = friction_kinetic * -v_t;

  return friction_force;
}

#endif
```

```
#ifndef __K_Boundaries_Terrain_cu__
#define __K_Boundaries_Terrain_cu__

#include "K_Boundaries_Common.cu"

#define EPSILON     0.00001f      //for collision detection


__device__ int2 getTerrainPos(float3 const &pos, int const &dTerrainSize, float const &dTerrainWorldSize)
{
  int2 terrainPos;
```

```
12   terrainPos.y = floor(pos.z*(dTerrainSize/dTerrainWorldSize));
13   terrainPos.x = floor(pos.x*(dTerrainSize/dTerrainWorldSize));
14   return terrainPos;
15 }
16
17 __device__ float getTerrainHeight(int const &terrainPosX, int const &
     terrainPosZ, float const *dTerrainHeights, int const &dTerrainSize)
18 {
19   return dTerrainHeights[((dTerrainSize) * (dTerrainSize) - 1) - (((
       dTerrainSize) * terrainPosZ)) + terrainPosX];
20 }
21
22 __device__ float getTerrainHeight(int2 const &terrainPos, float const *
     dTerrainHeights, int const &dTerrainSize)
23 {
24   return getTerrainHeight(terrainPos.x, terrainPos.y, dTerrainHeights,
       dTerrainSize);
25 }
26
27 __device__ float getTerrainHeightInterpolate(
28   float3 const &pos,
29   int const &dTerrainSize,
30   float const &dTerrainWorldSize,
31   float const *dTerrainHeights)
32 {
33   int2 tpos = getTerrainPos(pos, dTerrainSize, dTerrainWorldSize);
34
35   int Xa = tpos.x;       // x on one side
36   int Xb = tpos.x + 1; // x on the other side
37   int Za = tpos.y;       // z on one side
38   int Zb = tpos.y+1; // z on the other side
39
40   float Xd = pos.x-floor(pos.x);
41   if (Xd < 0.0f)
42     Xd *= -1.0f;
43   float Zd = pos.z-floor(pos.y);
44   if (Zd < 0.0f)
45     Zd *= -1.0f;
46
47   float b = lerp(getTerrainHeight(Xa,Zb, dTerrainHeights, dTerrainSize),
       getTerrainHeight(Xb,Zb, dTerrainHeights, dTerrainSize),Xd);
48   float a = lerp(getTerrainHeight(Xa,Za, dTerrainHeights, dTerrainSize),
       getTerrainHeight(Xb,Za, dTerrainHeights, dTerrainSize),Xd);
49   return lerp(a,b,Zd);
50
51 }
52
53 __device__ float3 getTerrainNormal(
54                  int2 const &terrainPos,
55                  float4 const *dTerrainNormals,
56                  int const &dTerrainSize)
57 {
58   // TODO: perhaps interpolate normals (with curve estimation?)
59   float4 normal = (dTerrainNormals[((dTerrainSize) * (dTerrainSize)) - (((
       dTerrainSize) * terrainPos.y)) + terrainPos.x]);
60   return make_float3(normal);
61 }
```

```
62
63 __device__ float3 calculateTerrainNoPenetrationForce(
64   float3 & pos,
65   float3 const& vel,
66   float3 const& fluidWorldPosition,
67   TerrainData const &dTerrainData,
68   float const& boundary_distance,
69   float const& boundary_stiffness,
70   float const& boundary_dampening,
71   float const& scale_to_simulation
72   )
73 {
74   float3 repulsion_force = make_float3(0,0,0);
75   float diff;
76
77   int2 terrainPos = getTerrainPos(pos+fluidWorldPosition+dTerrainData.
       position, dTerrainData.size, dTerrainData.world_size);
78
79   if(terrainPos.x >= 0 && terrainPos.x < dTerrainData.size && terrainPos.y
        >= 0 && terrainPos.y < dTerrainData.size)
80   {
81     //float terrainHeight = getTerrainHeightInterpolate(pos, dTerrainData.
         size,dTerrainData.world_size, dTerrainData.heights);
82     float terrainHeight = -dTerrainData.position.y - fluidWorldPosition.y
         + getTerrainHeight(terrainPos, dTerrainData.heights, dTerrainData.
         size);
83     float3 terrainNormal = getTerrainNormal(terrainPos, dTerrainData.
         normals, dTerrainData.size);
84
85     if(pos.y < terrainHeight)
86       pos.y  = terrainHeight;
87
88     diff = 2 * boundary_distance - (pos.y  - terrainHeight) *
         scale_to_simulation;
89     if (diff > EPSILON)
90     {
91       repulsion_force += calculateRepulsionForce(vel, terrainNormal, diff,
            boundary_dampening, boundary_stiffness);
92     }
93   }
94   return repulsion_force;
95 }
96
97
98 __device__ float3 calculateTerrainFrictionForce(
99   float3 const& pos,
100  float3 const& vel,
101  float3 const& force,
102  float3 const& fluidWorldPosition,
103  TerrainData const &dTerrainData,
104  float const& boundary_distance,
105  float const& friction_kinetic,
106  float const& friction_static_limit,
107  float const& scale_to_simulation
108  )
109 {
110  float3 friction_force = make_float3(0,0,0);
```

```
111    float diff;
112
113    int2 terrainPos = getTerrainPos(pos+fluidWorldPosition+dTerrainData.
           position, dTerrainData.size, dTerrainData.world_size);
114
115    if(terrainPos.x >= 0 && terrainPos.x < dTerrainData.size && terrainPos.y
            >= 0 && terrainPos.y < dTerrainData.size )
116    {
117      //float terrainHeight = getTerrainHeightInterpolate(pos, dTerrainData.
             size,dTerrainData.world_size, dTerrainData.heights);
118      float terrainHeight = -dTerrainData.position.y - fluidWorldPosition.y
             +getTerrainHeight(terrainPos, dTerrainData.heights, dTerrainData.
             size);
119      float3 terrainNormal = getTerrainNormal(terrainPos, dTerrainData.
             normals, dTerrainData.size);
120
121      // simple limit for terrain collision
122      diff = 3 * boundary_distance - (pos.y - terrainHeight) *
             scale_to_simulation;
123      if (diff > EPSILON)
124      {
125        friction_force += calculateFrictionForce(vel, force, terrainNormal,
               friction_kinetic, friction_static_limit);
126      }
127    }
128    return friction_force;
129  }
130
131  #endif
```

```
1  #ifndef __K_Boundaries_Walls_cu__
2  #define __K_Boundaries_Walls_cu__
3
4  #include "K_Boundaries_Common.cu"
5
6  #define EPSILON      0.00001f       //for collision detection
7
8  __device__ float3 calculateWallsNoPenetrationForce(
9    float3 const& pos,
10   float3 const& vel,
11   float3 const& grid_min,
12   float3 const& grid_max,
13   float const& boundary_distance,
14   float const& boundary_stiffness,
15   float const& boundary_dampening,
16   float const& scale_to_simulation)
17 {
18   float3 repulsion_force = make_float3(0,0,0);
19   float diff;
20
21   // simple limit for "wall" in Y direction (min of simulated volume)
22   diff = boundary_distance - ((pos.y - grid_min.y ) * scale_to_simulation)
         ;
23   if (diff > EPSILON) {
24     float3 normal = make_float3(0,1,0);
25     repulsion_force  += calculateRepulsionForce(vel, normal, diff,
           boundary_dampening, boundary_stiffness);
```

```
26    }
27
28    // simple limit for "wall" in Y direction (max of simulated volume)
29    diff = boundary_distance - ((grid_max.y - pos.y ) * scale_to_simulation)
        ;
30    if (diff > EPSILON) {
31      float3 normal = make_float3(0,-1,0);
32      repulsion_force  += calculateRepulsionForce(vel, normal, diff,
          boundary_dampening, boundary_stiffness);
33    }
34
35    // simple limit for "wall" in Z direction (min of simulated volume)
36    diff = boundary_distance - ((pos.z - grid_min.z ) * scale_to_simulation)
        ;
37    if (diff > EPSILON ) {
38      float3 normal = make_float3(0,0,1);
39      repulsion_force  += calculateRepulsionForce(vel, normal, diff,
          boundary_dampening, boundary_stiffness);
40    }
41
42    // simple limit for "wall" in Z direction (max of simulated volume)
43    diff = boundary_distance - ((grid_max.z - pos.z ) * scale_to_simulation)
        ;
44    if (diff > EPSILON) {
45      float3 normal = make_float3(0,0,-1);
46      float adj =  boundary_stiffness * diff - boundary_dampening * dot(
          normal, vel);
47      repulsion_force  += adj * normal;
48    }
49
50    // simple limit for "wall" in X direction (min of simulated volume)
51    diff = boundary_distance - ((pos.x - grid_min.x ) * scale_to_simulation)
        ;
52    if (diff > EPSILON ) {
53      float3 normal = make_float3(1,0,0);
54      repulsion_force  += calculateRepulsionForce(vel, normal, diff,
          boundary_dampening, boundary_stiffness);
55    }
56
57    // simple limit for "wall" in X direction (max of simulated volume)
58    diff = boundary_distance - ((grid_max.x - pos.x ) * scale_to_simulation)
        ;
59    if (diff > EPSILON) {
60      float3 normal = make_float3(-1,0,0);
61      repulsion_force  += calculateRepulsionForce(vel, normal, diff,
          boundary_dampening, boundary_stiffness);
62    }
63
64    return repulsion_force;
65  }
66
67
68  __device__ float3 calculateWallsNoSlipForce(
69    float3 const& pos,
70    float3 const& vel,
71    float3 const& force,
72    float3 const& grid_min,
```

