NTNU
Innovation and Creativity

Parallel Methods for Real-Time Visualization of Snow

Ingar Saltvik

Master of Science in Computer Science
Submission date: June 2006
Supervisor: Anne Cathrine Elster, IDI
Co-supervisor: Henrik Nagel, IDI

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

Problem Description

In this thesis, we look at the underlying numerical fluid equations for simulation of snow and look at how to optimize and parallelize these routines as well as how to display the results efficiently through OpenGL.

Multiprocessor platforms will be considered. The ultimate goal is to acheive near real-time realistic simulations of snow.

Assignment given: 20. January 2006
Supervisor: Anne Cathrine Elster, IDI

Abstract

Using computer generated imaging is becoming more and more popular in areas such as computer gaming, movie industry and simulation. A familiar scene in the winter months for most us in the Nordic countries is snow. This thesis discusses some of the complex numerical algorithms behind snow simulations. Previous methods for snow simulation have either covered only a very limited aspect of snow, or have been unsuitable for real-time performance. In this thesis, some of these methods are combined into a model for real-time snow simulation that handles both snowflake motion through the air, wind simulation, and accumulation of snow on objects and the ground.

With a goal towards achieving real-time performance with more than 25 frames per second, some new parallel methods for the snow model are introduced. Focus is set on efficient parallelization on new SMP and multi-core computer systems. The algorithms are first parallelized in a pure data-parallel manner by dividing the data structures among threads. This scheme is then improved by overlapping inherently sequential algorithms with computations for the following frame, to eliminate processor idle time. A speedup of 1.9 on modern dual CPU workstations is achieved, while displaying a visually satisfying result in real-time. By utilizing Hyper-Threading enabled dual CPU systems, the speedup is further improved to 2.0.

Acknowledgments

I would like to give my gratitude to Dr. Anne C. Elster for being my primary supervisor for this thesis. She has been a source of great inspiration with her positive attitude and deep understanding of the field, as well as providing valuable advises throughout the process. Also, thanks to Dr. Henrik R. Nagel for valuable ideas, particularly in the early stages of the project.

Many thanks go to Karstein Kristiansen and Rune Havnung Bakken for being helpful with providing and preparing the necessary computer facilities.

Also, I would like to thank the people at the Fiol computer lab for many quality breaks and interesting discussions that made this thesis work worthwhile.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Realistic visualization of natural phenomena has been subject to intensive research for decades. Together with the tremendous increase in computing capacity, this has indeed resulted in beautiful imagery both in still images, animations, and computer games. Still, there is an ever present demand for further developments in the field with respect to both realism and speed, as well as utilization of the seemingly ever increasing power available in today's computer hardware.

Snow-covered landscapes and snowfall are familiar conditions in the winter months for most of us in the Nordic countries. Snow has the ability to completely change the appearance and atmosphere of a scene by placing a white blanket over the landscape, and filling the air with falling and fluttering snowflakes. If the magical feeling present in a snow-covered area is captured in a computer graphics environment, it can strengthen the viewer's feeling of being immersed in the virtual environment.

The current trend in microprocessor design is an increased level of parallelism. As the current transistor technology is approaching the physical limit for the maximum clock frequency, it is becoming more relevant to design computers with more than one processor, as well as chips with more than one processor core, to utilize the transistors efficiently. This kind of parallelism has previously only been available in expensive workstations, servers and supercomputers for industry and research, but is now entering the consumer market.

This thesis' goal is to exploit parallel computer systems for real-time snow visualization. To achieve this, the feasibility of already known algorithms for snow visualization will be analyzed with respect to parallel efficiency, real-time requirement, and visual quality. An optimized, parallel implementation that combines snow visualization techniques from previous work will be developed to explore parallel methods and algorithms for snow visualization.

1.1 Problem Definition

The main objective is to visualize snow in real-time by utilizing parallel computer systems. This is an unclear task, so some requirements must be defined. For the performance part, the following requirements are established:

1. The application must run in real-time. To achieve this, a minimum of 25 frames per second is desirable, as this is the frame rate most movies operate with.
2. The code should be optimized for speed to utilize the power of a single CPU (central processing unit, or processor) well before parallelization is applied.
3. The application must be able to take advantage of at least two processors efficiently, without leaving any processor idle for significant periods of time.

As pointed out in later chapters, there are many interesting phenomena related to snow that can be visualized. The following items are chosen for visualization:

4. The motion of snowflakes on their way down to the ground must be visualized.
5. When snowflakes hit surfaces in the scene, they should pile up and eventually leave a blanket of snow.

There is of course a trade-off between performance and visual quality, as a high level of detail and physical realism will degrade performance, and likewise, a high frame rate can be achieved by reducing the scene complexity. However, the interesting aspect of this is scalability, where the relevant question to ask is: What happens with the frame rate when the number of processors and scene complexity are increased simultaneously?

1.2 Focus and Limitations

To narrow down the scope of this project, the following limitations are drawn:

- Methods for snow visualization should mainly be based on previous work. Instead of developing new visualization and simulation techniques, optimization and parallelization of existing ones are of first priority.
- Use of general purpose computations on the graphics processing unit (GPU) is an interesting direction. However, because of the scope of this thesis, this direction will not be pursued any further.
- Realistic rendering is second priority.
- Among important physical phenomena related to snow which will not be handled are: Creation and shape of snowflakes, merging of colliding snowflakes in the air, avalanches, compression, and melting.
- Static objects in the scene are limited to simple shapes such as cubes.

1.3 Definitions

Some clarifications regarding the use of terminology are necessary.

The term *real-time* is used in many contexts. Informally, in computer graphics it means a continuous stream of images that are generated at the same rate as they as displayed. When the images are displayed fast enough consecutively, they give the impression of motion. More formally, it can be defined as [Vik03]:

> Real-time visualization systems are time-critical software and/or hardware producing a continuous flow of output based on a continuous, unpredictable flow of input.

As movies and television use 24-25 frames per second to give the impression of motion, 25 frames per second is required in the snow visualization.

Throughout the text, the terms "timestep" and "frame" are used interchangeably, and mean the same, unless otherwise noted.

There may also be some confusion regarding threads, processes and processors. A *processor* is a physical unit in a computer that can execute programs. A *process* is a program under execution with its own address space, and can consist of one or more *threads* that can be scheduled to run on a processor.

A *voxel* is a 3D volume element. The volume occupied by a voxel is associated with a *grid point* in a discretization of space.

In the literature, many different mathematical notations for vectors are used. In this thesis, vectors are written as bold variables. For instance, u is a scalar, and $\mathbf{u}$ is a vector.

1.4 Thesis Outline

In Chapter 2, some background and state-of-the-art for concepts that are central to this thesis are presented, to give the reader the proper context. Next, Chapter 3 goes more in detail into previous work and techniques that have been applied to, or is meaningful for real-time snow visualization. In Chapter 4, some techniques from previous work are chosen for more work in this thesis. These are described more in detail, and combined to form a new model that handles the requirements set up in Section 1.1. Chapter 5 discusses how the model from Chapter 4 is implemented and presents parallel techniques used to achieve the necessary performance. Chapter 6 presents the attained results, both from a performance and visual point of view. Finally, the thesis work is concluded in Chapter 7.

Chapter 2

Background

This chapter presents background and state-of-the-art of concepts that are important for this thesis. First, the features of snow are described from a natural, physical point of view. Next, computer graphics is discussed with special attention to use of snow. Finally, an overview of parallel computing is given.

2.1 Snow As a Natural Phenomenon

One of the most noteworthy features of snow is its dynamic appearance. It interacts closely with its environment and changes its look and shape continuously. For instance, temperature may cause it to melt, wind and avalanches move it to new locations, and creatures leave footprints. Figure 2.1 shows a scene from the real world covered in snow, where some of the mentioned effects are apparent.

Snow crystals are formed up in the clouds when the temperature is low enough, so that water vapor condenses directly into ice [Lib99]. An assertion that is hard to prove right or wrong, is that no two snowflakes are alike. What is true however, is that the geometry of snowflakes is incredibly complex.

As snowflakes fall down towards the ground, they float, flutter and fly with the wind. Individual snowflakes are extremely light, and therefore follow air movements easily. The small scale motions are caused by friction, or viscid, forces between snowflakes and the surrounding air. Also, snowflakes create small air vortices behind them as they fall, leaving turbulent air that disturbs subsequent snowflakes.

Wind has a strong influence on the movement of snowflakes in general. The turbulent air flows around objects and creates air vortices. The snowflakes follow this air flow, and therefore move chaotically around objects, as well as following the wind direction on a larger scale. In addition, more snow hits the ground where the wind speed is slow, resulting in an uneven snow cover that reflects some of the characteristics of the air flow.

After some time of snowfall, the ground and other objects will be covered in a blanket of snow, rendering the scene different than before. The distribution of snow will rarely be evenly distributed throughout the scene, as some places will receive more or less snow because of

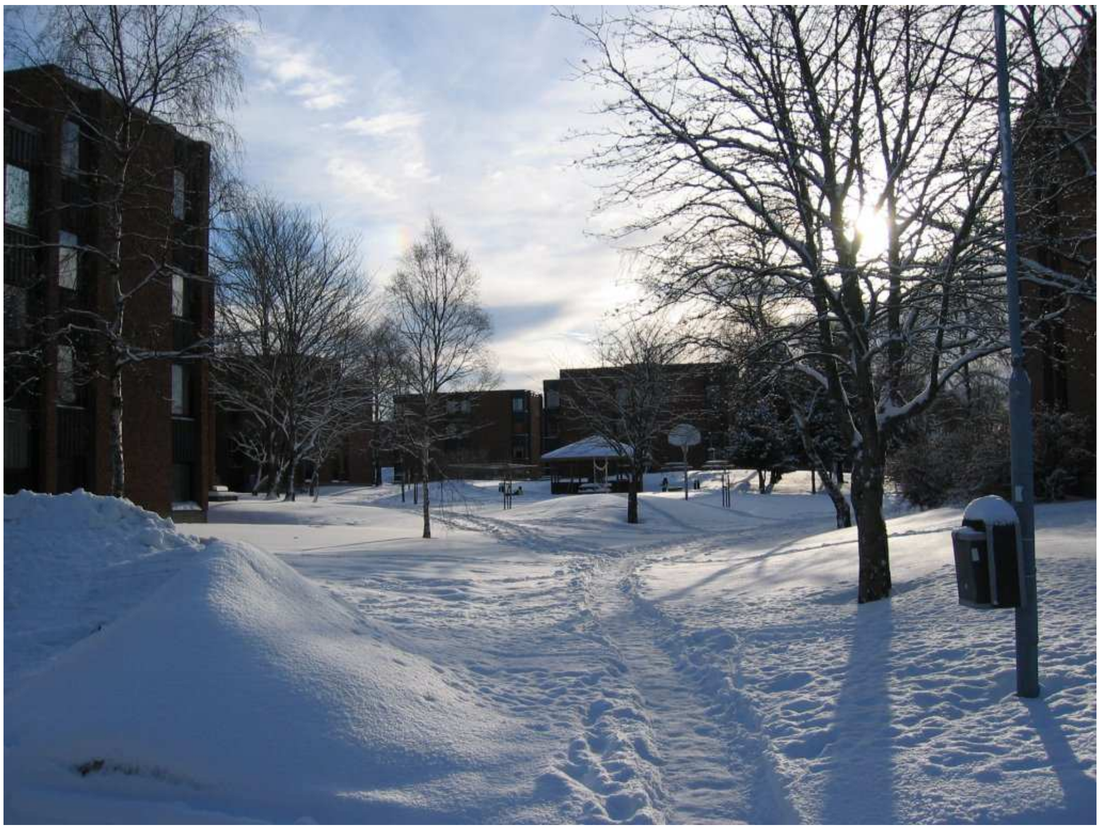


Figure 2.1: A scene from the real world covered in snow.

wind, avalanches, and blocking objects. This is evident in Figure 2.1, where branches of the trees have accumulated snow, leaving less snow on the ground below Also notice that other artifacts in the snow arise with human intervention, such as footprints, wheel tracks, and snowplough. On a smaller scale, the snow surface is not glossy but have random patterns. This is clearly visible in the snow heap in the lower left corner. When the temperature rises, the snow turns into water, pours away, and leaves the ground naked again.

2.2 Computer Graphics

Computer graphics (CG), or visual computing, is the science of using computers to generate imagery. Ever since the birth of the transistor based computer, artists and researchers have explored the possibilities of using computers for image processing and to generate images synthetically.

Traditionally, computer graphics applications have been very CPU intensive. However, in the last decade high performance special purpose graphics hardware has become common in consumer computing equipment. This hardware is able to take much of the load off the CPU, by processing a massive amount of display primitives in parallel. This has caused the transfer of data between the graphics processing unit (GPU) and the CPU to become a performance bottleneck instead of the CPU power itself.

There are two main approaches to modeling a phenomenon for computer graphics. First, there is the traditional approach where a modeler creates a precise description by trial, error and artistic skills. The other approach is to use a mathematical description of a physical phenomena, and simulate how the parameters are evolving. While the former usually requires less computation power than the latter, it brings along a difficult job for the modeler. The simulation approach only requires the initial conditions, and eventual other parameters to be defined, which eases the modeler job significantly. In practice, however, something in between these approaches is the most common.

2.2.1 Snow in Computer Graphics

A clear distinction can be made between real-time and offline applications for snow in computer graphics. When performance is not of importance, the fine details of snow can be included in comprehensive models of high detail. However, the demand for speed in computer games and other real-time applications, simplifications must be made to achieve the necessary frame rate. Also, as much computation time as possible must remain for other tasks.

Figure 2.2 shows a scene with snow from the computer game SSX On Tour. Some light snowfall is present, and the scene is covered in snow. It is likely that the snow cover is modeled as textured polygons, however, the trace after the snowboard shows that somehow the snow cover is perturbable.

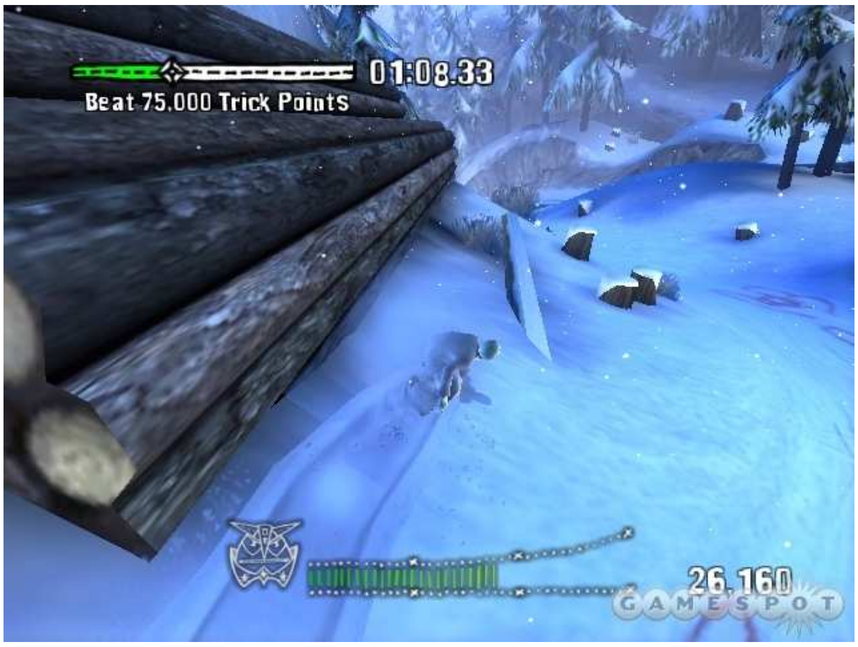


Figure 2.2: A screenshot from the computer game SSX On Tour. Used with permission from CNET Networks, Inc., Copyright 2006. All rights reserved.

In Figure 2.3, a scene from the motion picture Ice Age is shown. It includes heavy snowfall under the influence of wind. The snow on the ground look quite impressive. Notice the strong motion blur effect used to give a smoother impression of the snowflake motion. It is difficult to tell whether the snow was simulated or modeled, but it is certain that is took a lot of resources to render.

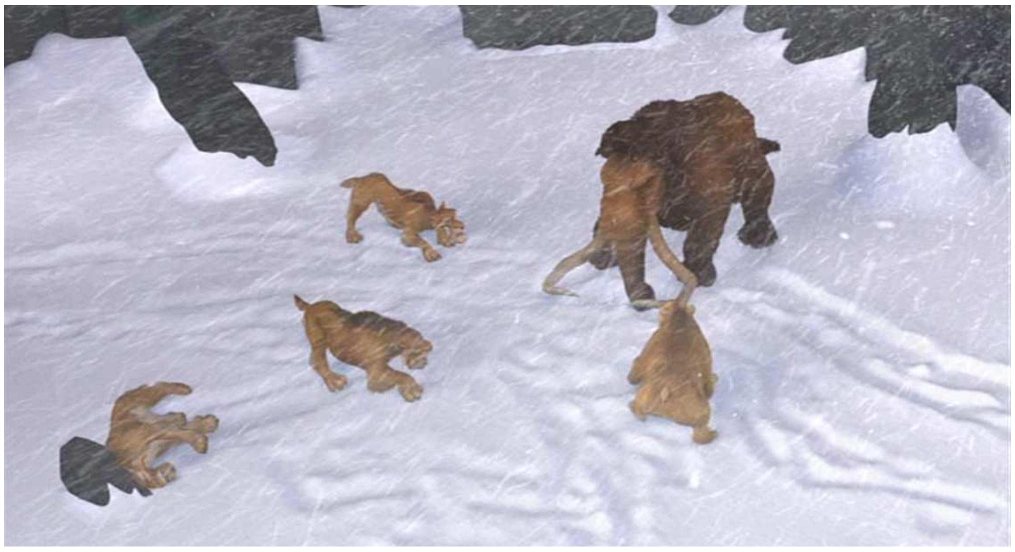


Figure 2.3: Screenshot of a scene with snow from the motion picture Ice Age

2.3 Parallel Computing

Parallel computing is the concept of breaking up a task into smaller units of work that can be executed simultaneously in order to finish the task faster.

Historically, parallel computing has been a niche market for research and other applications that benefit from massive amounts of computing power. Hardware with parallel capabilities has to a large extent been reserved for expensive workstations and servers, as well as supercomputers. As mentioned in the introductory paragraphs, this kind of parallelism is now entering the consumer market with multi-core processors, making parallel computing more relevant for the man in the street.

2.3.1 Parallel Hardware Architecture

Parallel techniques are applied on many levels in computer architecture. On the lowest level, an operation on a word can be viewed as operations on each individual bit of the word in parallel. A technique referred to as *pipelining* is employed in virtually all of today's processors, where the execution of an instruction may be started before the previous instruction has finished. Some processors referred to as *vector processors* are able to execute the same instruction on a set of related data in parallel. However, the distinction may be a little unclear, as most processors today include instructions to operate on multiple data simultaneously in their instruction set, for example Intel's Streaming SIMD Extension (SSE) and AltiVec for PowerPC.

Nevertheless, the term parallel computing is usually used in connection with computer systems that are able to execute multiple threads or processes in parallel. A common way to categorize parallel computer systems is based on their memory architecture [Pac97]:

- Shared memory: The common property of shared memory machines is that all processors in the machine can access the same global memory. The typical architecture is the *Symmetric Multiprocessor* (SMP), which is a computer with two or more CPUs and a common memory.
- Distributed memory: A distributed memory machine is collection of interconnected computers enabled to cooperate to complete a task. In contrast to the shared memory architecture, every processor has its own separate memory, and can not access the memory of other processors directly.

Many architectures fall in between these two categories. For instance, the *cache coherent non-uniform memory access* (ccNUMA) is a shared memory architecture, but the memory is physically distributed in the machine, causing the access speed to memory to vary with location of the physical memory. Also, the individual computers in a cluster may as well be SMPs.

As multi-core processors are becoming increasingly popular, the interest in parallel architectures has increased. To make use of the parallelism in these architectures, besides being able to run program simultaneously, programs must be designed in such way that they can take advantage of it.

2.3.2 Parallel Software Design

Parallel programs can be designed in two principal ways; data-parallel and task-parallel. In the data-parallel approach the domain of computation is divided among the participating processors. In essence, the same instructions are executed on different data simultaneously. For instance, to sum a set of numbers, one processor sums the first half of them, another processor sums the second half, and their results are aggregated to form the final result. A task-parallel approach suggests that processors executes different parts of the program. For instance, one processor may run a simulation, and another render a visualization of the simulation data on a screen.

