NTNU
Norwegian University of Science and Technology

Utilizing GPUs for Real-Time Visualization of Snow

Robin Eidissen

Master of Science in Computer Science
Submission date: February 2009
Supervisor: Anne Cathrine Elster, IDI

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

Problem Description

Animating realistic-looking snow is a complex and
computationally expensive task, and achieving it in real time an even further challenge.
Several subproblems need to be solved: A large number of snow particles have to be simulated, as well as the wind
velocity field that will affect them. Build-up of snow
on objects also require computationally expensive collision detection since it needs to be done for every snowflake.

This thesis will build upon work done previously[Saltvik06] where a parallel solution was implemented on a dual core computer.
The focus this time will be to utilize modern GPUs to achieve a more realistic and larger-scale simulation
compared to what was possible on just CPUs.

Techniques for solving the various subproblems across one
or several GPUs will be investigated using CUDA, NVIDIA´s
GPGPU compiler and set of development tools which enable
programming in a variation of C to code algorithms for
execution on the graphics processing unit (GPU).

Assignment given: 15. September 2008
Supervisor: Anne Cathrine Elster, IDI

NTNU
Norwegian University of
Science and Technology

Utilizing GPUs for Real-Time Visualization of Snow

Robin Eidissen

Master of Science in Computer Science
Submission date: February 2009
Supervisor: Anne Catherine Elster, IDI

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

Abstract

In this thesis, we present a realistic snow simulation, utilizing modern GPUs to achieve real-time performance. Our simulation of snowfall is a computationally expensive problem since each snowflake is simulated interacting with a dynamic windfield.

Three main components come together to form the complete snow simulation: The first is highly scalable particle simulation for simulating millions of individual snowflakes. The second is a CFD implementation for simulating wind phenomena which will affect the snow flakes. Finally, the third component is a terrain height map model for the geometry which lets us create a rich high resolution environment with which the snow and wind can interact, without introducing performance-limiting complexity. We implement the buildup of snow on this geometry is it collides with it, and provide a method for redistributing fallen snow, modelling sinking snow and the movement of snow down-slope.

We show that the simulation really benefits from a parallel CUDA GPU implementation, since it is able to maintain real-time frame rates on a modern NVIDIA GPU with particle counts exceeding two million, all of which are interacting both with the wind field and the ground.

The number of fluid cells simulated in the wind field can be scaled up beyond four million while maintaining our real-time requirement. Finally, we show that the performance hit of increasing geometry resolution to high values like over one million vertices was not significant.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank Dr. Anne C. Elster for supervising this thesis, and Rune Erlend Jensen for spending so much time looking at my code with me. Finally, I would like to thank NVIDIA for donating the GPUs in the NTNU/IDI HPC-Lab through Dr. Elster's membership in their Professor Partnership program.

Trondheim, Feb. 2009

Robin Eidissen

Chapter 1

Introduction

Realistic simulation of snowfall is a computationally expensive problem. First of all, a large number of snow particles are needed to achieve realism, each of which must be updated individually, taking a variety of factors into account including the air around it. Second, the air that affects all particles is also a very dynamic and important part of the simulation, and it needs to be modeled convincingly. This means that we have to model the turbulent flow of air around and between objects in our scene. Lastly, the snow itself needs to interact with the geometry of the scene. That is, we want snow to build up where it hits the ground.

In our application we implement a real-time[1] version of the above, utilizing the power of modern GPUs to do the expensive calculations. We build upon work by Mr. Ingar Saltvik[17] where a similar program was implemented for multi-core CPUs. Parallelizing for the GPU both require and enable somewhat different techniques, but as we shall see, this computationally expensive problem benefits much from a parallel GPU implementation.

By using a terrain height map model for the geometry we are able to create a rich high resolution environment with which the snow and wind can interact, without introducing performance-limiting complexity. This and the fact that the GPU handles much lager particle counts and wind field resolutions enables scenes that are more visually compelling than what was done in[17].

We also improve upon work done by Saltvik by making the built up snow on the ground affect both wind and falling snow. A computationally inexpensive method for redistributing and smoothing snow on the ground is introduced

[1]An often used criterion for real-time is over 25 FPS

and implemented, to more closely mimic observable patterns in nature, and avoid anomalous buildup. Finally, to make the simulation really come alive, we implement support for stereo rendering, enabling the use of 3D monitors.

1.1 Outline

In Chapter 2 we examine the state of the art in parallel computing as it relates to this thesis. We look at the mathematical models on which we will build our simulation, such as fluid dynamics, and present previous work on relevant topics.

Chapter 3 is a discussion of many of the details involved in the implementation of our application, and the techniques that are used. Important topics here are rendering and the utilization of CUDA in our implementation.

Our results are presented in Chapter 4, where the performance characteristics of the implementation are examined. We also present and evaluate the visuals that are achieved in the application.

Finally, we summarise our findings and draw conclusions in Chapter 5.

Chapter 2

Background

2.1 Parallel Computing

Improvements in single-processor design has been the main cause of our performance gains up until very recently. These improvements have been in the form of increased clock frequency and instruction level parallelism[13], and much complexity has been introduced in the process. Recent years have seen diminishing gains in these areas, and while single-threaded performance continues to increase, most gains come from increasing parallelism. Mainstream computers are now symmetric multiprocessing(SMP) configurations with two to four CPU cores. Saltvik's implementation of real-time snow simulation[18] utilized the parallelism yielded by this kind of system, with good results. This is just one of the avenues that are being explored in the search for more parallelism.

Taxonomies

Flynn introduced a taxonomy which can still encompass the entire design-space of parallel computer design[13] (though it my be imprecise for some purposes). It established categories basted on the presence of parallelism in the instruction and data streams of the computer. These are the categories:

SISD(Single Instruction Single Data) This corresponds to the traditional von Neuman architecture. The name implies that there is no parallelism exhibited in the instruction or data streams.

SIMD(Single Instruction Multiple Data) A single instruction stream operating on multiple independent data streams. The operations on each data element may be parallelized. GPUs operate on a SIMD principle.

MISD(Multiple Instruction Single Data) Mainly used for fault tolerance, where multiple different systems operate on the same data, and compare results.

MIMD(Multiple Instruction Multiple Data) Most parallel computers operate on this principle(for example SMP machines and cluster computers).

As mentioned, SISD computers have come a long way but are now coming up against something of a wall, and MIMD techniques have largely been responsible for the gains of recent years. Most of the Top 500 supercomputers of the last decade[1] have been MIMD architectures(Figure 3.13). These are mostly clusters of SMP machines, and they are exploited by partitioning problems into chunks that are handled by each individual node(most problems also require communication between nodes).

At the level of each physical chip, there is still a large design space. Current CPU architectures are well suited to certain problem types, but far from optimal for all. A SIMD architecture can sometimes improve performance substantially. Many CPUs have SIMD instructions that can perform arithmetic on small vectors(SSE in Intel processors), but for some problem types it pays to go further. The GPU shows what can be achieved by venturing far in this direction, and it may work well in conjuntion with the prevailing MIMD paradigm.

2.2 The GPU Platform

Today we see graphics hardware beginning to supplant the CPU as the main computational unit in many problem areas. We still know it by the name GPU(Graphics Processing Unit), and it is still by this function that the vendors are able to sell them in large numbers. Acceleration of graphics in computer games is their Raison d'être, but games have placed a steadily increasing demand on programmability. At the same time, researchers have

[1]http://www.top500.org/overtime/list/32/archtype

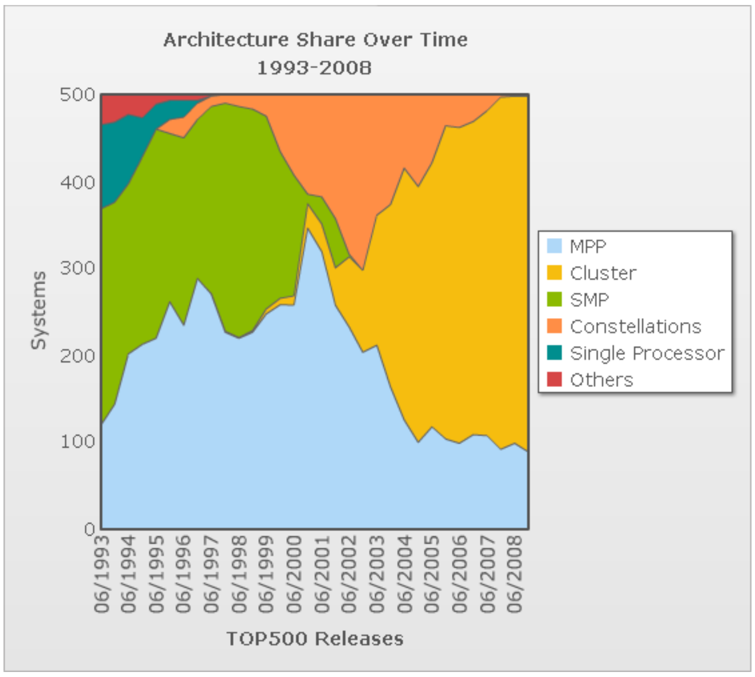


Figure 2.1: Architecture share over time among the Top 500

increasingly been able to exploit their power for computation. Thus the modern GPU is a very different beast to the GPU of just a few years ago, and applicability to general computation may soon be vital to the commercial success of a GPU. Much has been said of the history of GPUs and of GPGPU. [6] provides a thorough investigation of the subject up to the types of architectures that preceded the newest generation.

This generation however, deserves a more in-depth analysis here, since it's the platform we will be using for our program. More specifically we will be using NVIDIA's CUDA language which is a C-derived language that facilitates programming of newer NVIDIA hardware. Because of this, only NVIDIA's current architecture will be investigated.

2.2.1 Direct3D 10

With the evolution of the programmable graphics pipline being the main driving force behind GPU development, it is fitting to look more closely at the its most recent incarnation. It is supported in Microsoft's DirectX, through the Direct3D 10 API. OpenGL supports roughly the equivalent functionality, though some of it requires the use of extensions which have yet to be included in an official version of the standard. With most modern game developers using Direct3D, it is the API in which most new developments have been introduced. These are the components of the Direct3D pipeline[5]:

Input Assembler(IA) 3D rendering begins with geometry data. The Input Assembler gathers vertex data from up to 8 input streams bound to vertex buffers, and assembling vertices to be sent to the following stage.

Vertex Shader(VS) Normally used to transform vertices from object to screen space, but it can perform arbitrary transformation. It has access to a common featureset shared by all programmable shader stages.

Geometry Shader(GS) Transformed vertices pass through this stage, not one by one, but as part of the geometric primitive they make up(point, line or triangle). It may produce additional primitives, by outputting more than the input primitive, or it may delete by refraining from outputting. It can also add or modify attributes to the vertices of a primitive.

Stream Output(SO) Produces new vertex buffers from the output of the GS.

Set-up and Rasterization Stage(RS) This stage prapares the primitives for rendering. This includes clipping, culling, perspective divide, viewport transform, primitive set-up, scissoring, depth offset and fragment generation.

Pixel Shader(PS) Reads input attributes for fragments, and produces output fragments consisting of 1 to 8 attributes (which in most cases are color components), as well as an optional depth value that will override the one carried over from the RS stage. May discard fragments.

Output Merger(OM) Merges fragments from the PS stage into a set of render buffers, while performing blending and depth/stencil testing.

The main new development that we saw in D3D10 was unified shaders. Earlier we had vertex shading and pixel shading, each of which were performed by dedicated processing units. However through the generations, these two types of processing unit converged toward a similar featureset and toward high generality. D3D10 introduces a third type of shader(the geometry shader), and executes all shaders on one type of stream processor.

NVIDIA also supports compute kernels as a fourth program type. These are programs that can do general purpose computation on the GPU. They are made possible in part by the hardware required to support the unified shaders of Direct3D 10. In addition to this they require the ability to do random access writes to GPU memory, as well as explicit control of on-chip memory. As such they are not supported as a direct consequence of supporting D3D10, but were easily within reach, and the GPGPU support was developed alongside the Direct3D 10 support[15].

2.2.2 The Tesla Architecture

NVIDIA's newest generation cards are based on the scalable Tesla architecture[15], which was introduced in November 2006 with the release of the Geforce 8. It has subsequently been improved, but the basic design survives. It was the first architecture from NVIDIA designed to support DirectX 10, and as such an important goal was the required support for unified shaders. The generality offered by a compliant design would be of tremendous benefit for parallel-computing on the GPU.

Model	9400 GT	9800 GTX	GTX 280
Stream Processors	16	128	240
Min. RAM	256 MB	512 MB	1 GB
Core freq.	550 MHz	676 MHz	600 MHz
RAM freq.	1600 MHz	2200 MHz	2600 MHz
Bus width	128 bit	256 bit	512 bit

Table 2.1: Example models based on the Tesla architecture[2]

As shown in Table 2.1, the available cards that use this architecture range from the low-end to the very high-end. Importantly, the same base functionality is offered independent of computational power, so programs made for this architecture will run on all cards provided they are scalable.

This last statement is actually a half-truth. NVIDIA has deviced a versioning scheme to indicate the capabilities supported by an architecture, called *compute capability*[1]. The compute capability consists of a major an a minor revision number, where the major revision is the same for models that share the same core architecture, and the minor revision number signifies just that: Minor revision of the same core architecture. The base functionality supported by all Tesla architecture models is defined under compute capability 1.0, and revisions 1.1, 1.2 and 1.3 add support for a few select things like atomics and double precision, as well as expanding some capacities.

Overall Architecture

Before going into details of its operation and how this affects the programmer, we will give a high level overview of the architecture as seen in Figure 2.2. Two main elements are responsible for the mentioned scalability. The first is the streaming processor array(SPA) which is responsible for performing all programmable calculations, and the second is is the memory architecture which supports it. Both of these are designed from smaller units that can be aggregated to facilitate higher performance.

The SPA is comprised of one or more texture/processing cluster(TPC, Figure 2.3). The TPCs in turn comprise the following components:

2+ Streaming Multiprocessors(SM) Each of which contains the following:

- 8 stream processors(SP) which support general floating point and integer instructions. The SMs and SFUs operate at twice the core frequency.
- 2 special function units(SFU) that can do complex calculations like trancendental functions and interpolation.
- 8192 registers(16384 for compute capability 1.2 devices).
- 16KB high-speed on-chip memory called shared memory. Successive words in this memory fall into one of 16 banks, each of which can be accessed independently. When there are no bank conflicts between simultaneously running threads access to this memory is as fast as register access.
- 8KB constant cache which is as fast as registers when all threads read the same value. This constant cache works against a 64KB per-device constant memory.
- An multi-threaded instruction fetch and issue unit.
- An instruction cache.

1 Streaming Multiprocessor Controller(SMC) Controls multiple SMs, arbitrating acces to shared texture unit, load/store and I/O path.

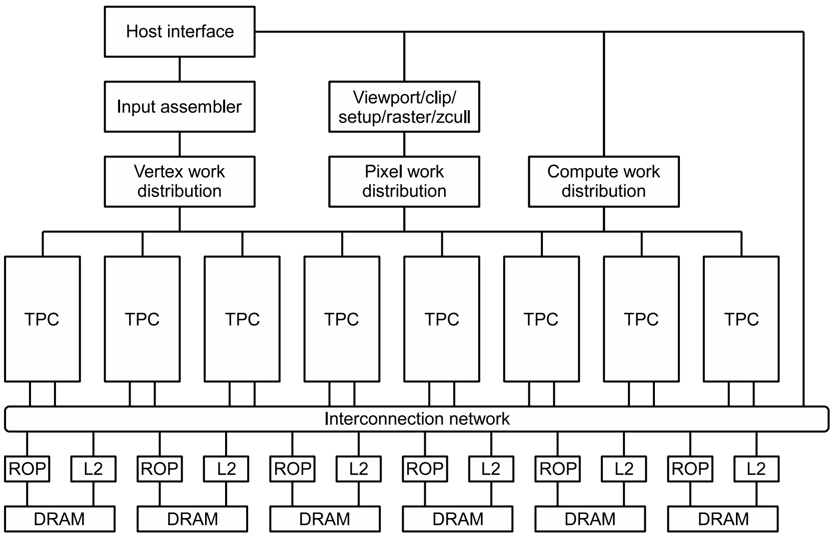


Figure 2.2: High level overview of the GeForce 8 architecture[15]

1 Texture Unit Inputs texture coordinates, and outputs filtered texture samples. Can process 4 threads per cycle, producing a four-component result. Caches samples to exploit filtering locality.

1 Geometry Controller This component replicates the functionality of parts of the traditional graphics rendering pipeline. It has storage space for input and output attributes, and manages vertices through the vertex shader and geometry shader stages.

The global memory DRAM is partitioned, with one ROP(Raster Operations Processor) unit responsible for each partition. They can handle hundreds of in-flight memory access requests from the parallel threads, and has logic to coalesce requests from threads running in the same warp accessing consecutive memory locations(subject to some restrictions). An interconnection network connects the TPCs to the ROPs, and any TPC can access memory governed by any ROP.

Operation

The SMs can process four different types of thread programs: Pixel shaders, vertex shaders, geometry shaders and compute threads. To balance shifting workloads the SM can concurrently execute different types of threads, and

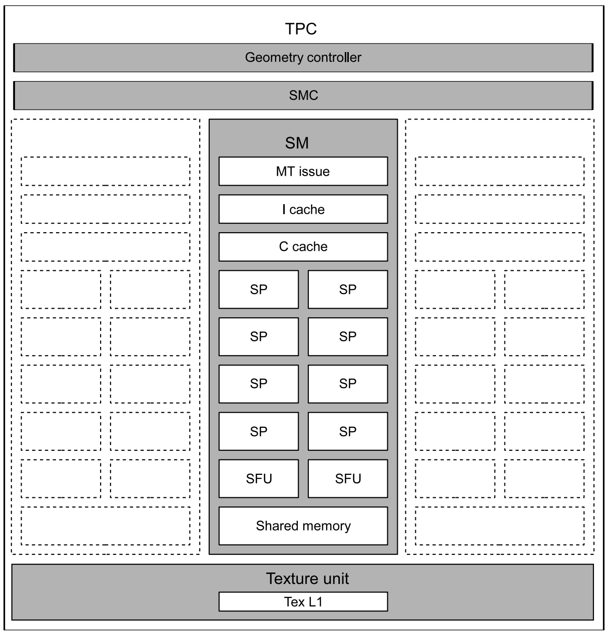


Figure 2.3: Detailed view of a TPC unit[15]

different programs[15]. It is hardware multithreaded to make this as efficient as possible, and can manage up to 768(1024 for compute capability 1.2) concurrent threads with zero scheduling overhead. The threads are executed in groups of 32 called a warp, and multiple warps are grouped into one block. This means that there can be a maximum of 24(or 32) active warps per SM. Each thread has its own state and may execute its own code path. All threads in a block running on a multiprocessor may synchronize using a barrier instruction. This is essential because access to the on-chip shared memory needs to be synchronized.