```
73   float3 const& grid_max,
74   float const& boundary_distance,
75   float const& friction_kinetic,
76   float const& friction_static_limit,
77   float const& scale_to_simulation)
78 {
79   float3 friction_force = make_float3(0,0,0);
80   float diff;
81 
82   // simple limit for "wall" in Y direction (min of simulated volume)
83   diff = boundary_distance - ((pos.y - grid_min.y ) * scale_to_simulation)
         ;
84   if (diff > EPSILON) {
85     float3 normal = make_float3(0,1,0);
86     friction_force += calculateFrictionForce(vel, force, normal,
           friction_kinetic, friction_static_limit);
87   }
88 
89   // simple limit for "wall" in Y direction (max of simulated volume)
90   diff = boundary_distance - ((grid_max.y - pos.y ) * scale_to_simulation)
         ;
91   if (diff > EPSILON) {
92     float3 normal = make_float3(0,-1,0);
93     friction_force += calculateFrictionForce(vel, force, normal,
           friction_kinetic, friction_static_limit);
94   }
95 
96   // simple limit for "wall" in Z direction (min of simulated volume)
97   diff = boundary_distance - ((pos.z - grid_min.z ) * scale_to_simulation)
         ;
98   if (diff > EPSILON ) {
99     float3 normal = make_float3(0,0,1);
100    friction_force += calculateFrictionForce(vel, force, normal,
           friction_kinetic, friction_static_limit);
101  }
102 
103  // simple limit for "wall" in Z direction (max of simulated volume)
104  diff = boundary_distance - ((grid_max.z - pos.z ) * scale_to_simulation)
         ;
105  if (diff > EPSILON) {
106    float3 normal = make_float3(0,0,-1);
107    friction_force += calculateFrictionForce(vel, force, normal,
           friction_kinetic, friction_static_limit);
108  }
109 
110  // simple limit for "wall" in X direction (min of simulated volume)
111  diff = boundary_distance - ((pos.x - grid_min.x ) * scale_to_simulation)
         ;
112  if (diff > EPSILON ) {
113    float3 normal = make_float3(1,0,0);
114    friction_force += calculateFrictionForce(vel, force, normal,
           friction_kinetic, friction_static_limit);
115  }
116 
117  // simple limit for "wall" in X direction (max of simulated volume)
118  diff = boundary_distance - ((grid_max.x - pos.x ) * scale_to_simulation)
         ;
```

```
119   if (diff > EPSILON) {
120     float3 normal = make_float3(-1,0,0);
121     friction_force += calculateFrictionForce(vel, force, normal,
          friction_kinetic, friction_static_limit);
122   }
123
124   return friction_force;
125 }
126
127 #endif
```

B.5 SPH Neighbor Iteration Framework Code

```
1  #ifndef __K_SPH_Common_cu__
2  #define __K_SPH_Common_cu__
3
4  template<class O, class D>
5  class SPHNeighborCalc
6  {
7  public:
8    // this is called before the loop over each neighbor particle
9    static __device__ void PreCalc(D &data, uint index_i)
10   {
11     O::PreCalc(data, index_i);
12   }
13
14   static __device__ void ForNeighbor(D &data, uint const &index_i, uint
         const &index_j, float3 const &r, float const &rlen)
15   {
16     O::ForNeighbor(data, index_i, index_j, r, rlen);
17   }
18
19   // this is called after the loop over each particle in a cell
20   static __device__ void PostCalc(D &data, uint index_i)
21   {
22     O::PostCalc(data, index_i);
23   }
24
25   // this is called inside the loop over each particle in a cell
26   static __device__ void ForPossibleNeighbor(D &data, uint const &index_i,
          uint const &index_j, float3 const &position_i)
27   {
28     // check not colliding with self
29     if (index_j != index_i)
30     {
31       // get the particle info (in the current grid) to test against
32       float3 position_j = make_float3(FETCH(data.dParticleDataSorted,
             position, index_j));
33
34       // get the relative distance between the two particles, translate to
              simulation space
35       float3 r = (position_i - position_j) * cFluidParams.
             scale_to_simulation;
36
```

```
37        float rlen_sq = dot(r,r);
38        // |r|
39        float rlen = sqrtf(rlen_sq);
40
41        // is this particle within cutoff?
42        if (rlen <= cFluidParams.smoothing_length)
43        {
44          O::ForNeighbor(data, index_i, index_j, r, rlen, rlen_sq);
45        }
46      }
47    }
48
49  };
50
51  #endif
```

B.6 SPH Smoothing Kernels Code

```
1  #ifndef __K_SPH_Kernels_cubic_cu__
2  #define __K_SPH_Kernels_cubic_cu__
3
4  // TODO
5  // see crespo_thesis.pdf for summary of kernels and tensile correction
       terms etc!
6  // add tensile correction terms!
7
8  // used by "A fully explicit three-step SPH algorithm for simulation of
       non-Newtonian fluid flow"
9
10 //third order B-spline
11 class Wcubic
12 {
13 public:
14
15   static __device__ __host__ float Kernel(float smoothing_length, float3 r
         , float rlen)
16   {
17     float Q = rlen / smoothing_length;
18
19     if(Q <= 1)
20     {
21       // for 2D
22       //float c = 10 * M_1_PI / 7 * smoothInvSq;
23       // for 3D
24       float c = 1/(M_PI*(smoothing_length*smoothing_length*
           smoothing_length));
25       return c * (1 - 1.5f*Q*Q + 0.75f*Q*Q*Q);
26     }
27     else if(Q <= 2)
28     {
29       // for 2D
30       //float c = 10 * M_1_PI / 28 * smoothInvSq;
31       // for 3D
```

```
32        float c = 0.25f/(M_PI/(smoothing_length*smoothing_length*
              smoothing_length));
33
34        float dif = Q-2;
35        return - c * dif * dif * dif;
36      }
37      return 0;
38    }
39
40    static __device__ __host__ float3 Gradient(float smoothing_length, float
          smoothing_length_pow2, float smoothing_length_pow3, float
        smoothing_length_pow4, float3 r, float rlen, float rlen_sq)
41    {
42      float Q = rlen / smoothing_length;
43
44      if(Q <= 1)
45      {
46        // for 3D
47        float c =  1 / (M_PI *(smoothing_length_pow3));
48        return - r * c * ( 3/(smoothing_length_pow2) - (9*rlen)/(4*
              smoothing_length_pow3) );
49      }
50      else if(Q <= 2)
51      {
52        // for 3D
53        float c = 3 / ( 4* M_PI * (smoothing_length_pow4));
54        float dif = Q-2;
55        return - r * (c * dif * dif) / rlen;
56      }
57      return make_float3(0.0f);
58    }
59
60  };
61  #endif
```

```
1    #ifndef __K_SPH_Kernels_gaussian_cu__
2  #define __K_SPH_Kernels_gaussian_cu__
3
4  // from "PhD Thesis: Application of the Smoothed Particle Hydrodynamics
       model SPHysics to free-surface hydrodynamics
5  class Wgaussian
6  {
7  public:
8
9    static __device__ __host__ float Kernel_Constant(float smoothing_length,
         float smoothing_length_pow2)
10   {
11     // for 2d
12     //float c = 1/(M_PI * smoothing_length_pow2);
13     // for 3d
14     float c = 1/(powf(M_PI, 1.5f)*smoothing_length_pow2*smoothing_length);
15     return c;
16   }
17
18   static __device__ __host__ float Kernel_Variable(float smoothing_length,
         float smoothing_length_pow2, float3 r, float rlen, float rlen_sq)
19   {
```

```
20     float Q = rlen/smoothing_length;
21
22     if(0<=Q && Q<=2)
23     {
24       return 1/expf((smoothing_length_pow2*rlen_sq));
25     }
26     return 0.f;
27   }
28
29   static __device__ __host__ float Kernel(float smoothing_length, float
         smoothing_length_pow2, float3 r, float rlen, float rlen_sq)
30   {
31     return Kernel_Constant(smoothing_length,smoothing_length_pow2) *
           Kernel_Variable(smoothing_length, smoothing_length_pow2, r, rlen,
           rlen_sq);
32   }
33
34   static __device__ __host__ float Gradient_Constant(float
         smoothing_length, float smoothing_length_pow2)
35   {
36     // for 3d
37     float c = -2/(powf(M_PI, 0.5f)*smoothing_length_pow2*
           smoothing_length_pow2*smoothing_length);
38     return c;
39   }
40
41   static __device__ __host__ float3 Gradient_Variable(float
         smoothing_length, float smoothing_length_pow2, float3 r, float rlen,
          float rlen_sq)
42   {
43     float Q = rlen/smoothing_length;
44
45     if(0<Q && Q<2)
46     {
47       return r/expf((smoothing_length_pow2*rlen_sq));
48     }
49     return make_float3(0.f);
50   }
51
52   static __device__ __host__ float3 Gradient(float smoothing_length, float
          smoothing_length_pow2, float3 r, float rlen, float rlen_sq)
53   {
54     return Gradient_Constant(smoothing_length, smoothing_length_pow2) *
           Gradient_Variable(smoothing_length, smoothing_length_pow2, r, rlen
           , rlen_sq);
55   }
56
57 };
58
59
60 #endif
```

```
1 #ifndef __K_SPH_Kernels_Wpoly6_cu__
2 #define __K_SPH_Kernels_Wpoly6_cu__
3
4 class Wpoly6
5 {
```

```
6 public:
7
8   static __device__ __host__ float Kernel_Constant(float smoothing_length)
9   {
10    return 315.0f / (64.0f * M_PI * pow(smoothing_length, 9.0f) );
11  }
12
13  static __device__ __host__ float Kernel_Variable(float
        smoothing_length_pow2, float3 r, float rlen_sq)
14  {
15    float hsq_rlensq = smoothing_length_pow2 - rlen_sq;
16    return hsq_rlensq * hsq_rlensq * hsq_rlensq;
17  }
18
19
20  static __device__ __host__ float Gradient_Constant(float
        smoothing_length)
21  {
22    return -945.0f / (32.0f * M_PI * pow(smoothing_length, 9.0f) );
23  }
24
25  static __device__ __host__ float Gradient_Variable(float
        smoothing_length, float smoothing_length_pow2, float3 r, float rlen)
26  {
27    // h - |r|^2
28    float hsq_rlensq = smoothing_length_pow2 - (rlen*rlen);
29    return hsq_rlensq * hsq_rlensq;
30  }
31
32  static __device__ __host__ float Gradient(float smoothing_length, float
        smoothing_length_pow2, float3 r, float rlen)
33  {
34    return Gradient_Constant(smoothing_length) * Gradient_Variable(
          smoothing_length, smoothing_length_pow2, r, rlen);
35  }
36
37  static __device__ __host__ float Laplace_Constant(float smoothing_length
        , float smoothing_length_pow2, float3 r, float rlen)
38  {
39    return -945.0f / (32.0f * M_PI * pow(smoothing_length, 9.0f) );
40  }
41
42  static __device__ __host__ float Laplace_Variable(float smoothing_length
        , float smoothing_length_pow2, float3 r, float rlen)
43  {
44    // |r|^2
45    float rlen_sq = rlen*rlen;
46    // h - |r|^2
47    float part1 = smoothing_length_pow2 - rlen_sq;
48    // 3h - 7|r|^2
49    float part2 = 3.0f*smoothing_length_pow2 - 7.0f*rlen_sq;
50    return part1 * part2;
51  }
52 };
53
54 #endif
```

```
#ifndef __K_SPH_Kernels_quadratic_cu__
#define __K_SPH_Kernels_quadratic_cu__

// from "PhD Thesis: Application of the Smoothed Particle Hydrodynamics
    model SPHysics to free-surface hydrodynamics
class Wquadratic
{
public:

  static __device__ __host__ float Kernel_Constant(float smoothing_length)
  {
    // for 2d
    //float c = 2/(M_PI * smoothing_length * smoothing_length);
    // for 3d
    float c = 5/(4*M_PI*smoothing_length*smoothing_length*smoothing_length
        );

    return c;
  }

  static __device__ __host__ float Kernel_Variable(float smoothing_length,
       float smoothing_length_pow2, float3 r, float rlen)
  {
    float q = rlen/smoothing_length;

    if(0<=q && q<=2)
    {
      return 0.1875f*q*q - 0.75*q + 0.75;
    }
    return 0.f;
  }

  static __device__ __host__ float Gradient_Constant(float
      smoothing_length)
  {
    //TODO
  }

  static __device__ __host__ float Gradient_Variable(float
      smoothing_length, float smoothing_length_pow2, float3 r, float rlen)
  {
  }

};

#endif
```

```
#ifndef __K_SPH_Kernels_quintic_cu__
#define __K_SPH_Kernels_quintic_cu__

// TODO
// see crespo_thesis.pdf for summary of kernels and tensile correction
    terms etc!