The design is also dependent on the memory architecture. In shared memory machines, processors can communicate by reading from and writing to the same memory locations. In a distributed memory machine, processors must explicitly send and receive data by means of an interconnection network. In addition, the data structures must manually be distributed among the processor's memories.

The only motivation for parallel computing, besides the pure intellectual exercise, is to shorten the time required to solve a problem. Development of parallel programs is an expensive and time-consuming task, and not just a question of having the necessary computing resources available. Twice the number of processors does not necessarily mean half the execution time, if the program is not designed to utilize the processors.

A number of sources of *overhead* can be identified in parallel programs. Overhead is work that is superfluous in the serial equivalent [Pac97]:

- Extra computation. Parallel programs always introduce additional code complexity and need to aggregate results.
- Communication. Sending and receiving of data between processors is not required in a serial code, and can be solely be regarded as overhead.
- Idle time. A processor that needs to wait for another one to finish its computations, is a waste of resources. This can be due to load imbalance, where a processor has been given more work than the other, and its result is needed to continue execution. Another reason is the fact that some operations are inherently serial, and can only be executed by a single processor.

2.3.3 Parallel Computing in Computer Graphics

As CG is a resource intensive branch of computer science, it is naturally a candidate to apply parallel computing to gain the necessary speed and realism. With more computational power comes the opportunity for increased physical realism, bigger and more detailed models of reality, and more power to render high resolution imagery in less time.

Today, CG benefits from the highly parallel, tailored GPU for processing millions of polygons in parallel. In addition, workstations and supercomputers with parallel capabilities are used for heavy graphics tasks in engineering and virtual reality. As multi-core CPUs are currently entering the consumer market, it is likely that future CG applications will be designed to take advantage of this easily accessible parallelism in the future.

Chapter 3

Previous Work

With the strong effect snow can have on a scene, numerous attempts have been made in order to use the different aspects of snow in computer graphics. This chapter presents previously published techniques that are directly related to snow simulation, or is relevant for later chapters.

3.1 Falling snow

Two main approaches can be identified in previous work on falling snow; particle systems and billboards. The particle system approach is the most physically realistic, as it treats each individual snowflake as an object whose motion is determined by the influence of various forces. This requires that each snowflake is handled separately for updating and rendering.

The idea behind the billboarding approach is to render an image of a distant object on a plane, to fool the viewer into believing that it is the original object. This method sacrifices realism and flexibility in how the camera can be moved around in the scene, in trade for speed. This is indeed a common technique in computer games and the movie industry. [LZK$^+$04] achieves the effect of falling snow in real-time by rendering snowflakes to a texture that is composited on the scene.

[Ree83] introduces the concept of a particle system for modeling fuzzy objects such as smoke, fire, clouds and water. However, it is also indeed applicable to more pronounced particle-like phenomena as snow and dust [Sim90]. A particle system is basically a collection of objects known as particles, a set of rules for how they behave in 3D space, and how they should be rendered. Animating a particle system is just a matter of moving the particles according to the velocity vector and calculating new forces. Particles may also enter or leave the scene, or go through transformations when certain events occur, for instance collisions with objects or other particles.

3.1.1 Forces Influencing Falling Snowflakes

In their master's thesis [AL04, MMAL05], Aagaard and Lerche derive a physically based model for the forces acting on falling snowflakes. Using this, an algorithm for calculating the movement of snowflakes is presented, using the particle system approach. The model identifies four main forces that act on a falling snowflake, as illustrated in Figure 3.1:

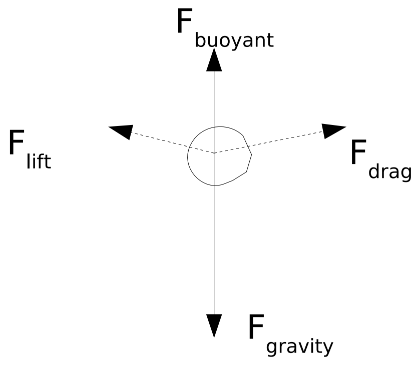


Figure 3.1: Forces acting on a snowflake. Constant forces are represented by solid arrows.

$\mathbf{F}_{\text{gravity}}$ is the gravitational force, pointing downwards with a constant value of mg, where m is the mass of the snowflake, and $g = 9.81$ is the gravitational constant. $\mathbf{F}_{\text{buoyant}}$ is the force that is caused by the density difference in an object and the surrounding fluid. The latter is in this case so small that is can be neglected without affecting the realism to a considerably extent.

The drag force is the result of the difference in velocity between the snowflake and the surrounding fluid. This motion is due to wind as well as turbulence created behind other snowflakes when they fall, and the force depends on the size, shape and velocity of the snowflake. The direction of the drag force is the same as the velocity direction of the surrounding fluid, and the magnitude of this force is derived by [AL04] as:

$$F_{drag} = \frac{V_{fluid}^2 \cdot m_{snow} \cdot g}{V_{max,z}^2} \tag{3.1}$$

$\mathbf{V}_{fluid}$ is the velocity of the surrounding fluid, m_{snow} is the mass of the snowflake, and $V_{max,z}$ is the terminal velocity in the z-direction (upwards). The terminal velocity is the steady state velocity a snowflake will arrive at when falling freely through the air. The reader should consult the original work for a more through derivation of this formula.

The lift force is the force that gives the chaotic, fluttering motion when snowflakes fall with no wind. This effect is caused by a phenomenon known as vortex shredding, which is the formation of air vortices behind an object that moves through a fluid, and resulting in varying pressure differences on around the object. Basically, it is the same reason as the up-drift of an airplane wing. In addition, vortex motion in the air caused by other snowflakes will contribute to the lift force. However, the problem is that an analytic solution to this is extremely hard to derive, as well as to solve numerically. Instead, a simplification is used, based on the observation that the characteristic movement pattern of snowflakes is a periodic, oscillating path. A vertical spiral path, the helix, is used to model this motion, as illustrated in Figure 3.2.

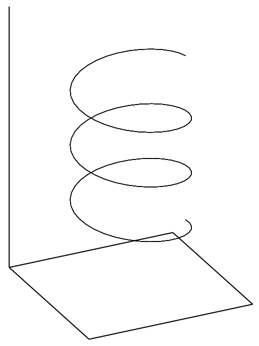


Figure 3.2: Spiral path, or helix, used to model lift force on snowflakes in [AL04].

3.2 Wind

Wind is of the utmost importance in order to simulate the motion of snowflakes. The basic idea is to simulate a wind field, or more specific, a velocity field, and use this field to calculate the force the air exerts on the snowflakes ($\mathbf{F}_{\text{drag}}$).

Many phenomena in nature exhibits fluid-like behavior, and as a result, much research has been put into simulating these phenomena. Mainly two approaches for simulation of fluid motion can be identified; Computational Fluid Dynamics (CFD) and the Lattice Boltzmann Model (LBM).

3.2.1 Computational Fluid Dynamics

The underlying equations for fluid flow are the Navier-Stokes equations, which are widely regarded as being an accurate description of the dynamics of fluids. A common simplification is to assume that the fluid in question is incompressible, leading to the *Navier-Stokes equations for incompressible flow* (NSE) [Sta00]:

$$\nabla \cdot \mathbf{u} = 0 \tag{3.2}$$
$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \frac{1}{\rho}\nabla \mathbf{p} + \nu \nabla^2 \mathbf{u} + \mathbf{F} \tag{3.3}$$

Among the various literature on computational fluid dynamics for computer graphics, can [Sta00] and [Fer04] be pointed out as good introductions. The latter also shows how the GPU can be used to speed up computations.

The NSE operates on two fields; a velocity field $\mathbf{u}$ and a pressure field $\mathbf{p}$. The first equation states that the velocity field is divergence-free, which means that the field is flow conserving. In other words, the inflow equals the outflow at every position in the field.

The second equation is a partial differential equation. In fact, there are as many equations as there are dimensions, in most cases two or three, because $\mathbf{u}$ is a vector field. The four terms in the equation are described in turn:

- Advection, $-(\mathbf{u} \cdot \nabla)\mathbf{u}$: As the fluid flows, it transports itself along the flow. This term expresses that the velocities transports themselves along the flow.

- Pressure, $-\frac{1}{\rho}\nabla\mathbf{p}$: When molecules move, they push each other around, causing pressure. This pressure is in fact a force that naturally leads to acceleration in the fluid. ρ is the density of the fluid.

- Diffusion, $\nu\nabla^2\mathbf{u}$: Some fluids are thicker than others, and a act out a resistance to flow. This is called the viscosity of the fluid, and is quantified by ν. This term accounts for this resistance.

- External forces, $\mathbf{F}$: External factors, e.g. a fan or a moving body, that cause an acceleration of the fluid.

If physical accuracy is of importance, the NSE are hard to solve. However, for visual purposes, it is only important that the fluid flow *looks like* a real flow, not that it is physically correct. The work "stable fluids" by Stam (1999) [Sta99] may be regarded as the most successful attempt to solve the NSE for use in computer graphics. It improves the work by [FM97] by introducing an unconditionally stable method for solving the advection term by using a semi-Lagrangian integration scheme, in contrast to [FM97] who used a finite difference scheme. In the semi-Lagrangian scheme, the velocities at grid points are traced backwards in time to find the point of departure for the particles that happened to be passing the grid point in question. The off-grid velocity at the origin is then interpolated from the neighbor grid points. This work has later been improved [FSJ01] with a method called "vorticity confinement" to restore the characteristic swirling flow, because numerical dissipation causes a rapid decay of this kind of flow in Stam's method.

In [FSJ01], it is demonstrated that this method is suitable for real-time applications on a coarse grid, by simulating the chaotic, swirling motions of smoke. Various techniques have later been applied to optimize performance of the stable fluid solver. [Sta01] provides a fast implementation of a 2D fluid solver by assuming periodic boundary conditions and using the fast Fourier-transform (FFT). [Vik03] uses dual CPU systems to achieve real-time performance, and [LLW04] and [Fer04] implement the fluid solver on the GPU to accelerate computations.

3.2.2 Lattice Boltzmann Model

An interesting and fundamentally different approach to fluid simulation is to use the Lattice Boltzmann Model (LBM) [Suc01] instead of CFD. LBM is based on the concept of lattice gas automata. Instead of trying to solve the macroscopic NSE, the fluid is viewed as a collection of microscopic flow particles moving on a discrete lattice. The lattice is basically a regular grid, where at each grid point a set of variables describing the state are defined. When time is advanced, collision and propagation of particles on the lattice are calculated, and a velocity field can be maintained. On a global basis, these collisions and propagations follow the same rules as the NSE. [WLMK04] describes this method in detail, and [WZF$^+$03] uses the GPU to achieve real-time performance for an application with a feather blowing in the wind.

There is at least one clear advantage of using LBM instead of solving the NSE. The rules for updating the grid are completely local, which means that an update of a grid point only depends upon the values of neighbor grid points. This makes the method easily parallelizable.

3.3 Accumulating Snow

There are many different approaches to visualization of snow cover. Two important distinctions can made based on the methods found in previous work; the amount of necessary computing resources, and how much work is needed beforehand by the modeler to place the snow.

In real-time applications and computer games, so-called *zero-depth* snow cover is widely used, which basically is to use textures with snow-like patterns to resemble the look of snow lying on various surfaces. With merely primitive graphics hardware, this method requires little extra computing resources. However, it is only applicable for very small amounts of snow, as it does not lift the snow surface from the ground. Also, the scene creator must explicitly map textures to surfaces.

3.3.1 Real-Time Accumulating Snow

To render accumulated snow in real-time, [OS04] uses a technique that is similar how shadow maps are used shadow calculation. The idea is to create an occlusion map to decide how the surface should be rendered in terms of snow depth, and to use a noise function to generate the texture of the snow. To detect which surfaces are exposed to the sky and therefore should receive show, a depth buffer is used. The algorithms are implemented on the GPU to achieve real-time frame rates. No extra modeling effort is required for the method to work.

In [HAH02], Haglund et al. present a method for real-time simulation of accumulation of snow on surfaces. They place height matrices on all surfaces that can receive snow, to store the snow depth. When snowflakes hit a surface, the nearest height value is increased. To render the scene, triangulations are created from the height matrices, and rendered using Gouraud shading by means of OpenGL's functionality to efficiently do so. Their focus is not on physical correctness, but to find a good trade-off between visual result and performance.

3.3.2 Non-Real-Time Accumulating Snow

[NIDN97] uses metaballs to model the appearance of snow cover. A metaball is basically a particle with a density distribution. It handles placement of snow on polygons, Bezier surfaces, and surfaces implicitly defined by metaballs. By taking the anisotropic light scattering properties of light into account, the snow surface is rendered nicely. However, because the rendering process uses raytracing and volume rendering techniques, it is very resource hungry.

In his doctoral thesis [Fea00a, Fea00b], Fearing (2000) presents a method for automatically generating thick snow cover in existing scenes with a minimum amount of interaction with the modeler. It leaves the original scene unchanged, and generates a set of new snow surfaces approximated by a triangulation. The algorithm consists of two components; accumulation and stability:

- Accumulation model: Determines how much snow each surface should receive. It is based on the counter-intuitive idea to shoot snowflakes up towards the sky from the surfaces, and detecting how many snowflakes hits the sky. At locations where finer detail is needed, the triangles are split up into smaller ones.
- Stability computation: As layers of snow are added to the scene, snow at unstable areas will fall down in small avalanches to places beneath. The method is based on the idea of calculating the angle of reprose (AOR) between neighboring triangles, and redistribute snow to neighbors if the angle is too steep.

To render the scene, Fearing uses commercial rendering software. While the method is visually superior, it requires a lot of computing resources, and is therefore restricted to use in offline rendering.

In [FO02], Feldman and O'Brien present a method for accumulating snow, including the effects of wind. They use the fluid simulation from [FSJ01] to create a velocity field, and use this field to transport snow. They accumulate snow by storing the amount of snow on the horizontal surface of the voxels that are marked as occupied. New voxels are marked as occupied after enough snow has been accumulated to fill the voxels beneath. Rendering is performed by creating a triangle representation of the snow surfaces, and can thereby be viewed as a combination of the contributions by Fearing [Fea00b] and Haglund et al. [HAH02], in that the triangle resolution is fixed, but it is created for offline use.

3.4 Parallel Methods for Simulation of Snow

The literature is sparse with respect to the direct application of parallel computing to snow simulation.

[Sim90] presents a framework for data parallel calculations on particle systems. He uses the Connection Machine CM-2, a data parallel supercomputer to simulate and render particle systems, by assigning each particle a virtual processor for simulation, and each pixel a virtual processor for rendering.

For simulation of a wind field using computational fluid dynamics, Bryborn et al. [BKM$^+$00] present a method for parallelization of the unconditionally stable solution to the Navier-Stokes equations presented by [Sta99]. Their implementation runs on shared-memory architectures, and achieves a speedup up to 3.7 on a 4 processor workstation in some parts of the code.

In his master's thesis, Torbjørn Vik [Vik03, VEH03] develops optimized, parallel algorithms for smoke simulation based on [FSJ01]. He uses dual processor workstations and the SIMD instruction set found in today's modern processors to achieve real-time frame rates. Because the smoke simulation is based on the fluid simulation in [Sta99], it is applicable to wind as well.

Chapter 4

Snow Modeling

The model for snow in this thesis is divided into three components; falling snow, wind, and accumulating snow. Here, these three components are described in turn, with special attention to computational efficiency and the real-time requirement both in simulation and rendering. Finally, the connection between them is described.

4.1 Falling Snow

In Chapter 3, previous work related to simulation of falling snow was presented. Two fundamentally different approaches were identified; image-based techniques and particle systems. Based on this information, the image-based approach is considered unsuitable for this thesis, and the particle system approach is chosen. The reason is the need to track the motion and position of each individual snowflake in three dimensions to detect when snowflakes hit objects in the scene, as well as interaction with wind. This would have been difficult, if not impossible, with an image-based approach, where snowflakes does not move around in the scene, but are merely moved around in two dimensions to imitate three dimensional motion.

In Appendix A, a movement model for a falling paper strip [TK94, BEM98] is described. This model is based on how thin paper strips move through the air when they are allowed to fall freely. Because big snowflakes have much of the same properties as paper strips, namely thin, flat and light, it was hoped for that the paper strip model would be applicable to snowflakes as well. However, by implementing the model, it became clear that it does not reproduce the motion of falling snowflakes well. Also, it is only defined in two dimensions, and it is not obvious how to generalize it to the required three dimensions.

To simulate the movement of snowflakes, the movement model from [AL04, pp. 66–84], also briefly described in Section 3.1.1, is used. It is further described next.

4.1.1 Simulation Model

Because the model of [AL04] was found to produce the best results and being the least resource demanding, is was chosen as the movement model for falling snowflakes. The model keeps

Property	**Description**	**How chosen?**
$\mathbf{p}$	Position	Randomly inside scene boundaries
$\mathbf{V}_{snowflake}$	Velocity	z-component equal to $V_{max,z}$
D	Diameter	$0.015 \cdot \lvert T\rvert^{-0.35}$ for $T \leq -0.061$ 0.04 for $T > -0.061$
ρ_{snow}	Density	$C_{humidity}/D$, where $C_{dry} = 0.170, C_{wet} = 0.724$
R	Rotational radius for circular movement	Uniformly random in the interval $(0, 2)$
ω	Angular speed for circular movement	Uniformly random in the interval $[-\pi/4, -\pi/3]$ or $[\pi/4, \pi/3]$
$V_{max,z}$	Terminal vertical velocity (maximum velocity)	Uniformly random in the interval $[0.5, 1.5]$ for wet snow, and $[1, 2]$ for dry snow

Table 4.1: Properties defining a snowflake. The temperature, T, is a global parameter. Dry snow occurs when $T \leq -1.0$ Celsius, and wet snow when $T > -1.0$ Celsius

track of the velocity and position of every snowflake. In addition, each snowflake has a set of properties attached to them, as summed up in Table 4.1. The position and velocity change as time advances, and together define the state of the snowflake. The last five properties describe physical features, and are initialized with stochastic values to imitate the apparent randomness and diversity in the movement and shape of snowflakes. [AL04] describes how to choose these values from a physical point of view. This is summed up together with the properties in Table 4.1.

As pointed out in Section 3.1.1, and illustrated in Figure 3.1, the model is based on four different forces. The forces are gravitational, buoyancy, drag and lift, where the two first are constant, and the latter two varies with time and wind velocity.

The equations from Section 3.1.1 are combined into Algorithm 1, which shows how to simulate the movement of a single snowflake [AL04].

Algorithm 1 Simulate snowflake motion

```
1: Initialize $D$, $\rho_{snow}$, $R$, $\mathbf{p}^0$, $\omega$, $V_{max,z}$ and $\mathbf{F}_{gravity}$ according to Table 4.1.
2: while new timestep is needed do
3:    Interpolate $\mathbf{V}^t_{wind}$ from wind field.
4:    $\mathbf{V}^t_{fluid} = \mathbf{V}^t_{wind} - \mathbf{V}^t_{snowflake}$
5:    $\mathbf{V}^t_{circ} = |\mathbf{V}^t_{fluid}|/|\mathbf{V}^t_{snowflake}| \cdot \omega R\,[-\sin \omega t, \cos \omega t, 0]$
6:    $F_{drag} = ((\mathbf{V}^t_{fluid})^2 \cdot m_{snow} \cdot g)/(V^2_{max,z})$ in the direction of $\mathbf{V}^t_{fluid}$
7:    $\mathbf{a} = (\mathbf{F}_{gravity} + \mathbf{F}_{drag})/m_{snow}$
8:    $\mathbf{p}^{t+\Delta t} = \mathbf{p}^t + (\mathbf{V}^t_{snowflake} + \mathbf{V}^t_{circ}) \cdot \Delta t + \frac{1}{2} \cdot \mathbf{a} \cdot \Delta t^2$
9:    $\mathbf{V}^{t+\Delta t}_{snowflake} = \mathbf{a} \cdot \Delta t + \mathbf{V}^t_{snowflake}$
10: end while
```

First, when a snowflake is spawned, its properties are drawn from a random distribution (line 1). Then, when the time is incremented by Δt at time t, its new position and velocity are calculated. This process starts by interpolating the wind velocity at the snowflake's current position (line 3). This is elaborated further in Section 4.2. Recall that the drag force is the force resulting from movements in the surrounding air. This force is based on the relative difference in air and snowflake velocity, $\mathbf{V}^t_{fluid}$, calculated on line 4. The drag force itself is calculated using Equation (3.1) on line 6. The circular velocity is found using a standard parametrized circular motion in the xy-plane, and adjusted according to the difference in snowflake and wind velocity, as a heavy influence of wind yields more chaotic motion. On line 7, the snowflake acceleration is computed based on gravity and drag forces. Based on this acceleration, the new position and velocity can be calculated using Newton's laws of motion (line 8 and 9).