NVIDIA calls this SIMT(Single Instruction Multiple Thread), although it sounds very much like SIMD(and it arguably is), but some details are different. Each thread in a warp starts at the same instruction, but threads may branch differently. However if there are divergent branches within a warp, they have to be serialized, so it still operates in SIMD mode with some threads doing nothing until they all return to the same execution path(Figyre 2.4). With typical SIMD units the programmer works with vector types as big as the SIMD width, and has to be careful how the data is packed into these vectors to achieve full utilization. On the Tesla architecture, programmers are only exposed to scalar operations and the hardware automatically fills the execution units by exploiting thread parallelism, if possible. This means

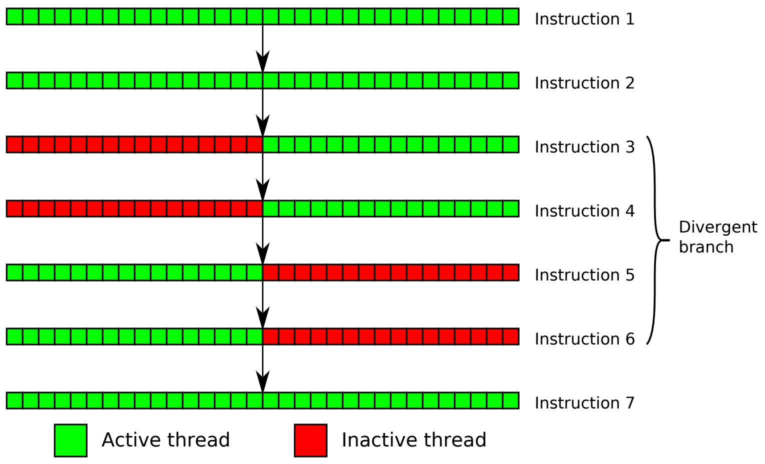


Figure 2.4: SIMD execution and divergent branches

that the programmer may think about the threads as running independent code-paths and forget about SIMD width and other concerns, as far as correctness is concerned. But to achieve good performance the actual execution of threads in a warp and the consequences of branching should be kept in mind.

2.2.3 CUDA

To be able to use the general computation capabilities of the new GPUs, we need a suitable language. The traditional shader languages like Cg are specialized toward graphics, and fail to expose some of the capabilities, while also being cumbersome to use. Every problem needs to be framed as a graphics rendering problem, and all data must be mapped to textures and vertices.

The solution was CUDA[1], a language developed alongside the new compute capable hardware. It is based on C++, with a set of syntax extensions and library support needed to make it suitable to this new hardware platform. Not all of the C++ language is supported, with exceptions being notable in their absense(thus leaving out much of the Standard Template Library).

A function in CUDA may execute on the CPU or on the GPU, and therefore new keywords were introduced to annotate this:

- `__host__` This is the default classifier, and may therefore be omitted.

These are our traditional C functions that run on the CPU, and ar only callable from other host functions.

- `__global__` Functions with this classifier are called kernels. They run on the GPU and are callable only from host functions. Special syntax is required for their invocation.
- `__device__` These run on the GPU and are only callable from kernels or other device functions.

While the execution on the CPU of a normal procedure is well defined by the information given in the C language, more information is needed to invoke a GPU kernel. Such an invocation is intended to spawn many execution threads, and information is needed on just what the execution parameters are. As mentioned in the section describing the Tesla architecture(page 7), threads are grouped into blocks that are executed on a single SM. Blocks are subdivided into warps(32 threads) which is the basic unit of execution as far as the scheduler is concerned.

A block is a 1 to 3 dimensional domain, and for each point in this domain a thread is created. It is up to the threads to decide how their block coordinates will map to the data that they operate on. With the block dimensions defined, a computational grid finally needs to be defined. This is a one to three dimensional grid of blocks, that will cover the entire computation domain. An image processing kernel might for example define blocks that are $block_x \times block_y$, and given the image dimensions $image_x \times image_y$ we would define the grid(Figure 2.5): $grid_x = image_x/block_x, grid_y = image_y/block_y$.

The syntax to support this is addid in the form of a parameter list between triple angle brackeds just after the function name. This is an example kernel invokation:

```
dim3 grid(IMAGE_X/BLOCK_X, IMAGE_Y/BLOCK_Y, 1);
dim3 block(BLOCK_X, BLOCK_Y, 1);
myKernel<<<grid, block>>>(param1, param2);
```

For an executin kernel thread, a few implicit parameters makes the grid information available:

- `gridDim` is of type `dim3`, and contains the grid dimension.
- `blockDim` is of type `dim3`, and contains the block dimension.

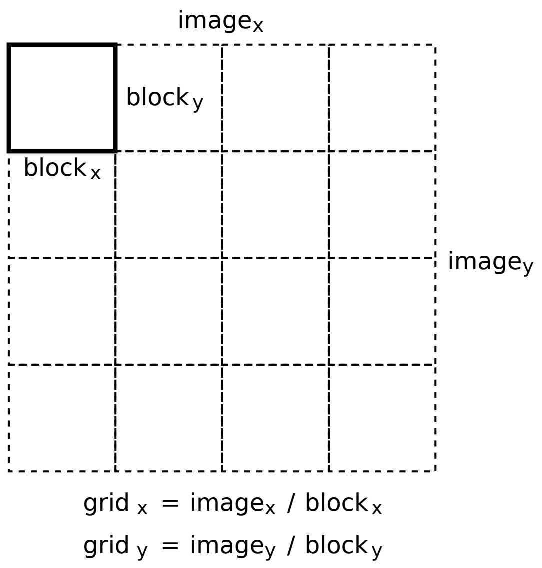

image$_x$
block$_y$
block$_x$
image$_y$
grid$_x$ = image$_x$ / block$_x$
grid$_y$ = image$_y$ / block$_y$

Figure 2.5: The composition of a 2D computation grid

- `blockIdx` is of type `uint3`, and contains the location of the current block within the grid.
- `threadIdx` is of type `uint3`, and contains the location of the current thread within the current block.

The invocation syntax supports two more optional parameters. One is the amount of shared memory that is to be allocated to each kernel block(to enable dynamic change of this setting), and the last one is the stream variable. Kernel invocations and other asynchronous operations may specify a stream, and all operations in a stream must be performed in order. Different streams however, may execute concurrently. When omitted, the default null-stream is assumed. To show the full kernel invocation syntax, for a kernel requiring a float of shared mamory per thread, and running in a user specified stream:

```
cudaStream_t stream;
cudaCreateStream(&stream);
dim3 grid(IMAGE_X/BLOCK_X, IMAGE_Y/BLOCK_Y, 1);
dim3 block(BLOCK_X, BLOCK_Y, 1);
myKernel<<<grid, block, BLOCK_X*BLOCK_Y*sizeof(float), stream>>>(param1, param2);
```

Similar to how functions may be classified by their execution environment, variable declarations may include an extra storage classifier.

- `__device__` Resides in global memory, and has the lifetime of the application. Accessible from all threads, and from the host through the API.
- `__constant__` Resides in constant memory, and has the lifetime of the application. Accessible from all threads, and from the host through the API.
- `__shared__` Resides in the shared memory of a block, and has the lifetime of the block. Accessible from all threads within the block.

`__shared__` declarations deserve some more discussion. One usage pattern to declare it like this within the kernel function:

```
__shared__ float block[BLOCK_X*BLOCK_Y];
```

Doing this means that the shared memory available to the kernel block is determined at compile time. To be able to specify this at runtime like in the above example, the array can be defined like so:

```
extern __shared__ float block[];
```

Memory Model

When programming the CPU, we are used to dealing with one large memory that is transparently backed by a high-speed cache on the processor. Access to main memory is high latency, and access to the cache is very low latency. All memory access is handled in a uniform manner, and for many problems we do not even have to think to much about cache effects to achieve good performance(although in scientific computing this is an entire reaserch topic in itself).

The GPU also has a large high latency main memory called global memory, as well as fast on-chip memory called shared memory for each SM. The latter does not function like a cache, but is completely under programmer control. Also achieveble bandwidth against global memory is determined by adherence to a set of restrictions on access pattern by threads in a warp. These things combine to make efficient memory access something the programmer has to put considerable thought into.

Shared Memory

The 16KB of shared memory available on each SM is divided into 16 separate banks that can be accessed simultaneously by all threads in the active half-warp. Successive 32-bit memory locations fall into separate banks. If care is not taken when accessing this memory, bank conflicts can occur. A bank conflict will occur any time more than one thread tries to access memory within the same bank, and all such accesses will lead to serial execution. The hardware will try to split accesses into into as few and big conflict-free sets as possible, and these will be run serially. A simple way to ensure no conflicts arise is to access memory in a pattern like `array[BaseIndex + tid]`(where `tid` is the thread ID, and the element type for `array` is four bytes).

A common access pattern is `array[BaseIndex + s*tid]`, where `s` is a stride value. This pattern will lead to conflict between threads `tid` and `tid+1`, if `s*n` is a multiple of the number of banks(16). On all current devices(compute capability 1.x) this will not happen as long as `s` is odd.

The following code will be checked for conflicts with Type substituted for various values:

```
__shared__ Type shared[32];
float data = shared[BaseIndex + tid];
```

Type	Conflict?	Explanation
`char`	Yes	Each set of four consecutive values belong to the same bank.
`float3`	No	Base type 32 bits wide, and odd stride(3).
`float4`	Yes	Even stride.

Global Memory

To utilize as much as we can of the global memory bandwidth available to us, we need to be aware of some details of its operation. As mentioned it is uncached, and very high latency (400-600 cycles, equivalent to 100 floating point MADD instructions). But it is also very high bandwidth, under the right conditions.

Good bandwidth is achieved by a technique called coalescing. As all the threads in an active half-warp requests a read or write operation against global memory, the memory controller may coalesce all these separate accesses into one single access, which will complete in about the same time as any one single uncoalesced access. Coalescing is done automatically if a few conditions are met. These conditions are relaxed for compute capability 1.2 and above, so they will be discussed for both cases.

For compute capability 1.0 and 1.1 cards:

- Threads must access words of size:

  32-bit Results in one 64 byte transaction.

  64-bit Results in one 128 byte transaction.

  128-bit Results in two 128 byte transactions.

- All 16 words must lie in the segment equal to the transaction size(or twice that for 128-bit words).

- The accesses must be in order: The *kth* thread must access the *kth* word.

For compute capability 1.2 cards:

- Accesses will be coalesced when they fall into the same segment of size:

  32 bytes When threads acces 8-bit words.

  64 bytes When threads acces 16-bit words.

  128 bytes When threads acces 32-bit words.

- When accesses fall into n segments, they will be coalesced into n transactions. For access against 128-bit words this results in two transactions for example.

We summarize that the major differences here is that for earlier cards we need to access words in order, and also that an access pattern spanning more than one segment would immediately result in 16 separate accesses, instead of the two(if it spans two segments) that will result on newer cards.

2.3 Computational Fluid Dynamics

Numerical simulation brings together the different worlds of theoretical science and experimental science. Theoreticians develop mathematical models that accurately describes phenomenons occuring in the physical world. By discretisizing the equations that make up these models, we are able to find approximate solutions to them. This, coupled with the use of modern computers, allow us to perform simulations of physical phenomena. Simulation may in many cases be used instead of physical experiment. This can be much cheaper, and is often the only option available to us if we want to observe the dynamics of certain processes[11].

2.3.1 The Navier-Stokes Equations

Incompressible flows can be described[11] by a set of partial differential equations called the Navier-Stokes equations(Eq. 2.1, 2.2). Eq. 2.1 is called the momentum equation, and Eq. 2.2 is the continuity equation.

$$\frac{\partial \mathbf{v}}{\partial t} = -(\mathbf{v} \cdot \nabla)\mathbf{v} - \frac{1}{\rho}\nabla p + \nu \Delta \mathbf{v} + \mathbf{f} \tag{2.1}$$

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

where $\mathbf{v}$ is the velocity vector, p is pressure, ρ is density, ν is the kinematic viscosity and $\mathbf{f}$ is the body forces acting on all particles in the fluid. Specifying initial($t = 0$) conditions $\mathbf{v} = \mathbf{v_0}$ at all points in the domain, as well as boundary conditions for $t > 0$, this forms and initial-boundary value problem.

Here is a short explanation[17] of each of the terms on the right hand side of Eq. 2.1:

Advection, $-(\mathbf{v} \cdot \nabla)\mathbf{v}$: The $\mathbf{v} \cdot \nabla$ part of this term is the advection operator for the vector field $\mathbf{v}$. This operator will transport a quantity along $\mathbf{v}$, and here it is being applied to itself. That is, the fluid is transported along its own flow.

Pressure, $-\frac{1}{\rho}\nabla p$: The velocity field also moves along the gradient of the pressure field. The physical explanation of this is that the random motion of fluid particles statistically favors motion from high to low pressure areas.

Diffusion, $\nu\Delta\mathbf{v}$: Δ is the Laplace operator and is also written ∇^2. $\Delta\mathbf{v}$ is equivalent to the second derivative of the vector field, and the diffusion term acts to resist flow changes. Fluids with higher kinematic viscosity factor ν will be more resistant, and fluids with $\nu = 0$ are called inviscid.

External Forces, $\mathbf{f}$: This is volumetric forces acting on the fluid, an example of which is gravity.

In some fluid models, some of these terms fall out. There is no diffusion in inviscid gases for example. The Navier-Stokes equations with the diffusion term dropped are called the Euler equations(Eq. 2.3, 2.4).

$$\frac{\partial\mathbf{v}}{\partial t} = -(\mathbf{v}\cdot\nabla)\mathbf{v} - \frac{1}{\rho}\nabla p + \mathbf{f} \tag{2.3}$$

$$\nabla\cdot\mathbf{v} = 0 \tag{2.4}$$

2.3.2 Discretization

To be able to compute solutions to these equations, we need to discretisize them[11]. This is the process of moving the problem from a continous domain to a discrete domain. By doing this we reduce the differential equation to a system of algebraic equations, with the amount of unknowns and the amount of equations equaling the number of discrete points in the domain. There are many discretization methods, such as the finite element, finite volume and the finite difference method. Here the finite difference method will be explained.

The continous domain within which a solution is sought is divided into equal-sized parts, and the solution will be sought on the points separating these. For the one-dimensional case an interval $\Omega := [0, l]$ is divided into n subintervals of length $\delta x := l/n$. We then have a grid comprised of the points $x_0, x_1, ..., x_n$ at the boundaries of these subintervals.

Given a function $u(x_i)$ defined for all grid points, we approximate the derivative for a point x_i by

$$\left[\frac{du}{dx}\right]_i = \frac{u(x_{i+1}) - u(x_{i-1})}{2\delta x}, \tag{2.5}$$

and the second derivative by

$$\left[\frac{d^2u}{dx^2}\right]_i = \frac{u(x_{i+1}) - 2u(x_i) + u(x_{i-1})}{\delta x^2}. \tag{2.6}$$

These approximations will be more accurate when the grid resolution is higher.

The advection operator $\mathbf{v} \cdot \nabla$ on the cartesian vector field $\mathbf{v}$ is defined as

$$\mathbf{v} \cdot \nabla = v_x \frac{\partial}{\partial x} + v_y \frac{\partial}{\partial y} + v_z \frac{\partial}{\partial z}. \tag{2.7}$$

Given this, the component form of Eq. 2.1 and 2.2 in cartesian coordinates are as follows:

$$\frac{\partial v_x}{\partial t} = -\left(\frac{\partial v_x^2}{\partial x} + \frac{\partial v_x v_y}{\partial y} + \frac{\partial v_x v_z}{\partial z}\right) - \frac{1}{\rho}\frac{\partial p}{\partial x} + \nu\left(\frac{\partial^2 v_x}{\partial x^2} + \frac{\partial^2 v_x}{\partial y^2} + \frac{\partial^2 v_x}{\partial z^2}\right) + f_x \tag{2.8}$$

$$\frac{\partial v_y}{\partial t} = -\left(\frac{\partial v_x v_y}{\partial x} + \frac{\partial v_y^2}{\partial y} + \frac{\partial v_y v_z}{\partial z}\right) - \frac{1}{\rho}\frac{\partial p}{\partial y} + \nu\left(\frac{\partial^2 v_y}{\partial x^2} + \frac{\partial^2 v_y}{\partial y^2} + \frac{\partial^2 v_y}{\partial z^2}\right) + f_y \tag{2.9}$$

$$\frac{\partial v_z}{\partial t} = -\left(\frac{\partial v_x v_z}{\partial x} + \frac{\partial v_y v_z}{\partial y} + \frac{\partial v_z^2}{\partial z}\right) - \frac{1}{\rho}\frac{\partial p}{\partial z} + \nu\left(\frac{\partial^2 v_z}{\partial x^2} + \frac{\partial^2 v_z}{\partial y^2} + \frac{\partial^2 v_z}{\partial z^2}\right) + f_z \tag{2.10}$$

$$\frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z} = 0 \tag{2.11}$$

All the terms can be approximated by finite-difference operators like those in Eq. 2.5 and 2.6, or other methods with different tradeoffs.

2.3.3 Numerical Simulation

We want to facilitate a time stepping simulation of Eq. 2.8-2.10. We have just the time derivative of the velocity field on the left-hand side of the equations, and all spatial derivatives on other side. To obtain the value of $\mathbf{v}^{(n+1)}$ at time

t_{n+1} given $\mathbf{v}^{(n)}$we can compute $\frac{\partial \mathbf{v}}{\partial t}$ and then we can employ a simple Euler integration scheme like

$$\mathbf{v}^{(n+1)} = \mathbf{v}^{(n)} + \delta t \frac{\partial \mathbf{v}}{\partial t}, \tag{2.12}$$

where n is the time step and δt is the time delta. We choose to evaluate the velocity spatial derivatives at time t_n which is called an *explicit method* - the values at time t_{n+1} can be calculated directly from the velocity at t_n. The pressure derivative is evaluated at time t_{n+1}, which is called an *implicit method*, and it requires us to solve an equation involving both the current and the next state of the system.