// the quintic wendland kernel [Wendland, 1995]
class Wquintic
```

```
9 {
10 public:
11
12   static __device__ __host__ float Kernel(float smoothing_length, float3 r
        , float rlen, float rlen_sq)
13   {
14     float Q = rlen / smoothing_length;
15     if(Q < 2)
16     {
17       // for 2D
18       //float c = 7.0f/(4.0f*M_PI*rlen_sq);
19       // for 3D
20       float c = 7.0f/(8.0f*M_PI*rlen*rlen_sq);
21       return c * pow(1-0.5f*Q, 4) * (2*Q+1);
22     }
23     return 0;
24   }
25
26   static __device__ __host__ float3 Gradient(float smoothing_length,
        float3 r, float rlen, float rlen_sq)
27   {
28     float Q = rlen / smoothing_length;
29     if(Q < 2)
30     {
31       // for 2D
32       //scalar c = (-35 * M_1_PI / 4 * rlen_sq*rlen_sq);
33       // for 3D
34       float c =  (-35 * M_1_PI) / (8 * rlen_sq*rlen_sq);
35       float dif = 2 - Q;
36       return r * (c * dif * dif / r);
37     }
38     return make_float3(0,0,0);
39   }
40
41
42 };
43 #endif
```

```
1 #ifndef __K_SPH_Kernels_Wspiky_cu__
2 #define __K_SPH_Kernels_Wspiky_cu__
3
4 // Spiky kernel by Desbrun and Gascuel, also used by Müller et al.
5 class Wspiky
6 {
7 public:
8
9   static __device__ __host__ float Kernel_Constant(float smoothing_length)
10  {
11    return 15.0f / (M_PI * pow(smoothing_length, 6.0f) );
12  }
13
14  static __device__ __host__ float Kernel_Variable(float smoothing_length,
         float3 r, float rlen)
15  {
16    // h - |r|
17    float h_rlen =  (smoothing_length - rlen);
18    return h_rlen*h_rlen*h_rlen;
```

```
19    }
20
21    static __device__ __host__ float3 Gradient(float smoothing_length,
        float3 r, float rlen)
22    {
23      return Gradient_Constant(smoothing_length) * Gradient_Variable(
          smoothing_length, r, rlen);
24    }
25
26    static __device__ __host__ float Gradient_Constant(float
        smoothing_length)
27    {
28      return -45.0f / (M_PI * pow(smoothing_length, 6.0f) );
29    }
30
31    static __device__ __host__ float3 Gradient_Variable(float
        smoothing_length, float3 r, float rlen)
32    {
33      // h - |r|
34      float h_rlen = (smoothing_length-rlen);
35      return r*(1.0f/rlen)*(h_rlen*h_rlen);
36    }
37
38    static __device__ __host__ float Laplace_Constant(float smoothing_length
        )
39    {
40      return -90.0f / (M_PI * pow(smoothing_length, 6.0f) );
41    }
42
43    static __device__ __host__ float3 Laplace_Variable(float
        smoothing_length, float3 r, float rlen)
44    {
45      // h - |r|
46      float h_rlen = (smoothing_length-rlen);
47      float h_2rlen = (smoothing_length-2*rlen);
48      return (1.0f/r) * (h_rlen*h_2rlen);
49    }
50  };
51
52  #endif
```

```
1   #ifndef __K_SPH_Kernels_Wviscosity_cu__
2   #define __K_SPH_Kernels_Wviscosity_cu__
3
4   // Viscosity kernel from Müller et al.
5   class Wviscosity
6   {
7   public:
8
9     static __device__ __host__ float Kernel_Constant(float smoothing_length)
10    {
11      return 15.0f / (M_PI * pow(smoothing_length, 6.0f) );
12    }
13
14    static __device__ __host__ float Kernel_Variable(float smoothing_length,
         float3 r, float rlen)
15    {
```

```
16      float h_rlen =  (smoothing_length - rlen);
17      return h_rlen*h_rlen*h_rlen;
18    }
19
20    static __device__ __host__ float Gradient_Constant(float
        smoothing_length)
21    {
22      return 15.0f / (2 * M_PI * pow(smoothing_length, 3.0f) );
23    }
24
25    static __device__ __host__ float Gradient_Variable(float
        smoothing_length, float3 r, float rlen)
26    {
27      float part1 = (-3*rlen)/(2*pow(smoothing_length, 3.0f));
28      float part2 = (2/smoothing_length*smoothing_length);
29      float part3 = -smoothing_length/(2*pow(rlen,3.0f));
30      return part1 + part2 + part3;
31    }
32
33    static __device__ __host__ float Laplace_Constant(float smoothing_length
        )
34    {
35      return 45.0f / (M_PI * pow(smoothing_length, 6.0f) );
36    }
37
38    static __device__ __host__ float Laplace_Variable(float smoothing_length
        , float3 r, float rlen)
39    {
40      float h_rlen = (smoothing_length-rlen);
41      return h_rlen;
42    }
43  };
44  #endif
```

B.7 Simple SPH Implementation Code

```
1  #ifndef __K_SimpleSPH_Step1_cu__
2  #define __K_SimpleSPH_Step1_cu__
3
4  #include "K_UniformGrid_Utils.cu"
5  #include "K_SPH_Kernels.cu"
6  #include "K_SPH_Common.cu"
7
8  class Step1
9  {
10 public:
11
12   struct Data
13   {
14     float sum_density;
15
16     SimpleSPHData dParticleDataSorted;
17   };
18
```

```
19   class Calc
20   {
21   public:
22
23     static __device__ void PreCalc(Data &data, uint const &index_i)
24     {
25       // read particle data from sorted arrays
26       data.sum_density = 0;
27     }
28
29     static __device__ void ForNeighbor(Data &data, uint const &index_i,
         uint const &index_j, float3 const &r, float const& rlen, float
         const &rlen_sq)
30     {
31       // the density sum using Wpoly6 kernel
32       data.sum_density += SPH_Kernels::Wpoly6::Kernel_Variable(
           cPrecalcParams.smoothing_length_pow2, r, rlen_sq);
33     }
34
35     static __device__ void PostCalc(Data &data, uint index_i)
36     {
37       // Compute the density field at the current particle,
38       // Calculate the W smoothing function for this particle, mass and
           the poly6_grad_coeff has been moved outside the sum because they
            are constant.
39       float density = max(1.0, cFluidParams.particle_mass * cPrecalcParams
           .kernel_poly6_coeff * data.sum_density);
40       data.dParticleDataSorted.density[index_i]= density;
41
42       // ideal gas equation of state (by Desbrun and Cani in "Smoothed
           particles: A new paradigm for animating highly deformable bodies
           ")
43       data.dParticleDataSorted.pressure[index_i] = cFluidParams.
           rest_pressure + cFluidParams.gas_stiffness * (density -
           cFluidParams.rest_density);
44     }
45   };
46 };
47
48
49 __global__ void K_SumStep1(uint         numParticles,
50                  NeighborList   dNeighborList,
51                  SimpleSPHData  dParticleDataSorted,
52                  GridData const dGridData
53                  )
54 {
55   // particle index
56   uint index = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
57   if (index >= numParticles) return;
58
59   Step1::Data data;
60   data.dParticleDataSorted = dParticleDataSorted;
61
62   float3 position_i = make_float3(FETCH(dParticleDataSorted, position,
       index));
63
64   // Do calculations on particles in neighboring cells
```

```
65 #ifdef SPHSIMLIB_USE_NEIGHBORLIST
66   UniformGridUtils::IterateParticlesInNearbyCells<SPHNeighborCalc<Step1::
         Calc, Step1::Data>, Step1::Data>(data, index, position_i,
         dNeighborList);
67 #else
68   UniformGridUtils::IterateParticlesInNearbyCells<SPHNeighborCalc<Step1::
         Calc, Step1::Data>, Step1::Data>(data, index, position_i, dGridData)
         ;
69 #endif
70
71 }
72
73 #endif
```

```
1  #ifndef __K_SimpleSPH_Step2_cu__
2  #define __K_SimpleSPH_Step2_cu__
3
4  #include "K_UniformGrid_Utils.cu"
5  #include "K_SPH_Kernels.cu"
6  #include "K_SPH_Common.cu"
7
8
9  class Step2
10 {
11 public:
12
13   struct Data
14   {
15     float3 veleval_i;
16     float density_i;
17     float pressure_i;
18
19     float3 veleval_j;
20     float density_j;
21     float pressure_j;
22
23     float3 f_viscosity;
24     float3 f_pressure;
25
26     SimpleSPHData dParticleDataSorted;
27   };
28
29   template <SPHSymmetrization symmetrizationType>
30   class Calc
31   {
32   public:
33
34     // this is called before the loop over each neighbor particle
35     static __device__ void PreCalc(Data &data, uint index_i)
36     {
37       // read particle data from sorted arrays
38       data.veleval_i  = FETCH_FLOAT3(data.dParticleDataSorted, veleval,
             index_i);
39       data.density_i  = FETCH(data.dParticleDataSorted, density, index_i);
40       data.pressure_i = FETCH(data.dParticleDataSorted, pressure, index_i)
             ;
41
```