4.1.2 Rendering

As pointed out in Section 2.1, snowflakes have an extremely complex geometry. However, the viewer must be very close to the snowflake to spot these fine details. In an application with thousand of snowflakes spread throughout the scene, most of the snowflakes will only be perceived as small dots by the viewer. The geometry of only a few snowflakes relatively close to the "camera" will be significant.

Based on this motivation, and that the rendering has a low priority in this thesis, a simple approach based on three rectangles is used. One rectangle is placed in each plane (xy, xz and yz), with the center in the center of the snowflake. The sides of the rectangles have the same length as the radius of the snowflake. This is illustrated in Figure 4.1(a).

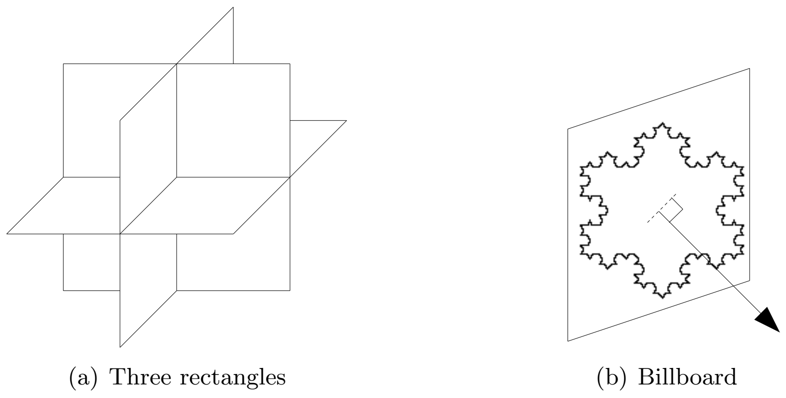

(a) Three rectangles (b) Billboard

Figure 4.1: Strategies for modeling the appearance of a snowflake. (a) is used in this thesis, and is composed of three perpendicular rectangles. (b) uses a textured rectangle always oriented towards the camera.

The main advantage of this approach is its simplicity, because it requires only 12 easily computed coordinates for each snowflake. Also, it looks approximately like a small dot from all angles, when observed from a distance. However, on a close view, it lacks the necessary detail to look like a real snowflake.

A better solution could be to use a billboarding technique as illustrated in Figure 4.1(b), where each snowflake is represented by a rectangle which always has its normal oriented towards the camera. These rectangles can be textured with an image of a real snowflake. As textures can be uploaded to graphics memory beforehand, this approach would require only four coordinates to be uploaded. However, rotation of the rectangles would require more computations to take place.

An interesting direction would be to simulate the growth of snow crystals to generate the textures for the billboard approach for rendering snowflakes. Such techniques exist, for instance [KL03], but will not be investigated any further due to the scope of this thesis.

The method from [AL04] also includes a way to simulate the creation of snowflakes. The result is snowflake models composed of about 100 triangles per snowflake. With thousands of snowflakes in a typical scene, this will become infeasible for real-time use, because of the vast amount of polygon data that needs to be transferred from memory to the graphics processing unit for each frame.

4.2 Wind

Step 3 of Algorithm 1 requires the wind velocity at the position of the snowflake that is being simulated. This velocity could simply be defined and stored in advance. However, while requiring little extra computations, it would require much effort from the scene modeler to create a realistic wind field, as well as providing little flexibility in how the field evolves with time. Instead, in this thesis a wind field is simulated.

In Section 3.2 two methods for wind simulation were identified; Computational Fluid Dynamics (CFD) and the Lattice Boltzmann Model (LBM). In this thesis CFD has been chosen, because it is well-tried and has an unconditionally stable numerical solution [Sta99]. The method used here is largely based on the method from [Sta99] and [FSJ01]. The vorticity confinement has been left out, because the small-scale swirling motion is not so important for the movement of snowflakes, as it is for the chaotic behavior of smoke, fire and such.

By assuming that the air has zero viscosity and a density equal to one, the incompressible Navier-Stokes equations (NSE), see Equation (3.3), reduce to the incompressible Euler equations [FSJ01]:

$$\nabla \cdot \mathbf{u} = 0 \qquad (4.1)$$

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \nabla \mathbf{p} \qquad (4.2)$$

The last term in Equation (3.3) has been dropped out, because no external forces are modeled. [Sta99] and [FSJ01] describe how to move a density such as smoke around in the fluid, but for our purposes that is unnecessary, as only the velocity field is needed. To solve these equations, two steps are needed; self-advection and projection. These steps are described next. The self-advection step produces a temporary velocity field $\mathbf{u}^*$. This field is then forced to be incompressible, or divergence-free, by the projection step, to form the velocity field for the next timestep.

4.2.1 Self-Advection Step

The first term on the right side of Equation (4.2) describes the self-advection of the velocity field, which is the fluid's ability to transport velocities along its own velocity field. The self-advection step solves for this term by tracing the particle which happens to be traveling past a grid point back in time, to find the velocity at the departure point. See Figure 4.2.

To implement the advection step, an integration scheme is needed to trace the particles back in time to find the departure points. [Sta99] uses a second order Runge-Kutta integration scheme. In this thesis the Eulerian scheme from [Sta03] is used, where the departure point d_p is traced linearly back in time from the on-grid arrival position a_p, expressed mathematically as:

$$\mathbf{u}^*(\mathbf{p}, t + \Delta t) = \mathbf{u}(\mathbf{p} - \Delta t\mathbf{u}(\mathbf{p}, t), t) \tag{4.3}$$

This is illustrated in Figure 4.2. After the departure point has been found, the velocity is linearly interpolated from the nearest grid points. This integration scheme is chosen because of its computational simplicity, as only one off-grid value needs to be interpolated (the right hand side of Equation (4.3)). However, this breaks the unconditional stability of the method, because if the timestep is large or the distance between grid points is small, entire voxels can be jumped over in the backtrace. In practice this is not regarded as a showstopper, because the timesteps will be quite small and the grid will be coarse. Only in exceptional circumstances is this going to make a difference.

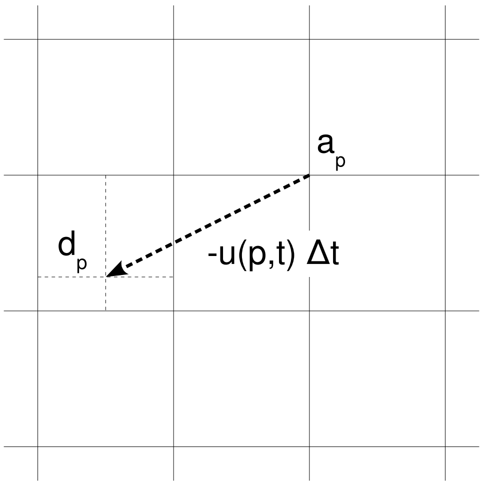


Figure 4.2: Self-advection of velocities using the semi-Lagrangian integration scheme of [Sta99], where on-grid velocities are traced backwards in time. The off-grid velocity at the departure point d_p is linearly interpolated from the nearest four grid points. This can easily be generalized to 3D, by interpolating from 8 grid points instead.

4.2.2 Projection Step

After the intermediate velocity field $\mathbf{u}^*$ has been computed, Equation (4.1) and the last term in Equation (4.2) are combined into one step that computes the final, divergence-free velocity field $\mathbf{u}$. As in [FM97], [Sta99] and [FSJ01], this is achieved by using a mathematical property known as the Helmholtz-Hogde decomposition, which states that any vector field $\mathbf{w}$ can be decomposed as:

$$\mathbf{w} = \mathbf{v} + \nabla s \tag{4.4}$$

where $\mathbf{v}$ is a divergence-free vector field, ∇ is the gradient operator, and s is a scalar field. By rearranging Equation (4.4) into Equation (4.5), this property can be used to project $\mathbf{u}^*$ into a divergence-free vector field, as shown in Equation (4.6), where $\mathbf{u}$ is the divergence-free velocity field and p is the pressure field.

$$\begin{aligned} \mathbf{v} &= \mathbf{w} - \nabla s & (4.5) \\ \mathbf{u} &= \mathbf{u}^* - \nabla p & (4.6) \end{aligned}$$

To solve Equation (4.6), the pressure field p must be computed. By applying the divergence operator $(\nabla \cdot)$ on both sides of Equation (4.6), results in the Poisson equation:

$$\begin{aligned} \nabla \cdot \mathbf{u} &= \nabla \cdot \mathbf{u}^* - \nabla \cdot \nabla p & (4.7) \\ \nabla^2 p &= \nabla \cdot \mathbf{u}^* & (4.8) \end{aligned}$$

Note that $\nabla \cdot \mathbf{u} = 0$, since $\mathbf{u}$ is divergence-free. The Poisson equation (Equation (4.8)) is a partial differential equation, which can be solved numerically by using a finite difference approximation scheme. Equation (4.8) corresponds to a system of linear equations on the form:

$$Ap = b \tag{4.9}$$

where A is a sparse matrix, p is the pressure field, and b is a vector of scalar values. Each row in the system corresponds to one voxel in the discretization of the domain, and the system therefore consists of $N = N_x N_y N_z$ equations.

The three steps required to perform the projection step are described in turn.

1. Create b-vector

The right hand side of Equation (4.8), $\nabla \cdot \mathbf{u}^*$, requires the discrete divergence of $\mathbf{u}^*$ to be computed. The discrete divergence operator is defined as [Fer04]:

$$(\nabla \cdot \mathbf{u})_{i,j,k} = \frac{(x_{i+1,j,k} - x_{i-1,j,k} + y_{i,j+1,k} - y_{i,j-1,k} + z_{i,j,k+1} - z_{i,j,k-1})}{h} \tag{4.10}$$

where $\mathbf{u}_{i,j,k} = [x, y, z]_{i,j,k}$. This vector must be computed once for each timestep.

2. Solve Poisson Equation

Once the b-vector is computed, the linear system in Equation (4.9) is ready to be solved. This is a matter of choosing and implementing a suited method for this problem instance. The characteristic for this particular case, is that the coefficient matrix A is very sparse. To store this matrix in memory would be a waste of resources, as the majority of the elements are equal to zero[1]. Instead, the coefficient matrix elements are generated when needed. This requires less memory to be used, and should perform better because less data needs to be transported from memory to the CPU. The drawback is that it makes it difficult to use a high performance linear algebra library like BLAS or PETSc.

To solve the system, the Successive Over-Relaxation (SOR) method is used. SOR is an extrapolation of the Gauss-Seidel (GS) method, which takes a weighted average of the previous and the computed GS iterate to speed up convergence [BBC+94]. This can be expressed as:

$$\mathbf{x}^k = (1 - \omega)\overline{\mathbf{x}}^k + \omega \mathbf{x}^{k-1} \tag{4.11}$$

where $\overline{\mathbf{x}}^k$ is the k-th GS iterate, and $\omega \in [0, 2]$ is the extrapolation factor. The optimal ω is problem dependent, and cannot be found in general for most problem instances. Based on trial and error, a value of around 1.7 has been found to provide good results.

Both [Sta99], [FSJ01], [Vik03] and [AL04] use the Conjugate Gradient (CG) method [She94] with good results. However, SOR is chosen because of its simplicity and ease of implementation. In addition, SOR is easier to parallelize than CG, which has some intricate data dependencies between the computations in the algorithm. Chapter 5 discusses in detail how SOR is implemented and parallelized. It is based on the idea of [Bau78], which is basically to disregard any synchronization in the GS iteration.

This part of the thesis could benefit from a little more research into use of other linear system solvers, as well as how to choose the optimal ω. However, because of time limitations and scope, this is not given priority.

3. Project Velocities

After the pressure field has been computed, the last step is to use the pressure field to force $\mathbf{u}^*$ to be incompressible. That is, apply Equation (4.6) by subtracting the pressure gradient

[1]More specific, the diagonal element in each row is equal to the negative number of unoccupied neighbor voxels. Off-diagonal elements in column i are equal to 1 if the neighbor voxel corresponding to row i is unoccupied and not on the boundary of the domain. If the voxel is occupied, the entire row of the system is set equal to zero.

from the intermediate velocity field. The discrete gradient is computed using the following finite difference scheme [Fer04]:

$$(\nabla p)_{i,j,k} = \left[\frac{p_{i+1,j,k} - p_{i-1,j,k}}{2h}, \frac{p_{i,j+1,k} - p_{i,j-1,k}}{2h}, \frac{p_{i,j,k+1} - p_{i,j,k-1}}{2h}\right] \tag{4.12}$$

4.2.3 Boundary Conditions

Because computations are performed on a finite domain, the domain boundaries have to be given special treatment. [Sta99, FSJ01, Sta03] use closed boundaries to prevent any flow into or out of the domain. This is achieved by setting the velocity component normal to the boundaries to zero. That way, no flow is allowed to pass through the boundary. For internal boundaries as well as the ground, these boundary conditions are applied here as well.

However, if no flow is allowed to enter the domain there will be no wind, because no other forces create any flow. Thus, for the purpose of the wind simulation, the flow into the domain will actually to a large extent define the velocity field, except for flow around objects inside the domain.

In this thesis, the boundary conditions are based on the method from [AL04]. The idea is to set the velocities on boundary voxels to some chosen value instead of zero. The boundary velocity values can also be varied with time to give a more dynamic and chaotic appearance, but this is not done in this thesis.

For the pressure field, the Neumann boundary condition is used, as in [Sta99, FSJ01, Sta03]. It says that the pressure gradient is equal to zero across the boundaries, or $\frac{\partial p}{\partial n} = 0$ on ∂D. This is easily accomplished by setting the pressure on the boundary voxels to the pressure on neighbor voxels.

The initial internal velocities are set to the defined boundary value when the simulation is started.

4.2.4 Final Algorithm for Wind

The final algorithm for simulation of wind is summed up as Algorithm 2.

4.3 Accumulating Snow

In Chapter 3, different methods for modeling accumulation of snow were presented. For this thesis, there are the requirements that snow must be accumulated gradually when snowflakes hit obstacles, and that rendering can be performed in real-time.

While Fearing's model [Fea00b] produces very good visual results, it is unsuitable for use here for a number of reasons. First, rendering is done by means of commercial rendering software, and requires a lot of time. Second, snow accumulation does not track each snowflake, but insert snow into the scene in bursts of a large number of snowflakes. Third, the process of

Algorithm 2 Simulate wind

```
1: Initialize $\mathbf{u}$
2: Set boundaries on $\mathbf{u}$
3: while new timestep is needed do
4:    Self-advect velocities using the semi-Lagrangian method, Eq. (4.3)
5:    Set boundaries on $\mathbf{u}^*$
6:    Create $b$-vector, Eq. (4.10)
7:    Solve Poisson's equation using SOR
8:    Set boundaries on $p$
9:    Project velocity field into mass-conserving field, Eq. (4.6) and (4.12).
10:   Set boundaries on $\mathbf{u}$
11: end while
```

accumulation is not made with real-time applications in mind. As the snow cover is made of a lot of small triangles, collision detection will be expensive because potential collisions between all snowflakes and triangles must be checked.

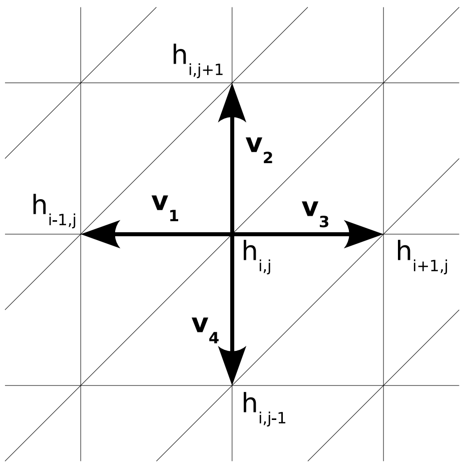
The method presented by [HAH02] is chosen for use here, because it is considered to be a reasonable trade-off between visual quality, complexity, and resource requirements. The method is based on the idea of placing 2D height matrices on all surfaces where snow can accumulate, to store the snow depth. When a snowflake hits a surface, the nearest height value is increased. This fits nicely into the falling snow model used, because for each timestep, when snowflakes are moved, intersections between surfaces and snowflakes can be checked. Also, [HAH02] suggested in the future work section of the paper, that wind should be included in the model, so more snow would accumulate near walls and around corners where turbulent air exists. This will be realized in this thesis, with the falling snow model which includes wind.

If a snowflake hits a surface, the nearest height value is increased. An advantage with this approach is that collisions only need to be checked for surfaces, not for all the triangles in a polygon representation of the snow cover. If the snow cover is thin this is an insignificant approximation, but for thick snow cover it may be imprecise. However, for small or moderate amounts of snow, this it should be a reasonable approximation.

4.3.1 Rendering

To render a snow surface, the height matrix is used to create a triangulation. [HAH02] discusses different ways to create this triangulation to avoid patterns resulting from the orientation of the triangles. However, only a simple, regular triangulation as shown in Figure 4.3 is used here. The reason is that it is desirable to set focus on other aspects than how to create polygonal approximations.

To create a smooth and nice looking surface based on the triangle representation of the snow cover, some technique must be employed to smooth out the edgy surface. This part is also kept simple, by using OpenGL's shading facilities [HB04]. This requires specification of the normal vector at each triangle vertex. To compute this vector, the trick from [HAH02] is used, where four gradient vectors are computed from the height values. These vectors are



$$\begin{aligned}
\mathbf{v_1} &= [-1, 0, h_{i-1,j} - h_{i,j}] \\
\mathbf{v_2} &= [0, 1, h_{i,j+1} - h_{i,j}] \\
\mathbf{v_3} &= [1, 0, h_{i+1,j} - h_{i,j}] \\
\mathbf{v_4} &= [0, -1, h_{i-1,j} - h_{i,j}]
\end{aligned}$$

Figure 4.3: Triangulation of height matrix. Gradient vectors used for shading calculation are shown.

shown in Figure 4.3. From these vectors, two normal vectors can be found by computing the cross product. The average of these vectors is used as the vertex normal:

$$\mathbf{n} = \frac{\mathbf{v_3} \times \mathbf{v_2} + \mathbf{v_1} \times \mathbf{v_4}}{2}$$

At the boundaries however, only two gradients can be computed, so the cross product itself is used as the normal. This simple shading technique has proved to be sufficient to create a smooth surface, and has the advantage of being hardware accelerated if the available graphics hardware supports it.

In the beginning of the accumulation process, the snow cover is so sparse that the color and texture of the ground shine through. With the presented approach, this is will look unrealistic, as the surface will change suddenly from having no snow to being completely covered in opaque snow. This is solved by taking a weighted average of the color of the ground and the snow when the snow height is below a predefined threshold. When the color values are defined at the triangle vertices, OpenGL will average the color over the triangle, to give it a smooth appearance [HB04].

4.3.2 Performance Considerations

The performance critical aspects of this snow model component, is the number of height matrices and the resolution of them. The number of matrices will increase with the scene complexity, and slow down collision detection between snowflakes and surfaces. This could be solved by employing some spacial subdivision scheme, reducing the asymptotic time complexity for collision calculations from $O(sn)$ to $O(s \log n)$, where s is the number of snowflakes, and n is the number of surfaces. This is not pursued any further because of the relatively small scenes that will be used here, but is highly relevant is the scene complexity increases significantly.

Ideally, each height value should cover an area of the same order as a single snowflake. However, that would be impracticable for real-time applications. The computation time for generation and rendering of the snow cover are directly dependent on the resolution of the height matrices. Each vertex of a triangle requires one coordinate, one normal vector, and one color to be sent the GPU for processing. For every triangle, this results in nine vectors with three components each. When single precision floating point numbers are used, this is 108 bytes per triangle. At 25 fps and 15000 triangles, 40.5 MB/s must be sent to the GPU.