By calculating the explicit parts of $\mathbf{v}^{(n+1)}$ we can obtain an intermediate velocity field. We introduce an abbreviation $\mathbf{F}$ for this:

$$\mathbf{F}^{(n)} = \mathbf{v}^{(n)} + \delta t \left[-(\mathbf{v} \cdot \nabla)\mathbf{v} + \nu \Delta \mathbf{v} + \mathbf{f} \right]. \tag{2.13}$$

We can now express the velocity at the next time step as a modification of this intermediate field:

$$\mathbf{v}^{(n+1)} = \mathbf{F}^{(n)} - \frac{\delta t}{\rho} \nabla p^{(n+1)}. \tag{2.14}$$

When this final velocity is substituted into the continuity equation (Eq. 2.4) we obtain the *Poisson equation for pressure*[11]:

$$\Delta p^{(n+1)} = \frac{\rho}{\delta t} \nabla \cdot \mathbf{F}^{(n)}. \tag{2.15}$$

Using the continuity equation to arrive at this ensures that the resulting velocity field will be divergence-free. What remains before we can calculate $\mathbf{v}^{(n+1)}$ is to solve this equation numerically to obtain the pressure field.

Solving the Poisson Equation

Here we will discuss the solution of systems of differential equations like Eq. 2.15. Let us name the unknown solution(which is the pressure) of Eq. 2.15 u, and let us name the right-hand side f. We may then write the equation like this:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = f \tag{2.16}$$

Using a central difference approximation this equation may then be discretized as

$$\begin{aligned}\frac{-u_{i+1,j,k}+2u_{i,j,k}-u_{i-1,j,k}}{h^2} &+ \\ \frac{-u_{i,j+1,k}+2u_{i,j,k}-u_{i,j-1,k}}{h^2} &+ \\ \frac{-u_{i,j,k+1}+2u_{i,j,k}-u_{i,j,k-1}}{h^2} &= f_{i,j,k},\end{aligned}$$

where h is the point spacing of the discretization. Then we can solve this equation for $u_{i,j,k}$

$$u_{i,j,k}=\frac{1}{6}(u_{i+1,j,k}+u_{i-1,j,k}+u_{i,j+1,k}+u_{i,j-1,k}+u_{i,j,k+1}+u_{i,j,k-1}+h^2f_{i,j,k}) \quad (2.17)$$

Now that we have an expression for $u_{i,j,k}$, we can use an iteration scheme called the *Jacobi method*[7]. We will use the symbol v to denote an approximation to the exact solution u. Here an improved approximation $v^{(1)}$ is computed from the previous approximations $v^{(0)}$:

$$v^{(1)}_{i,j,k}=\frac{1}{6}(v^{(0)}_{i+1,j,k}+v^{(0)}_{i-1,j,k}v^{(0)}_{i,j+1,k}+v^{(0)}_{i,j-1,k}v^{(0)}_{i,j,k+1}+v^{(0)}_{i,j,k-1}+h^2f_{i,j,k}). \quad (2.18)$$

This function is applied to all grid cells for each iteration. An initial guess is used to start the iteration process. We see that this update function is a function of all the neighboring approximations(Figure 2.6), as well as the right-hand side f of the Poisson equation at that point.

This method will converge under certain conditions[20], but in many cases it will require a large amounts of iterations. The *Gauss-Seidel*(GS) method is a variation of this method that will boost convergence rates by as much as 50%. This method utilizes updated approximations $v^{(1)}$ whenever they are available, instead of always using $v^{(0)}$ values. If the order of update is increasing x, y and z then for $v_{i-1,j,k}$, $v_{i,j-1,k}$ and $v_{i,j,k-1}$, the new approximations will be used. Other orderings such as *red-black*[11] may in some cases lead to better convergence.

The *Successive Over-Relaxation*(SOR) method by Young[20] is a modification of GS that can significantly improve convergence in certain cases. He

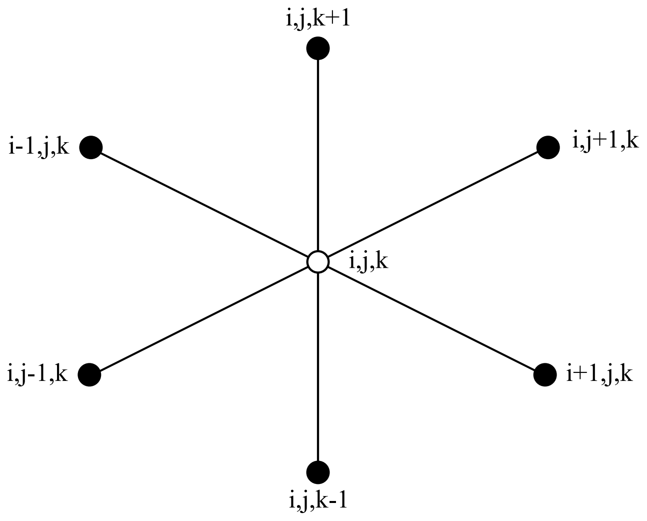


Figure 2.6: Stencil used for updating the approximation $v_{i,j,k}$ at each point

provides a rigorous analysis, but for our purposes a simple explanation of the method will suffice.

Like with GS, new approximations are used when available, but each new approximation i are set to a weighted sum of the old and the updated values:

$$v_i^{(1)} = (1-\omega)v_i^{(0)} + \omega\overline{v}_i \tag{2.19}$$

where $\omega \in [0, 2]$ is the relaxation factor, and $\overline{v}_i$ is the updated GS iterate. With $\omega = 1$ this method reduces to regular GS, while values above 1 helps speed up convergence, and values below 1 may induce convergence in divergent iterations.

2.4 Snow Modeling

Armed with the fluid dynamics of previous sections we can set out to describe a complete snow model. The fluid dynamics will enable us to simulate wind effects realistically, and in conjunction with the particle model described below forms the basis for the snow simulation. To complete the model we need some kind of geometry with which the wind and the snow will interact, and a way for falling snow to build up on its surfaces.

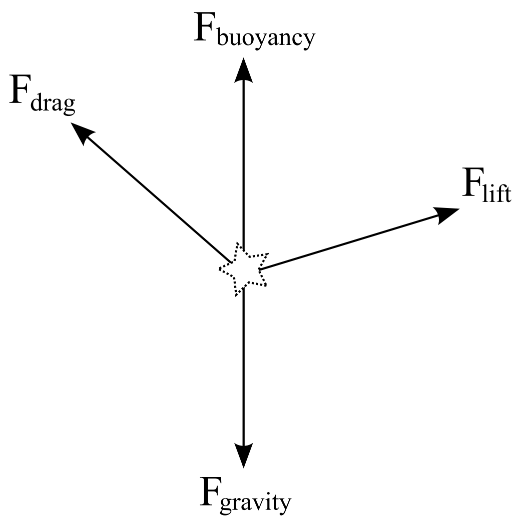


Figure 2.7: Forces influencing a snowflake

2.4.1 Snow Particle Simulation

The movement model used here for the falling snowflakes is the same as in and [3], and we follow their descriptions. It is a particle-based approach that keeps track of the position and velocity of each individual snowflake. Four different forces(Figure 2.7) influencing the particles are identified:

$\mathbf{F}_{gravity}$ This is constant for each snow flake and is determined its mass m, and the gravitational constant g. It is always directed down along the negative z-axis and is of magnitude mg.

$\mathbf{F}_{buoyancy}$ The buoyancy force is caused by density differences between objects and their surrounding medium. For falling snow it is small enough that we can ignore it.

$\mathbf{F}_{lift}$ Vortices created by the snow flake and nearby snowflakes as they fall through the air generate the chaotic looking lift force, which result in the irregular motion of each snow flake.

$\mathbf{F}_{drag}$ The drag force is born from the difference in velocity between a snow flake and that of the air itself. Its magnitude varies according to some properties of individual snow flakes.

Symbol	Description	Initialization
$\mathbf{p}$	Position	Randomly within the scene boundaries
$\mathbf{V}_{snow}$	Velocity	$([-1,1],[-1,1],V_{max,z})$
R	Radius of circular movement	$(0,2)$
ω	Angular velocity of circular movement	$[-\frac{\pi}{4},-\frac{\pi}{3}]$ or $[\frac{\pi}{4},\frac{\pi}{3}]$
$V_{max,z}$	Vertical terminal velocity	$[0.5,1.5]$ for wet snow, $[1,2]$ for dry

Table 2.2: Properties of a snow flake[17]

Each snow flake has a series of properties[2], as seen in Table 2.2, that define it and give rise to its behavior in its environment. All of these attributes are randomized at the beginning of the simulation, and then some of them are continuously updated according to the equations that we describe below. The mass m, by the way, is not included here since it drops out of our calculations as we shall see, and its effect is otherwise incorporated in $V_{max,z}$.

When a snow flake is moving just as fast and in the same direction as the air around it we understand that the air will exert no force on it. Likewise if it is moving in the opposite direction the force exerted will be greater than if it was standing still. In this way we can intuitively understand that the drag force is dependent on $\mathbf{V}_{fluid}$, the velocity difference between the snow flake en the wind:

$$\mathbf{V}_{fluid} = \mathbf{V}_{wind} - \mathbf{V}_{snow} \tag{2.20}$$

The force has the direction of $\mathbf{V}_{fluid}$, but its magnitude is C_{drag} which is defined for a snow flake as

$$C_{drag} = \frac{\mathbf{V}^2_{fluid} mg}{V^2_{max,z}} \tag{2.21}$$

where $V_{max,z}$ is its vertical terminal velocity under the influence of gravity. The drag force is then

[2]X and Y coordinates of position are randomized to ensure they are a source of noise at all times as explained on p. 44

$$\mathbf{F}_{drag} = \frac{\mathbf{V}_{fluid}}{|\mathbf{V}_{fluid}|} \cdot C_{drag}. \tag{2.22}$$

To see how this makes sense let's define as per Newton's second law, the combined acceleration $\mathbf{a}$ caused by $\mathbf{F}_{drag}$ and $\mathbf{F}_{gravity}$ as

$$\mathbf{a} = \frac{\mathbf{F}_{gravity} + \mathbf{F}_{drag}}{m}. \tag{2.23}$$

If wind is zero a particle's velocity will be increasing downwards under the influence of gravity. And as it approaches $V_{max,z}$ the drag force will increase until it matches that of gravity(in the opposite direction), after which acceleration will be zero.

As mentioned, the lift force is highly chaotic in nature, due to the chaotic and complex nature of the turbulence phenomena that cause it. Because of this it is difficult to model accurately. Instead we will use a simplified model, where each snow flake follows a spiral path on descent. If spiral radius R and angular velocity ω varies between individual snow particles their combined motion looks fairly convincing. We define the velocity contribution of this force at time t:

$$\mathbf{V}^t_{circ} = \frac{|\mathbf{V}_{fluid}|}{|\mathbf{V}_{snow}|} \cdot \omega R[-\sin \omega t, \cos \omega t, 0]. \tag{2.24}$$

The first factor scales it so that it diminishes as $\mathbf{V}_{fluid}$ grows. We can now write expressions for updated particle position and velocity at time $t + \Delta t$:

$$\mathbf{p}^{t+\Delta t} = \mathbf{p}^t + (\mathbf{V}^t_{snow} + \mathbf{V}^t_{circ})\Delta t + \frac{1}{2}\mathbf{a}\Delta t^2 \tag{2.25}$$

$$\mathbf{V}^{t+\Delta t}_{snow} = \mathbf{V}^t_{snow} + \mathbf{a}\Delta t \tag{2.26}$$

2.4.2 Wind Field Simulation

The wind field simulation will be based on the CFD model described in previous sections, also described by Saltvik[17]. We will assume a zero viscosity fluid with density equal to 1. This reduces the Navier-Stokes equations to the Euler equations.

The first part of the Euler equations is repeated here with volume forces **f** omitted:

$$\frac{\partial \mathbf{v}}{\partial t} = -(\mathbf{v} \cdot \nabla)\mathbf{v} - \nabla p \tag{2.27}$$

We will assume that the force of gravity on air is negligible compared to advection and pressure forces for the scenarios we will be modeling, thus **f** may be omitted.

We recall that this equation is implicit in pressure, so for p we need to solve a separate equation system. This was the Poisson equation for pressure(again assuming density $\rho = 1$):

$$\Delta p^{(n+1)} = \frac{1}{\delta t} \nabla \cdot \mathbf{F}^{(n)} \tag{2.28}$$

where n is the time step at which the term is evaluated and $\mathbf{F}^{(n)}$ for the Euler equations is given by

$$\mathbf{F}^{(n)} = \mathbf{v}^{(n)} - \delta t(\mathbf{v} \cdot \nabla)\mathbf{v}. \tag{2.29}$$

Looking at Eq. 2.27 we see that to compute the time derivative of **v**, we need to calculate the advection forces, and then the pressure forces. Given the time derivative we can compute an updated velocity field(as described on page 20). This process will be divided into three steps:

1. **Advection**: Calculate advective forces, and update the current velocity field $\mathbf{v}^{(n)}$, resulting in an intermediate field $\mathbf{F}^{(n)}$.
2. **Solve Poisson**: Solve the Poisson equation, resulting in a pressure field $p^{(n+1)}$.
3. **Projection**: Calculate pressure forces as per the second right-hand term in Eq. 2.27, and modify the intermediate velocity field $\mathbf{F}^{(n)}$ resulting in a final divergence-free field $\mathbf{v}^{(n+1)}$.

Advection Step

Eq. 2.8-2.10 show that the advection force is given by

$$-(\mathbf{v}\cdot\nabla)\mathbf{v} = -\left(\frac{\partial v_x^2}{\partial x} + \frac{\partial v_x v_y}{\partial y} + \frac{\partial v_x v_z}{\partial z}, \; \frac{\partial v_x v_y}{\partial x} + \frac{\partial v_y^2}{\partial y} + \frac{\partial v_y v_z}{\partial z}, \; \frac{\partial v_x v_z}{\partial x} + \frac{\partial v_y v_z}{\partial y} + \frac{\partial v_z^2}{\partial z} \right). \tag{2.30}$$

$$\begin{tikzcd}
 & \textbf{Advect} & & \textbf{Project} \\
v^{(n)} \arrow[r] & F^{(n)} \arrow[rr] \arrow[dr] & & v^{(n+1)} \\
 & & p^{(n+1)} \arrow[ur] & \\
 & & \textbf{Solve Poisson} &
\end{tikzcd}$$

Figure 2.8: Calculating the updated velocity field

This may be computed by doing a finite difference approximation of the partial derivatives as explained on p. 18. However this may lead to instability unless great care is taken according to Stam in[19], and another approach was described.

Let $\mathbf{x}$ be a grid point in the wind velocity field, and $\mathbf{u}(\mathbf{x}, t)$ be the velocity at this point. The intuition of this method is that to since the advection step reflects the motion of fluid particles along the velocity of the fluid itself, we can obtain the new velocity $\mathbf{u}(\mathbf{x}, t + \Delta t)$ by backtracing through the velocity as illustrated in Figure 2.9. We follow a path $\mathbf{p}$ from $\mathbf{x}$ from t to $t - \Delta t$, which corresponds to a partial stream line in the velocity field. At the end-point of $\mathbf{p}$ we can sample the velocity of the particle previously located there, which will now have moved to $\mathbf{x}$, and we update the velocity of field:

$$\mathbf{u}(\mathbf{x}, t + \Delta t) = \mathbf{u}(\mathbf{p}(\mathbf{x}, -\Delta t), t). \tag{2.31}$$

The velocity will change continously along path $\mathbf{p}$, but we will make the assumption that it will stay equal to $\mathbf{u}(\mathbf{x}, t)$ all along its length. For small time steps this approximation should prove adequate. This simplification yields this expression for the updated velocities:

$$\mathbf{u}(\mathbf{x}, t + \Delta t) = \mathbf{u}(\mathbf{x} - \Delta t \mathbf{u}(\mathbf{x}, t), t). \tag{2.32}$$

The point $\mathbf{x} - \Delta t \mathbf{u}(\mathbf{x}, t)$ is almost certainly off-grid, and the velocity at this point will have to be linearly interpolated from the 8 nearest grid-points(for a 3D field).

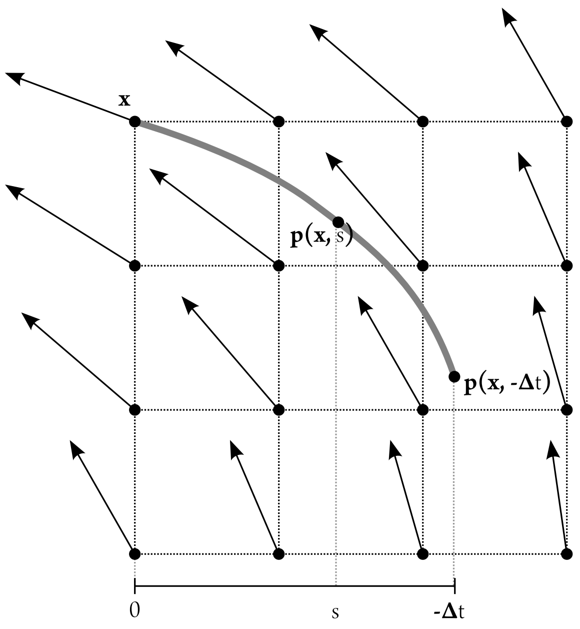


Figure 2.9: Tracing a path backward in time in the velocity field

Solve Poisson Step

To solve the poisson equation we first need the right-hand side $\frac{1}{\delta t}\nabla \cdot \mathbf{F}^{(n)}$ of Eq. 2.28. This expression says simply to calculate the divergence of the intermediate velocity field scaled by the reciprocal of the timestep. The discretization of this at each point is given by:

$$\left(\frac{1}{\delta t}\nabla \cdot \mathbf{F}^{(n)}\right)_{i,j,k} = \frac{(x_{i+1,j,k} - x_{i-1,j,k}) + (y_{i,j+1,k} - y_{i,j-1,k}) + (y_{i,j,k+1} - y_{i,j,k-1})}{\delta t \cdot h} \tag{2.33}$$

where h is the point spacing and x, y and z refers to the components of the intermediate velocity field.