```
42        data.f_pressure    = make_float3(0,0,0);
43        data.f_viscosity   = make_float3(0,0,0);
44      }
45
46      static __device__ void ForNeighbor(Data &data, uint const &index_i,
            uint const &index_j, float3 const &r, float const& rlen, float
            const &rlen_sq)
47      {
48        data.veleval_j  = FETCH_FLOAT3(data.dParticleDataSorted, veleval,
              index_j);
49        data.density_j  = FETCH(data.dParticleDataSorted, density,  index_j)
              ;
50        data.pressure_j = FETCH(data.dParticleDataSorted, pressure, index_j)
              ;
51
52
53        // pressure  force calc
54        switch (symmetrizationType)
55        {
56          //mueller symmetrization of density
57        case SPH_PRESSURE_MUELLER:
58          {
59            // in the mueller paper, density_i is placed outside the force
                  defs..., but we calc it here.. easier(atm)
60            // from paper:  f_pressure = -(1/rho_i)* SUM(m_j * ((p_i + p_j)
                  / (2rho_j)) DELWpress
61            // we move the mass the 1/2 and the Wpress constants to precalc.
62            data.f_pressure  += ( (data.pressure_i + data.pressure_j) / (
                  data.density_j * data.density_i) ) * SPH_Kernels::Wspiky::
                  Gradient_Variable(cFluidParams.smoothing_length, r, rlen);
63          }
64          break;
65          //from "Particle-based viscoplastic fluid/solid simulation", also
                see "SPH survival kit"
66        case SPH_PRESSURE_VISCOPLASTIC:
67          {
68            data.f_pressure  += ( (data.pressure_i/(data.density_i*data.
                  density_i)) + (data.pressure_j/(data.density_j*data.
                  density_j)) ) * SPH_Kernels::Wspiky::Gradient_Variable(
                  cFluidParams.smoothing_length, r, rlen);
69            break;
70          }
71        }
72
73        // viscosity from mueller paper : f_viscosity = (ţ/rho_i)SUM(m_j * (
              v_j-v_i)/(rho_j)DEL^2Wvis
74        // we move the mass and the Wvis constants to precalc
75        data.f_viscosity += ( (data.veleval_j  - data. veleval_i ) / (data.
              density_j * data.density_i) ) * SPH_Kernels::Wviscosity::
              Laplace_Variable(cFluidParams.smoothing_length, r, rlen);
76      }
77
78      // this is called after the loop over each particle in a cell
79      static __device__ void PostCalc(Data &data, uint index_i)
80      {
81        float3 sum_sph_force = (cPrecalcParams.kernel_pressure_precalc *
              data.f_pressure + cPrecalcParams.kernel_viscosity_precalc * data
```

```
            .f_viscosity );
82
83       // Calculate the force, the particle_mass is added here because it
            is constant and thus there is no need for it to be inside the
            sum loop.
84       data.dParticleDataSorted.sph_force[index_i] = make_vec(sum_sph_force
             * cFluidParams.particle_mass);
85     }
86   };
87
88 };
89
90 template <SPHSymmetrization symmetrization>
91 __global__ void K_SumStep2(uint      numParticles,
92                SimpleSPHData  dParticleDataSorted,
93 #ifdef SPHSIMLIB_USE_NEIGHBORLIST
94                NeighborList dNeighborList,
95 #else
96                GridData   dGridData
97 #endif
98                )
99 {
100   // particle index
101   uint index = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
102   if (index >= numParticles) return;
103
104   Step2::Data data;
105
106   data.dParticleDataSorted = dParticleDataSorted;
107
108   float3 position_i = FETCH_FLOAT3(data.dParticleDataSorted, position,
        index);
109
110   // Do calculations on particles in neighboring cells
111 #ifdef SPHSIMLIB_USE_NEIGHBORLIST
112   UniformGridUtils::IterateParticlesInNearbyCells<SPHNeighborCalc<Step2::
        Calc<symmetrization>, Step2::Data>, Step2::Data>(data, index,
        position_i, dNeighborList);
113 #else
114   UniformGridUtils::IterateParticlesInNearbyCells<SPHNeighborCalc<Step2::
        Calc<symmetrization>, Step2::Data>, Step2::Data>(data, index,
        position_i, dGridData);
115 #endif
116 }
117
118 #endif
```

```
1 #ifndef __K_SimpleSPH_Integrate_cu__
2 #define __K_SimpleSPH_Integrate_cu__
3
4 #include "K_Coloring.cu"
5 #include "K_Boundaries_Terrain.cu"
6 #include "K_Boundaries_Walls.cu"
7
8 template<SPHColoringSource coloringSource, ColoringGradient
     coloringGradient>
9 __global__ void K_Integrate(int       numParticles,
```

```
10                    bool      gridWallCollisions,
11                    bool      terrainCollisions,
12                    float     delta_time,
13                    bool      progress,
14                    GridData    dGridData,
15                    SimpleSPHData dParticleData,
16                    SimpleSPHData dParticleDataSorted,
17                    float3      fluidWorldPosition,
18                    TerrainData   dTerrainData
19                    )
20  {
21    int index = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
22    if (index >= numParticles) return;
23
24    float3 pos       = make_float3(FETCH_NOTEX(dParticleDataSorted, position,
         index));
25    float3 vel       = make_float3(FETCH_NOTEX(dParticleDataSorted, velocity,
         index));
26    float3 vel_eval    = make_float3(FETCH_NOTEX(dParticleDataSorted, veleval
        , index));
27
28    float3 sph_force   = make_float3(FETCH_NOTEX(dParticleDataSorted,
        sph_force, index));
29    float  sph_pressure = FETCH_NOTEX(dParticleDataSorted, pressure, index);
30    //float  sph_density  = FETCH_NOTEX(dParticleDataSorted, density, index)
        ;
31
32  //  if(pos.x < cGridParams.grid_max.x/3 && pos.z < cGridParams.grid_max.z
      /3)
33  //  {
34  //    // negate gravity
35  //    //external_force.y += 9.8f;
36  //
37  //    external_force.x += 3.f;
38  //    external_force.y += 12.f;
39  //    external_force.z += 2.f;
40  //  }
41
42    float3 external_force = make_float3(0,0,0);
43    // add gravity
44    external_force.y -= 9.8f;
45
46
47    // add no-penetration force due to terrain
48    if(terrainCollisions)
49      external_force += calculateTerrainNoPenetrationForce(
50        pos, vel_eval,
51        fluidWorldPosition, dTerrainData,
52        cFluidParams.boundary_distance,
53        cFluidParams.boundary_stiffness,
54        cFluidParams.boundary_dampening,
55        cFluidParams.scale_to_simulation);
56
57  //  // add no-slip force due to terrain..
58    if(terrainCollisions)
59      external_force += calculateTerrainFrictionForce(
60        pos, vel_eval, sph_force+external_force,
```

```
61        fluidWorldPosition, dTerrainData,
62        cFluidParams.boundary_distance,
63        cFluidParams.friction_kinetic/delta_time,
64        cFluidParams.friction_static_limit,
65        cFluidParams.scale_to_simulation);
66
67
68    // add no-penetration force due to "walls"
69    if(gridWallCollisions)
70      external_force += calculateWallsNoPenetrationForce(
71        pos, vel_eval,
72        cGridParams.grid_min,
73        cGridParams.grid_max,
74        cFluidParams.boundary_distance,
75        cFluidParams.boundary_stiffness,
76        cFluidParams.boundary_dampening,
77        cFluidParams.scale_to_simulation);
78
79    // add no-slip force due to "walls"
80    if(gridWallCollisions)
81      external_force += calculateWallsNoSlipForce(
82        pos, vel_eval, sph_force + external_force,
83        cGridParams.grid_min,
84        cGridParams.grid_max,
85        cFluidParams.boundary_distance,
86        cFluidParams.friction_kinetic/delta_time,
87        cFluidParams.friction_static_limit,
88        cFluidParams.scale_to_simulation);
89
90    float3 force = sph_force + external_force;
91
92    // limit velocity
93    float speed = length(force);
94    if (speed > cFluidParams.velocity_limit ) {
95      force *= cFluidParams.velocity_limit / speed;
96    }
97
98    // Leapfrog integration
99    // v(t+1/2) = v(t-1/2) + a(t) dt
100   float3 vnext = vel + force * delta_time;
101   // v(t+1) = [v(t-1/2) + v(t+1/2)] * 0.5
102   vel_eval = (vel + vnext) * 0.5;
103   vel = vnext;
104
105   // update position of particle
106   pos += vnext * (delta_time / cFluidParams.scale_to_simulation);
107
108   if(progress)
109   {
110     uint originalIndex = dGridData.sort_indexes[index];
111
112     // writeback to unsorted buffer
113     dParticleData.position[originalIndex] = make_vec(pos);
114     dParticleData.velocity[originalIndex] = make_vec(vel);
115     dParticleData.veleval[originalIndex]  = make_vec(vel_eval);
116
```

```
117     float3 color = CalculateColor(coloringGradient, coloringSource, vnext,
          sph_pressure, sph_force);
118     dParticleData.color[originalIndex]  = make_float4(color, 1);
119   }
120 }
121
122 #endif
```

B.8 Complex SPH Implementation Code

```
1  #ifndef __K_SnowSPH_Density_cu__
2  #define __K_SnowSPH_Density_cu__
3
4  #include "K_UniformGrid_Utils.cu"
5  #include "K_SPH_Kernels.cu"
6  #include "K_SPH_Common.cu"
7
8  class Step1
9  {
10 public:
11
12   struct Data
13   {
14     float sum_density;
15
16     SnowSPHData dParticleDataSorted;
17   };
18
19   class Calc
20   {
21   public:
22
23     static __device__ void PreCalc(Data &data, uint const &index_i)
24     {
25       // read particle data from sorted arrays
26       data.sum_density = 0;
27     }
28
29     static __device__ void ForNeighbor(Data &data, uint const &index_i,
         uint const &index_j, float3 const &r, float const& rlen, float
         const &rlen_sq)
30     {
31       // the density sum using Wpoly6 kernel
32       data.sum_density += SPH_Kernels::Wpoly6::Kernel_Variable(
           cPrecalcParams.smoothing_length_pow2, r, rlen_sq);
33
34       //data.sum_density += SPH_Kernels::Wcubic::Kernel(cPrecalcParams.
           smoothing_length_pow2, r, rlen_sq);
35     }
36
37     static __device__ void PostCalc(Data &data, uint index_i)
38     {
39       data.sum_density *= cFluidParams.particle_mass * cPrecalcParams.
           kernel_poly6_coeff;
```