4.4 Complete Snow Model

Figure 4.4 shows how the different components of the snow simulation works together to simulate one timestep (or frame). The boxes denote a computation, and the arrows denote a data dependency or data flow between the computations.

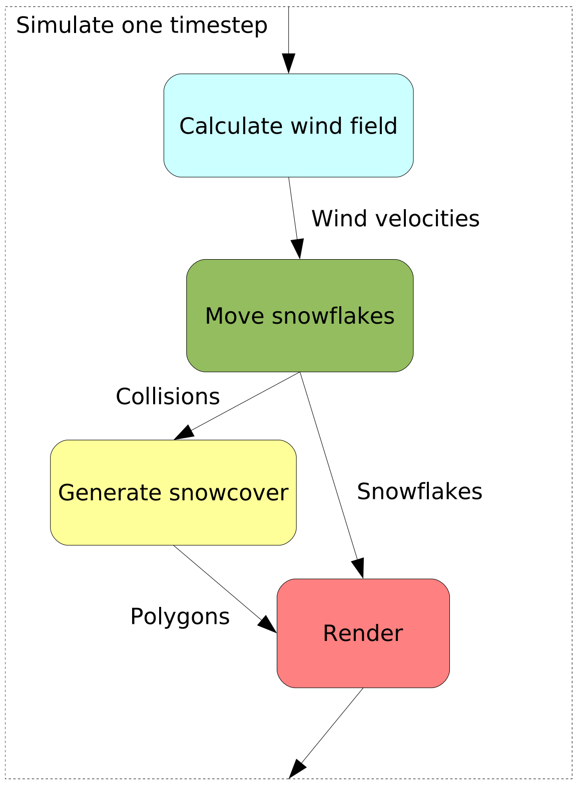


Figure 4.4: Overview of dependencies between model components

First, a new wind field must be computed, using Algorithm 2 for wind simulation. After this, new positions for all snowflakes are computed, using Algorithm 1. This also includes a step to detect collisions between snowflakes and surfaces. Actually, the figure may be a little misleading by regarding step 3 of Algorithm 1, because V^t_{wind} is needed, not $V^{t+\Delta t}_{wind}$. In other words, the movement step in Figure 4.4 depends on the wind computation from the previous timestep, not the current. Effectively, this means that the wind computations can be performed in parallel with the rest of the computations for the current timestep. This fact will be exploited in the next chapter.

After new snowflake positions and collisions have been computed, the polygonal representation of the snow cover with vertex normals can be created. When the snow cover representation is ready, the data describing scene objects, snow cover and snowflakes are sent to OpenGL for rendering in the render step. Finally, the computations for a new timestep can be started.

4.5 Discussion

The snow model presented in this chapter has been put together from various techniques found in previous work, and adjusted to be compatible with each other for a snow simulation model that handles both falling and accumulating snow in real-time. There are, however, a number of elements that have been given lower priority, and can therefore be viewed as weaknesses in the work. This is particularly present in the rendering portion, but also in the dynamics of the model.

One can argue that by incorporating more elements to improve the model, it will eventually be unsuitable for real-time use. However, the author believes that the most prominent shortcomings can be remedied with relatively simple techniques that will not sacrifice the performance requirements.

Regarding the snow accumulation model, first, there is no back-coupling between the wind field and the snow cover. As snow accumulates, the space occupied by the snow cover will not be treated as occupied. Second, intersection testing takes place between snowflakes and objects, not the snow cover. The technique from [FO02] could be used to treat these situations correctly. Third, large amounts of snow will probably not look good, because effects as bridging and avalanches are not handled. This is more problematic, but could be subject to further research, for instance to improve the performance of Fearing's model [Fea00b].

A relatively small amount of work would be necessary to render the snowflakes and the snow cover better. By using the billboarding technique outlined in Section 4.1.2, the snowflakes would give a more realistic impression.

The snow cover will look too smooth and glossy compared to reality. A solution to this could be bump mapping [HB04], which is a technique that perturbs the normal vector over surfaces to imitate the small scale details.

Chapter 5

Implementation

This chapter describes how the model presented in Chapter 4 has been implemented to achieve good performance. First, the platform that has been used to implement the model is presented. Next, the implementation itself is presented and discussed with emphasis on performance issues and techniques to achieve good performance. Finally, parallel techniques used to boost performance on symmetric multiprocessors (SMPs) are presented.

5.1 Platform Specification

The implementation is targeted towards graphics-enabled desktop computers and workstations. That is, computer systems with the capability of displaying high-speed graphics on a screen, accelerated by special purpose hardware. It is very common to find such hardware in today's computers if a screen is connected. Two or more processors in a symmetric multi processor (SMP) organization is preferred, but systems with only one processor are also of interest. It is not common to find more than one processor in today's consumer hardware, but the motivation for choosing this platform is based on the fact that chips with more than one processor core are entering the market with full momentum. The same parallelization and optimization techniques should apply to these processors as well, even though additional problems imposed by this new architecture are likely to arise. In addition, processors with Hyper-Threading [Int06] support are explored.

An interesting direction could be to parallelize for PC clusters, using MPI. However, the interconnection network for clusters is usually too slow too make such a solution feasible. As the delay often can be in the order of 1ms, at a frame rate of 30fps, only the delay may constitute a total of 30ms for one data exchange per frame. In addition, time is needed for the data exchange itself, as well as computations. Because there will probably be many data exchanges per frame, this approach is not investigated any further. However, it may be an interesting direction for large offline applications, but this not a topic in this thesis.

As execution speed is crucial in this project, the programming language C++ is chosen. Based on personal preferences and previous experience, the GNU/Linux operating system along with the GNU C++ compiler, is used. For graphics, the de-facto standard library for high performance graphics, OpenGL[1], is used. For user input and window management, the Simple DirectMedia Layer (SDL)[2] is used. SDL is a cross-platform library for access to low level media devices as keyboard, mouse, OpenGL and the frame buffer.

The natural parallel programming model for SMPs is shared memory by means of multi-threading. In this model, a program consists of multiple threads that share the same memory address space. These threads can be scheduled to run on the available processors by the operating system. To create and manage threads, library support from the operating system is needed. The POSIX Threads, or Pthreads library is chosen for this purpose, because is it the de facto standard for threads management on UNIX systems. For cross-platform compatibility with Microsoft Windows, the pthreads-win32[3] library is suggested as a solution.

5.2 Performance Considerations and Programming Guidelines

Parallel computing may not be the ultimate answer to all performance critical applications. Before parallel techniques are applied, it should be assured that the program code is optimized for speed. The optimal solution is to avoid parallelization after all if the serial program can optimized to reach the required performance. Also, each processor used by a parallel program should be utilized to its maximum. General optimization techniques are presented by for instance [Ger02].

Ideally, the compiler should take care of generating optimized code from the high-level program code written by the programmer. Today's compilers are indeed very complex pieces of software that do their job well, however, the compilers have no further insight into the nature of the algorithms than the mere code itself. To give the compiler the best possible basis to create well performing code, the programmer should keep in mind some of the most fatal performance degrading pitfalls.

Two characteristic features of today's computer and processor architectures are long execution pipelines and extensive reliance on effective utilization of the memory hierarchy. The long pipelines allow many instructions to be executed in parallel by starting to execute new instructions before the previous instruction has finished. This introduces a problem with conditional jumps, because the processor can not foresee which instruction to branch to before the condition has been evaluated. If the processor makes the wrong guess, all instructions started after the jump instruction must be flushed from the pipeline, and thereby a performance penalty is imposed. To avoid the penalty of many branch-mispredictions, it is highly desirable to eliminate conditional branches in performance-critical loops as demonstrated in pseudo code in Table 5.1. Also, by keeping the loop bodies simple, it is easier for the compiler to apply techniques as loop unrolling and vectorization.

Another important technique to improve performance, is to reduce the number of cache misses. If the processor requests a data item not present in cache memory, it must wait for the much slower main memory to fetch the item. When a cache miss is encountered, not only the data item in question is transferred to cache, but data items close in virtual memory are

[1]`http://www.opengl.org/`
[2]`http://www.libsdl.org/`
[3]`http://sourceware.org/pthreads-win32/`

Inefficient	**Efficient**
	`special case for 0`
`for i = 0..n:`	
`   if i = 0 or i = n-1:`	`for i = 1..n-1:`
`      special case`	`   normal case`
`   else`	
`      normal case`	`special case for n-1`

Table 5.1: Optimization for branch prediction

retrieved to fill an entire cache line. This fact can be exploited by the program by letting consecutive instructions work on data close to the previous data to avoid new cache misses. Table 5.2 illustrates how this technique can be applied by incrementing the lowest dimension in multi-dimensional arrays in the innermost loop. This way the array elements are accessed sequentially to work with data that is already loaded into cache, to minimize the number of cache misses.

Inefficient	**Efficient**
`for j = 0..m:`	`for i = 0..n:`
`   for i = 0..n:`	`   for j = 0..m:`
`      access array[i * m + j]`	`      access array[i * m + j]`

Table 5.2: Optimization for cache efficiency

5.3 Sequential Implementation

This section describes how the algorithms presented in Chapter 4 are implemented, and discusses techniques applied to maximize performance before any explicit parallelization is introduced.

5.3.1 Falling Snow

The basic data object for simulating snowfall is the snowflake. The properties describing a snowflake are listed in Table 4.1 on page 20. The algorithm is implemented as a loop which traverses an array with all snowflake objects. With a total of 11 floating point numbers needed to describe each snowflake and 10.000 snowflakes ($10.000 \cdot 11 \cdot 4$bytes $\approx$ 430KB), this data structure may be small enough to fit into L2 cache on many computers. In any case, the snowflakes are accessed sequentially to minimize the number of cache misses. However, to compute the next position and velocity for a snowflake, the wind velocity at the snowflake's position must be interpolated. These look ups in the velocity field occurs seemingly at random, because the positions for subsequent snowflakes are independent, and may therefore result many in cache misses. Depending on the wind field resolution, the velocity field may also fit into cache, so the penalty will be minimal.

The second step in the falling snow simulation is to check for intersections between snowflakes and objects in the scene to determine where snow should be accumulated. This is realized in the same loop as the movement simulation in order to use the snowflake data while they are still in cache anyway, as shown in pseudo code:

```
for each snowflake:
    snowflake.updatePosition()
    snowflake.checkForCollisions()
```

Collision checking is considerably simplified by the assumption that objects in the scene are restricted to rectangular "blocks". To check if a snowflake has intersected a surface, it is enough too determine if the current and previous position are on each side of the surface. If they are, the point of intersection is computed to find out if the point is within the bounds of the surface. To save computation time, only horizontal surfaces that are enabled for snow accumulation are checked for collisions. Even though snowflakes are allowed to enter objects, this will happen infrequently because the wind velocity is zero near the vertical surfaces, or walls. It is therefore considered a reasonable approximation, and gives better performance because the collision detection is exposed to computations involving code with many branches.

5.3.2 Wind Simulation

The algorithms for wind simulation operate on voxel data objects, where the properties related to the wind simulation are defined. All of the four steps (self-advect, build b-vector, SOR and project), are implemented as loops over all the voxels. The boundary conditions are applied before or after the main loop, as the principle in Figure 5.1 suggests.

Another optimization is performed when the b-vector is built and the linear system is solved with SOR. With every voxel there is an associated bit mask that tells whether the neighbor voxels, including itself, is occupied, as in [LLW04]. If the voxel is occupied, the corresponding element of the b-vector is set to zero. Instead of using an if-statement to check this, the expression for calculating the b-vector can be multiplied by an expression that is equal to 0 if the voxel is occupied and 1 if not:

$$b_{i,j,k} = ((m_{i,j,k} \oplus 1) \wedge 1)(\nabla \cdot \mathbf{u}_{i,j,k})$$

$m_{i,j,k}$ is a bit mask that has its least significant bit equal to 1 if the voxel is occupied, and 0 if not. $\oplus$ is the bit wise "xor" and $\wedge$ is bit wise "and".

5.3.3 Accumulating Snow

The implementation of accumulating snow visualization is basically concerned with translating the snow height matrices into a triangle representation of the snow cover. As the snow cover may change between timesteps, the representation must be regenerated for each frame, but only the coordinates that are able to change are updated (i.e. vertical coordinates). To efficiently transfer the vertex data to the GPU, the vertex array functionality in OpenGL [HB04] is used. The idea behind this is to lay out the vertex data linearly in memory, and hand over just a pointer to the data to OpenGL. This way it is up to the OpenGL implementation to utilize the Direct Memory Access (DMA) features of must computer systems to transfer the data from memory to the GPU without having to go through the CPU first, and letting the program continue with its operation while the data are being transferred. In this case this is particularly useful, because some vertex data are static, and does not need to be touched by the CPU for each timestep.

5.4 Parallelization

The algorithms for snow simulation are well-suited for the data-parallel approach to parallelization. The routines of computational significance are implemented as loops over arrays of data. Because the calculations in each iteration of the loops are independent of the other iterations, the iterations can be distributed among the threads.

Figure 5.1 shows more in detail how the steps in the simulation algorithm are related. The numbers in parenthesis denotes the number of threads that can be assigned to the task in parallel, by dividing the loop index range between the threads. In the wind field computations, the data structures are divided into disjoint sets of voxels, so that different threads work on different locations in the simulation domain. The snowflake movement is parallelized by assigning an equal number of snowflakes to each thread, and snow cover generation is parallelized by letting different threads generate snow cover on different parts of the objects. When doing this kind of splitting up, it is important that the division is done in the slowest increasing dimension, so that each thread works on a continuous sections in memory to utilize the cache memory best possible.

There is one exception to this fairly simple parallelism. More specifically, the SOR linear solver, which is based on the Gauss-Seidel iteration (GS), uses data computed in previous loop iterations to speed up convergence, compared to Jacobi iteration. To overcome this sequential obstacle, the algorithm is parallelized by pretending that these dependencies do not exist, and distributing loop iterations among threads as the straight forward parallelization of Jacobi would. Even though this may seem annoying in the eyes of purists, the technique has been given a theoretical analysis for GS by [Bau78], and is proved to converge under basically the same assumptions as GS and Jacobi. Intuitively, the algorithm can be viewed as “chaotic Gauss-Seidel”, because of the non-determinism in whether new data are available or not. No literature directly related to “chaotic SOR” has been found, but the results show that it works well for this particular problem instance.

Another potential race condition exists in the interface between snowflake movement and snow cover representation. When snowflakes collide with objects, there is no synchronization in the part of the code where snow height values are incremented. Because it is highly unlikely that more than one thread will attempt to increase a height value at the exactly same time, further work on this problem is abandoned. No visual degradation can be observed and attributed to this fact when adding another thread.

Compared to the fluid dynamics parallelization carried out by Bryborn et al. [BKM$^+$00], more computations run in parallel in this implementation. They leave the entire projection step unparallelized, arguing that their sequential implementation runs faster than a parallel

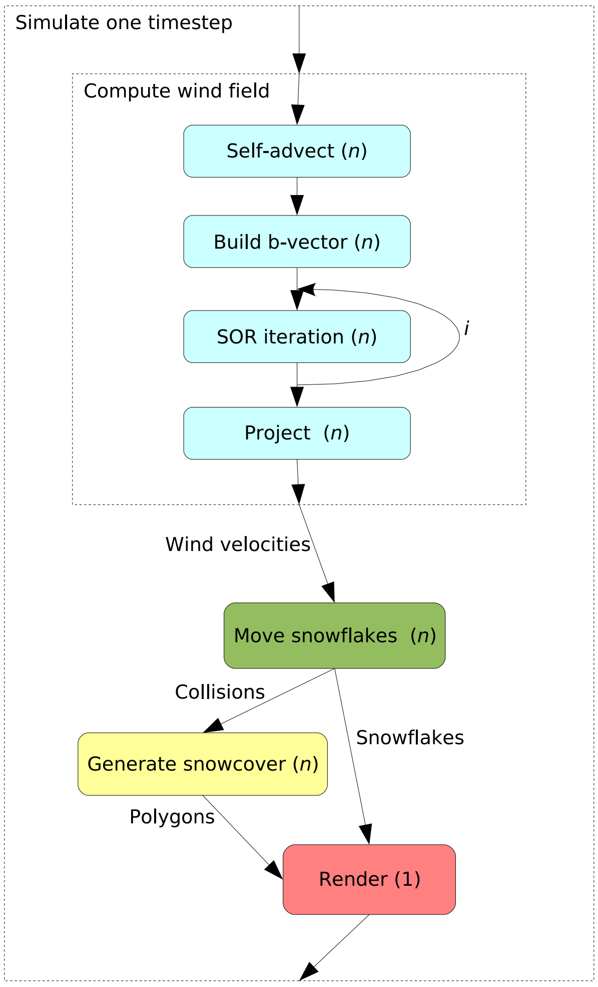


Figure 5.1: Schematic overview of units of work and data flow between them. The numbers in parenthesis denotes the number of subtasks that can be executed in parallel, where n is any number.

implementation because of synchronization in the Poisson solver. However, they let other computations run in parallel with the projection, so that the penalty for the serial section is reduced. It is true that synchronization is needed between each SOR iteration, however, as will be seen in the results in Chapter 6 a parallelization gives better performance than the serial equivalent anyway.

Up to the rendering step, everything can be parallelized in a data-parallel manner. However, the rendering is inherently sequential, because only one thread is allowed to make calls to OpenGL. This leaves the rest of the threads idle while one of them is making the necessary OpenGL calls.

In the previous chapter, it was pointed out that the outcome of the wind field computations in Figure 5.1 are not needed until the next timestep. In other words, the wind field computation boxes could have been placed anywhere in the dependency graph and carried out completely in parallel with the rest. By using a separate thread for wind field computations, the overhead of synchronization would be avoided. However, to utilize the computing resources efficiently, the wind field computations must have taken the exactly the same amount of time as computations they run in parallel with. Alternatively, some sophisticated load balancing system could be used to distribute work among threads. Because the computations need to be performed once for each timestep anyway, they are placed first and executed in parallel to utilize the processors to their maximum. As will be further described in the next section, the wind field computations for timestep $n + 1$ can be started before the rendering for timestep n have finished, to avoid leaving processors idle. This will be implemented by a fairly simple task-queue.

5.5 Task-Parallel Implementation

The pure data-parallel approach described in the previous section leaves all but one processor idle while rendering is performed. As rendering will take significant amounts of time, this seriously limits the possible speedup. If rendering takes 25% of the time for each frame with one processor, the maximum possible speedup for an infinite number of processors is 4, according to Amdahl's law. Also, it was pointed out that the wind simulation is independent of every step in the current timestep. The only requirement for the execution order in Figure 5.1 is that snowflake movement, snow cover generation and rendering are executed strictly sequential. Note that the steps themselves still can be parallelized as in the previous section.

This introduces the motivation for extending the data-parallel implementation with a task-parallel piece to start wind field computations for the next timestep before rendering has finished. To implement this, the notion of a *task* is introduced, which is the work one thread does in one step of Figure 5.1, for instance "perform self-advection on the first half of voxels". The basic idea, based on [Vik03], is to untie the coupling between threads and tasks, so that the threads choose the next available task, instead of waiting for the other threads for finish their share of the work. Figure 5.2 illustrates how the task-parallelism shortens the time needed to compute one timestep. Each gray box represents a task. The size of the tasks may be out of proportion with the actual time they take to execute in the figure, but the time will be dependent on the configuration with respect to wind field resolution, number of snowflakes and objects, and snow cover resolution. Also, the SOR-task is divided into one task per iteration, so the parallelism is even more fine-grained than what is apparent in the figure.