When this is calculated we may proceed to solve the equation using the methods outlined on p. 20.

Projection Step

Now that the pressure field p has been computed, we can perform the last step of the calculation. The contribution of this step to the final updated velocity field is $-\nabla p$. The discrete form for the point (i, j, k) is

$$(-\nabla p)_{i,j,k} = -\frac{1}{2h}(p_{i+1,j,k} - p_{i-1,j,k}\ ,\ \ p_{i,j+1,k} - p_{i,j-1,k}\ ,\ \ p_{i,j,k+1} - p_{i,j,k-1}) \tag{2.34}$$

This step forces the velocity field to be divergence-free.

Boundary Conditions

Like Saltvik[17], we will use the *Dirichlet boundary condition* for the wind velocity field, which is to specify the value that of the field at the boundary. We will linearly interpolate this velocity between a set of predefined velocities so that the wind field does not settle into a stable state.

For the internal boundaries(against obstacles) we will use a condition such that no flow leaves or enters the domain, which we achieve by setting the velocity component normal to the domain to zero.

The Von Neumann condition is used for the pressure field. This condition dictates that the gradient is zero across the boundaries. This condition will be fulfilled by setting the pressure value at the boundary equal to that of a neighboring live(fluid) voxel[17].

2.4.3 Geometry and Snow Buildup

To make the wind and snow interesting we need scene geometry which they can interact with. In reality all objects cause disturbances in the wind field, which in turn make the movements of snow more interesting in their proximity. Many surfaces will ratain snow that impacts them, and this will slowly build up and for a white carpet covering the objects.

To solve this problem Saltvik[17] used the method of overlaying snow matrices from[12]. Here any surface that can receive snowfall is overlayed with a triangle matrix that will represent the covering snow. It starts out invisible, but as more snow hits it, it will gradually fade into whiteness and its vertices slowly grow vertically. In [17] and [12] these matrices are used to cover planar rectangular shaped objects, like a flat ground and rooftops. A modified version will be adopted here that is simplified in some ways and further develped in others.

Instead of useing planes and boxes for geometry we will be using a terrain

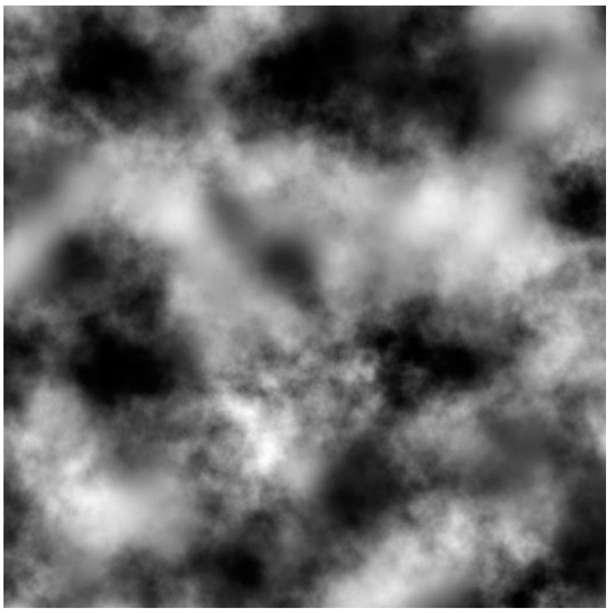


Figure 2.10: A terrain height map

height map that spans the entire scene. A terrain height map is a triangle matrix similar to what Saltvik used, but it will start out with its vertices offset vertically to make it look like a piece of terrain. The offset values will be read from an image such as the one in Figure 2.10. That particular height map was generated using an application called Terragen[3].

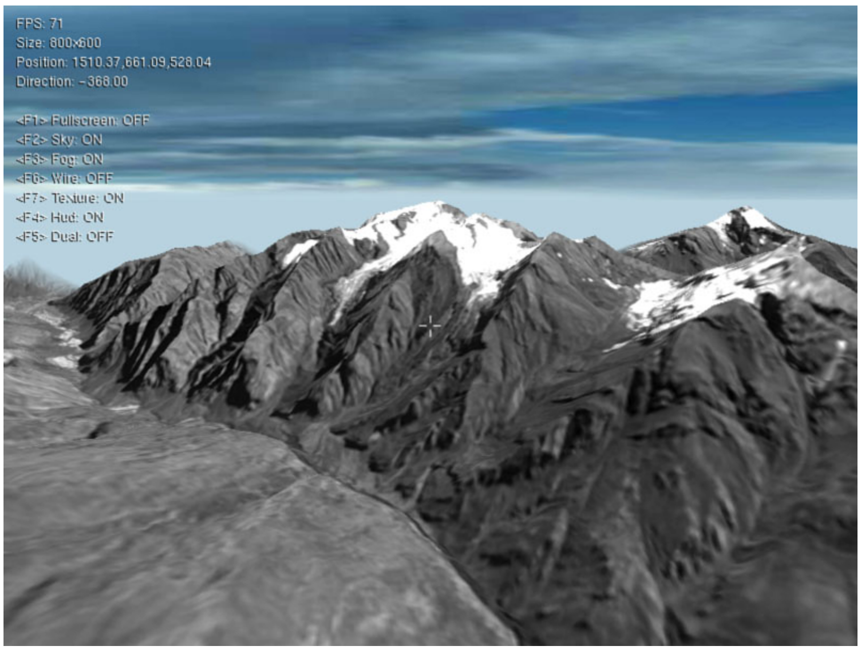
Figure 2.11 show an example[4] of a height map visualization, and it shows that very compelling scenes and landscape may be modeled using this simple design.

By using this for our geometry, we will be able to make certain assumptions that enable us to efficiently do some things that were not done in[17]:

- We know that the height map spans the whole scene, so for a particle within scene boundaries, collision detection reduces to an array lookup in the vertex array, and we can easily take built-up snow into account. This also lets us increase terrain resolution without running into problematic scaling issues.

- For each vertex we know precisely what geometry neighbors it, so we

[3]www.planetside.co.uk/terragen/
[4]http://web.iiit.ac.in/ shiben/cgi-bin/site/projects/terrain.jpg



```
FPS: 71
Size: 800x600
Position: 1510.37,661.09,528.04
Direction: -368.00

<F1> Fullscreen: OFF
<F2> Sky: ON
<F3> Fog: ON
<F6> Wire: OFF
<F7> Texture: ON
<F4> Hud: ON
<F5> Dual: OFF
```

Figure 2.11: Example height map visualization

are able to move snow that is located in steep areas, or allow a big pile of snow to sink.

The second point is similar to Fearing's *stability criterion*[10]. However the implementation is tailored for real-time use, and will not resolve all instabilities at each invocation. Instead an iterative, incremental approach is taken, where snow is gradually moved from unstable positions to stable ones from frame to frame. For each terrain vertice we compare the slope of the lines to its neighbors to the angle of response of the snow(AOR). If it surpasses it we move a fraction of it to neighbors that are located down-slope. The AOR represents the static friction of the material, but unlike reality, when this is surpassed it will not trigger an avalanche. Instead snow will continuously be incrementally redistributed whenever the stability criterion is not fulfilled, and the process will stop immediately when it is. The slopes considered will include both the slope of the underlying terrain, as well as the slope introduced by unevenly distributed snow.

Chapter 3

Implementation

In this chapter we describe specifics of how our snow model is implemented. We begin by describing the libraries and languages forming platform on which the application is built, and we proceed to explain what is done to achieve good parallel performance on the GPU for each simulation component. We also explain what is done to achieve convincing visual results.

3.1 Platform

We have chosen NVIDIA GPUs as target for the implementation due to the availability of the featureful and mature CUDA language for performing general purpose computation on them. It may have been possible to use the Cg shader language but CUDA is specifically tailored for general computation and is often more suitable[8]. Another alternative for writing data-parallel programs for the GPU is the vendor independent OpenCL API, but at the time of writing it is still very new and hardware support is somewhat lacking. CUDA is based on C/C++ and the GPU kernels must be written in this language. For easy interoperability, the rest of the application was written in C++. In the end the choice of CUDA proved a successful one.

Preferring to stay platform independent we selected the OpenGL API for our graphics rendering purposes. CUDA has an interface to both Direct3D and OpenGL, but Direct3D is only available on the Windows operating system, while OpenGL is also available on other systems like Linux and MacOS X, both of which also support CUDA. OpenGL alone does not provide input and window handling, so this handled by different libraries. A lightweight library called GLFW[1] was used here, and it is portable to a wide variety of platforms. It also provides threading primitives, but this has not been used, since no computation is performed by the CPU.

The native OpenGL header files on most platforms only support older features, and newer features need to be handled through the OpenGL extension mechanism. It allows functions to be dynamically resolved at runtime from name strings. A portable library called GLEW[2] has been used to simplify this.

Instead of using the OpenGL fixed function pipeline, the programmable pipeline was used. This allows much more flexibility in determining how rendering is to be performed. The OpenGL Shader Language(GLSL) is used to program the vertex and fragment shaders.

3.2 Rendering

There are two components involved in the rendering of the scenes in this simulation. The first is the geometry that shall represent both the ground and the snow covering it. The second is the snow particles themselves. For both of these, care has been taken to keep rendering overhead to a minimum by making sure data is batched and located on the GPU[4].

3.2.1 Geometry and Ground Snow

Geometry in this simulation will be represented by a height map. It is implemented in OpenGL as a simple Vertex Buffer Object(VBO), with a vertex for each grid point in the height map, and an index buffer defining triangles over these vertices.

VBO is the OpenGL term for a data buffer containing vertex data residing in GPU memory. Using a VBO allows us to avoid costly transfers of geometry data from system memory each frame, and rendering it is simply a matter of a few function calls, even if it contains millions of vertices. This is very advantageous if the data is static, and will not change from frame to frame. In our case we actually want the terrain to be dynamic to facilitate snow buildup, but as we shall see CUDA has a feature which combines with the

[1]http://glfw.sourceforge.net/
[2]http://glew.sourceforge.net/

OpenGL VBO to make this very straight forward and fast by letting compute kernels directly operate on VBOs.

For this part of the application we are able to significantly reduce overhead compared to Saltvik[17], where geometry was recalculated on the CPU each frame, and then transfered to the GPU for rendering. In comparison, once our VBO has been created it is never again modified by the CPU, and so we avoid expensive bus transfers. The geometry is also never recalculated from scratch, but instead it is modified as snow impacts it. For shading we use a precalculated normal map that is never updated. A normal map is a texture that holds normal vectors in its RGB components. If the geometry is significantly changed by snow-fall the normal map will not reflect this, and this is considered a weakness in the implementation. However, we do not estimate the cost of recalculating this per frame to be significant, and this is a relatively small change that could feasibly be made. By switching to vertex normals as in[17], it would also be possible to only modify normals that are affected by changed snow depth, and in this case the cost would be close to zero.

Representation of Snow Cover

The actual vertices are 4-component vectors, with the first 3 being the spatial coordinates(x, y and z), and the fourth component(w) being the snow depth at this location. Storing this in the position vertex allows us to leave it entirely up to the vertex and fragment shaders how the snow will be visualized. The strategy that is currently used to render the snow is to gradually blend a snow texture over the ground texture as snow cover gets higher, and then when it exceeds a certain threshold the vertical position of the vertices will be modified(Figure 3.1). So the vertex shader produces a possibly modifed position vertex, and computes a blend factor α that will be interpolated across the triangles. The fragment shaders then blend the two textures based on this interpolated factor.

Vertex Indexing

One way of representing geometry data is to simply say that every triplet of vertices in the vertex buffer defines a triangle. This however, will lead to duplication of vertices that are shared by many triangles. Duplication is unwanted because it means wasted space, and because it means modification

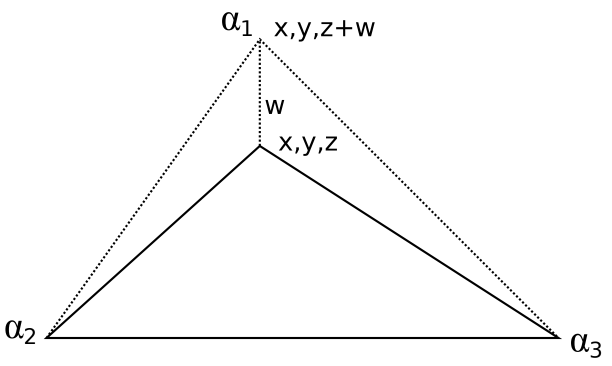


Figure 3.1: Triangle with blend factors and a modified vertex

of any one terrain vertex has to touch many actual vertices in the array.

The alternative, as mentioned, is to use an auxiliary index buffer to define triangles, by saying that every triplet of indices in this buffer defines a triangle with the its vertices being the corresponding vertices in the vertex array. This is the simplest indexing scheme, and may be improved upon.

An more attractive alternative is the *triangle strip* indexing scheme, wherein a triangle is defined by one new index as well as the previous two indices(except at the very beginning). This allows us to store approximately $k + 2$ indices for k triangles instead of $3k$ indices. By using triangle strips in a straight forward manner we can only cover one row of triangles for each strip. A trick will enable us to glue row strips together and use one strip for the whole terrain[9]: Duplicating the last index at the end of a row and the first index at the beginning of the next row(Fig 3.2). This trick will generate four *degenerate*(zero area) triangles which will move the strip to the beginning of the next row. In the figure the extra indices are numbered 8 and 9, and they have been moved from their actual locations(on top of 7 and 10) to show the degenerate triangles(dotted lines), which of course would be invisible otherwise. The index buffer can also be uploaded to GPU memory in the form of a Vertex Element Buffer Object.

3.2.2 Snow Particles

The particle rendering routine utilizes an OpenGL feature called *point sprites*. OpenGL has traditionally had support for *points*, that would be represented with a single vertex and rendered as a single colored square(camera facing

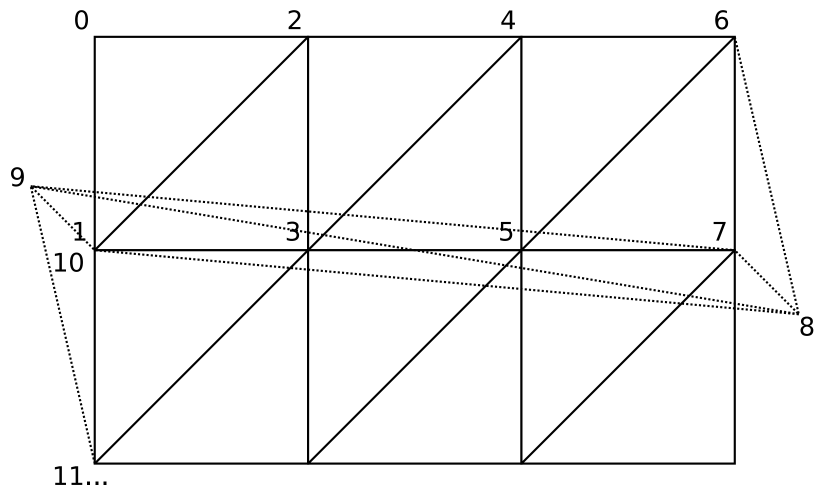


Figure 3.2: Triangle strip across multiple rows using degenerate triangles

quad) ranging from a pixel in size an upward. Point sprites is a a hardware supported feature for rendering *textured* quads facing the camera, without having to store its four corner vertices, and without manually having to transform them to achieve correct orientation. This is just what's needed for the potentially millions of snow particles that we want to render. Another name for this type of object is *billboard*. Figure 3.3 shows an example texture[3] that might be used for texturing the billboards.

Using this, we can store all center positions in a large Vertex Buffer Object, and render them with one draw call. This VBO may be directly manipulated by a CUDA kernel, so like with the terrain geometry we are able to isolate both particle update and rendering from the CPU, and do everything on the GPU.

The use of point sprites results in snow flakes that look more realistic than in[17], and they blend together in a better way when overlapping, due to their alpha transparency. In contrast, Saltvik used three white intersecting quads which were rendered using OpenGL's immediate mode(where vertices are submitted to OpenGL one at a time by the CPU). This resulted in both appearance and performance suffering compared to our approach. Tests show that our implementation is able to handle millions of snow particles.

[3]http://www.iayork.com/MysteryRays/2007/08/29/snowflakes-in-a-blizzard-counting-t-cells/

Figure 3.3: Snow particle texture example

3.2.3 Stereo Rendering

Some screens available today have the ability to display alternating rows of pixels with light of opposite polarization. If we render two different images, with one image on even rows and one displayed on odd rows, we can effectively display a stereo image. 3D glasses only let light of one polarization through to each eye, and thus each image will be seen by only one eye.

Our brains are able to estimate the distance to an object that our eyes focus on. It can do this because of the parallax provided our pair of eyes' different positions. In short, faraway objects will have moved less than nearby objects when comparing the image from one eye that of the other.

We can utilize the stereo functionality of a monitor to enable us to percieve depth in 3D graphics. This can be done by rendering the scene at half vertical resolution twice, each time from the position of each eye. Then interleaving the rows of each image as we draw a frame to the screen, we end up with an image in which we can percieve depth.