```
40        //data.sum_density *= cFluidParams.particle_mass;
41
42        // Compute the density field at the current particle,
43        // Calculate the W smoothing function for this particle, mass and
              the poly6_grad_coeff has been moved outside the sum because they
               are constant.
44        //float density = max(1.0, data.sum_density);
45        data.dParticleDataSorted.density[index_i]= data.sum_density;
46
47        // ideal gas equation of state (by Desbrun and Cani in "Smoothed
              particles: A new paradigm for animating highly deformable bodies
              ")
48        data.dParticleDataSorted.pressure[index_i] = cFluidParams.
              rest_pressure + cFluidParams.gas_stiffness * (data.sum_density -
               cFluidParams.rest_density);
49      }
50
51    };
52 };
53
54
55 __global__ void K_SumStep1(uint      numParticles,
56                NeighborList    dNeighborList,
57                SnowSPHData     dParticleDataSorted,
58                GridData const dGridData
59                )
60 {
61    // particle index
62    uint index = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
63    if (index >= numParticles) return;
64
65    Step1::Data data;
66    data.dParticleDataSorted = dParticleDataSorted;
67
68    float3 position_i = make_float3(FETCH(dParticleDataSorted, position,
          index));
69
70    // Do calculations on particles in neighboring cells
71 #ifdef SPHSIMLIB_USE_NEIGHBORLIST
72    UniformGridUtils::IterateParticlesInNearbyCells<SPHNeighborCalc<Step1::
          Calc, Step1::Data>, Step1::Data>(data, index, position_i,
          dNeighborList);
73 #else
74    UniformGridUtils::IterateParticlesInNearbyCells<SPHNeighborCalc<Step1::
          Calc, Step1::Data>, Step1::Data>(data, index, position_i, dGridData)
          ;
75 #endif
76
77 }
78
79 #endif
```

```
1 #ifndef __K_SnowSPH_Force_cu__
2 #define __K_SnowSPH_Force_cu__
3
4 #include "K_UniformGrid_Utils.cu"
5 #include "K_SPH_Kernels.cu"
```

```
 6 #include "K_SPH_Common.cu"
 7 #include "K_Common.cuh"
 8
 9 class Step2
10 {
11 public:
12   struct Data
13   {
14     float density_i;
15     float3 veleval_i;
16     matrix3 sum_velocity_tensor;
17
18     SnowSPHData dParticleDataSorted;
19   };
20
21   class Calc
22   {
23   public:
24     // this is called before the loop over each neighbor particle
25     static __device__ void PreCalc(Data &data, uint index_i)
26     {
27       // read particle data from sorted arrays
28       data.density_i  = FETCH(data.dParticleDataSorted, density, index_i);
29       data.veleval_i  = FETCH_FLOAT3(data.dParticleDataSorted, veleval,
             index_i);
30       data.sum_velocity_tensor = make_matrix3(0,0,0,0,0,0,0,0,0);
31     }
32
33     static __device__ void ForNeighbor(Data &data, uint const &index_i,
           uint const &index_j, float3 const &r, float const& rlen, float
           const &rlen_sq)
34     {
35       float density_j = FETCH(data.dParticleDataSorted, density, index_j);
36       float3 veleval_j = FETCH_FLOAT3(data.dParticleDataSorted, veleval,
             index_j);
37
38       float3 gradW = SPH_Kernels::Wcubic::Gradient(cFluidParams.
             smoothing_length, cPrecalcParams.smoothing_length_pow2,
             cPrecalcParams.smoothing_length_pow3, cPrecalcParams.
             smoothing_length_pow4,   r, rlen, rlen_sq);
39
40       // calculate the velocity tensor sum
41       data.sum_velocity_tensor += outer(
42         (veleval_j  - data.veleval_i)/(density_j)
43         , gradW
44         );
45     }
46
47
48     // this is called after the loop over each particle in a cell
49     static __device__ void PostCalc(Data &data, uint index_i)
50     {
51       //data.sum_velocity_tensor *= SPH_Kernels::Wviscosity::
             Gradient_Constant(cFluidParams.smoothing_length);
52
53       // velocity tensor derivative (DELv)
```

```
54        matrix3 velocity_tensor_i = cFluidParams.particle_mass * data.
              sum_velocity_tensor;
55
56        // rate-of-deformation/rate-of-strain tensor (E on wiki(NSE), D in
              viscoplastic paper)
57        matrix3 deformation_tensor_i = 0.5*(velocity_tensor_i + transpose(
              velocity_tensor_i));
58        //matrix3 deformation_tensor_i = (velocity_tensor_i + transpose(
              velocity_tensor_i));
59
60        // from "Particle-based viscoplastic fluid/solid simulation"
61        float t = trace(deformation_tensor_i);
62        float deformation_amount = sqrtf(t*t);
63
64        // stress tensor
65        matrix3 stress_tensor;// = make_matrix3(0,0,0,0,0,0,0,0,0);
66
67        //viscoplastic fluid (exp-power model w/jump number for melting
              stuff.. e.g. lava)
68        //float n = 0.5f;
69        //float J = 10;
70        //stress_tensor = (1-__expf(-(J+1)*deformation_amount))*(pow(
              deformation_amount, n-1.0f)+1/deformation_amount)*
              deformation_amount;
71
72        // newtonian fluid
73        // 3-step says: ( t = 2*ţ*D )
74        //stress_tensor = 1*deformation_amount*deformation_tensor_i;
75
76        // non-newtonian POWER-LAW fluid
77  //      float n = 3.2;
78  //      float K = 1.74;
79  //      float viscosity = K*pow(deformation_amount,n-1.0f);
80  //      viscosity = clamp(viscosity, 1.0f,300.0f);
81  //        stress_tensor = viscosity*deformation_tensor_i;
82
83        // non-newtonian BINGHAM fluid
84  //      float K = 10;
85  //        float yield_stress = 1.5;
86  //        stress_tensor = yield_stress + K * deformation_tensor_i;
87  //      float s = trace(stress_tensor);
88  //      float stress_amount = sqrtf(s*s);
89  //        if(stress_amount <= yield_stress)
90  //        {
91  //        stress_tensor = 500*deformation_amount*deformation_tensor_i;
92  //        }
93
94
95        // non-newtonian HERSCHEL-BULKLEY fluid
96  //        float K = 1;
97  //        float yield_stress = 1.5;
98  //        float n = 1.74;
99  //        stress_tensor = (K*pow(deformation_amount,n-1.0f) +yield_stress/
        deformation_amount)*deformation_tensor_i;
100 //        float stress_amount = trace(stress_tensor)/3.0f;
101 //        if(stress_amount < yield_stress) {
102 //        stress_tensor = 500*deformation_amount*deformation_tensor_i;
```

```
103 //        }
104
105      // non-newtonian cross fluid
106        float K = 2.1f;
107      float visco_inf= 1.07;
108      float visco_zero = 300;
109      float n = 1.0f;
110      float viscosity = visco_inf+(visco_zero-visco_inf)/(1+K*pow(
             deformation_amount, n));
111      viscosity = clamp(viscosity, 1.0f,300.0f);
112      stress_tensor = viscosity*deformation_tensor_i;
113
114      // store stress tensor
115      data.dParticleDataSorted.stress_tensor[index_i] = stress_tensor;
116
117      //data.dParticleDataSorted.color[index_i] = make_vec(
             deformation_amount,0.3,0.3);
118    }
119  };
120 };
121
122 __global__ void K_SumStep2(uint     numParticles,
123             NeighborList  dNeighborList,
124             SnowSPHData dParticleDataSorted,
125             GridData    dGridData
126             )
127 {
128   // particle index
129   uint index = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
130   if (index >= numParticles) return;
131
132   Step2::Data data;
133
134   data.dParticleDataSorted = dParticleDataSorted;
135
136   float3 position_i = FETCH_FLOAT3(data.dParticleDataSorted, position,
        index);
137
138   // Do calculations on particles in neighboring cells
139 #ifdef SPHSIMLIB_USE_NEIGHBORLIST
140   UniformGridUtils::IterateParticlesInNearbyCells<SPHNeighborCalc<Step2::
        Calc, Step2::Data>, Step2::Data>(data, index, position_i,
        dNeighborList);
141 #else
142   UniformGridUtils::IterateParticlesInNearbyCells
143     <
144     SPHNeighborCalc
145       <Step2::Calc, Step2::Data>
146       ,
147       Step2::Data
148     >
149   (data, index, position_i, dGridData);
150 #endif
151 }
152
153
154 #endif
```

```
1  #ifndef __K_SnowSPH_Step3_cu__
2  #define __K_SnowSPH_Step3_cu__
3
4  #include "K_UniformGrid_Utils.cu"
5  #include "K_SPH_Kernels.cu"
6  #include "K_SPH_Common.cu"
7
8  class Step3
9  {
10 public:
11
12   struct Data
13   {
14     float3 veleval_i;
15     float density_i;
16     float pressure_i;
17     matrix3 stress_tensor_i;
18
19     float3 veleval_j;
20     float density_j;
21     float pressure_j;
22     matrix3 stress_tensor_j;
23
24
25     float3 f_viscosity;
26     float3 f_pressure;
27     float3 f_stress;
28     float3 f_xsph;
29
30
31     SnowSPHData dParticleDataSorted;
32   };
33
34   class Calc
35   {
36   public:
37
38     // this is called before the loop over each neighbor particle
39     static __device__ void PreCalc(Data &data, uint index_i)
40     {
41       // read particle data from sorted arrays
42       data.veleval_i  = FETCH_FLOAT3(data.dParticleDataSorted, veleval,
             index_i);
43       data.density_i  = FETCH(data.dParticleDataSorted, density, index_i);
44       data.pressure_i = FETCH(data.dParticleDataSorted, pressure, index_i)
             ;
45       data.stress_tensor_i  = FETCH_MATRIX3(data.dParticleDataSorted,
             stress_tensor, index_i);
46
47       data.f_pressure   = make_float3(0,0,0);
48       data.f_viscosity  = make_float3(0,0,0);
49       data.f_stress    = make_float3(0,0,0);
50       data.f_xsph      = make_float3(0,0,0);
51     }
52
53     static __device__ void ForNeighbor(Data &data, uint const &index_i,
         uint const &index_j, float3 const &r, float const& rlen, float
```