To realize the task-parallel approach, some mechanism for assigning tasks to threads is needed. This is implemented as simple as possible, but not simpler. To allow a certain degree of flexibility, a queue-system is implemented. When a thread finishes a task, it queries the queue for the next available task, as demonstrated in pseudo code:

```
while (running):
    task = queue.getTask()
    task.execute()
    queue.finishTask(task)
```

The queue is realized as a linear array of tasks, where all tasks are preallocated at startup and reused for each frame to avoid the overhead of allocation and deallocated of task objects,

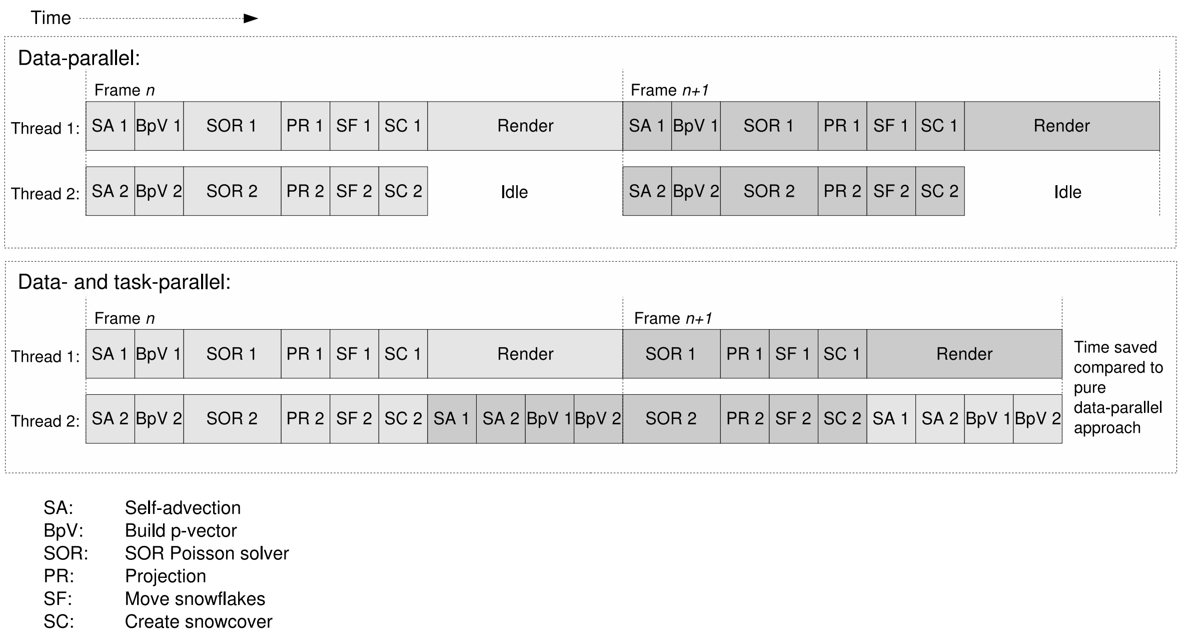

Time
Data-parallel:
Frame n
Frame n+1
Thread 1: SA 1 BpV 1 SOR 1 PR 1 SF 1 SC 1 Render SA 1 BpV 1 SOR 1 PR 1 SF 1 SC 1 Render
Thread 2: SA 2 BpV 2 SOR 2 PR 2 SF 2 SC 2 Idle SA 2 BpV 2 SOR 2 PR 2 SF 2 SC 2 Idle
Data- and task-parallel:
Frame n
Frame n+1
Thread 1: SA 1 BpV 1 SOR 1 PR 1 SF 1 SC 1 Render SOR 1 PR 1 SF 1 SC 1 Render
Thread 2: SA 2 BpV 2 SOR 2 PR 2 SF 2 SC 2 SA 1 SA 2 BpV 1 BpV 2 SOR 2 PR 2 SF 2 SC 2 SA 1 SA 2 BpV 1 BpV 2
Time saved compared to pure data-parallel approach
SA: Self-advection
BpV: Build p-vector
SOR: SOR Poisson solver
PR: Projection
SF: Move snowflakes
SC: Create snowcover

Figure 5.2: Illustration of how the task-parallel implementation eliminates idle threads while rendering. Note that the size of the tasks in the figure may be out of proportion with the actual time they require to execute.

as in [Vik03]. When a thread requests a task, the array is traversed and the first task marked as ready, is returned and marked as running. To manage the dependencies between tasks, each task object holds pointers to all tasks (successors) that require the task in question to be completed before execution is started. Also, each task object counts how many prerequisite tasks have finished (`parentsDone`). When a task finishes, the `parentsDone` variable of all successors is increased, and if it is equal to the number of parents, the task's status is set to ready. Figure 5.3 illustrates the concept for three tasks, where task 3 requires both task 1 and 2 to be finished. Task 1 is being executed by a thread, and task 2 has finished.

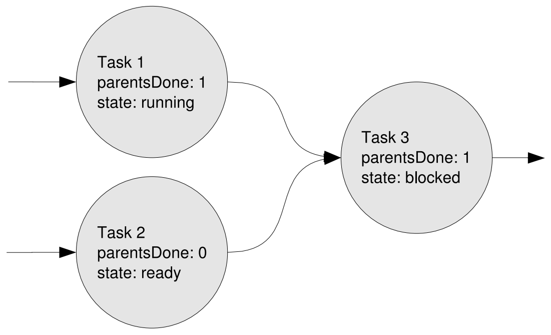


Figure 5.3: Example of setup of tasks and dependencies. Task 3 requires task 1 and 2 to be finished in order to execute. Task 2 has finished, while task 1 is still running.

Figure 5.4 shows in details how the dependencies between the tasks are set up for two threads. Compared to Figure 5.1, an extra step has been introduced; advance snowflakes. Its purpose is to copy snowflake positions to a location in memory that is used by the render task to draw the snowflakes. The benefit is that rendering can happen while snowflake computations are taking place, because the computations do not touch the data structures that are used to render the snowflakes. This is particularly useful when there are many snowflakes and the wind field is coarse, to avoid that the second thread has to wait for the first one to finish rendering.

The step “increment time” is just a maintenance and synchronization task to increment the timer, and collect some statistics about the frame rate between timesteps.

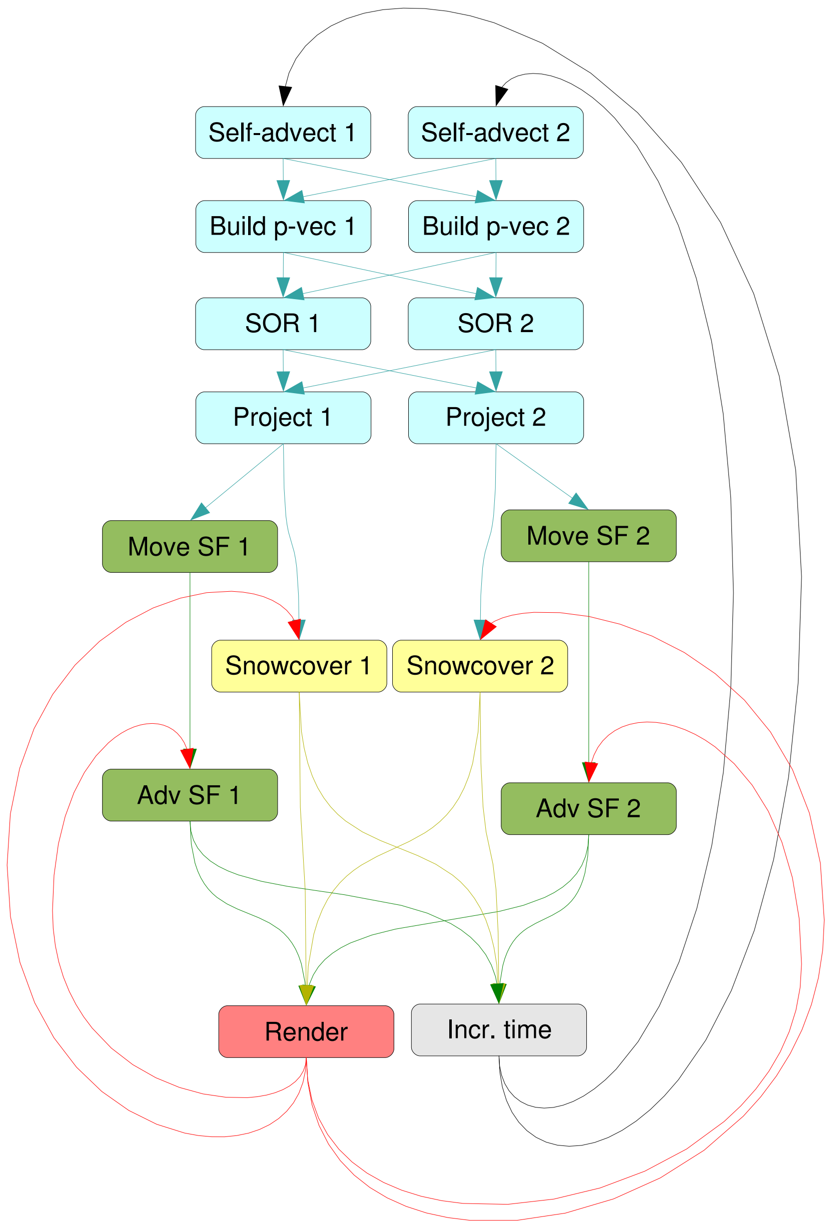


Figure 5.4: Detailed setup of tasks and dependencies for two threads

Chapter 6

Results

In this chapter, the implementation from the previous chapter is investigated in terms of performance, parallel efficiency, and visual quality.

6.1 Test Hardware

Specification of the test machines used for development and performance measurements are given in Table 6.1. The first row shows the abbreviation used for the respective computer later on.

[SingleHyper] is a desktop computer with a Hyper-Threading (HT) enabled processor. Hyper-Threading is a technology that duplicates the architectural state of the processor—but not the execution units—so that the operating system sees the processor as two logical processors. As two threads can be scheduled to run simultaneously, the processor can be utilized by other threads when it otherwise would stall due to cache misses, branch mispredictions or data dependencies. [Int06] reports that a performance boost of 15-30% can be achieved with this technology.

	[SingleHyper]	**[Dual]**	**[DualHyper]**
Processor	Pentium 4	Pentium Xeon	Pentium Xeon
Parallelism	Hyper-Threading	Dual SMP	Dual SMP with Hyper-Threading
Clock Freq.	3.00 GHz	3.2 GHz	3.4 GHz
Memory	1 GB	2 GB	3 GB
L2 Cache	1 MB	2x 1 MB	2x 1 MB
GPU	NVidia GeForce FX 5500	NVidia Quadro FX 3400	NVidia Quadro FX 3400
Video Memory	128 MB	256 MB	256 MB
Bus type	AGP	PCI Express 16x	PCI Express 16x
OS	Linux 2.6.15	Linux 2.6.16	Linux 2.6.15

Table 6.1: Test hardware used for collecting performance data

[Dual] is a workstation with two processors organized in a SMP fashion.

[DualHyper] is very much like [Dual], but has slightly better processors, both in terms of clock frequency, and that they are Hyper-Threading enabled. In this arrangement, the operating system's awareness of the Hyper-Threading technology is important for good performance. A performance issue can arise if the operating system believes that there are four physical processors, and schedules two threads on one CPU, and leaving the other idle. The 2.6-series of the Linux kernel is aware of this issue, and knows how to optimize thread scheduling [Opd04]. Other operating systems' capabilities have not been investigated, as they will not be considered for the measurements.

6.2 Test Parameters

A variety of means for configuring the snow simulation is available, as summed up here:

1. **Number of threads:**
   Number of threads concurrently performing computations. To achieve maximum performance, this number should be equal to the number of threads the computer is capable of scheduling simultaneously, i.e. the number of processors.

2. **Type of parallelism:**
   Sequential, data-parallel or task-parallel implementation, as described in the previous chapter.

3. **SOR iterations:**
   Number of iterations used to solve the Poisson equation for computing the pressure field. By altering this parameter, realism can be sacrificed for speed.

4. **Wind field resolution:**
   The number of grid points used to define the wind field. Less grid points means better performance, but if the distance between grid points becomes too big, details in the interaction between wind and objects are lost.

5. **Number of snowflakes:**
   At all times, the same number of falling snowflakes are present in the simulation.

6. **Resolution of snow cover:**
   How many height values used to store snow depth. This can be configured specifically for every surface.

7. **Scene objects:**
   Objects are limited to being simple cubes. By adding more objects, performance is degraded, mainly because of more intersection testing.

For pure visual purposes, the following can be configured:

8. **Temperature:**
   The physical temperature in the scene. It affects the properties of the snowflakes, such as density and size.

	Small	**Default**	**Large**	**Gigant**
Wind field resolution:	20x20x5	40x40x10	48x48x12	54x54x9
Number of snowflakes:	5.000	20.000	30.000	40.000
Height value spacing:	0.5m	0.3m	0.25m	0.35m
Triangles in snow cover:	5.352	12.050	20.800	21.760

Table 6.2: Configuration for four different scene sizes

9. **Boundary wind velocity:**
   The wind velocity defined at boundaries. Intuitively, this is the large-scale wind velocity surrounding the scene.

The scene which is used for testing, is based on the scene that is used in [FO02], and shown from above in Figure 6.1. It contains three 3 meters high "buildings", and spans 25x25x8 meters. The wind field spans 20x20x5 meters, and is placed in the middle.

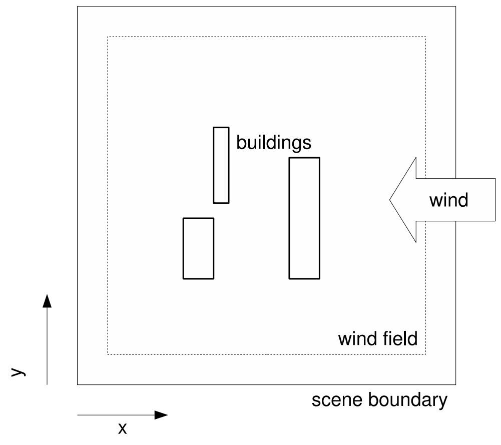


Figure 6.1: Scene setup from [FO02] seen from above.

Four different scene configurations are used in this chapter to measure performance. They are summarized in Table 6.2. Unless otherwise noted, 5 SOR iterations are used, and the temperature is set to -1° Celsius. The boundary wind velocity is set to 2m/s in the negative x-direction. This scenario has been chosen because, based on trial and error, it provides a reasonable visual quality, and is of considerable size. Also, it provides a foundation for comparison with [FO02].

6.3 Sequential Performance

To get a grasp of performance properties without parallelization, the program was profiled. Also, two different compilers were tried out.

6.3.1 Profiling

To see how the implementation behaves without regard to any parallelization, it was profiled in sequential mode. The GNU Profiler (gprof) was used to generate the profile, with one processor with the default size scene on [Dual]. Based on statistical information obtained by sampling the program counter at regular intervals, the profiler creates a distribution of time spent in different parts of the code. The distribution of time spent in the main parts of the program is shown in Figure 6.2.

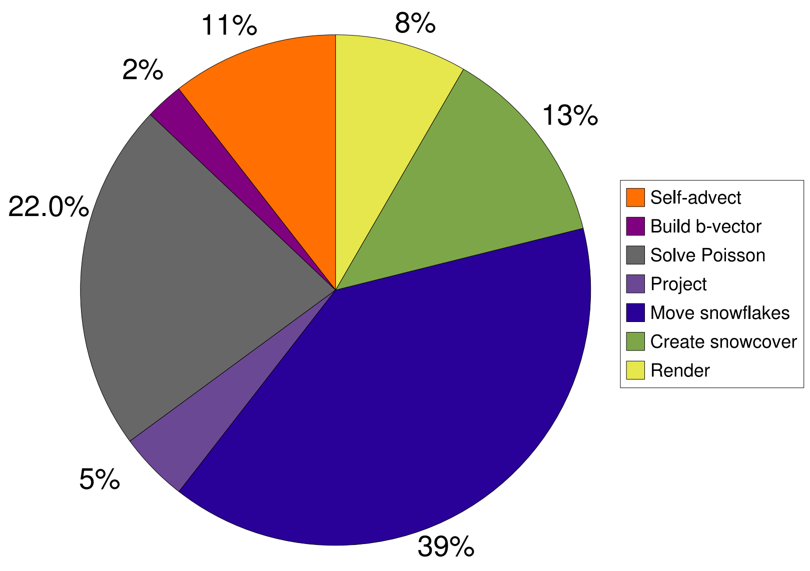


Figure 6.2: Profiling results of sequential version with the default configuration

A profile of this kind can be manipulated in every direction by altering the parameters. The parameters chosen here are considered to be reasonable, and are chosen because they are also used in later tests. In any case, it shows that the simulation of snowflake movement and wind field simulation are the most resource hungry components of the model, by together occupying 79% of the execution time.

The profile is probably slightly biased because of the monitoring overhead that comes with profiling, but also because of the more conservative compiler settings that have to be used with the profiler to obtain reliable results. In particular, the rendering part of the code will be relatively larger with aggressive compiler settings, because it is mainly concerned with shuffling data over to the GPU, and is limited by the speed of the bus between the CPU and GPU. It has therefore little benefit of optimization. The floating point intensive simulation code derive much more benefit from compiler optimization, and is likely to execute faster. This means that everything but the rendering part of Figure 6.2 is going to decrease, resulting in a distribution with a relatively bigger rendering fraction.

6.3.2 Compiler Settings and Vector Instructions

While most of the development was carried out using the GCC C++ compiler, the Intel C++ compiler was tried for the sake of curiosity. The result was a speedup of 8.1% on the default scene configuration. Because of this finding, all of the following measurements are performed with the Intel compiler instead of GCC.

By comparing the assembly output from the two compilers, some observations were made. First, the code they produce is quite different from each others. Second, a considerable more widespread application of SSE instructions is evident in the code output from Intel's compiler. This is particularly clear in the vector like operations in many of the heavy loops in the main simulation routines, i.e. where the same operation is applied in each dimension of the domain.

Even better performance could probably be achieved by using SSE instructions manually, and thereby executing four loop iterations simultaneously. This was not pursued any further because of time constraints.

6.4 Parallel Speedup

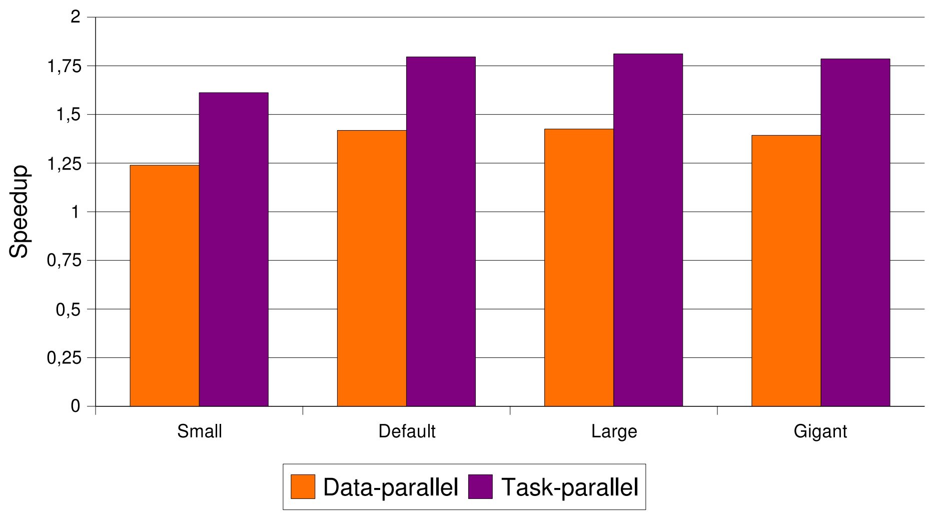
This section presents the results for parallel performance in different configurations and computer systems.

6.4.1 Data- vs. Task-Parallel Implementation

In Chapter 5, two different parallelization strategies were presented; a pure data-parallel approach that left processors idle while rendering, and a task-parallel approach that started computations for the next frame while rendering proceeded. Figure 6.3 shows a a comparison between the two approaches on [Dual] for different scene sizes.

For all scene sizes, the task-parallel implementation performs better, and achieves a significantly higher speedup than the data-parallel implementation. The absolute difference in speedup seems to be approximately constant equal to 0.4 for all scene sizes. As expected, the speedup increases with scene size, because the ratio between the overhead introduced by parallelization and the benefit of parallelization decreases. However, contradictory to this, the speedup decreases again at the gigant scene. In Section 6.5 an explanation for this is presented. Is has to do with a weakness in the parallelization of snow cover generation. Also, when the size of the tasks that are distributed among threads is increased, load balancing becomes increasingly important. When the units of work become large, there is a bigger potential for a performance penalty with uneven load.

With the gigant scene, the task-parallel implementation makes the real-time requirement, while the data-parallel fails to do so.