OpenGL makes this easy with its Frame Buffer Objects(FBO) which allow us to set up textures as render targets. The scene is rendered twice as described, to two different textures. To interleave their rows we render a fullscreen quad using a GLSL shader which samples the two textures. It decides which texture to sample based on the y-coordinate of the fragment. The *modulo* operator or bitwise *and* is not supported in GLSL so we emulate `if(y%2)` by

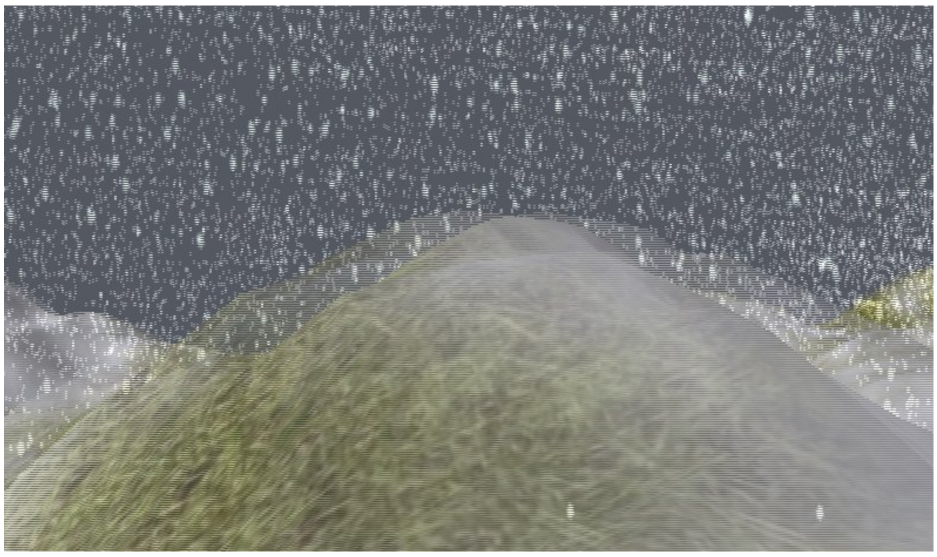


Figure 3.4: Example stereo rendering

writing `if(int(y) * -2147483648 == 0)`. This exploits the fact that since integers here are 32-bit, for an even `y` the expression overflows to zero, and for an odd `y` it overflows to itself(-2147483648).

As we see in Figure 3.4 this results in two slightly different points of view that are alternated. We see the same objects duplicated but seen from a different angle on alternating lines. The effect can be seen to be strongest closer to the camera, as expected(the two white dots on the bottom right are actually the same snow flake).

3.3 Simulation Components

Here the implementation of the two major parts of the simulation is discussed: The particle simulation, and the wind simulation. The particle simulation includes the movement of particles, as well as the handling of collisions and snow buildup. The wind simulation component covers everything required to compute the velocity field, including obstacle management. In both parts we will discuss some CUDA-specific issues that we face.

3.3.1 Particle Simulation

Updating all the snow particles is an embarrassingly parallel problem; there are no dependencies between any pairs of snowflakes. This is the ideal situation that we want to be facing when equipped with a GPU, which excels at just this kind of operation. Note that while in reality there are dependencies between snowflakes since they can collide, we ignore that possibility here.

The snow particles are represented by arrays in GPU memory which hold all the properties of each particle. An interesting part of this is the position property of the particles, which is also needed by OpenGL for rendering purposes. We will exploit CUDA functionality which allows OpenGL and CUDA to share a VBO located in device memory. Thus we can invoke the render operation directly on the computed position vector, without having to copy or modify any data.

We divide the job of updating the particles evenly between a number of thread blocks, each handling a set of contiguous particles, and allocate one thread per particle. The threads then load all required properties in a series of coalesced reads from their respective arrays.

Not all per-particle properties in the mathematical model are really required to be unique for each. Three examples are mass, rotation speed and rotation radius. Instead of allocating storage for these on a per-particle basis, an array of 32 different values is allocated in constant memory. Each thread indexes these array by its global index modulo 32. This provides the needed variation between individual snow flakes.

The position and the velocity are the only two properties that are allocated their own space per particle. These were four-component vectors from the beginning for alignment reasons. However only three components are needed, so the fourth(w) component of each may hold one floating point attribute. The position.w slots are used to hold the rotation angle that the particle has reached in its circular movement. The velocity.w slots holds the the value of the $\frac{1}{V_{max,z}^2}$ expression for this particle(see p. 2.4.1).

A fixed number of particles will be simulated at all times. No particles are actually removed even when they collide with the ground. Instead they are moved to a random place in the sky, so as to resemble more incoming snowfall.

Wind Field Lookup

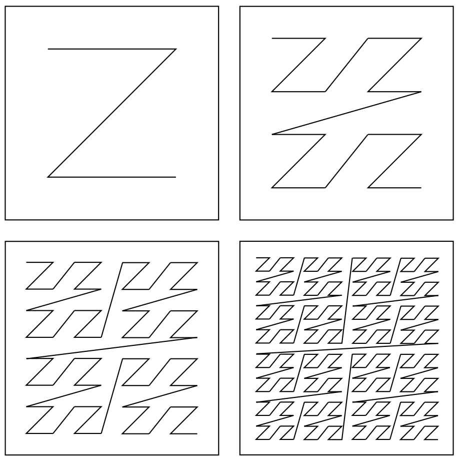
To update each particle, we need know not only its attributes, but we also need to sample the simulated wind velocity field at the position of the snowflake. The velocity field is a discrete set of vectors, while the particle positions vary continuously in all three dimensions. Therefore we need to perform a triliniear interpolation of the 8 nearest velocity vectors, based on the particle position. Lookup will necessarily involve as much as 8 suboptimal random access to device memory for each particle.

Instead of using a flat array to store the velocity field, we can use a 3D texture. Texture sampling in CUDA is backed by the hardware texture units, which basically gives us interpolation for free, as well a cached access. Access is cached based on spatial locality. The particular implementations used in modern cards are proprietary, but spatial locality is often achieved by use of a special space-filling curve such as a Z-order curve, as opposed to ordinary linear ordering(Fig. 3.5). In contrast the CPU implementation by Saltvik[17] must perform the interpolation in software, and may also suffer from the fact that system memory is cached based only on 1D spatial locality.

To exploit this maximally, we could sort the particles in lexicographic order based on coordinates. The ordering can not be based on the unmodified coordinates, because we can't expect any particles to have the exact same major coordinate. Instead we must divide each axis into segments and sort based on which segment the particle's coordinates fall into. A simple method is to convert the coordinates to integers and mask out the lower bits. This will divide the coordinate space into cubes, and particles falling into the same cube will be grouped together in memory. Suitable segment sizes may be determined empirically.

Tests with an initially sorted array show that this does indeed boost performance of the particle update step by a not insignificant factor, but sorting of the particles on the GPU has not yet been implemented here. Figure 3.6 shows the frame rate decline as the ordering of the presorted array more and more fails to reflect the spatial relationship between the particles. This test was done on a 256x256x32 wind field with 512K particles on an NVIDIA 8800 GT.

The repositioning of particles when they reach boundaries(either terrain or scene edges) will have en impact on a sorting scheme like this. It may be possible for a repositioning routine that is aware of the cube division to minimize decay in the ordering, but it is unclear whether this would lead to


(a) Z-order curves of order 1-4

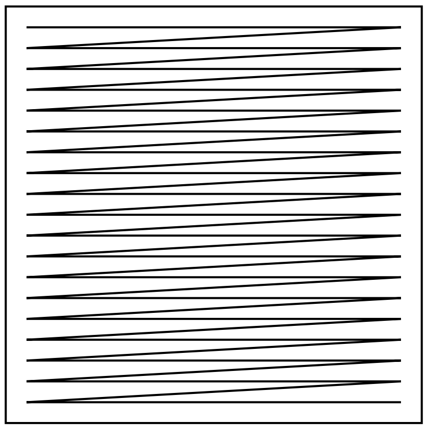

(b) Linear ordering

Figure 3.5: Z-order curves vs. normal linear ordering for spatial locality

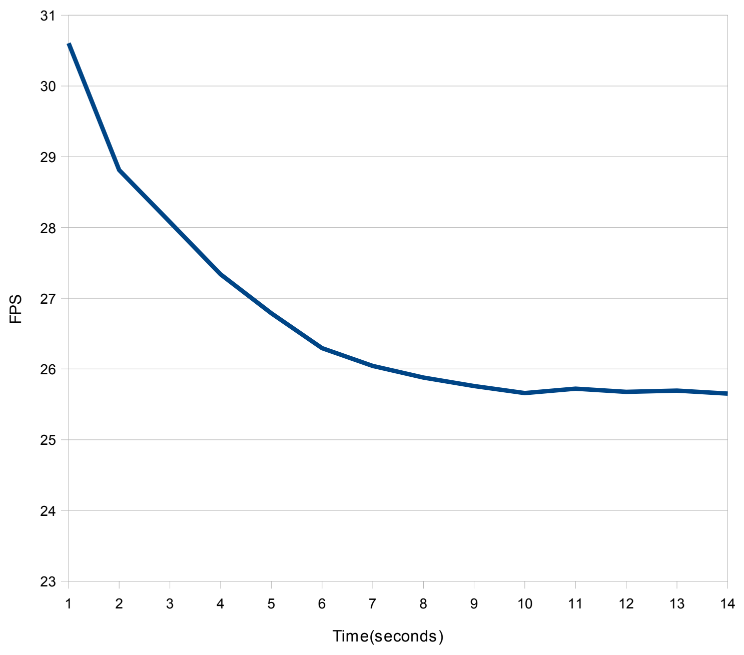


Figure 3.6: FPS decline as spatial particle ordering diminishes

non-uniform looking spawn patterns. Th best solution may prove to be more frequent sorting.

Geometry Collision

We also need to check for collision against the local geometry. Given that the only geometry that will be collidable is a height map, we can do this quite efficiently. Checking for collision in this case merely involves converting the x and y coordinates of the particle position to indices into the terrain vertex array, and then reading one vertex. If the particle's height is lower than that of the vertex, there is a collision(the height of the vertex is its z-coordinate plus its w-coordinate which holds the snow depth). Now we need to respawn the particle elsewhere, and increase the snow depth at this location.

Not all particles have to perform an uncoalesced read into the terrain vertex buffer. One simple optimization we can do is to calculate the the height of the highest terrain vertex, and the only do the collision check for particles which fall below this value. This still leaves a lot of reads. A second possible optimization is to generate a 2D height map texture based on terrain height values. Reading from this texture will be faster than the uncoalesced read. Especially if the particles are sorted, because then the texture cache will be maximally exploited. The texture lookup optimization has not been implemented.

Particle Repositioning

When ”dead” snowflakes need to be respawned we need to determine the position to which they will be moved. Merely modifying their z-component to move them to the top of the scene again will not be sufficient, as this will result in unrealistic patterns in the appearance of new snow.

The obvious answer is to not only move the particles up again, but also to randomize their planar(x and y) coordinates in a way that will produce uniform snowfall across the scene. This is not entirely straight forward on the GPU, as a random number generator is stateful in that it maintains a seed variable, which is updated upon generating a number. This seems highly incompatible with the GPU model where may have many threads simultaneously requesting random numbers. This problem has actually been solved[14], but we were looking for a more light-weight solution.

Instead of using a random number generator as a source of randomness, we use inherent noise in the floating point representation of the position of each particle. The method has no specific formal basis and was developed by trial and error, but the resulting distribution appears uniform and pleasing to the eye.

Here is the actual code used for repositioning a particle(note that in the code the vertical axis is y and not z):

```
__device__ float4 reposition(float4 pos) {
    int or_term = 0x3F000000 | ((*(int*)&pos.y & 0xFF) << 2);
    float temp = 4.0f * (float)SCENE_X;

    int ival = ((*(int*)&pos.x & 0xFFF) << 10) | or_term;
    pos.x = (*(float*)&ival - 0.5f) * temp;

    ival = ((*(int*)&pos.z & 0xFFF) << 10) | or_term;
    pos.z = (*(float*)&ival - 0.5f) * temp;

    pos.y = (float)(SCENE_Y - 2);
    return pos;
}
```

For this to work the velocity numbers must be a source of noise at all times, and to ensure that this is the case the particles start with low non-zero random velocities along all axes. If we don't do this then initial collisions will cause the particles to be repositioned in very obvious non-uniform patterns.

3.3.2 Managing Snow Buildup

When a collision is detected between a particle and an the terrain geometry, an expected consequence is that there be deposited snow at the location of impact. One solution is to simply increment the snow depth at the nearest vertex by some fixed amount. For coarse terrain grid resolutions this produces reasonable results, but for finer resolutions the density of collisions may not be high enough that snow is distributed in a nice uniform manner across the ground.

If snow buildup per collision is very low, snow levels will even out before changes start to become visible, but a compromise will allow for higher buildup rates without visual artifacts. Instead of only increasing the snow depth at the nearest vertex, we may add snow to the nearest 9 for example (with most buildup in the middle, and least for the corner vertices).

This works fairly well for cases where snowfall is distributed fairly uniformly across the surface, but wind phenomena may often cause more snow in certain areas. This will lead to a dramatic buildup of snow at these locations, which will look unnatural after some time due to the absence of any mechanism for redistributing snow.

A simple kernel was devised to avoid this problem. It is based on an idea that resembles the stability criterion in[10]. Each frame the kernel will compare the height and snow depth of each vertex with that of its neighbors, and possibly adjust the snow depth value by some small amount. The first precondition for adjustment is that the height difference Δh between vertices (where height includes snow depth) is greater than some threshold t. The second is that the highest vertex(from which snow will be drawn) has more snow than a threshold value m. If these hold true, then a certain fraction k will be taken from the snow of the higher and added to the lower vertex. Only the amount of snow s that is actually above the receiving vertex is considered when calculating the amount to transfer.

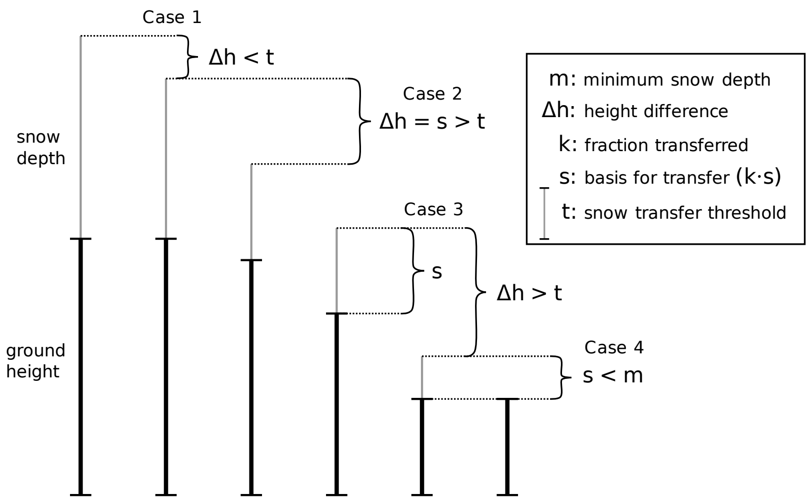


Figure 3.7: Snow transfer illustration

This is illustrated in Figure 3.7, where four case examples are shown where snow transfer from four vertices to their right neighbor is considered:

1. No transfer because $\Delta r < t$.
2. $k \cdot s$ transfered where s is less than the total amount of snow for the vertex.
3. $k \cdot s$ transfered where s is the total amount of snow for the vertex.
4. No transfer because total amount of snow is less than m.

The Δh threshold condition allows for some slope in build up snow, while total snow $> m$ condition ensures that the ground will not be stripped bare by snow transfer. By transferring only a small amount per frame, we ensure that the effect is virtually imperceptible, but across many frames snow will slide downslope and build up in low points, which leads to an evening out of the snow levels.

This method does not model all forms of migration for fallen snow, and what it does model it merely approximates. Tweaking the thresholds t and m is necessary in order to achieve a believable effect, and values for both have been found by trial and error. A possible improvement would be to also take wind into account to model wind-driven snow drift. It may also be feasible to delay redistribution until a certain stability level is reached, whereupon snow will be redistributed until it reaches some lower level. This might be enough to model avalanches, but that remains speculative at this point.

3.3.3 Wind Simulation

This is at the core of the simulation, and everything depends on a good implementation. Many different components come together to make this work. We need a way to handle obstacles in the scene, and we need to implement many different kernels representing different parts of the fluid simulation. Some of these are similar, and some are not.

The wind simulation operates on a set of core data structures that are 3D arrays covering the domain volume:

Velocity field A vector field containing wind flow velocity, primarily represented by a 3D array of float4 values. For each frame this array is copied into a 3D texture, to enable cheaper random-access lookup and interpolation, but this can not be the main velocity data structure since textures in CUDA are read-only(except the copy operation needed to initialize them).

Pressure field A scalar field containing pressure values, represented by a 3D array of floats.

Solution vector A 3D array of float values representing the right-hand side of the Poisson equation governing the pressure field. Calculated each frame from the velocity field.

Obstacle map This is a 3D array of int values. The first bit of each int is set when the cell is not a fluid cell(it is an obstacle cell). Each of the next 26 bits represent whether or not the corresponding neighbor cell is an obstacle or not. By storing the neighbor information in each cell we avoid redundant loads when this information is needed.

Before development started on the kernels forming the simulation, a set of functions for visualizing the information in these data structures were made. This have both been used to facilitate debugging during development, and as windows into the running simulation for no other reason than to see what is happening. Figure 3.8 shows the pressure field visualized, with yellow areas being high pressure, red areas low, and everything else something in between. A CUDA kernel will generate an array of 4-component vectors, one for each voxel cell, with the xyz-components holding position and the w-component holding pressure. A GLSL shader colors the points based on the pressure. In Figure 3.9 we see the velocity field which is also created using a CUDA kernel and shaded in GLSL. Vectors pointing from low to

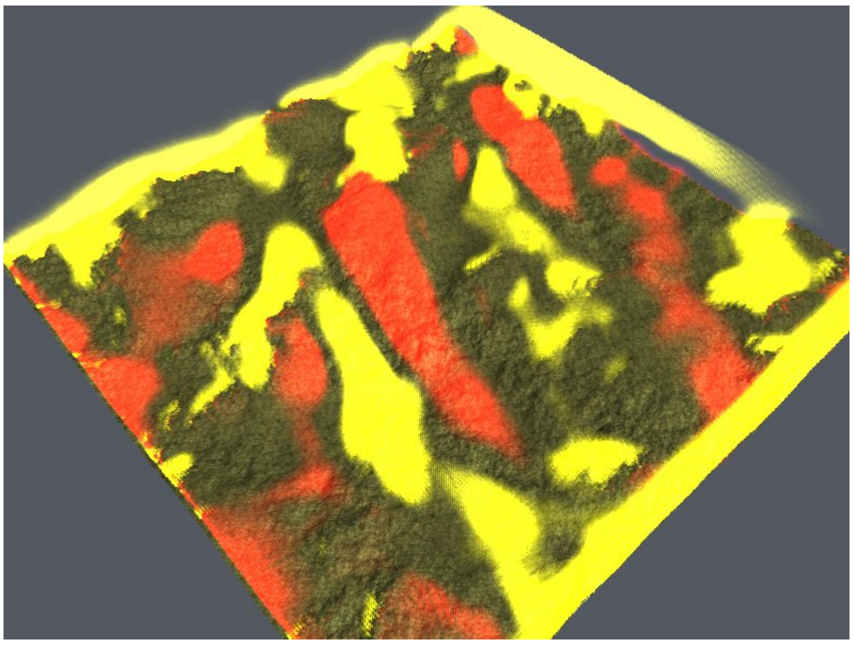


Figure 3.8: Rendering the pressure field

high velocity cells are yellow, and for the opposite case they are red, with a continuum in between. Finally we see in Figure 3.10 the obstacle field, showing to us how the scene geometry have been voxelized.