```
        const &rlen_sq)
54    {
55      data.veleval_j  = FETCH_FLOAT3(data.dParticleDataSorted, veleval,
            index_j);
56      data.density_j  = FETCH(data.dParticleDataSorted, density,  index_j)
            ;
57      data.pressure_j = FETCH(data.dParticleDataSorted, pressure,  index_j
            );
58      data.stress_tensor_j  = FETCH_MATRIX3(data.dParticleDataSorted,
            stress_tensor, index_j);
59
60      // XSPH velocity correction, Monaghan JCP 2000
61      data.f_xsph += ( (data.veleval_j - data.veleval_i) / (data.density_i
            +data.density_j) ) *  SPH_Kernels::Wpoly6::Kernel_Variable(
            cPrecalcParams.smoothing_length_pow2, r, rlen_sq);
62
63      //from "Particle-based viscoplastic fluid/solid simulation", also
            see "SPH survival kit"
64      data.f_pressure  += ( (data.pressure_i/(data.density_i*data.
            density_i)) + (data.pressure_j/(data.density_j*data.density_j))
            ) * SPH_Kernels::Wspiky::Gradient_Variable(cFluidParams.
            smoothing_length, r, rlen);
65
66      // viscosity from mueller paper : f_viscosity = (ţ/rho_i)SUM(m_j * (
            v_j-v_i)/(rho_j)DEL^2Wvis
67      // we move the mass and the Wvis constants to precalc
68      data.f_viscosity += ( (data.veleval_j  - data. veleval_i ) / (data.
            density_j * data.density_i) ) * SPH_Kernels::Wviscosity::
            Laplace_Variable(cFluidParams.smoothing_length, r, rlen);
69
70      // stress force calculation
71      data.f_stress += dot(
72        (data.stress_tensor_i+data.stress_tensor_j)/(data.density_j)
73        , SPH_Kernels::Wcubic::Gradient(cFluidParams.smoothing_length,
              cPrecalcParams.smoothing_length_pow2,  cPrecalcParams.
              smoothing_length_pow3, cPrecalcParams.smoothing_length_pow4,
                r, rlen, rlen_sq)
74        );
75    }
76
77    // this is called after the loop over each particle in a cell
78    static __device__ void PostCalc(Data &data, uint index_i)
79    {
80      //data.f_stress *= SPH_Kernels::Wviscosity::Gradient_Constant(
            cFluidParams.smoothing_length);
81      data.f_stress   *= (cFluidParams.particle_mass/data.density_i);
82
83      // Calculate the forces, the particle_mass/constants are added here
            because there is no need for it to be inside the sum loop.
84      data.f_pressure   *= cFluidParams.particle_mass * cPrecalcParams.
            kernel_pressure_precalc;
85      data.f_viscosity  *= cFluidParams.particle_mass * cPrecalcParams.
            kernel_viscosity_precalc;
86      data.f_xsph      *= 2 * cFluidParams.particle_mass;
87
88      //data.dParticleDataSorted.color[index_i] = make_vec(data.
            stress_tensor_i.r1.x, data.stress_tensor_i.r2.x, data.
```

```
            stress_tensor_i.r3.x);
89        //data.dParticleDataSorted.color[index_i] = make_vec(data.f_stress.x
            , data.f_stress.y, data.f_stress.z);
90        //data.dParticleDataSorted.color[index_i] = make_vec(1,1,1);
91
92        // store xsph val
93        data.dParticleDataSorted.xsph[index_i] = make_vec(data.f_xsph);
94
95        float3 sph_force =  (
96          data.f_pressure
97          + data.f_stress
98          + data.f_viscosity
99          );
100
101       data.dParticleDataSorted.sph_force[index_i] = make_vec(sph_force);
102     }
103   };
104 };
105
106
107
108 __global__ void K_SumStep3(uint      numParticles,
109             NeighborList  dNeighborList,
110             SnowSPHData dParticleDataSorted,
111             GridData    dGridData
112             )
113 {
114   // particle index
115   uint index = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
116   if (index >= numParticles) return;
117
118   Step3::Data data;
119
120   data.dParticleDataSorted = dParticleDataSorted;
121
122   float3 position_i = FETCH_FLOAT3(data.dParticleDataSorted, position,
        index);
123
124   // Do calculations on particles in neighboring cells
125 #ifdef SPHSIMLIB_USE_NEIGHBORLIST
126   UniformGridUtils::IterateParticlesInNearbyCells<SPHNeighborCalc<Step3::
        Calc, Step3::Data>, Step3::Data>(data, index, position_i,
        dNeighborList);
127 #else
128   UniformGridUtils::IterateParticlesInNearbyCells<SPHNeighborCalc<Step3::
        Calc, Step3::Data>, Step3::Data>(data, index, position_i, dGridData)
        ;
129 #endif
130 }
131
132 #endif
```

```
1 #ifndef __K_SnowSPH_Integrate_cu__
2 #define __K_SnowSPH_Integrate_cu__
3
4 #include "K_Coloring.cu"
5 #include "K_Boundaries_Terrain.cu"
```

```
6  #include "K_Boundaries_Walls.cu"
7
8  template<SPHColoringSource coloringSource, ColoringGradient
       coloringGradient>
9  __global__ void K_Integrate(int      numParticles,
10                bool     gridWallCollisions,
11                bool     terrainCollisions,
12                float      delta_time,
13                bool     progress,
14                GridData   dGridData,
15                SnowSPHData  dParticleData,
16                SnowSPHData  dParticleDataSorted,
17                float3     fluidWorldPosition,
18                TerrainData  dTerrainData
19                //,float*      dCFL
20                )
21 {
22   int index = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
23   if (index >= numParticles) return;
24
25   float3 pos       = make_float3(FETCH_NOTEX(dParticleDataSorted, position,
          index));
26   float3 vel       = make_float3(FETCH_NOTEX(dParticleDataSorted, velocity,
          index));
27   float3 vel_eval    = make_float3(FETCH_NOTEX(dParticleDataSorted, veleval
        , index));
28   float3 xsph      = make_float3(FETCH_NOTEX(dParticleDataSorted, xsph,
        index));
29
30   float3 sph_force   = make_float3(FETCH_NOTEX(dParticleDataSorted,
        sph_force, index));
31   float   sph_pressure = FETCH_NOTEX(dParticleDataSorted, pressure, index);
32   //float   sph_density   = FETCH_NOTEX(dParticleDataSorted, density, index)
        ;
33
34   float3 external_force = make_float3(0,0,0);
35
36   // add gravity
37   external_force.y -= 9.8f;
38
39   // add no-penetration force due to terrain
40   if(terrainCollisions)
41     external_force += calculateTerrainNoPenetrationForce(
42     pos, vel_eval,
43     fluidWorldPosition, dTerrainData,
44     cFluidParams.boundary_distance,
45     cFluidParams.boundary_stiffness,
46     cFluidParams.boundary_dampening,
47     cFluidParams.scale_to_simulation);
48
49   //  // add no-slip force due to terrain..
50   if(terrainCollisions)
51     external_force += calculateTerrainFrictionForce(
52     pos, vel_eval, sph_force+external_force,
53     fluidWorldPosition, dTerrainData,
54     cFluidParams.boundary_distance,
55     cFluidParams.friction_kinetic/delta_time,
```

```
56      cFluidParams.friction_static_limit,
57      cFluidParams.scale_to_simulation);
58
59    // add no-penetration force due to "walls"
60    if(gridWallCollisions)
61      external_force += calculateWallsNoPenetrationForce(
62      pos, vel_eval,
63      cGridParams.grid_min,
64      cGridParams.grid_max,
65      cFluidParams.boundary_distance,
66      cFluidParams.boundary_stiffness,
67      cFluidParams.boundary_dampening,
68      cFluidParams.scale_to_simulation);
69
70
71    // add no-slip force due to "walls"
72    if(gridWallCollisions)
73      external_force += calculateWallsNoSlipForce(
74      pos, vel_eval, sph_force + external_force,
75      cGridParams.grid_min,
76      cGridParams.grid_max,
77      cFluidParams.boundary_distance,
78      cFluidParams.friction_kinetic/delta_time,
79      cFluidParams.friction_static_limit,
80      cFluidParams.scale_to_simulation);
81
82
83    float3 force = sph_force + external_force;
84
85    // limit velocity
86    float speed = length(force);
87    if (speed > cFluidParams.velocity_limit ) {
88      force *= cFluidParams.velocity_limit / speed;
89    }
90
91    // Leapfrog integration
92    // v(t+1/2) = v(t-1/2) + a(t)*dt
93    float3 vnext = (vel) + force * delta_time;
94
95    // xsph
96    vnext += cFluidParams.xsph_factor * xsph;
97
98    // Leapfrog integration
99    // v(t+1) = [v(t-1/2) + v(t+1/2)] * 0.5
100   vel_eval = (vel + vnext) * 0.5;
101   vel = vnext;
102
103   // update position of particle
104   pos += (vnext) * (delta_time / cFluidParams.scale_to_simulation);
105
106
107   // Calculate CFL val
108   //dCFL[index] = length(vel_eval) + sqrt(cFluidParams.gas_stiffness);
109
110   if(progress)
111   {
112     uint originalIndex = dGridData.sort_indexes[index];
```