	Small	Default	Large	Gigant
Data-parallel frame rate	133.2	45.8	29.5	23.4
Data-parallel speedup	1.24	1.42	1.43	1.39
Task-parallel frame rate	173.3	58.0	37.5	30.0
Task-parallel speedup	1.61	1.80	1.81	1.79

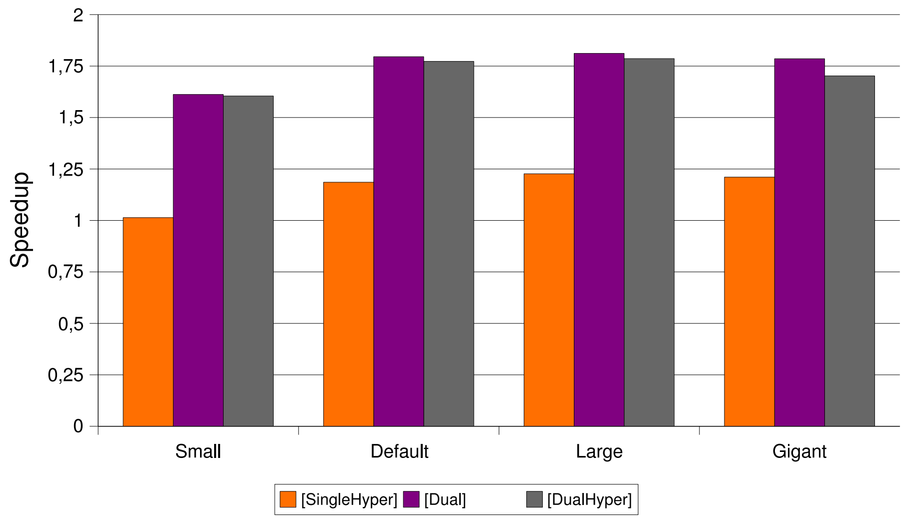
Figure 6.3: Speedup comparison between data-parallel and task-parallel implementation on [Dual] for different scene sizes.

6.4.2 Different Computer Systems

Figure 6.4 compares the achieved speedup with two threads for the computers listed in Table 6.1. Hyper-Threading is disabled on [DualHyper], so it works as a pure dual CPU computer. [SingleHyper] has only one processor, and runs two threads with Hyper-Threading enabled.

Except for the small scene, [SingleHyper] performs according to Intel, who suggests a performance improvement of 15-30% with Hyper-Threading [Int06]. For floating point intensive applications like this one, only a small speedup should be expected, as the execution units probably are kept busy a lot of the time already. However, because of the poor graphics hardware in this machine, a lot of the time used for rendering is spent waiting for data to be transferred to the GPU, as well as for the GPU to do its work. This leaves many idle cycles ready to be taken advantage of by the other thread, and thereby potential for a significant speedup. There is one catch, though, in that the synchronization is implemented as a spin-lock. If the second thread must wait for the first thread to finish rendering, it will slow down the rendering when it waits for its completion. This is probably what happens for the small scene, when the second thread only has a small amount of work to do concurrently with rendering.

Judging by the speedup figure, [DualHyper] seems to perform worse than [Dual]. However, this is not the case. Because [DualHyper] has better processors, it executes the computationally intensive routines faster, while the parallel overhead stays approximately constant. To account for this, the problem size, or scene size, should ideally have been scaled according to processor



	Small	Default	Large	Gigant
[SingleHyper] frame rate	37.2	13.4	9.2	6.9
SingleHyper speedup	1.01	1.19	1.23	1.21
Dual framrate	173.3	58.8	37.5	30.0
Dual speedup	1.61	1.80	1.81	1.79
DualHyper frame rate	184.4	60.1	39.3	30.3
DualHyper speedup	1.60	1.77	1.79	1.70

Figure 6.4: Speedup for task-parallel implementation with two threads on different machines.

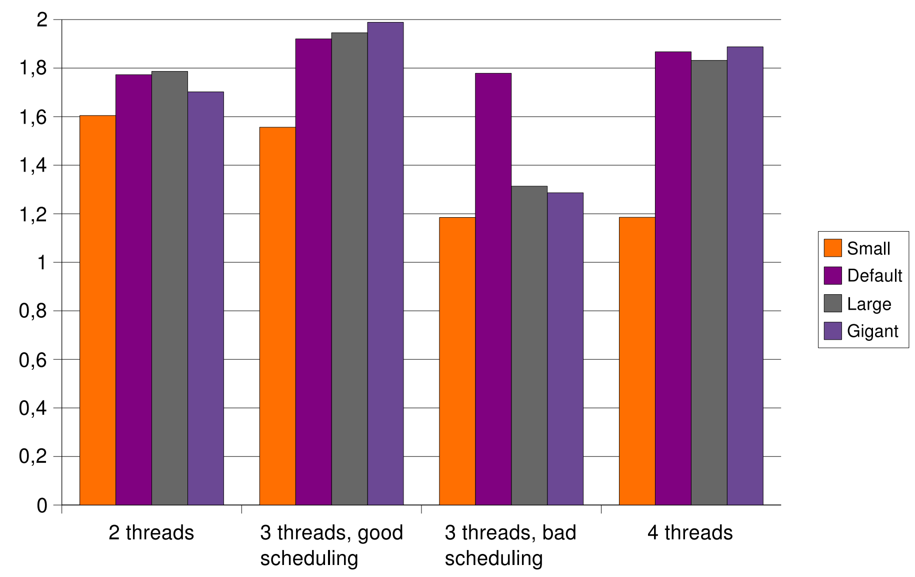
speed. In Section 6.5 the influence of varying scene size is investigates further.

6.4.3 Scaling Beyond Two

So far only one and two threads have been considered. Another interesting problem to be addressed is how more than two processors can be utilized efficiently. An inherent problem in this implementation, and to a certain degree in the approach to the problem itself, is the sequential bottleneck related to real-time rendering on a screen. As a lot of data need to be transferred to a single GPU, this approach does not scale infinitely.

Unfortunately, the practical circumstances did not allow experiments to be carried out with more than two CPUs. However, Figure 6.5 shows the speedup for 2-4 threads on [DualHyper] with Hyper-Threading enabled. This was the maximum amount of parallelism that was available due to the practical limitations. Even though four separate processors would be ideal, this experiment proves that the implementation can take advantage of more than two processors.

When executing the measurements in Figure 6.5, a considerable variation in frame rate was observed when running with three threads. The reason is that the operating system has four logical CPUs amongst which to schedule three threads, and it is not indifferent how it decides to do this. As the threads scheduled to run on the same physical CPU must compete for resources, it is highly desirable that the rendering thread is given a physical CPU for itself to



	Small	Default	Large	Gigant
2 threads frame rate	183.4	60.1	39.3	30.3
2 threads speedup	1.60	1.77	1.79	1.70
3 threads, good, frame rate	177.9	65.1	42.8	35.4
3 threads, good, speedup	1.56	1.92	1.95	1.99
3 threads, bad, frame rate	135.4	60.3	28.9	22.9
3 threads, bad, speedup	1.18	1.78	1.31	1.29
4 threads, frame rate	135.5	63.3	40.3	33.6
4 threads, speedup	1.19	1.87	1.83	1.89

Figure 6.5: Parallel speedup for two, three, and four threads on [DualHyper].

complete the critical rendering task as quickly as possible. Also, because two threads on the same physical CPU share the cache memory, they may destroy each others memory access patterns.

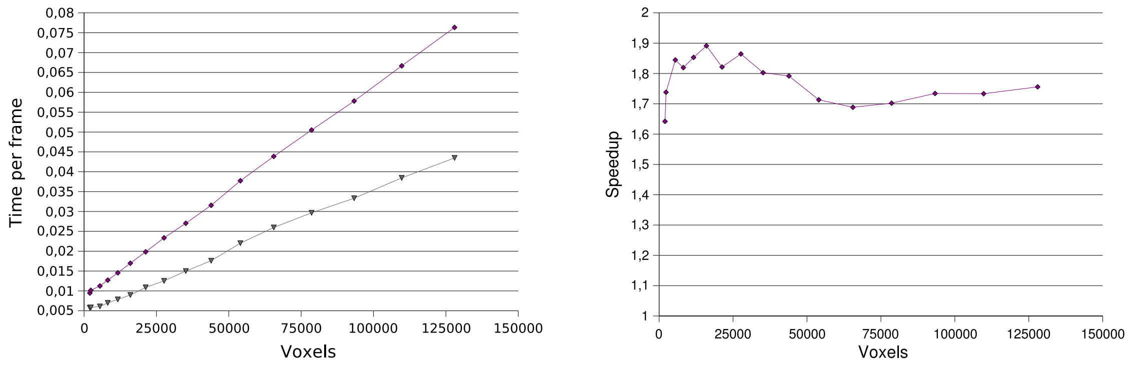
The highest measured speedup occurs with three threads scheduled in the better way. This suggests that the rendering thread is best left with its own processor. However, it is problematic to set this up properly, because of the limited practical facilities for manually mapping threads to processors in Linux. The processor on which a thread is run is largely decided non-deterministically when the program starts. Later, the threads are scheduled to run on the processor they ran on previously.

6.5 Scaling of Parameters

In Section 6.4.1 and Section 6.4.2, a decrease in speedup was observed for the gigant scene. This section investigates how selection of parameters, motivated this observation, as well as the more general desire to understand how different parameters affect performance and scalability.

The basis for the measurements are the small scene, where one of the parameters: wind field resolution, snow cover resolution, or number of snowflakes, is increased in turn. [Dual] is used for measurements.

6.5.1 Wind Field Resolution



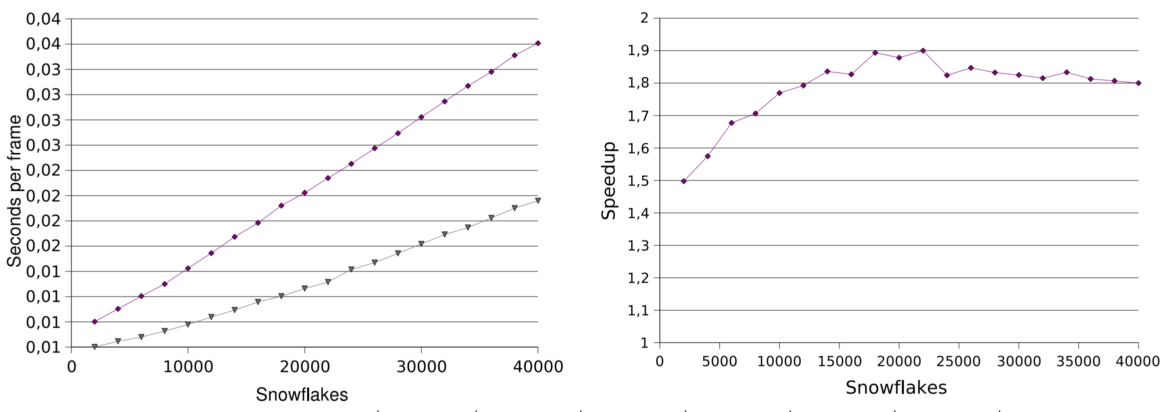
Wind field size:	20x20x5	32x32x8	44x44x11	56x56x14	68x68x17	80x80x20
Frame rate, 1 thread	105.6	78.6	50,4	31.7	19.8	13.1
Frame rate, 2 threads	173.4	143.0	91.8	56.8	33.7	23.0
Speedup	1.64	1.82	1.82	1.79	1.70	1.76

Figure 6.6: Second per frame and speedup as function of wind field size, when the wind field size is varied from 20x20x5 to 80x80x20 voxels. [Dual] is used for measurements.

Figure 6.6 shows how the speedup is varying with wind field resolution, where the resolution is increased from 20x20x5 up to 80x80x20. The x-axis is linear in the number of voxels, however, note that the number of voxels is $O(n^3)$, where n is the number of voxels in each dimension. The y-axis measures *seconds per frame* instead of frame rate, to give a better impression of how the resource consumption varies. It is immediately clear that the computation time needed for the wind field is proportional to the number of voxels. However, there is some variation in the speedup curve. Three reasons behind this can be identified. First, at small scenes, the benefit of parallelization is small compared to the parallel overhead and inherently serial parts of the program. Second, the point at which the wind field data structure becomes too big fit into cache is reached at a later point in the parallel version than the sequential. Third, as the size of each task to be executed in the task-parallel version increases, and implies that the tasks may be scheduled among the threads in different ways. This may lead to either more or less waiting.

6.5.2 Number of Snowflakes

Figure 6.7 shows how the performance is affected by different numbers of snowflakes. Obviously, the computation time is approximately proportional to the number of snowflakes. However, the speedup seems to decrease a little bit beyond 20.000 snowflakes. By regarding Figure 5.4, it can be observed that not all snowflake tasks can be run in parallel with rendering, namely the “advance snowflakes”. At some point, the render task becomes so big that the second thread must wait for the render task to complete, to start on advancing snowflakes.



Snowflakes:	4000	10000	16000	22000	28000	34000	40000
Frame rate, 1 thread	113.8	78.1	57.8	46.0	38.2	32.4	28.5
Frame rate, 2 threads	179.2	138.2	105.6	87.4	70.0	59.4	51.3
Speedup	1.57	1.77	1.83	1.90	1.83	1.83	1.80

Figure 6.7: Seconds per frame and speedup as function of number of snowflakes, on [Dual]

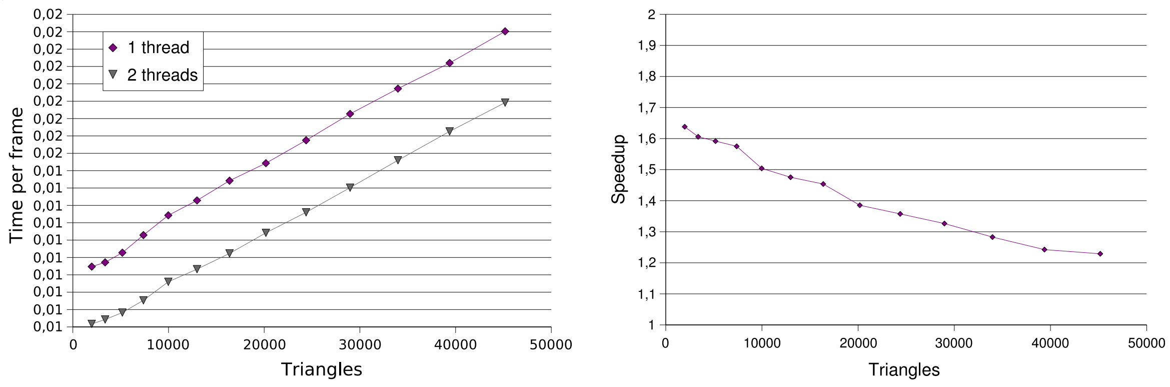
6.5.3 Resolution of Snow cover

Figure 6.8 shows how performance is affected by different snow cover resolutions. This figure is seemingly surprising, as the benefit of parallelization seems to be only a constant amount of time for each frame, independent of the snow cover resolution. The consequence is that the achieved speedup decreases with the snow cover resolution. Instead, one would have expected the speedup to stay approximately constant, and the time per frame for two threads to increase slower than for one thread. The reason behind this, is the dependencies between tasks described in Chapter 5, which requires rendering to complete before snow cover computations for the next frame can be started. In this case, both the wind field and snowflake calculations are kept small, and therefore finish quickly. Then the second thread must wait for the first thread to finish rendering before it can start computing the snow cover for the next frame. Even though the snow cover computations run in parallel, this gain is eaten up by the fact that rendering time is proportional to the snow cover resolution, and thereby increase with the same order of magnitude as the benefit of parallelization.

Figure 6.9 shows the same experiment as Figure 6.8, but uses the default scene small one. The consequence is that with a higher resolution wind field and more snowflakes, there is more work that can be performed in parallel with the rendering. This attenuates the phenomenon that caused the poor speedup to occur with the small scene. Still, there seems to be a performance issues, but they arise much later and with less momentum than in the case of Figure 6.8.

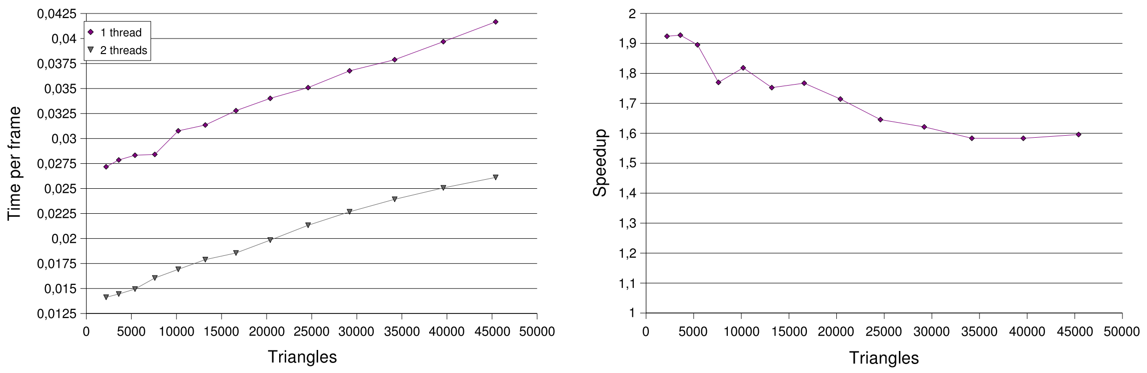
The poor scalability properties related to the snow cover implementation explain to some degree why the speedup decreases for the gigant scene configuration in Figure 6.3, Figure 6.4, and Figure 6.5. However, the problems are less evident in these figures, because the other parameters are scaled up as well. It is likely that rendering has finished before the snow cover computations are scheduled to run, when there is more work related for wind and snowflake computations.

Also, the snow cover computations could benefit from being divided into smaller tasks. As mentioned earlier, when the scene size is scaled up, the distribution of work units to processors becomes a bigger issue. Because the tasks are bigger, the potential load imbalance penalty becomes bigger. The disappointing speedup for high resolution snow cover can be attributed to this fact, because the parallelization of snow cover generation is too coarse grained.



Triangles:	1976	5176	9976	16376	24376	33976	45176
Frame rate, 1 thread	118.0	107.8	87.5	74.5	63.5	53.4	45.4
Frame rate, 2 threads	193.3	171.6	131.6	108.3	86.2	68.5	55.8
Speedup	1.64	1.59	1.50	1.45	1.36	1.28	1.23

Figure 6.8: Seconds per frame and speedup as function of number of triangles used to model the snow cover. [Dual] is used for measurements.



Triangles:	2200	5400	10200	16600	24600	34200	45400
Frame rate, 1 thread	36.8	35.3	32.5	30.5	28.5	26.4	24.0
Frame rate, 2 threads	70.8	66.9	59.1	53.9	46.9	41.8	38.3
Speedup	1.92	1.90	1.82	1.77	1.65	1.58	1.60

Figure 6.9: Seconds per frame and speedup as function of number of triangles used to model the snow cover, with the default scene size as starting point. [Dual] is used for measurements.