Domain Decomposition in the Wind Field

Common to all steps involved in the wind simulation is that we need to operate on individual values in three dimensional arrays representing the domain of the simulation. The computation domain in CUDA is subdivided into blocks, each of which runs on a single multiprocessor.

The decision of how to subdivide the domain rests on a variety of factors; specifics of the problem at hand dictate how to handle the borders between blocks; the dimensionality of the problem affects our choices for block shape. In our case most of the kernels will for each voxel need the values of the six directly neighboring voxels along the three axis lines.

An important consideration that we need to make is that memory access is best organized into sequentially aligned batches of 16 individual 4, 8 or 16-byte values. This corresponds to the 16 threads that are simultaneously active in a half-warp, whose memory accesses will be coalesced under these

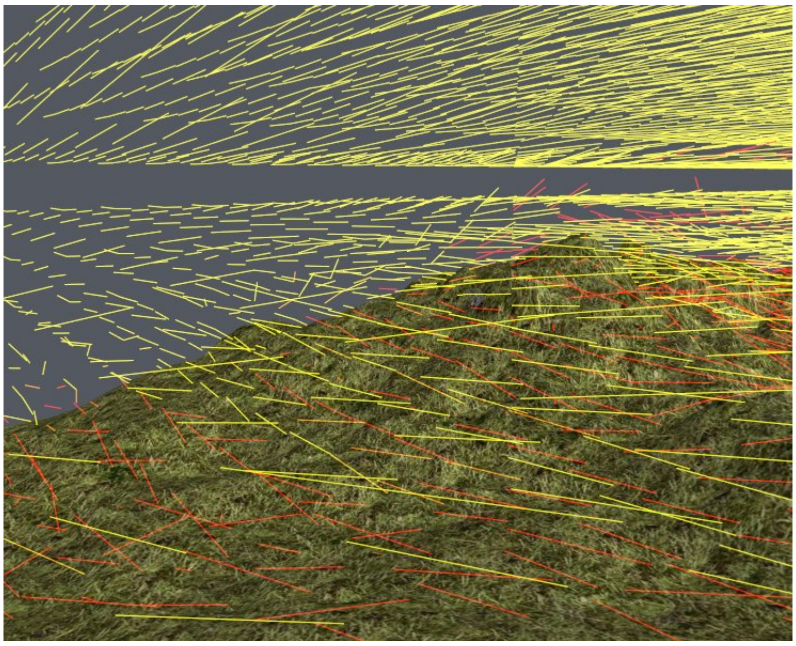


Figure 3.9: Rendering the wind velocity field

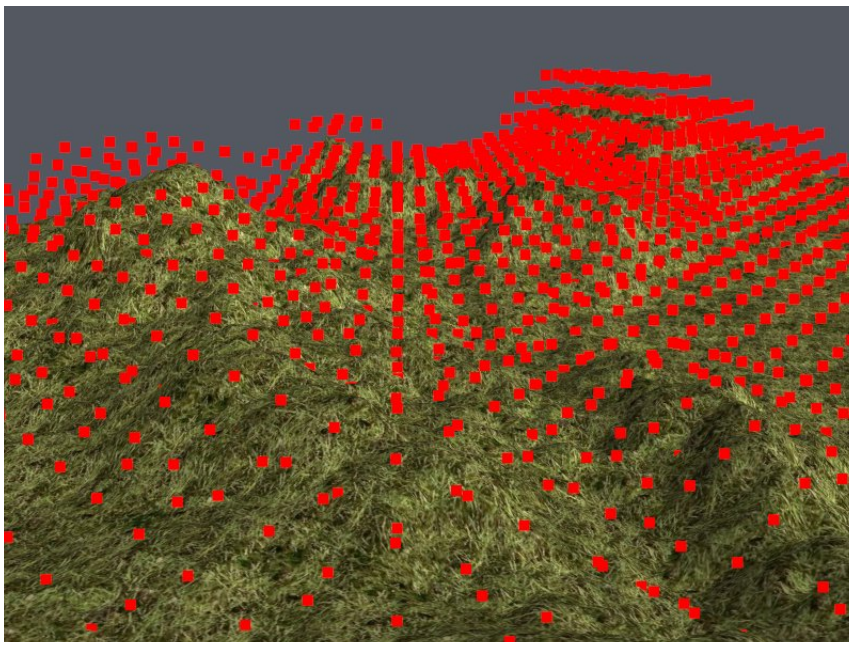


Figure 3.10: Rendering the obstacle field

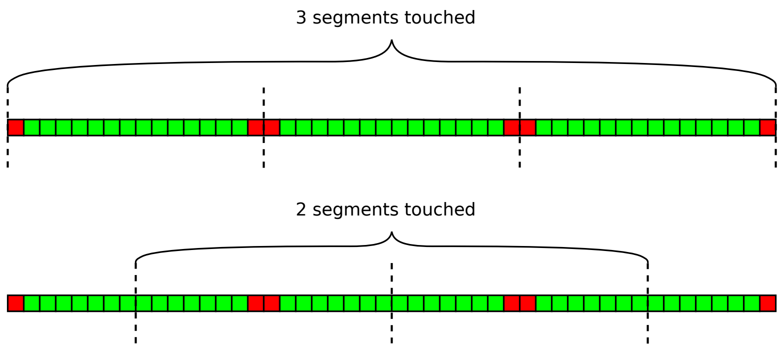


Figure 3.11: Borders across segment boundaries

conditions.

The restrictions posed mean that our blocks should be a minimum of 16 voxels wide along the main X-axis where sequential coordinates map to sequential memory locations. Another vital point is that one thread reading one voxel will take approximately the same time as 16 consecutive threads doing a coalesced read, due to the latencies involved. This has implications for how we must handle borders between blocks. We note that we will need one value from the neighboring block on each side. If our block size along the major axis is a multiple of 16 and start and end at this boundary, we will need to "waste" one whole transaction for one value. If it starts at one such boundary, we need one transaction for the one border value there also. Having the blocks start and end in the middle of a segment will minimize the "potential bandwidth" waste. These two scenarios are illustrated in Fig. 3.11.

One scheme which avoids this waste altogether involves storing the vertical borders of each block sequentially in a separate array(Fig. 3.12). Since horizontal borders can be read in one single coalesced read for each border, these are given no special treatment. Given block dimensions of 16x8 as in the figure, we can also read both the vertical borders in one coalesced read if these are stored as described. That is, for each column of blocks we store one row in an auxiliary border array. The width of this array needs to be twice the combined height of blocks in a column, since two vertical borders will be stored for each. The height must be the number of block columns. This method necessitates that each thread block must write the values from its interior vertical borders to its two neighbors' slots in the border array, in addition to storing its results back normally.

A compelling option is to designate one block to each whole row in the layered

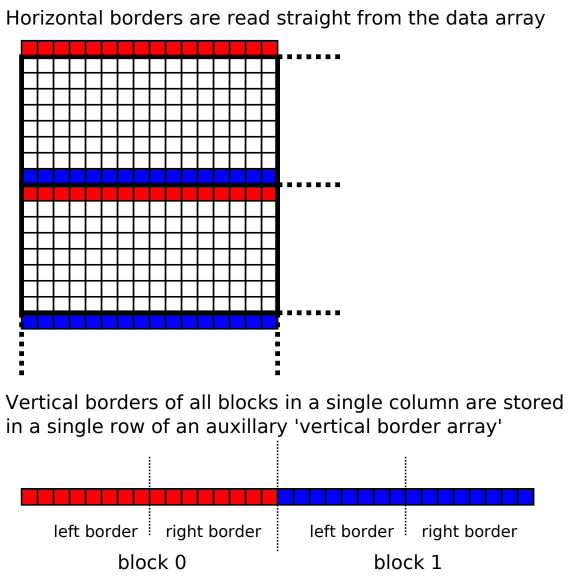

Horizontal borders are read straight from the data array
Vertical borders of all blocks in a single column are stored in a single row of an auxillary 'vertical border array'
left border
right border
left border
right border
block 0
block 1

Figure 3.12: Auxiliary array for vertical borders

planes that constitute the volume. This is the first configuration that was actually implemented. It is among the simplest, and while it may not be optimal, it is attractive for several reasons. The first reason is, as mentioned, simplicity: It allows us to entirely disregard complexity introduced by borders along the major axis. Also, by reading the whole row into shared memory it does still reduce memory overhead somewhat by making two of the six surrounding voxels immediately available at each point. This means that each cell is read 5 times instead of the full 7 times(once for the cell itself, and once for each of the 6 direct neighbors). Now this is hardly efficient but it works.

A modification of this scheme only allocates enough blocks to fill one single plane, and then each one iterates from the bottom of the volume to the top. While doing this we can retain immediately preceding voxels in registers, and thus avoid duplicating reads to obtain neighbors in the vertical direction(without requiring more shared memory). This results in each voxel being read a total of 3 times. Rewritten kernels using this iteration scheme caused a 66% increase in wind simulation performance on a test with a 256x256x32 field. Variables other than memory access pattern may have varied between the versions so this performance increase may not be wholly attributed to the vertical iteration scheme.

Below is pseudo-code for thread `i` in a block running this vertical iteration

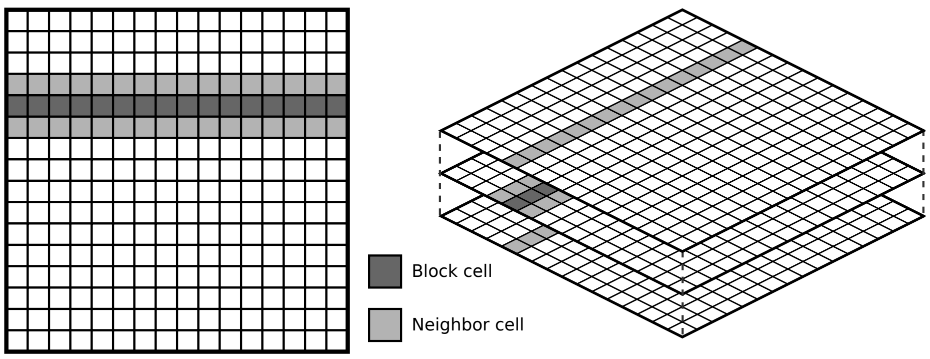


Figure 3.13: Block distribution

scheme. Note that `read` reads from global memory and `calcX` does the required calculation using the stencil values in the X directions.

```
shared[i] = read(x,y,0);
prev = shared[i];
shared[i] = read(x,y,1);

for(z = 1; z < Z_MAX-1; ++z) {
    calcX(shared[i+1], shared[i-1]);
    calcY(read(x,y+1,z), read(x,y-1,z));

    temp = shared[i]
    shared[i] = read(x,y,z+1);
    calcZ(shared[i], prev);
    prev = temp
}
```

Advection Step

The self-advection for the wind field involves using the velocity in each voxel in the field to produce a new point that is transported along the negative velocity vector. We do an interpolated sampling of the wind field at this new point, and store the sampled vector in place of the original voxel value.

The velocity vectors at the grid points are read using coalesced reads from a normal array in global memory. The block division employed is the simple scheme outlined above, where one block is one full line along the X-axis. Since only the self voxel is needed here, this should be close to optimal. Vertical iteration is unnecessary since we don't depend on the value of neighbors.

Having produced new positions using the obtained vectors, we need to sample the field again to produce the updated velocity value. This time we are sampling off-grid so we need to perform linear interpolation on the eight nearest grid points. It would be quite costly to lookup these and then do the interpolation manually. Instead we can use CUDAs support for 3D textures, which enables us to utilize hardware crafted to do this exact thing as efficiently as possible. We have already created a 3D texture from the last state of the wind field, for the purpose of updating the snow particles, and we can reuse this same texture here. Now what remains is to to a series of coalesced writes back to the original array.

It is worth noting that since we here write all the cells in the wind velocity volume, we write the boundary values at the same time, in order to avoid a separate stage for that. We also set the cells in the pressure field to zero(the initial "guess") to prepare for SOR iteration.

Other Simulation Steps

The wind simulation requires a number of additional steps in addition to the advection step:

Build solution This calculates the right-hand side of the Poisson equation.

Solve poisson Each iteration of this updates the pressure field according to Eq. 2.19. Saltvik[17] used an ω value of 1.7 but this leads to instability(like in Figure 3.14) here for reasons that are not clear. It has been found that using a list of relaxation factors such as $1.7, 1.5, 1.2, 1.1$ and 1.1 for over 5 iterations does lead to a stable simulation as opposed to using something like 1.7 for all, and it results in a more responsive wind simulation than using a low relaxation factor for all.

Set pressure boundaries Enforces the bondary condition of the pressure field. Performed once per Poisson iteration.

Project velocities Modifies the velocity field using the gradient of the computed pressure field.

All of these except *Set pressure boundaries*, access some of the array data structures with a 5-point stencil per cell in order to perform its function.

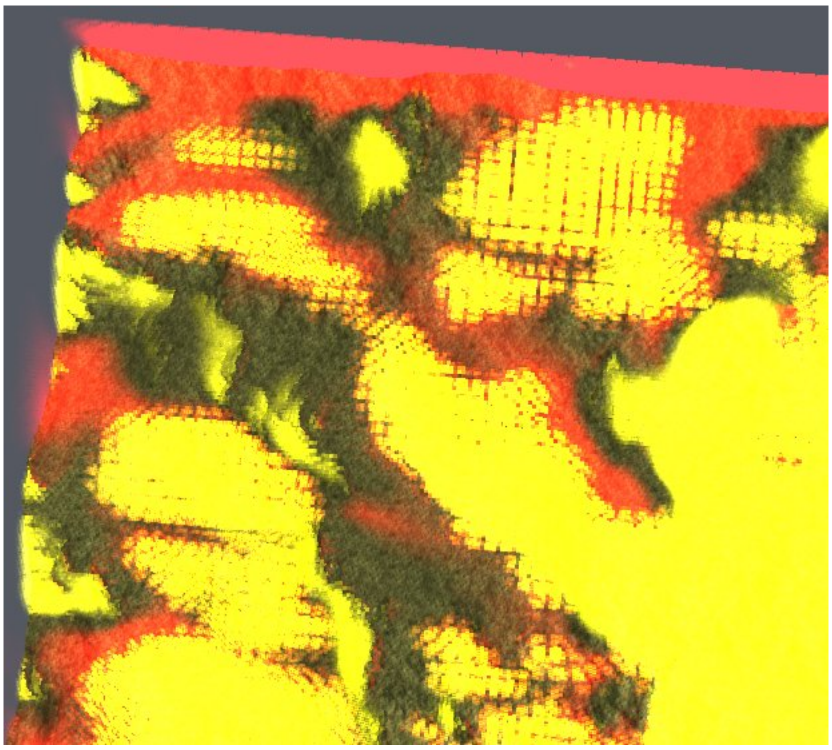


Figure 3.14: Numeric instability as seen in the pressure field

For these the domain decomposition and vertical iteration scheme explained previously is used. Since *Set pressure boundaries* does not read neighboring values(except at boundaries) it does not use vertical iteration.

Updating the Obstacle Field

The obstacle field conveys to the fluid simulation which cells are within obstacles and which are not. The simulation kernels will only run on cells that are outside of obstacles. Since the geometry of the scene is subject to change due to the buildup of snow, this field has to be updated to reflect this (this is not done by Saltvik). This may be done periodically since snow buildup is slow compared to typical voxel cell dimensions.

Right now it is done in the CPU, and it is in fact the only simulation component that CPU handles. We do this in a way that does not impact frame rate at all. One update cycle is divided into a series of steps:

1. Copy terrain geometry from GPU

2. Zero local obstacle map memory
3. Set *self bit* of each cell to 1 if below terrain(plus snow), 0 otherwise.
4. Set the remaining 26 bits to 1 or 0, according to which neighbors are obstacles.
5. Copy obstacle map to GPU

In addition to this step division, step 3 and 4 are divided into substeps with one layer in the volume computed per substep. The implementation currently performs one step per 0.1 seconds. This means that for a wind field 16 voxels high, there are 35 steps per cycle(the obstacle map is updated once per 3.5 seconds).

3.4 Performance Tuning with CUDA

The GPU multiprocessors provide limited resources that have to be shared among all active threads that are executing on them. There is much to be gained by controlling our usage of these resources, so that more threads may run simultaneously. We will here explore some techniques that we use when profiling and optimizing CUDA code.

If we invoke the compiler with the parameter '-cubin' when compiling our CUDA code the compiler creates a file with the cubin extension that contains some useful information on GPU program. Especially of interest are the sections for each compute kernels, where three values are of great importance. Here is part of an entry for a kernel:

```
code {
    name = __globfunc__Z11wind_advect14cudaPitchedPtr4dim3f6float4
    lmem = 16
    smem = 64
    reg = 8
    ...
}
```

The values to watch are `lmem`, `smem` and `reg`. These list the amount of local memory, shared memory and register space that is in use by the kernel. Ideally we want to use zero local memory, because of its slowness and it is easily achievable most of the time, but sometimes, like here, the compiler insists on generating code using it.