```
113
114     // writeback to unsorted buffer
115     dParticleData.position[originalIndex] = make_vec(pos);
116     dParticleData.velocity[originalIndex] = make_vec(vel);
117     dParticleData.veleval[originalIndex]  = make_vec(vel_eval);
118
119     float3 color = CalculateColor(coloringGradient, coloringSource, vnext,
            sph_pressure, sph_force);
120     dParticleData.color[originalIndex]  = make_float4(color, 1);
121   }
122
123 }
124
125 #endif
```

Appendix C

Extended abstract for paper submitted to PARA 2010

Fast GPU-based Fluid Simulations Using SPH

Smoothed Particle Hydrodynamics (SPH) on Graphics Processing Units (GPUs)

Øystein E. Krog*[1] and Anne C. Elster†[1]

[1]*Dept. of Computer and Information Science, Norwegian University of Science and Technology (NTNU)*

Abstract Graphical Processing Units (GPUs) are massive floating-point stream processors, and through the recent development of tools such as CUDA and OpenCL it has become possible to fully utilize the bandwidth and computational power they contain. A computationally challenging problem is how to model movements of liquids. We have developed a GPU-based framework for 3-dimensional Computational Fluid Dynamics (CFD) using Smoothed Particle Hydrodynamics (SPH). This paper describes the methods used for implementing fast simulations of fluids dynamics using GPUs, and compares the performance of the implementation to previous SPH implementations. Our implementation uses the acceleration data structures found in the NVIDIA "Particles" demo, but implements SPH instead of its simpler mass-spring system. The implementation uses CUDA and has been highly optimized to the point where a scaled simulation can run in "real-time". We implement two different SPH models, a simplified model for newtonian fluids, and a complex model for non-newtonian fluids. The complex SPH model is used to simulate flowing snow avalanches. Using our simple SPH model and a NVIDIA GeForce 260 GTX we achieve 38 FPS with 256K particles, 63 FPS with 128K particles and 99 FPS with 64K particles. Open source code will be provided. This should make our work very useful not only for our current work on simulating snow avalanches, but also for other CFD applications that need faster simulations of many particles.

Keywords GPU, CFD, SPH, GPGPU, CUDA, Fluid

1 Introduction

The simulation of fluids is an interesting problem due to the importance of fluids in the physical world. Simulating fluids also present a large challenge due to the large computational demands that arise from the complex behaviors of fluids, especially in 3 dimensions. Due to these demands most fluid simulations are not done in "real-time", in fact many previous SPH implementations have been constrained to 2D. Our framework makes it possible to do SPH simulations in 3D for large problem sizes in "real-time". So far it has not been possible to achieve beliveable "real-time" fluid simulations for all but the smallest and coarsest models. By using an accelerator such as the GPU, the number of particles modeled in the simulation can be increased considerably, and the overall simulation speed is greatly increased compared to CPU-based simulations.

*Email: `oystein.krog@gmail.com`

†Email: `elster@idi.ntnu.no`

2 Computational Fluid Dynamics

Computational Fluid Dynamics (CFD) is a large field, in which the Navier-Stokes equations play an important role. These equations can be solved numerically, and several such methods have been developed in order to simulate fluids on computers.

2.1 Navier-Stokes Equations

The Navier-Stokes equations in simplified Lagrangian form consist of mass and momentum conservation:

$$\frac{d\rho}{dt} = -\rho \nabla \cdot \mathbf{v} \tag{1}$$

$$\frac{d\mathbf{v}}{dt} = -\frac{1}{\rho}\nabla p + \frac{1}{\rho}\nabla \cdot \mathbf{S} + \mathbf{f} \tag{2}$$

Where $\mathbf{v}$ is the velocity field, ρ the density field, ∇p the pressure gradient field resulting from isotropic stress, $\nabla \cdot \mathbf{S}$ the stress tensor resulting from deviatoric stress and $\mathbf{f}$ an external force field such as gravity.

For incompressible newtonian fluids the momentum conservation reduces to:

$$\frac{d\mathbf{v}}{dt} = -\frac{1}{\rho}\nabla p + \frac{\mu}{\rho}\nabla^2 \mathbf{v} + \mathbf{f} \tag{3}$$

Where the term μ is the dynamic viscosity of the fluid.

2.2 Smoothed Particle Hydrodynamics

In SPH the different effects of Navier-Stokes are simulated by a set of forces that act on each particle. These forces are given by scalar quantities that are interpolated at a position $\mathbf{r}$ by a weighted sum of contributions from all surrounding particles within a cutoff distance h in the space Ω. In integral form this can be expressed as follows [5]:

$$A_i(\mathbf{r}) = \int_\Omega A(\mathbf{r}')W(\mathbf{r} - \mathbf{r}', h)d\mathbf{r}' \tag{4}$$

The numerical equivalent is obtained by approximating the integral interpolant by a summation interpolant [5]:

$$A_i(\mathbf{r}_i) = \sum_j A_j \frac{m_j}{\rho_j} W(\mathbf{r}_{ij}, h) \tag{5}$$

where j iterates over all particles, m_j is the mass of particle j, $\mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j$ where $\mathbf{r}$ is the position, ρ_j the density and A_j the scalar quantity at position $\mathbf{r}_j$.

For a more comprehensive introduction to SPH, please refer to [5].

2.3 Avalanche SPH

Snow avalanches wary greatly in behaviour, from powder-snow avalanches to so called dense-flow, or flowing snow avalanches. Snow avalanches usually appear as a viscous flow down a slope, and it is this obvious property which has prompted the use of fluid dynamics in avalanche simulation [1]. Several viscosity models exist for modelling non-newtonian fluids, and rheological parameters have been collected for flowing snow [6]. Many SPH models exist for viscoplastic fluids, from melting objects [8] to lava flows [9] and generalized rheological models [4].

3 Methods and Implementation

CUDA is a parallel computing architecture developed by NVIDIA. We use CUDA for C, which is basically the C language with some added syntax. GPUs are massively parallel, with several thousand threads available. We parallelize the calculation of SPH by assigning a thread to each particle in the simulation. Each thread is then responsible for calculating the SPH sums over the surrounding particles. When accessing memory on the GPU, *coalesced* (correctly structured) access is very important. Due to the nature of the algorithm, fully coalesced access is unfortunately not possible. By utilizing the texture cache this problem is greatly alleviated.

3.1 Nearest-Neighbor Search

The summation term in the SPH-formulation is computationally heavy, it requires looking at many nearby particles and computing interactions between them. To avoid a naive brute-force $O(N^2)$ search for neighbors, a nearest-neighbor search algorithm is commonly used, such as a linked list or a uniform grid. We use the algorithm presented in [2], which can be summarized as follows:

1. Divide the simulation domain into a uniform grid.
2. Use to the spatial position of each particle to find the cell it belongs to.
3. Use the particle cell position as input to a hash function (a spatial hash)
4. Sort the particle according to their spatial hash.
5. Reorder the particles in a linear buffer according to their hash value.

Particles in the same cell will then lie consecutively in the linear buffer, and finding ”neighbors” is simply a matter of iterating over the correct indices in the buffer. For the sorting we used the fastest radix sort available for the GPU at the time of implementation (by Satish *et al* [10]).

3.2 Non-Newtonian Fluids

non-newtonian fluids differ from newtonian fluids in that their viscosity is not constant. In a newtonian fluid the relation between shear stress and the strain rate is linear, with the constant of proportionality being the viscosity. For a non-newtonian fluid the relation is nonlinear and can even be time-dependent. There exist many classes of non-newtonian fluids, and many types of models, of which we implement several. The complex SPH model differs primarily in that it includes the much more complex stress calculation presented in [4].

3.3 SPH Models

We use our framework to implement two different SPH models, a simplified model for interactive use, based on a model by Müller *et al.*[7] and a complex model for non-newtonian fluids based on a model by Hosseini *et al.*[4]. The simplified model is focused on interactive performance and creates a visually pleasing water-like fluid, and the complex model is used to simulate flowing-snow avalanches through the use of different rheological(the study of the flow of matter) models.

By using the SPH formulation we end up with the following simulation equations:

$$\rho_i = \sum_j m_j W(\mathbf{r}_{ij}, h) \tag{6}$$

$$\mathbf{f}_i^{pressure} = -\frac{1}{\rho}\nabla p(\mathbf{r}_i) = \sum_{j\neq i} m_j\left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2}\right)\nabla W(\mathbf{r}_{ij}, h) \tag{7}$$

The incompressible fluid is simulated as a weakly compressible fluid where the incompressibility constraint is applied to the pressure p by using an equation of state given by the ideal gas law with an added rest density: $p = k(\rho - \rho_0)$

For calculating non-newtonian fluids we use the formulation presented by [4].

$$\nabla \mathbf{v}_i = \sum_j \frac{m_j}{\rho_j}(\mathbf{v_i} - \mathbf{v_j}) \otimes \nabla W(\mathbf{r}_{ij}, h) \tag{8}$$

Where $\otimes$ is the outer product and $\nabla\mathbf{v}$ is a tensor field (a 3x3 matrix). Giving us $\mathbf{D}_i = \frac{1}{2}(\nabla\mathbf{v} + (\nabla\mathbf{v})^T)$, and the stress tensor $\mathbf{S}_i = \eta(D)\mathbf{D}$, with $D = \sqrt{\frac{1}{2}trace(\mathbf{D})^2}$. The term $\eta(D)$ is essentially the viscosity for the fluid, which for a newtonian is constant. By using a rheological model for calculating this term, various non-newtonian fluids can be simulated. Finally the stress tensor is calculated:

$$\mathbf{f}_i^{stress} = \frac{1}{\rho}\nabla\cdot\mathbf{S}_i = \sum_{j\neq i}\frac{m_j}{\rho_i\rho_j}(\mathbf{S}_i + \mathbf{S}_j)\cdot\nabla W(\mathbf{r}_{ij}, h) \tag{9}$$

Thus we have that the acceleration for a particle is given by:

$$\mathbf{a}_i = f_i^{pressure} + f_i^{stress} + f_i^{external} \tag{10}$$

3.4 SPH Algorithm

Due to the data dependencies between the various force, some of them must be calculated separately. Each calculation step is essentially a summation over neighboring particles, and we combine the force calculations using loop

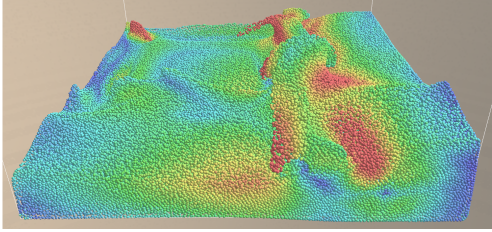


Figure 1: A screenshot of the simple SPH model with 256K particles. Hue-based gradient shading for the velocity of the particles.

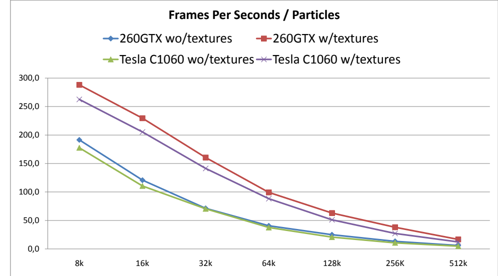


Figure 2: Performance scaling for the simple SPH model.

fusion as far as it is possible. For the complex SPH models we end up with the following steps:

1. Update the hashed, radix sorted, uniform grid.
2. Calculate the SPH density
3. Calculate the SPH velocity tensor
4. Calculate the SPH pressure and SPH stress tensor
5. Integrate in time.