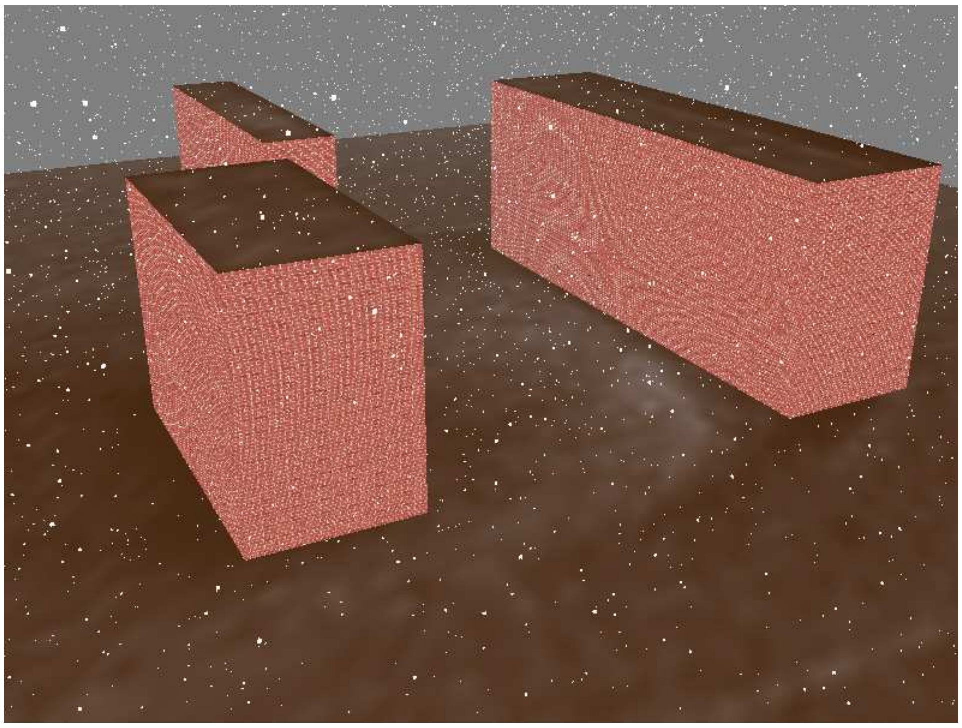


Figure 6.10: Large scene after snowing for 1 minute

6.6 Visual Results

Figure 6.10 through 6.15 shows the large scene configuration after snowing for certain amounts of time. The simulation is run on [DualHyper] with two threads, at a frame rate of 39-40 frames per second. After one minute, it is clear that some snow has reached the ground. At this point, the thin snow cover is formed by blending the color on the ground with the color of the snow. The individual snowflakes that have hit the ground are not visible, because only average snow heights for limited areas are stored. After ten minutes, the contours of how the wind influences where snowflakes land are clearly coming into play. At places where the snow depth changes abruptly, the limited resolution of the snow cover is visible as zigzag patterns. After one hour of snowing, the snow cover has gained a perceivable depth, in particular on top of the buildings. After two and three hours, it is possible to spot variations in snow depth on the ground, because of the uneven distribution of snow caused by the wind. After four hours, the snow cover has become so thick that previously neglectable physical phenomena starts to come into play. The most salient of them is that the wind field should be affected by the snow cover, and that snow should move from unstable places in small avalanches.

Figure 6.16 shows the same scene as Figure 6.15, but from different angles. It demonstrates better how the snow is distributed around the objects.

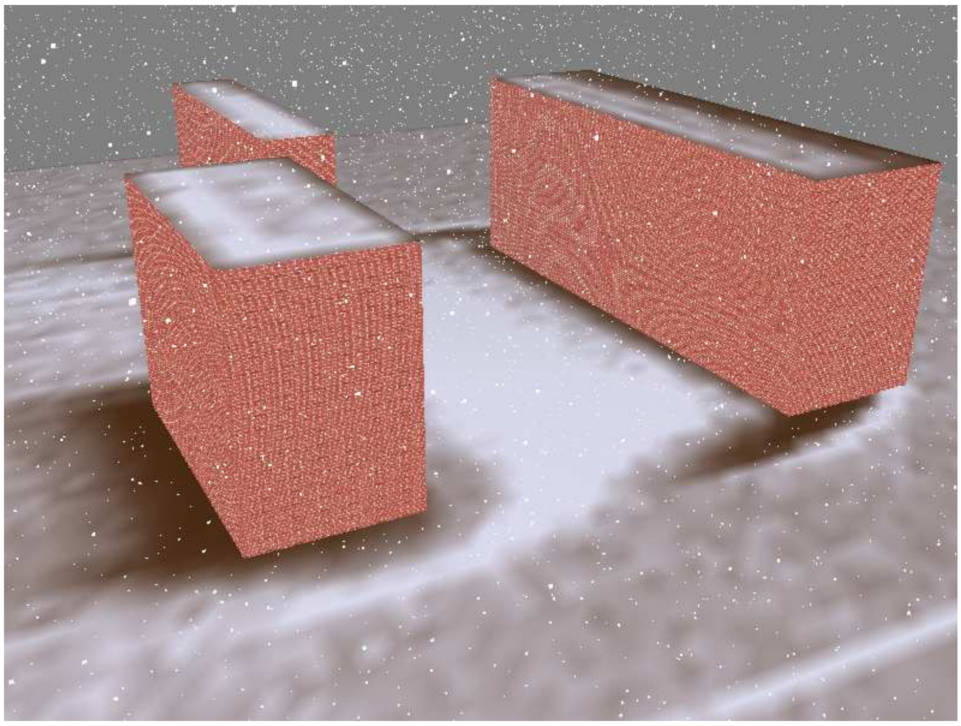


Figure 6.11: Large scene after snowing for 10 minutes

6.6.1 Resolution of Snow cover

Figure 6.17 shows the result of different resolutions of the snow cover representation. It is absolutely clear that a high resolution provides much more detail than the coarse alternative. Also, the weaknesses in regular way the triangulation is created are more pronounced with a low resolution, in particular the zigzag pattern in the lower left of the figure. On the other hand, the fine details achieved by increasing the resolution results in a significant performance penalty, as the frame rate drops from 46.6 to 28.8 frames per second, but the real-time requirement is still attained.

6.6.2 Resolution of Wind Field

Another parameter of visual significance is the wind field resolution. Even though the wind field is not visualized in any way, its realism affects how the snowflakes move around in the scene. If the wind field differs too much from reality, both the snowflake movements and thereby the accumulation will look unrealistic. In particular, objects in the scene are discretized, and the occupied voxels are marked as occupied. Consequently, if the wind field is coarse, it may fail to correctly represent the influence from objects.

Figure 6.18 compares two different wind field resolutions. The left picture clearly shows that this resolution is not fine enough to satisfactory represent the objects. The corners of the objects represented by the wind field do not coincide with the real object corners. Also, what does not appear in the picture, is that the snowflakes penetrate the walls in large quanta.

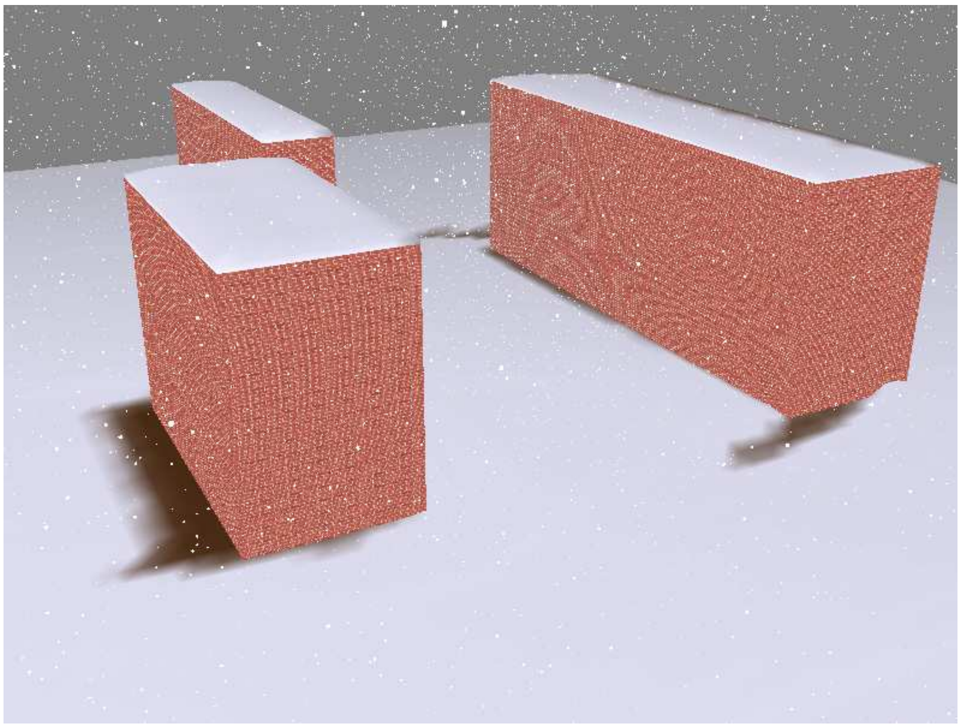


Figure 6.12: Large scene after snowing for 1 hour

The reason is that the walls does not coincide with voxel boundaries, and causing the velocity in parts of the inside of objects to be non-zero. The right picture shows that the snow accumulates nicely, as expected.

6.6.3 Movement of Snowflakes

As pictures can not show how snowflakes fly through the air, video clips of the snow simulation can be found enclosed with this thesis, or at [Sal06].

The videos show that snowflakes follow the air flow around objects in a way that is not too far from reality. They clearly follow the wind around corners of objects, and disappear when the hit objects or the ground. Also, the lift force that is modeled as circular movement is visible as the snowflakes do have some seemingly random motion.

The appearance of the snowflakes themselves obviously has potential for improvement. Particularly in the images, the snowflakes are visible as rectangles with sharp corners.

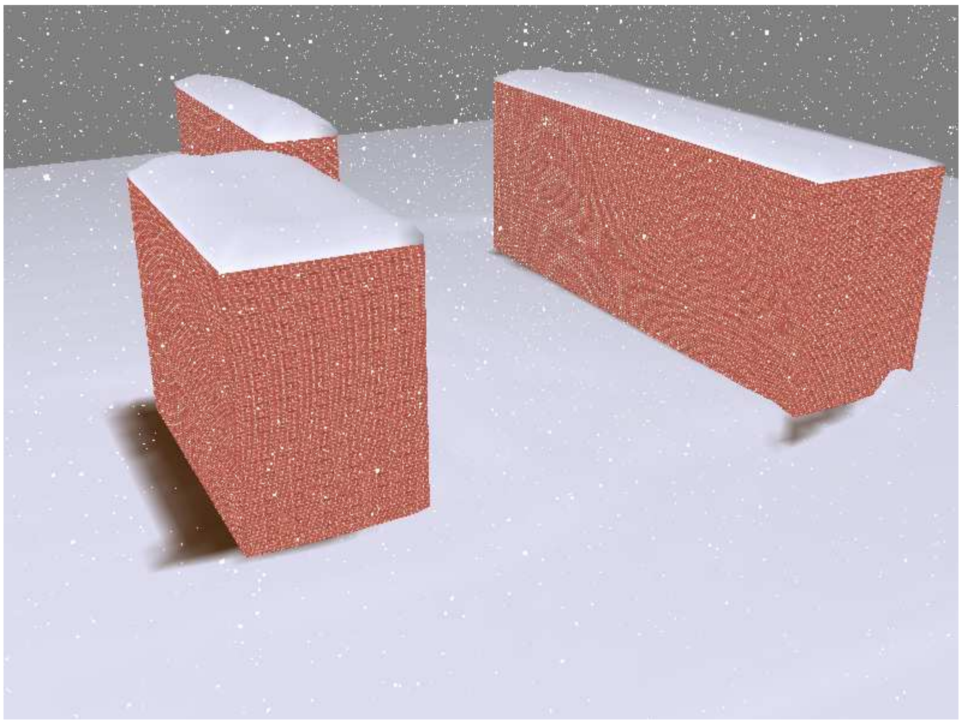


Figure 6.13: Large scene after snowing for 2 hours

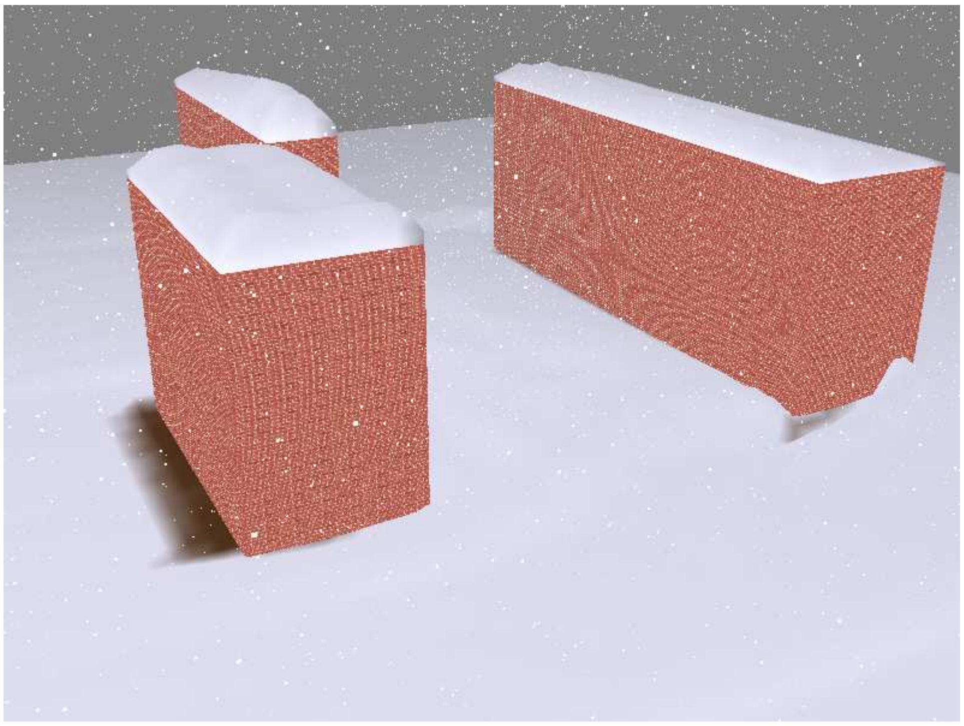


Figure 6.14: Large scene after snowing for 3 hours

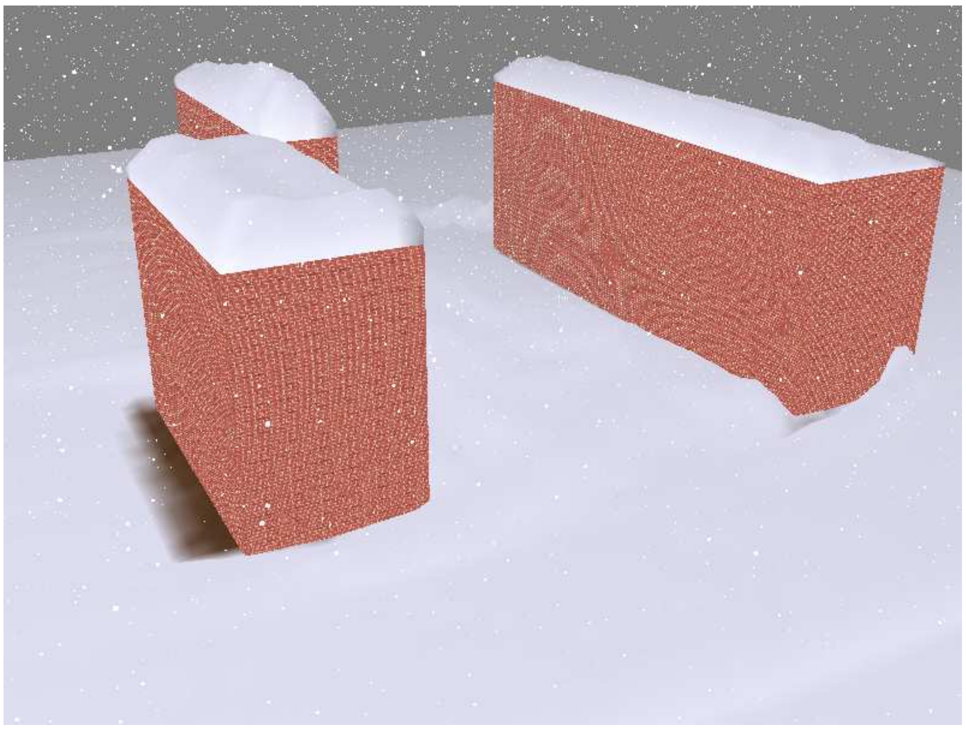


Figure 6.15: Large scene after snowing for 4 hours

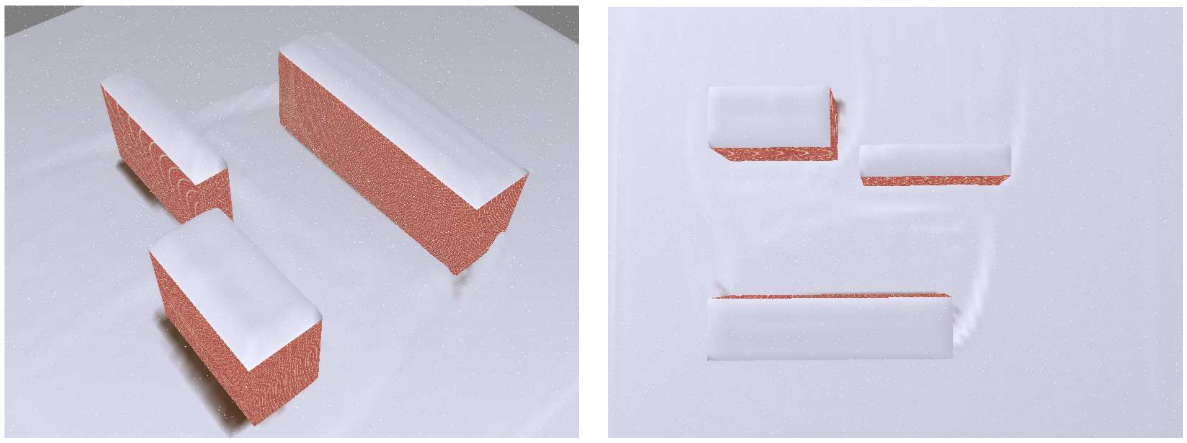
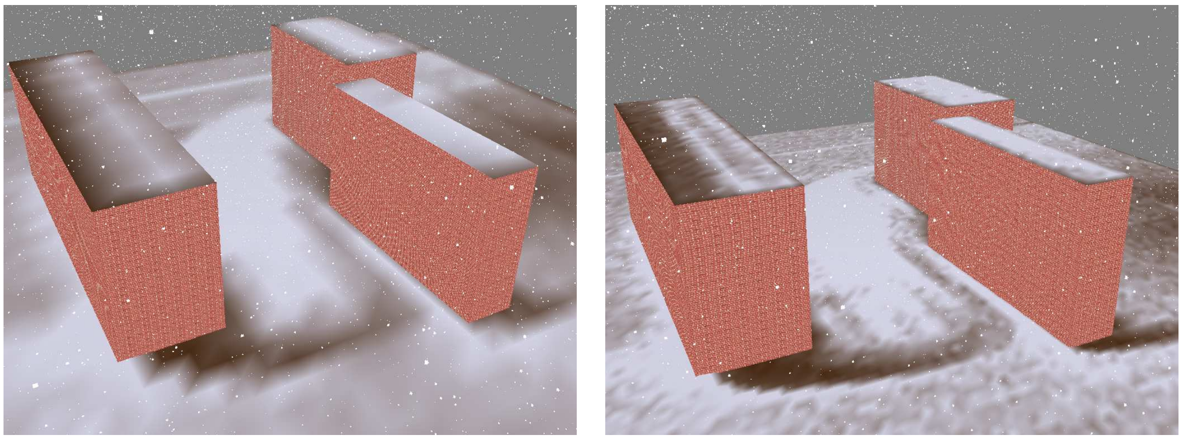
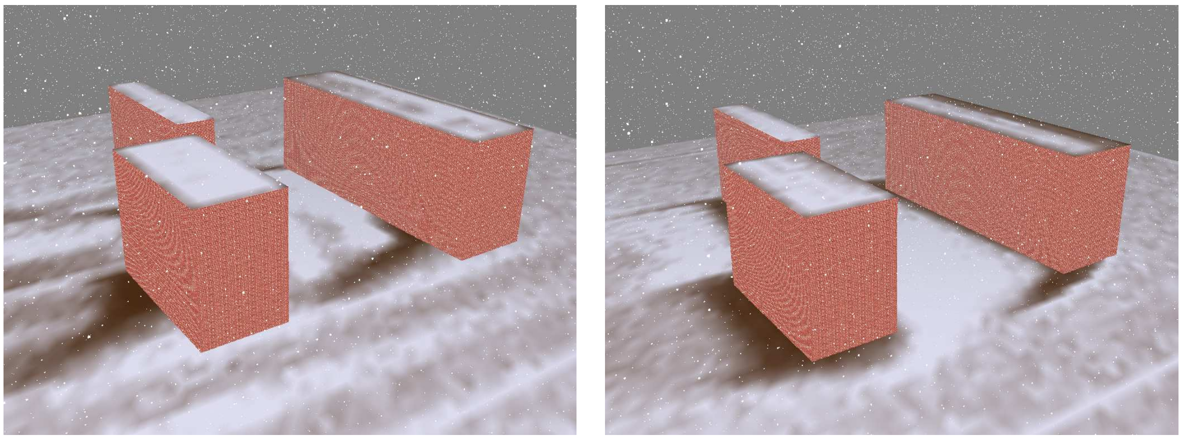


Figure 6.16: Scene seen from above after 4 hours of snowing



(a) 0.5m height value spacing (46,6fps) (b) 0.17m height value spacing (28,2fps)

Figure 6.17: Comparison of different snow cover resolutions.