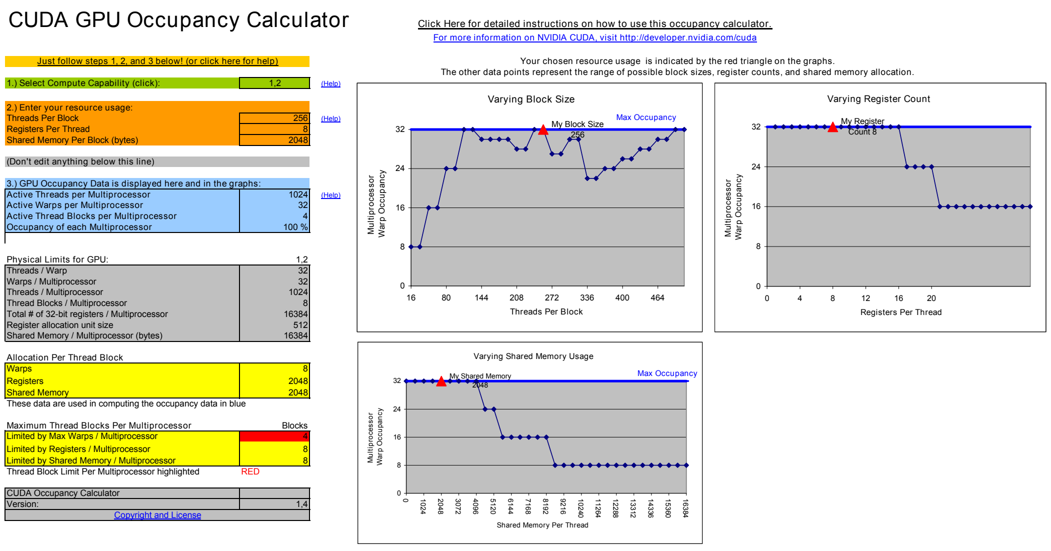
An important concept that we must consider when writing kernels is *occupancy.* Occupancy is the ratio of active warps to the maximum supported number of warps per multiprocessor. We want this number to be as close to 1 as we can. Factors that limit occupancy is the amount of shared memory and register space used. The number of active warps will be determined by the number of blocks that can be scheduled simultaneously, and this is directly determined by the resource use of each block. One block will use a specific amount of the total available shared memory, as well as a specific number of registers which will be split evenly among its constituent threads. Achieving hight occupancy then becomes an exercise in getting shared memory and register use as low as possible, and also keeping them balanced. It will not matter that register use is low if one block uses all available shared memory and vice versa[16].

To reduce register use we can evaluate our code to see if we need to use all the variables that we are using. This should be the first step, but we may not always be able to rewrite in ways that lower register use. It is useful to experiment, and watch for changes in the cubin output. If we are register constrained and no redundant variables are found we can move one or more variables to shared memory, which is just as fast. We can do this by simply allocating for each block a shared memory array of the type needed with as many elements as there are threads in the block.

It is not always easy to predict whether we are register or shared memory constrained just by looking at the numbers. To make this easier NVIDIA has released an Excel spreadsheet(Fig. 3.15) that will calculate occupancy, and display graphs of how occupancy will develop with changing resource use. A quick glance at this will tell us what we are constrained by. To use it one has to fill in the number of threads per block, as well as shared memory and register use per block. It also has an option to view occupancy for different compute capabilities, since maximum warp limit and register count has increased in newer revisions.

Also of tremendous use is the CUDA Visual Profiler application. Using this we can run our CUDA programs and get a precise account of the time use of all our kernels, as well as other statistics. Profile-guided optimization is always the best way to go, because then we can be sure of how much the problems we are addressing are actually contributing to computation time.

The profiler allows us to enable counters for certain events that are of interest



CUDA GPU Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	1,2	(Help)

2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	8	
Shared Memory Per Block (bytes)	2048	

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	1024	(Help)
Active Warps per Multiprocessor	32	
Active Thread Blocks per Multiprocessor	4	
Occupancy of each Multiprocessor	100 %	

Physical Limits for GPU:	1,2
Threads / Warp	32
Warps / Multiprocessor	32
Threads / Multiprocessor	1024
Thread Blocks / Multiprocessor	8
Total # of 32-bit registers / Multiprocessor	16384
Register allocation unit size	512
Shared Memory / Multiprocessor (bytes)	16384

Allocation Per Thread Block	
Warps	8
Registers	2048
Shared Memory	2048

These data are used in computing the occupancy data in blue

Maximum Thread Blocks Per Multiprocessor	Blocks
Limited by Max Warps / Multiprocessor	4
Limited by Registers / Multiprocessor	8
Limited by Shared Memory / Multiprocessor	8

Thread Block Limit Per Multiprocessor highlighted RED

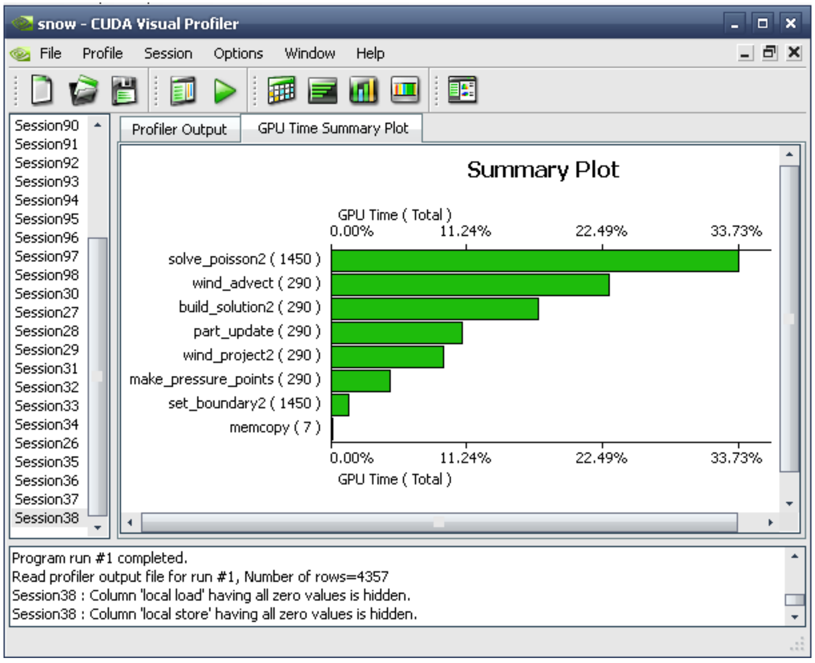
CUDA Occupancy Calculator	
Version:	1,4
Copyright and License	

Click Here for detailed instructions on how to use this occupancy calculator.

For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Your chosen resource usage is indicated by the red triangle on the graphs.
The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

Figure 3.15: Occupancy calculator spreadsheet



```
Program run #1 completed.
Read profiler output file for run #1, Number of rows=4357
Session38 : Column 'local load' having all zero values is hidden.
Session38 : Column 'local store' having all zero values is hidden.
```

Figure 3.16: Example profiler session

to us:

- Coalesced load
- Coalesced store
- Uncoalesced load
- Uncoalesced store
- Local load
- Local store
- Branch
- Divergent branch
- Instructions
- Warp serialize
- CTA launched(executed thread blocks per multiprocessor)

Of these the most interesting ones are those that pertain to uncoalesced memory access, local memory access, divergent branches and warp serializing. All of these should be kept at a minimum, or else performance will suffer.

We have attempted to optimize all of our simulation kernels according to the above techiques and guidelines. Table 3.4 lists the occupancy that was achieved on each kernel, on compute capability 1.1 and 1.2 cards. For all the kernels which on 1.1 cards achieve 67% occupancy, we are register constrained and unable to optimize further.

Kernel	1.1	1.2
Advect	67%	67%
Build solution	67%	100%
Solve poisson	67%	100%
Set pressure boundary	100%	100%
Project	67%	100%
Smooth snow	67%	100%
Update particles	67%	100%

Table 3.1: Occupancy achieved for all kernels on different compute capability

Chapter 4

Results

In this chapter we will evaluate several aspects of our implementation. We will examine its performance characteristics and scaling properties, under various different configurations, and running on different systems. Visual results will also be presented and evaluated.

4.1 Test Setup

To test the performance of the simulation, four test systems were selected(Table 4.1). They were selected for their wide range of GPU power while other factors are similar.

Three of the four GPUs in the test are consumer graphics cards, while one(the Tesla c1060) is targeted at the HPC market. It does not have a video output, and another graphics card is needed for output. System 1 and 2 are in reality the same system. It has two graphics cards: the Tesla and the GTX 280. For all tests involving those systems, the GTX 280 was used for graphics output, and for System 1 computation was also performed on this card, while for System 2 it was performed on the Tesla. For all tests the number of SOR iterations performed will be 5, as this has proved to strike a good balance between visual results and performance.

	System 1	System 2	System 3	System 4
CPU	2.83 GHz	2.83 GHz	2.83 GHz	2.5 GHz
RAM	4 GB	4 GB	2 GB	8 GB
GPU	GTX 280	Tesla c1060	8600 GT	9800 GTX
GPU RAM	1 GB	1 GB	256 MB	512 MB
Chipset	nForce 790i	nForce 790i	nForce 790i	Intel X48
OS	Ubuntu	Ubuntu	Ubuntu	Ubuntu
Driver	180.22	180.22	180.22	180.22

Table 4.1: Test systems(all CPUs are Intel Core 2 Quads)

4.2 Kernel Profiling

The Cuda Visual Profiler was used to examine the time distribution per frame of the different kernel functions involved in the simulation. For this test a 256x256x32 wind field was used, a particle count of 512K, and a 512x512 height map. Figure 4.1 shows the results, with the different kernels listed in descending order on the right. We see that the particle update kernel grabs the most time here, and this may be explained by the large amount of uncoalesced memory reads that are performed against the terrain vertex array, as well as non-coherent texture lookups in a large 3D texture.

We see that the major kernels involved with the wind field simulation fill most of the remaining time. Predictably the solve Poisson kernel is the most resource demanding among these, considering the fact that it is executed several time. In fact, we may have expected it to more expensive than it is in comparison to some other functions that are only called once per frame, particularly the build solution kernel, and the advect kernel. We explain the performance characteristics of the advect kernel by the fact that it performs a filtered texture sampling per voxel, which is less efficient than normal coalesced reads. The build solution kernel suffers from the inefficiency of operating on `float4` values compared to the 32-bit `float` values the solver works on.

The snow smoothing kernel, which is responsible for managing redistribution of fallen snow is shown to have an insignificant footprint compared to the rest.

We are motivated to examine the impacts on performance of varying the parameters that were fixed in this test. In particular, we want to see how performance scales when each is gradually increased.

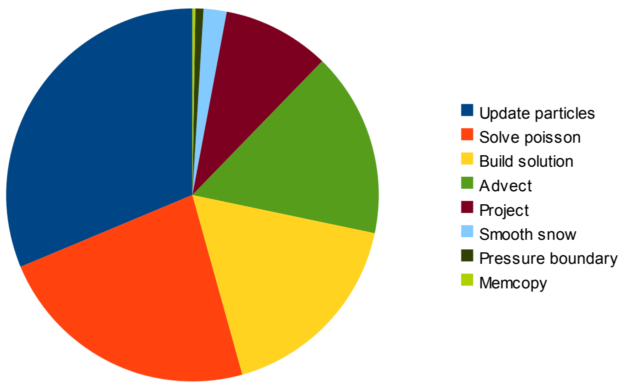


Figure 4.1: Time distribution of simulation kernels per frame

4.3 Benchmarks

4.3.1 Scaling Wind Field Resolution

Two scenarios were considered with respect to investigating scaling the wind field. In the first scenario(Figure 4.2 and 4.3) the particle count was zero, while the size of the wind field is scaled from low to high resolutions. This will let us observe how the different systems behave with regard to increasing amounts of pure computation. In particular, the computations on the wind field will not affect screen output.

No Particles

Fig. 4.2 shows the frame rate results of the first test. Here we see the powerful Tesla card leading. It is perhaps surprising that is leads by such a high margin over the equally powerful GTX 280. The key to understanding this lies in remembering the fact that since there are absolutely zero particles present. This means that nothing that is rendered depends on what is computed, and therefore the runtime system does not need to transfer data from the Tesla to the 280(which is doing the rendering). The same is true of the 280 when it is doing the simulation, but unlike the Tesla it has to perform rendering tasks simultaneously. It seems reasonable that this is the cause of this gap, and indeed we see that this gap is reduced as the computation overhead increases: Where the Tesla performs twice as good as the 280 for the lowest resolutions we see that the difference tends to zero with higher resolutions. The Tesla's initial flat curve is likely to be caused by the fact that for the two lowest

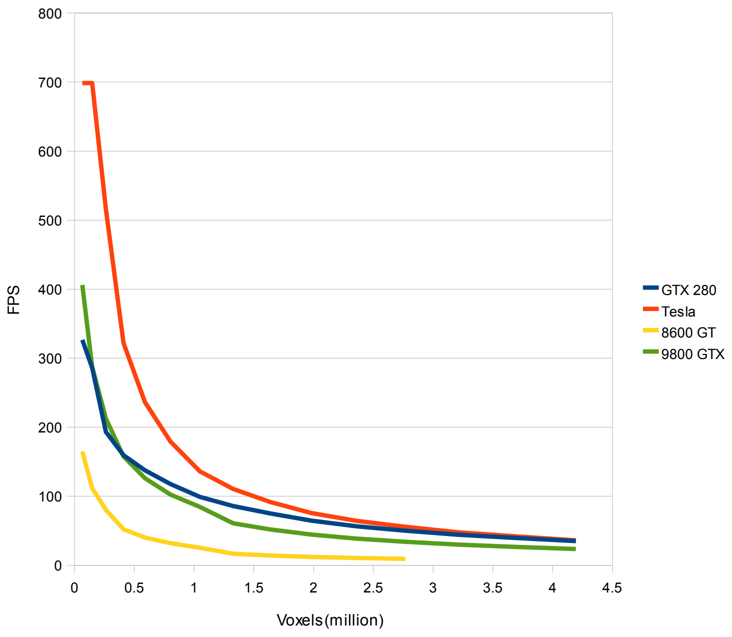


Figure 4.2: Scaling wind field with no snow(frame rate)

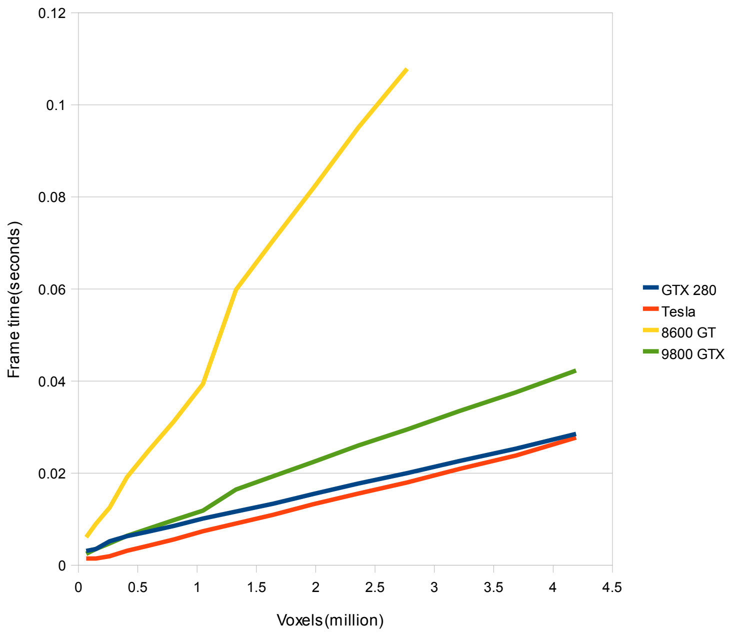


Figure 4.3: Scaling wind field with no snow(frame time)

problem sizes other factors than computation time limit its frame rate.

An explanation is also called for regarding the 9800 GTX's frame rate in comparison to the much more powerful GTX 280 on the lowest problem sizes. As was just mentioned the lower problem sizes seem not to be able to stretch computation time to the point where it overtakes other factors. Recall from Table 4.1 that these two systems have different chipsets, and we theorize that particulars in the operation of the system bus may affect the results when computation load is negligible, as was the case here.

Figure 4.3 is perhaps more interesting, because it shows us how frame time(as opposed to frame rate) scales with respect to increaseing voxel count. This is because frame time should increase linearly given a linear increase in computation(all else being equal). This is also what we see happening, at least for the Tesla and the GTX 280. The 280 and the Tesla scales very similary, while the 9800 GTX scales somewhat worse and the 8600 GT is worst, as expected. The latter two experiences a deterioration after about 1 million voxels, but their slopes recover their previous characteristics after this point.

Balanced Particle Count

In the second scenario(Figure 4.4) the particle count was set at what was considered a balanced number(512K), while the wind field was scaled like before. This is a more interesting case since it lets us see how the wind simulation scales in a normal situation where the particle simulation will run simultaneously. The particle count was chosen because it is not unreasonably high while it results in convincing visuals.

When we add the snow simulation(Figure 4.4) we see the comparative power differences between the cards reflected in the graphs, since now, even at low resolutions, there is still a significant amount of computation to do. We see that the Tesla is very much held back by the fact that for each frame, the updated particles need to be transfered across the system bus to the 280 card doing the rendering. This holds it back to such a degree that we don't see performance decreases until the problem size reaches 192x192x16.

All the cards except the 8600 GT seem to have similar scaling characteristics here, being separated only by a constant. It seems to be the case that the wind simulation is memory bound and that the more powerful cards are not able to fulle utilize their execution units, because if that were the case they would not scale so similarly. This may indicate to us that improvements could be made in the domain decomposition of the wind kernels, to reduce

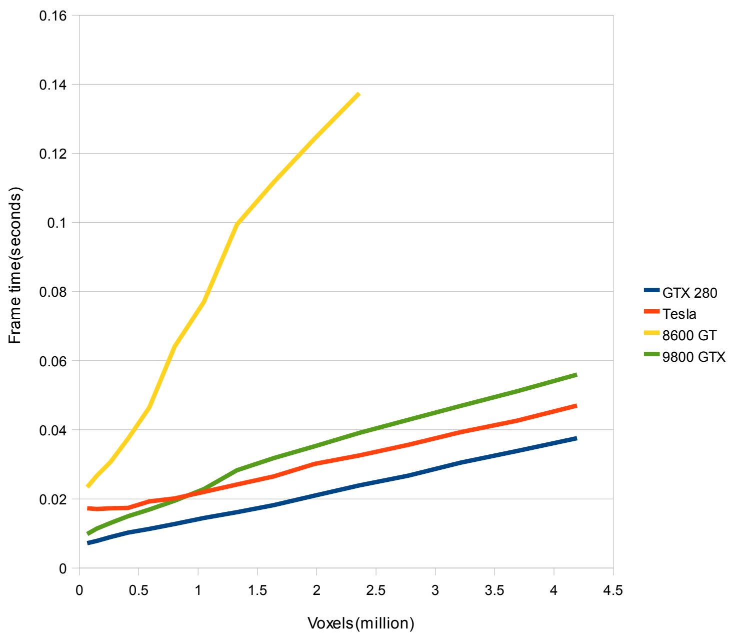


Figure 4.4: Scaling wind field with snow count = 512K

bandwidth use.