For the simplified SPH model, the stress tensor is replaced with a simplified viscosity approximation which ignores deviatoric stress, since it is not strictly nescessary for incompressible newtonian fluids. As a result the viscosity calculation does not need the velocity tensor, step 3 can be dropped, and the viscosity force can be computed together with the pressure in step 4. For integrating the velocity change we use the Leap-Frog method, since it is more accurate than simple Euler integration while still mainting low memory and computational costs.

3.5 Simulation Framework

The implementation of the simulation framework is coded in C++ and the GPU code using C for Cuda. The simulation itself is separated logically into a separate library. Since the entire simulation is done on the GPU, an API for integration of rendering is nescessary to avoid the high cost of copying rendering buffers to the CPU. Using the API it is possible to implement direct rendering of simulation buffers for both Direct3D and OpenGL. For the case of OpenGL this is possible using Vertex Buffer Objects (VBOs), which can be rendered directly using shaders with fairly low overhead. The CUDA simulation kernels have been highly optimized. By manually optimizing register usage, reording memory accesses and optimizing the block sizes for the CUDA code, performance gains as large as 40% were realized.

4 Results

For performance testing two different GPUs were used, a Geforce 260GTX and a Tesla C1060. The computer had a Intel Core2 Quad Q9550, 4GB DDR3 ram and was running Windows Vista 64-bit .

4.1 Choice of Metrics

Comparing and evaluating the performance of the simulations is difficult due to the large amount of parameters and their effect on performance. In addition it is hard to compare to other SPH implementations due to different SPH models and paramaters. For the simple SPH model we compare against Müller and Harada which use a very similar SPH model, using rest density selected to simulate water, with the dynamic viscosity set to 1. For the complex SPH model we have not completed our performance evaluation, but compare against the implementation of the simple SPH model. An additional difficulty with the non-newtonian model is that the viscosity is not constant, but varies according to stress, which means that it is hard to give exact performance numbers. For the purposes of measuring performance, we choose the same parameters as the simple model, and a simplified rheology model with a (very high) constant viscosity of 300. To obtain absolute performance numbers we use a fairly simple simulation setup; a square simulation domain with simple repulsive forces as walls where a cubic volume of fluid is dropped into a shallow pool of water. The performance numbers were measured when the fluid had reached a stable equilibrium. We use FPS as our performance unit, because our simulation is frame-locked with the rendering.

Memory Usage Due to the hashed uniform grid structure the memory usage is fairly sparse. Had the uniform grid been allocated directly the usage would be much larger. The memory usage in bytes is $176N$ where N is the number of particles. The memory usage of the complex SPH models is $212N$ bytes. This means that it is possible to simulate very large systems even on commodity hardware. We have tested systems up to 2048K particles on a NVIDIA Geforce 260GTX.

4.2 Performance

Müller *et al.*[7] achieves 20 FPS at 2200 particles on a 1.8 GHz Pentium 4. Harada *et al.*[3] achieves 17 FPS at 60000 particles on an NVIDIA GeForce 8800GTX using OpenGL and CG.

Using our simple SPH model and a NVIDIA GeForce

260 GTX we achieve 38 FPS with 256K particles, 63 FPS with 128K particles and 99 FPS with 64K particles. On the NVIDIA Geforce 260GTX the complex SPH model runs at 17 FPS with 256K particles, 31 FPS at 128K particles and 55 FPS at 64K particles.

It is worth noting that direct comparison of FPS is not always a good measure of performance. An SPH implementation can have large variations in performance depending on the parameters of the system.

By changing only the number of particles and the particle mass (to keep the fluid volume constant), the scaling in Figure 2 is observed. Comparing the 260GTX and the Tesla C1060, the performance on the Tesla is somewhat lower, which is likely due to the lower memory clock on the Tesla. This finding is congruent with the theory that the SPH algorithm is highly bandwidth constrained on the GPU. This is due to imperfect coalescing of memory reads.

4.3 Real-Time Appearance

By scaling the simulation domain, and relaxing the accuracy requirements by selecting large timesteps, the fluid simulations produce belivable ”real-time” behavior. We have tested up to 2048K particles at which point the simulation can no longer be considered ”real-time”, but it is still very fast compared to previous implementations. In fact the CPU implementation by Müller *et al.*[7] achieves 20 FPS with 2200 particles, in contrast to our implementation on the GPU which achieves 17 FPS with 512K particles. It should be noted that this is not truly ”real-time” simulation of a volume of liquid, since both the fluid volume and the time is not to scale. Our complex SPH model is not as well suited to ”real-time” simulation due to the nescessity of a somewhat lower timestep in order to support higher viscosities.

5 Conclusions

In this paper, we presented an implementation of Smoothing Particle Hydrodynamics (SPH) on the GPU. Our implementation achieves very good performance since we take advantage of the massive amounts of parallelism available on modern GPUs, as well as use specialized acceleration data structures. Our SPH implementations achieve very good performance compared to previous implementations, and can be used to produce believable ”real-time” behavior.

Using our simple SPH model and a NVIDIA GeForce 260 GTX we achieve 38 FPS with 256K particles, 63 FPS with 128K particles and 99 FPS with 64K particles.

5.1 Current and Future Work

Simulations of snow can be used for everything from gaming to avalanche prediction. For games snow simulation can help create complex environments, which can lead to numerous possibilities for game-play mechanics. Predicting the behavior of snow avalanches can help prevent loss of both life and property.

The resource usage of our model has been investigated and it was found that it does not consume much memory but is very memory bandwidth intensive. To further increase performance it might be interesting to investigate the possibility of using multiple GPUs.

Finally, the visualization of the fluid model can be improved, at the moment a very simple method of direct particle rendering is used. By using a surface reconstruction model such as Marching-Cubes a real surface can be rendered. It is also possible to use screen-space surface rendering techniques to approximate the fluid surface without the large computation cost associated with true surface reconstruction.

References

[1] E. Bovet, B. Chiaia, and L. Preziosi. A new model for snow avalanche dynamics based on non-newtonian fluids. *Meccanica*.

[2] S. Green. Cuda particles. Technical report, NVIDIA.

[3] T. Harada, S. Koshizuka, and Y. Kawaguchi. Smoothed particle hydrodynamics on gpus. 2007.

[4] S. M. Hosseini, M. T. Manzari, and S. K. Hannani. A fully explicit three-step sph algorithm for simulation of non-newtonian fluid flow. *International Journal of Numerical Methods for Heat & Fluid Flow*, 17(7):715–735, 2007.

[5] M. Kelager. Lagrangian fluid dynamics using smoothed particle hydrodynamics. Master’s thesis, University of Copenhagen, Department of Computer Science, 2006.

[6] M. A. Kern, F. Tiefenbacher, and J. N. McElwaine. The rheology of snow in large chute flows. *Cold Regions Science and Technology*, 39(2-3):181 – 192, 2004.

[7] M. Müller, D. Charypar, and M. Gross. Particle-based fluid simulation for interactive applications. In *SCA ’03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 154–159, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[8] A. Paiva, F. Petronetto, T. Lewiner, and G. Tavares. Particle-based non-newtonian fluid animation for melting objects. *Computer Graphics and Image Processing, Brazilian Symposium on*, 0:78–85, 2006.

[9] A. Paiva, F. Petronetto, T. Lewiner, and G. Tavares. Particle-based viscoplastic fluid/solid simulation. *Computer-Aided Design*, 41(4):306 – 314, 2009. Point-based Computational Techniques.

[10] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore gpus. NVIDIA Technical Report NVR-2008-001, NVIDIA Corporation, Sept. 2008.

Appendix D

Poster

Fast GPU-based Fluid Simulations using Smoothed Particle Hydrodynamics (SPH)

Øystein E. Krog (Master Student) and Dr. Anne C. Elster (Advisor)

Dept. of Computer and Information Science, Norwegian University of Science and Technology (NTNU)

Our Work

We use Smoothed Particle Hydrodynamics (SPH) in combination with a hashed, radix sorted, uniform grid [1] to achieve high performance fluid simulations on the GPU. We have developed a framework for creating GPU-based particle systems which require Nearest Neighbor Searches (NNS) and SPH calculations. We use this framework to implement two different SPH models; a "simple" model and a "complex" model.

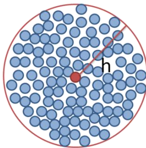
Optimizations

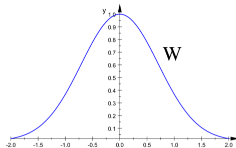
A large effort has been expended to ensure optimal performance of the simulations.
By optimizing register usage, occupancy, memory access patterns and the algorithms themselves, large gains in performance have been achieved.

Smoothed Particle Hydrodynamics

SPH is a Lagrangian interpolation method for approximating a solution to the Navier-Stokes Equations. Particles are affected by neighboring particles through a weighting function. For performance considerations the summation over neighboring particles is the most important.
Here the equation for the pressure force is shown:

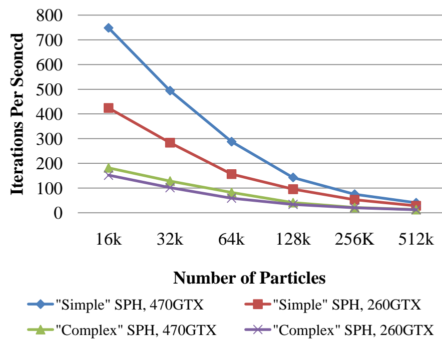
$$f^{pressure} \approx -\sum_{j=1}^{N} \frac{m_j}{\rho_j} \left(\frac{P_i}{\rho_j^2} + \frac{P_j}{\rho_j^2} \right) \nabla W(\mathbf{r}_i - \mathbf{r}_j, h)$$





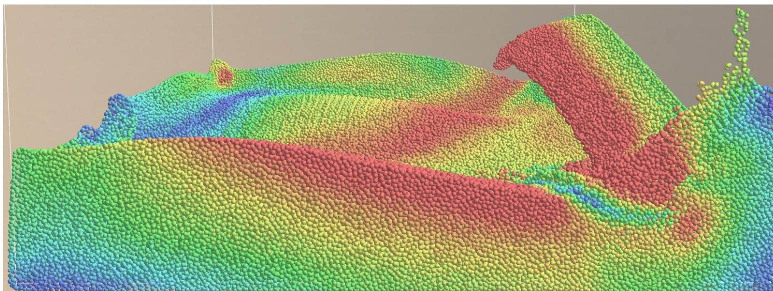
Results

We show the performance scaling of the system on two different GPUs; a NVIDIA Geforce 260GTX and a Geforce 470GTX (FERMI). The 470GTX shows much improved performance for the simple SPH model.
Compared to previous implementations of comparable SPH models on both CPUs and GPUs our implementation is significantly faster. A CPU implementation by Müller et al. [2] achieves 20 FPS with 2200 particles using a 1.8Ghz Pentium 4, which is not directly comparable but does provide some perspective.



Future Work

We are currently developing a real-time snow avalanche simulation which uses the complex SPH model to simulate avalanches as Non-Newtonian fluids.
A rendering technique for reconstructing the fluid surface is also something which should be investigated.



Here we show a screenshot of the simple SPH model with 256K particles running in a real-time rendering mode.
Real-time rendering imposes a small overhead, but the simulation is still interactive when running on the 470 GTX.

Acknowledgements

We would like to thank NVIDIA for their contribution of hardware through their professor afiliates program.

References

[1] Green *et al.* CUDA Particles
[2] Müeller *et al.* Particle-Based Fluid Simulation for Interactive Applications

HPC-Lab
NTNU
Norwegian University of Science and Technology