(a) 20x20x5 voxels (51,1fps) (b) 60x60x15 voxels (26,6fps)

Figure 6.18: Comparison of different wind field resolutions.

Chapter 7

Conclusions and Future Work

This chapter concludes the work by discussing weaknesses in previous chapters, evaluating the achieved results, and pointing out suggestions for future work.

7.1 Feasibility

A natural question to ask, is whether the snow simulation is suitable for use in other applications that the proof-of-concept implementation presented in this thesis. Because the simulation is very CPU intensive, it leaves very little resources left for other tasks. Most likely, the implementation itself is not applicable to anything but serving as a mere demonstration that the snow model is suitable for real-time use. However, the model may be incorporated into other applications, tailed to fit the specific needs. Also, if an application already simulates a wind field to model other phenomena, for instance rain, dust, falling leaves, or a feather as in [WZF$^+$03], the snow model could take advantage of it.

As pointed out in Chapter 4, there are a number of weaknesses in the model. One can argue that by incorporating more elements to improve the model, it will eventually be unsuitable for real-time use. However, the author believes that the most prominent shortcomings can be remedied with relatively simple techniques that will not sacrifice the performance requirements. Most of the visual weaknesses can probably be solved by using more sophisticated rendering techniques that can be handled by graphics hardware.

The most troublesome question is what is going to happen when the model is extended to handle large scenes with complex objects. The height matrix approach to snow accumulation may fail to handle objects with complex geometry, and collision detection between snowflakes and objects may become too expensive. However, with careful attention to and more research into these problem areas, nothing should be impossible.

In Chapter 6, a problem related to parallelization of the snow cover generation was discovered. If scenes with high resolution snow cover, coarse wind field, and sparse snowfall is required, this problem must be addressed to achieve good a speedup. A fairly easy solution is to maintain two vertex arrays, and use them in a “double buffer” fashion. One array can be updated with data for the following frame, while the other one is used for rendering the current frame. This strategy allows snow cover generation to overlap the rendering step in Figure 5.4, and is therefore expected to solve the problem.

7.2 Comparison

Compared to reality (e.g. Figure 2.1), it is clear that the snow model and implementation fails to copy the complex snow effects. However, this has never been the intention, and the implementation clearly captures some important aspects of both falling and accumulating snow. In particular, the snowflake movement is regarded as the most prominent strength of the model.

Both [Fea00b], [FO02], and [AL04] produces very nice results, and the model presented in this thesis definitively falls short of them. However, it is regarded as unfair to compare the model to them, because they are not bound by the computational constraint that follows the real-time requirement. With enough computational resources available, the model should be almost as good as [FO02], because they are both based upon the same underlying wind simulation. However, as the performance is not of importance in [FO02], they can use a much more detailed simulation grid and complex lighting effects. [Fea00b] does not model the influence of wind, while the model in this thesis does.

A comparison to [HAH02] can be justified, as the snow accumulation model is largely based on that work. The snow accumulation looks quite similar, as expected, but the model of this thesis includes the influence of wind, as [HAH02] suggests as a subject to future work.

Concerning the performance, the implementation definitely beats [Fea00b], [FO02] and [AL04], as it runs in real-time. [HAH02] does not mention anything about performance except that is runs in real-time, so it is difficult to make a detailed performance comparison.

As to parallelization, it is hard to make a fair comparison. The smoke simulation in [BKM$^+$00] only presents results for 4 processors, and does not present an overall speedup of the code. [Vik03] achieves a maximum speedup of 1.67 for the smoke simulation on two processors. As far as a comparison can be made, the results of this thesis that show a speedup of 2 on two processors with Hyper-Threading, they are regarded as at least just as good as [BKM$^+$00] and [Vik03],

7.3 Evaluation of Goals

Reciting the the introductory paragraphs, the goals for the thesis are:

> *...to exploit parallel computer systems for real-time snow visualization. To achieve this, the feasibility of already known algorithms for snow visualization will be analyzed with respect to parallel efficiency, real-time requirement, and visual quality.*

Chapter 3 describes different approaches to snow visualization, and Chapter 4 picks out the ones that are considered to be best suited for real-time use and parallelization. This resulted in three components; falling snow, wind simulation and accumulating snow. Chapter 5 describes how these techniques and algorithms can be parallelized, and presents two ways to parallelize them; a data-parallel and a task-parallel approach.

Further,

> *An optimized, parallel implementation that combines snow visualization techniques from previous work will be developed to explore parallel methods and algorithms for snow visualization.*

To put the implementation in concrete terms, some requirements were set up in Section 1.1. These are answered in turn:

1. **The application must run in real-time, with at least 25 frames per second**
   The results in Chapter 6 show that the simulation runs with frame rates greater than 25fps for different scene sizes on today's modern workstations.

2. **The application should be optimized for speed to utilize the power of a single processor well before parallelization is applied.**
   To design a high-performance snow simulation, two principal methods are applied to achieve the necessary execution speed. First, in Chapter 4 model simplifications are made to reduce the amount of computations, for instance reduce the number of iterations in linear solver, and the simplified normal vector computation. Second, code optimizations to avoid common performance pitfalls in today's processor architectures are applied, as described in Chapter 5.

3. **The application must take advantage of at least two processors efficiently**
   Two slightly different ways to parallelize the simulation are presented. One pure data-parallel approach, where all processors perform the same operations on different parts of the data structures where possible, and a task-parallel, where sequential parts of the code are overlapped with other computation. Chapter 6 shows that, depending on the configuration, a speedup of up to 1.99 is achieved on a workstation with two Hyper-Threading enabled processors, running three threads simultaneously.

4. **The motion of snowflakes on their way down to the ground must be visualized**
   The simulation of snowflake motion is implemented by simulating how the forces of a wind field affects snowflakes, as well as imitating the seemingly random motion observed as snowflakes fall. As the videos in [Sal06] show, the approach works pretty well.

5. **When snowflakes hit surfaces in the scene, they should pile up and eventually leave a blanket of snow**
   Snow accumulation is handled by storing the snow depth in height matrices in parts of the scene where snow can accumulate. To visualize the result, triangulations are created based on these height matrices. The images in Chapter 6 show that snow hitting surfaces piles up. Even though it could benefit from more work on rendering, it is proved to be a viable solution.

7.4 Contributions

The contributions of this thesis are twofold:

- A snow model intended for real-time applications. This model basically combines techniques from previous work, and consists of the components falling snow, wind simulation, and accumulating snow. Even though this is not in any way revolutionary, it is regarded as a contribution for two reasons. Previous work has either been narrowed down to one or two of these aspects, or has not been intended for real-time use.
- A proof-of-concept implementation that demonstrates that the snow model is feasible for real-time frame rates on today's modern workstations. This is achieved by parallelizing the algorithms involved in the simulation for shared-memory architectures.

7.5 Conclusion

In this thesis, a novel model for real-time visualization of snow was presented. This model was developed by integrating several components from previous work related to snow visualization, and consists of models for falling snow, wind simulation, and accumulating snow. To achieve real-time frame rates, parallel computer architectures were utilized to speed up computations. First, the algorithms were parallelized by dividing the data structures among processors. To further improve this parallelization scheme, a system was developed that divided the work into smaller tasks that were distributed to processors. This resulted in significantly better speedups, because inherently sequential parts of the code could be overlapped with computations.

Real-time frame rates on dual CPU systems were achieved for reasonable scene sizes. Systems with Hyper-Threading were also considered. On dual CPU systems, speedups up to 1.9 were obtained, and on dual CPU systems with Hyper-Threading, speedups up to 1.99 were achieved. Even though multi-core systems were not considered, the results are believed to generalize to this quickly growing architecture, as well.

7.6 Future Work

There are a number of elements that should be subject to future work. Of the computational aspect, the most important are:

- Multi-core processors have been mentioned several places throughout this thesis as a motivation for parallelization of the implementation. As practical circumstances only allowed exploration of dual CPU systems, it should be investigated how performance is affected by using this up-and-coming architecture.
- Only low-level threads have been used for parallelization on symmetric multiprocessors. An interesting direction could be to use other more high-level parallelization paradigms like OpenMP, or explore the possibilities for using distributed memory architectures as PC clusters with MPI and a high-speed interconnect.
- Some of the literature on snow modeling and simulation is targeted towards an implementation on programmable graphics processing units (GPUs). Speeding up computations by using the GPU is a promising direction that could be explored further.

Related to the model, there is room for improvements:

- When the boundary conditions are fixed, as in this implementation, the wind field converges to a stable state fairly quickly. By precomputing the wind field, as the case is in [FO02], significant amounts of computation time can be saved, as wind field computations are redundant after the wind field has converged. This is a direction that could be pursued. Likewise, precomputation of the snow cover would be of interest, because it takes a long time before a significant amount of snow accumulates.
- A more advanced snow accumulation model could also be developed. A suggestion is to make the necessary modifications for real-time application to Fearing's model [Fea00b]. The stability computations are of particular interest, as are accumulation on arbitrary surfaces.
- More realistic rendering would be highly desirable, both for snowflakes and snow cover. Textured billboards are already suggested for snowflakes, and bump mapping could be a solution for the snow cover.

Appendix A

Modeling the Motion of Snowflakes As Falling Paper

When thin, flat objects such as paper strips, feathers and leaves fall down to the ground, they follow complex, fluttering and tumbling paths. These motions are the result of a wide variety of interactions between object geometry and airflow motion, and may appear as chaotic and random. A natural approach to model this behavior would be to apply equations from fluid dynamics. However, the computing power needed would exceed what would be practical for most purposes.

[BEM98] presents a phenomenological model for falling paper strips, based on among others [TK94]. This model considers thin, flat strips falling through a fluid in two dimensions, and is a crude approximation. It does however, cover the governing motion of such objects, if parameters are carefully tuned.

The center of the paper is located at (x, y) and moves with velocity $(u, v) = (dx/dt, dy/dt)$. It forms an angle θ with the horizontal plane and rotates with an angular velocity of $\omega = d\theta/dt$. The model has four parameters: length l, ratio between the densities of the fluid and the paper ρ, perpendicular friction coefficient $k_\perp$, and parallel friction coefficient $k_\|$. Figure A.1 illustrates how the forces act upon the paper in the model. The following four ordinary differential equations describe the motion of the falling paper in two dimensions:

$$\begin{aligned}
\dot{u} &= -(k_\perp \sin^2\theta + k_\| \cos^2\theta)u + (k_\perp - k_\|)\sin\theta\cos\theta v \mp \pi\rho V^2 \cos(\alpha+\theta)\cos\alpha && \text{(A.1)}\\
\dot{v} &= -(k_\perp \cos^2\theta + k_\| \sin^2\theta)v + (k_\perp - k_\|)\sin\theta\cos\theta u \pm \pi\rho V^2 \cos(\alpha+\theta)\sin\alpha - g && \text{(A.2)}\\
\dot{\omega} &= -k_\perp\omega - (3\pi\rho V^2/l)\cos(\alpha+\theta)\sin(\alpha+\theta) && \text{(A.3)}\\
\dot{\theta} &= \omega && \text{(A.4)}
\end{aligned}$$

where $V = \sqrt{u^2+v^2}$ and $\alpha = \arctan(u/v)$. The upper sign denotes the situation when $(v < 0,\ 0 < \alpha+\theta < \pi)$ or $(v > 0,\ -\pi < \alpha+\theta < 0)$. Please consult the original papers for the derivation and empirical verification of the model. To solve these equations numerically, the paper suggests using a fourth order Runga-Kutta integration scheme.

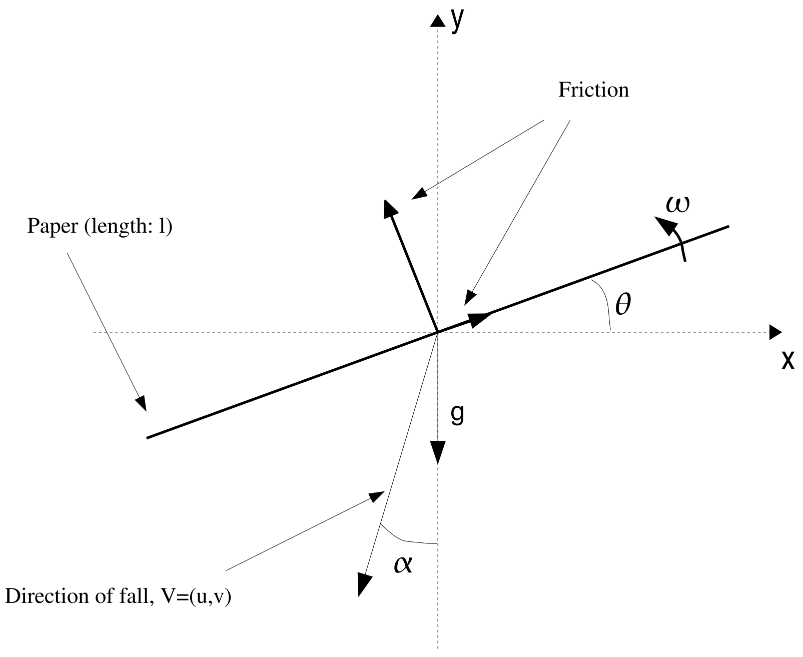


Figure A.1: Illustration of forces in paper strip model.

However, the paper strip model is considered unsuitable for snow simulation for three main reasons. First, it is only defined in two dimensions, and is it not obvious how to generalize it to three dimensions. Second, by implementing the model in 2D and choosing parameters according the properties of snowflakes, it became clear that the model does not reproduce the motions of falling snowflakes well. Basically, either a fast rotating motion, or a periodic motion not to different from a sinusoidal was observed. Last, it requires much more computation time than the method of [AL04].

Bibliography

[AL04] Michael Aagaard and Dennis Lerche. Realistic modelling of falling and accumulating snow. Master's thesis, Lab of Computer Vision and Media Technology, Aalborg University, Denmark, 2004.

[Bau78] Gérard M. Baudet. Asynchronous iterative methods for multiprocessors. *J. ACM*, 25(2):226–244, 1978.

[BBC$^+$94] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition.* SIAM, Philadelphia, PA, 1994.

[BEM98] Andrew Belmonte, Hagai Eisenberg, and Elisha Moses. From flutter to tumble: Inertial drag and froude similarity in falling paper. *Physical Review Letters*, 81(2):345–348, 1998.

[BKM$^+$00] M. Bryborn, R. Klein, T. May, S. Schneider, and A. Weber. A portable, parallel, real-time animation system for turbulent fluids. In M. Guizani and X. Shen, editors, *IASTED International Conference on Parallel and Distributed Computing and Systems '00*, pages 394–400. International Association of Science and Technology for Development, November 2000.

[Fea00a] Paul Fearing. *The Computer Modelling of Fallen Snow.* PhD thesis, UBC, 2000.

[Fea00b] 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.

[Fer04] Randima Fernando. *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*, chapter 38: Fast Fluid Dynamics Simulation on the GPU, pages 637–665. Addison-Wesley Professional, 2004.

[FM97] Nick Foster and Dimitris Metaxas. Modeling the motion of a hot, turbulent gas. *Computer Graphics*, 31(Annual Conference Series):181–188, 1997.

[FO02] Bryan E. Feldman and James F. O'Brien. Modeling the accumulation of wind-driven snow, 2002. Technical Sketch, SIGGRAPH 2002, San Antonio, TX.

[FSJ01] Ronald Fedkiw, Jos Stam, and Henrik Wann Jensen. Visual simulation of smoke. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer* *graphics and interactive techniques*, pages 15–22, New York, NY, USA, 2001. ACM Press.

[Ger02] Richard Gerber. *The Software Optimization Cookbook*. Intel Press, 2002.

[HAH02] Håkan Haglund, Mattias Andersson, and Anders Hast. Snow accumulation in real-time. In *SIGRAD2002, The Annual SIGRAD Conference. Special Theme – Special Effects and Rendering, November 28–29, 2002, Norrköping, Sweden*, 2002.

[HB04] Donald Hearn and M. Pauline Baker. *Computer Graphics with OpenGL*. Prentice-Hall, 2004.

[Int06] Intel. Hyper-threading technology, 2006. `http://www.intel.com/technology/hyperthread/`.

[KL03] Theodore Kim and Ming C. Lin. Visual simulation of ice crystal growth. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 86–97. Eurographics Association, 2003.

[Lib99] Kenneth G. Libbrecht. Snowcrystals.com, 1999. `http://snowcrystals.com`, accessed 18.04.2005.

[LLW04] Youquan Liu, Xuehui Liu, and Enhua Wu. Real-time 3d fluid simulation on gpu with complex obstacles. In *PG '04: Proceedings of the Computer Graphics and Applications, 12th Pacific Conference on (PG'04)*, pages 247–256. IEEE Computer Society, 2004.

[LZK$^+$04] Michael S. Langer, Linqiao Zhang, Allison W. Klein, Aditya Bhatia, Javeen Pereira, and Dipinder Rekhi. A spectral-particle hybrid method for rendering falling snow. In Alexander Keller and Henrik Wann Jensen, editors, *Rendering Techniques*, pages 217–226. Eurographics Association, 2004.

[MMAL05] Thomas B. Moeslund, T. B. Madsen, Michael Aagaard, and Dennis Lerche. Modeling falling and accumulating snow. *Second International Conference on Vision, Video and GraphicGraphics*, 2005.

[NIDN97] Tomoyuki Nishita, H. Iwasaki, Yoshinori Dobashi, and Eihachiro Nakamae. A modeling and rendering method for snow by using metaballs. *Comput. Graph. Forum*, 16(3):357–364, 1997.

[Opd04] Michael Opdenacker. What's new in linux 2.6?, 2004. `http://free-electrons.com/articles/linux26`.

[OS04] Per Ohlsson and Stefan Seipel. Real-time rendering of accumulated snow. In *SIGRAD 2004. The Annual SIGRAD Conference. Special Theme – Environmental Visualization*, pages 25–32, 2004.

[Pac97] Peter S. Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann Publishers, Inc., 1997.

[Ree83] William T. Reeves. Particle systems–a technique for modeling a class of fuzzy objects. *ACM Trans. Graph.*, 2(2):91–108, 1983.

[Sal06] Ingar Saltvik. Videos of snow visualization, 2006. `http://www.idi.ntnu.no/~saltvik/snow`.

[She94] Jonathan R Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, Pittsburgh, PA, USA, 1994.

[Sim90] Karl Sims. Particle animation and rendering using data parallel computation. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 405–413, New York, NY, USA, 1990. ACM Press.

[Sta99] 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.

[Sta00] Jos Stam. Interacting with smoke and fire in real time. *Commun. ACM*, 43(7):76–83, 2000.

[Sta01] Jos Stam. A simple fluid solver based on the fft. *J. Graph. Tools*, 6(2):43–52, 2001.

[Sta03] J. Stam. Real-time fluid dynamics for games. 2003.

[Suc01] Sauro Succi. *The Lattice Boltzmann Equation for Fluid Dynamics and Beyond.* Oxford University Press, 2001.

[TK94] Yoshihiro Tanabe and Kunihiko Kaneko. Behavior of a falling paper. *Physical Review Letters*, 73(10):1372–1377, 1994. Including comments to this article.

[VEH03] Torbjørn Vik, Anne C. Elster, and Torbjørn Hallgren. Real-time simulation of smoke through parallelizations. In *ParCo 2003, Dresden, Germany*, 2003.

[Vik03] Torbjørn Vik. Real-time visual simulation of smoke. Master's thesis, IDI, NTNU, 2003.

[WLMK04] Xiaoming Wei, Wei Li, Klaus Mueller, and Arie E. Kaufman. The lattice-boltzmann method for simulating gaseous phenomena. *IEEE Transactions on Visualization and Computer Graphics*, 10(2):164–176, 2004.

[WZF^{+}03] Xiaoming Wei, Ye Zhao, Zhe Fan, Wei Li, Suzanne Yoakum-Stover, and Arie Kaufman. Blowing in the wind. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 75–85. Eurographics Association, 2003.