4.3.2 Scaling Particle Count

When considering the scaling of the particle counts, the opposite two scenarios of the previous test were considered, namely scaling particle count with either no wind field[1] or a balanced wind field(192x192x16). Again the choice for fixed parameter in the second scenario was based on finding a reasonable but not too high value, such that the scaling properties of the other parameter may be examined under realistic circumstances.

No Wind Field

In Figure 4.5 we see fairly similar shapes for all four graphs, with some minor differences. The GTX 280 and the 9800 start out at the same level, suggesting that they are not compute bound and with other system factors(perhaps differing motherboards) putting them on an even level. The Tesla follows

[1]This is slightly inaccurate; there is a tiny 32x32x4 wind field, but it is static.

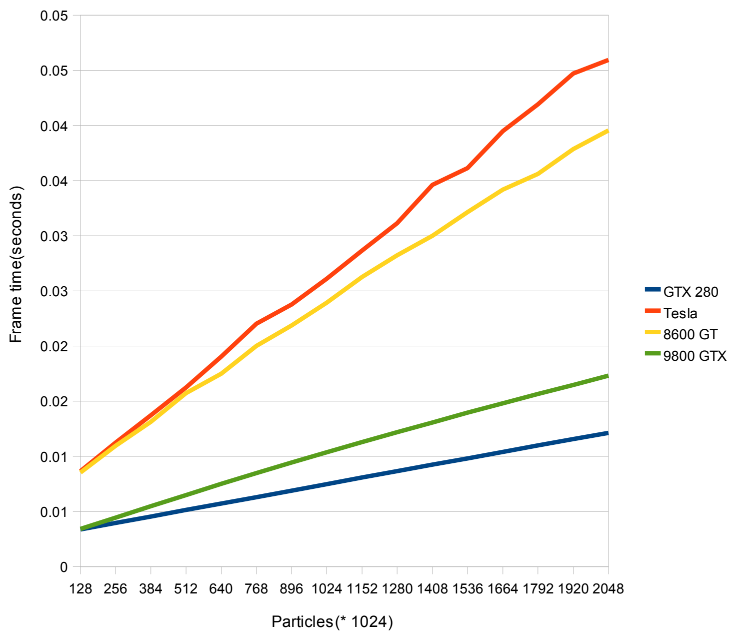


Figure 4.5: Scaling snow particle count with no wind

the curve of the comparatively very weak 8600 GT very closely. This is a bad scenario for the Tesla since absolutely all compute work is directly tied to display output and the amount of computations per particle nowhere near justifies the cost of going through the system bus. It seems to be the case here that the overhead for transferring the particle positions over the bus for the Tesla grows about as fast as the 8600 is slowed by its increasing compute load. At the highest particle count the frame time on the 9800 GTX is 43% higher than on the GTX 280. This is significantly faster, but it does not reflect the almost twice as numerous Stream Processors on the latter.

Balanced Wind Field

Figure 4.6 shows the Tesla doing much better than it did in the previous test, in comparison with the other cards. This is because now that the wind field is also simulated the amount of computation per particle increases dramatically. Not only do we have an increase in computation, but the cost of updating each particle increases to since we now have to do a texture lookup in a large 3D texture. All of this helps to amortize the cost of the needed data transfers. Here we do see the 280 perform as expected in relation to

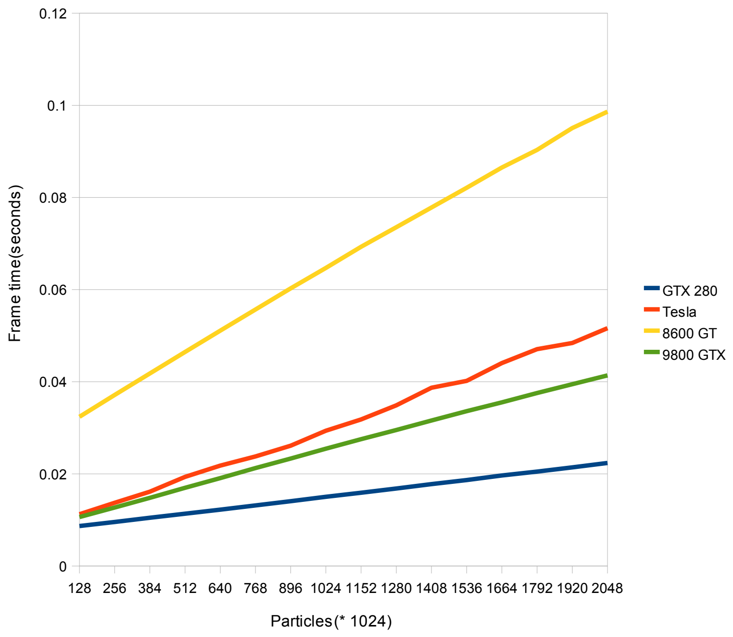


Figure 4.6: Scaling snow particle count with wind field = 192x192x16

the 9800 GTX, with each frame taking almost half the time. This indicates that the simulation scale well with increasing compute power, and that the computation required for the particle update alone is not enough to fully utilize the hardware.

4.3.3 Scaling Terrain Resolution

All colliding particles modify the terrain, and the snow redistribution kernel reads and writes each vertex in the terrain every frame. The former should not suffer from increasing terrain resolution since collisions happen fairly seldom, but the latter should make an FPS impact when resolution increases. In addition we note that the largest resolution(1024x1024) will cause 2M triangles to be rendered each frame. The test will run a reasonably heavy simulation with a 256x256x30 wind field and 512K particles.

The results(Figure 4.7) show that even with this large a problem size, scaling the terrain resolution still has a non-negligible impact. The 8600 GT is predictably paralyzed by the load even at the lowest terrain resolution, due to the field resolution and particle count. However, while impact is

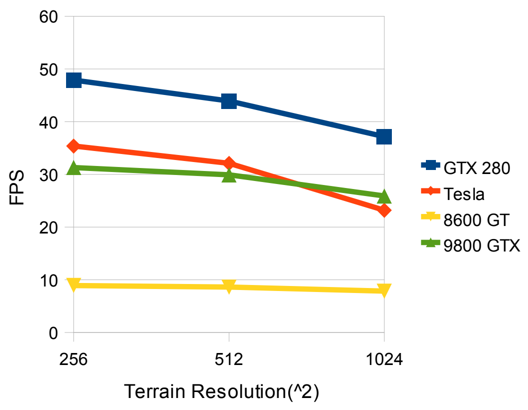


Figure 4.7: Scaling terrain resolution with particle count = 512K and wind field = 256x256x32

non-negligible it is very small compared to the impact we have seen from increasing particle count and field resolution, and as a result we may use very high resolution terrains to simulate snow phenomena over large landscapes.

4.4 Visuals

A very large number of particles can be simulated while maintaining real time performance in this application. Even in a large scenes this enables high particle densities in all spaces. This enables believable visualisations of dramatic snow storms, with snow seriously impairing visibility. High densities in itself would of course not be very interesting if the snowflakes did not behave realistically in scenarios with no wind, with violent gusts of wind, and everything in between. Reasonably detailed and realistic wind simulations are also achieved, while still maintaining real time frame rates.

In Figure 4.8 we can see a scene where the snowflake count is at a medium level, and where the wind acting upon the snow is obvious. Streaks composed of denser snow may be seen following the field lines. The streaks are formed by wind forcing the snow against hillsides which causes it to be compressed to the form seen as it passes over the hills.

Figure 4.9 shows the really dense scenario, where long distance visibility is effectively zero. This is possible to fake, but here it is achieved through sheer mass of particle density. The wind force is lesser here and this leaves a more uniform distribution of snow in the air, but details of wind interaction is also

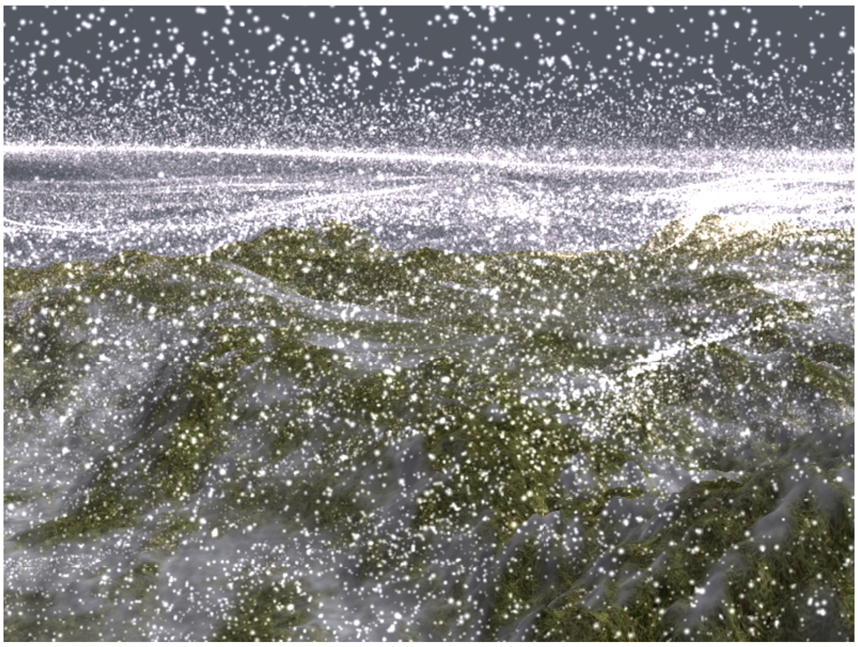


Figure 4.8: High wind speeds, medium particle count

hidden by the pure whiteness that results.

A simple test of the snow redistribution kernel is shown in Figure 4.10. Here the entire heightmap was flattened, except for a square box in the middle. What the figure shows is the result of snow being carried by high winds from the right impacting its side and on top of it. The snow hitting the wall increases the snow depth on the wall forming vertices, and this is redistributed to the lower neigboring vertices, eventually resulting in the buildup of snow at the base as shown. We also see a buildup on the flat top, that is deepest in the middle as we often see in reality.

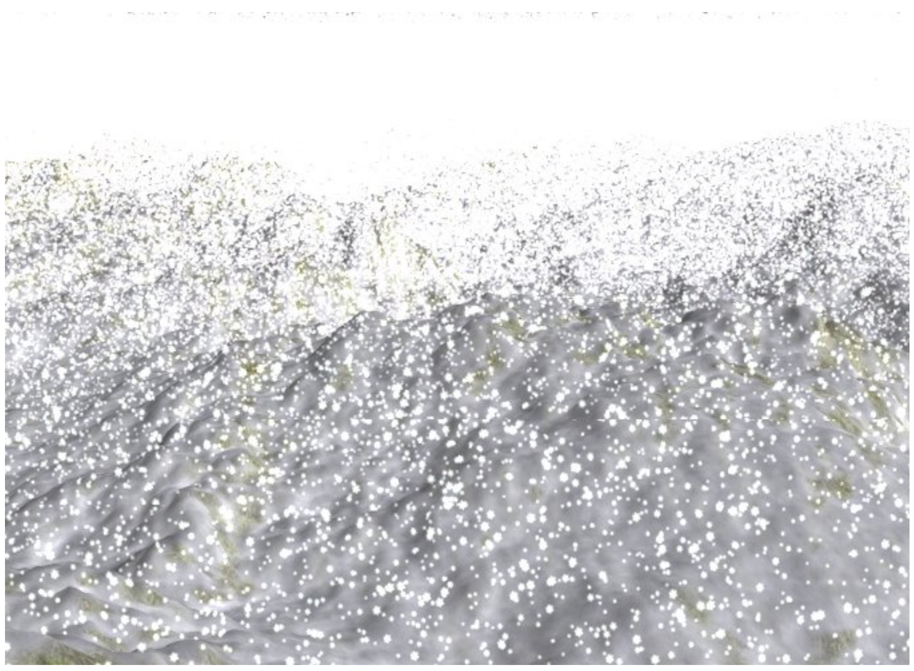


Figure 4.9: Medium wind speeds, high particle count

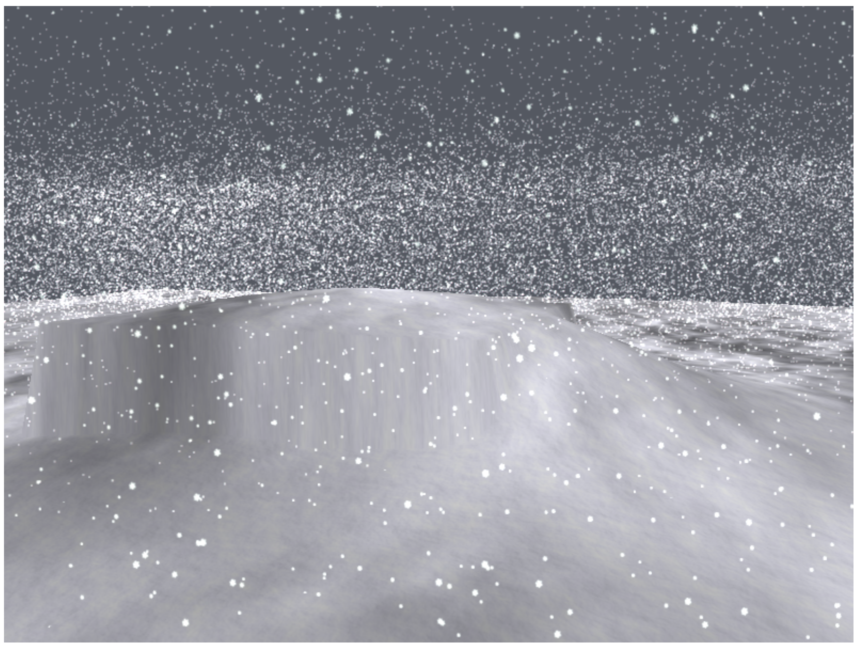


Figure 4.10: Snow buildup on top of and against square block shape

Chapter 5

Conclusions

The goal of this thesis was to investigate using modern GPUs for the purpose of realizing a real-time computationally intensive snow simulation. Snow was here interpreted to mean the behavior of airborne snow as it falls, as well as the buildup on snow on the ground.

Our simulation was capable of simulating complex air flows over and around the features of large landscapes, with very numerous snow particles carried on these winds. The snow particles behaved realistically, with respect to both air flow and geometry collision. We were able to model the dynamic buildup and distribution of snow over varied geometry.

With respect to our main goal, we feel out final snow simulation implementation was very successful. We implemented a simulation using the NVIDIA CUDA programming environment. This allowed us to implement the simulation in high level abstractions that older Cg-style implementations that are more graphics specific.

Our implementation was able to maintain real-time frame rates on a modern NVIDIA GPU with particle counts exceeding two million, all of which are interacting both with the wind field and the ground.

The number of fluid cells simulated in the wind field could be scaled up beyond four million while maintaining our real-time requirement. Finally, we showed that the performance hit of increasing geometry resolution to over to high values like over one million vertices was not significant.

The high particle counts that we achieve, in combination with the high resolution landscapes provide for very convincing visuals.

5.1 Future Work

The implementation provided here while illustrative of what can be done with regards to simulation of snow phenomena on a modern GPU, can, like most implementations, be improved even further.

One area where it could be significantly improved is in support for more complex geometry. The height map model does support fairly realistic scenes from the natural world, but it does not support objects like buildings well.

It may also prove useful to look at providing an even better performing fluid simulation, or to investigate alternative fluid models, such as Lattice Bolzman methods.

Bibliography

[1] Nvidia cuda compute unified device architecture. `http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf`, June 2007.

[2] Nvidia product catalog. `http://www.nvidia.com/page/products.html`, 2009.

[3] Michael Aagaard and Dennis Lerche. Realistic modelling of falling and accumulating snow. 2004. Master Thesis, Aalborg University.

[4] Louis Bavoil. Rendering huge triangle meshes with opengl. 2005. University of Utah.

[5] David Blythe. The direct3d 10 system. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 724–734, New York, NY, USA, 2006. ACM.

[6] Dirk Vanden Boer. General-purpose computing on gpus. 2005. School of Information TechnologyTransnationale Universiteit Limburg.

[7] William L. Briggs. *A multigrid tutorial.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1987.

[8] Robin Eidissen. Comparing cg and cuda implementations of selected transform algorithms. 2008. NTNU.

[9] Francine Evans, Steven Skiena, and Amitabh Varshney. Optimizing triangle strips for fast rendering. In *VIS '96: Proceedings of the 7th conference on Visualization '96*, pages 319–326, Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

[10] Paul Fearing. Computer modelling of fallen snow. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and*

interactive techniques, pages 37–46, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[11] Michael Griebel, Thomas Dornseifer, and Tilman Neunhoeffer. *Numerical simulation in fluid dynamics: a practical introduction.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1998.

[12] Hkan Haglund, Mattias Andersson, and Anders Hast. Snow accumulation in real-time. 2002. University of Gvle.

[13] John Hennessy and David Patterson. *Computer Architecture - A Quantitative Approach.* Morgan Kaufmann, 2003.

[14] Lee Howes and David Thomas. Efficient random number generation and application using CUDA. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 37, pages 805–830. Addison Wesley, 2007.

[15] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. Nvidia tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, March-April 2008.

[16] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, New York, NY, USA, 2008. ACM.

[17] Ingar Saltvik. Parallel methods for real-time visualization of snow. 2005. Master thesis, NTNU.

[18] Ingar Saltvik, Anne C. Elster, and Henrik R. Nagel. Parallel methods for real-time visualization of snow. In Bo Kgstrm, Erik Elmroth, Jack Dongarra, and Jerzy Wasniewski, editors, *PARA*, volume 4699 of *Lecture Notes in Computer Science*, pages 218–227. Springer, 2006.

[19] Jos Stam. Stable fluids. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 121–128, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[20] David M. Young. *Iterative Solution of Large Linear Systems.* 2003.