Examensarbete
LITH-ITN-MT-EX--06/036--SE

Animating Wind-Driven Snow Buildup Using an Implicit Approach

Tommy Hinks

2006-06-13

Linköpings universitet
TEKNISKA HÖGSKOLAN

Department of Science and Technology
Linköpings Universitet
SE-601 74 Norrköping, Sweden

Institutionen för teknik och naturvetenskap
Linköpings Universitet
601 74 Norrköping

LITH-ITN-MT-EX--06/036--SE

Animating Wind-Driven Snow Buildup Using an Implicit Approach

Examensarbete utfört i medieteknik
vid Linköpings Tekniska Högskola, Campus Norrköping

Tommy Hinks

Handledare Ken Museth
Examinator Ken Museth

Norrköping 2006-06-13

Avdelning, Institution
Division, Department

Institutionen för teknik och naturvetenskap

Department of Science and Technology

Datum
Date

2006-06-13

Språk
Language

☐ Svenska/Swedish
☒ Engelska/English

☐ _______________

Rapporttyp
Report category

Examensarbete
☐ B-uppsats
☐ C-uppsats
☒ D-uppsats

☐ _______________

ISBN

ISRN **LITH-ITN-MT-EX--06/036--SE**

Serietitel och serienummer
Title of series, numbering

ISSN

URL för elektronisk version

Titel
Title

Animating Wind-Driven Snow Buildup Using an Implicit Approach

Författare
Author

Tommy Hinks

Sammanfattning
Abstract

We present a method for stable buildup of snow on surfaces of arbitrary topology and geometric complexity. This is achieved by tracing quantities of snow, so-called snow packages, through a dynamic wind field. Dual compact level sets are used to represent geometry as well as accumulated snow. The level sets have also proven to be well suited for the internal boundaries for our Navier-Stokes solver, which produces a wind field that changes according to snow buildup. Our method is different from previous work in that all the addition of snow is done by local operations, avoiding computationally expensive global refinement procedures. The main contribution of this work is a dual level set method for particle interaction with level sets.

Nyckelord
Keyword

snow, level sets, navier-stokes, wind

LINKÖPING UNIVERSITY
ELECTRONIC PRESS

Upphovsrätt

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

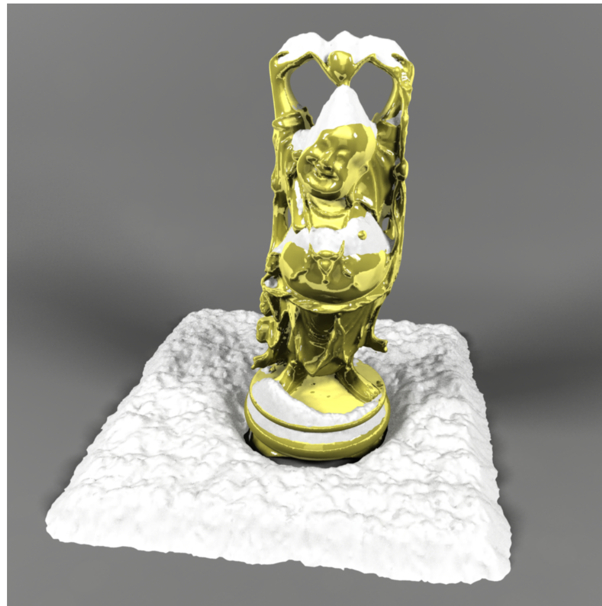
The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Tommy Hinks

Animating Wind-Driven Snow Buildup Using an Implicit Approach



Master Thesis in Media Technology

Supervisor:
Prof. Ken Museth

Author:
Tommy Hinks

Department of Science and Technology
Linköping University, Norrköping, Sweden

Abstract

We present a method for stable buildup of snow on surfaces of arbitrary topology and geometric complexity. This is achieved by tracing quantities of snow, so-called *snow packages*, through a dynamic wind field. Dual compact level sets are used to represent geometry as well as accumulated snow. The level sets have also proven to be well suited for the internal boundaries for our Navier-Stokes solver, which produces a wind field that changes according to snow buildup. Our method is different from previous work in that all the addition of snow is done by local operations, avoiding computationally expensive global refinement procedures. The main contribution of this work is a dual level set method for particle interaction with level sets.

Acknowledgements

I would like to acknowledge my co-members of the Graphics Group along with my supervisor Prof. Ken Museth for invaluable support and discussions throughout this thesis. Extra credit goes to Michael Bang Nielsen for letting me use his implementation of the DT-grid and Andreas Söderström for all the help with the fluid code.

Thank you Helene for love and patience during these months.

Contents

Abstract **i**

Acknowledgements **ii**

Contents **iii**

List of Figures **vi**

List of Tables **ix**

Acronyms & Abbreviations **x**

Notation **xi**

1 Introduction **1**

1.1 Problem Statement 1

1.2 Related Work 1

1.2.1 Snow 2

1.2.2 Wind 3

1.2.3 Level sets 3

1.3 Our Proposed Method 4

1.4 Structure of this thesis 5

2 Theory **6**

2.1 Level Sets 6

2.1.1 Euclidian Distance Fields 9

2.1.2 Dynamic Level Sets 10

2.1.3 Discrete Representation of Level Sets 12

2.1.4 Dynamic Tubular Grids 14
2.1.5 Level Set CSG Operations 15
2.2 Computational Fluid Dynamics 16
2.2.1 The Navier-Stokes equations 16
2.2.2 Stable Fluids 17

3 Implementation **23**
3.1 Dual Level Set Representation 25
3.2 The Wind Field 25
3.2.1 Grid Cell Classification 26
3.2.2 Update Cycle 28
3.3 Snow Particles 31
3.3.1 Snow Packages 31
3.3.2 Slide Packages 32
3.3.3 Snowflakes 32
3.4 Advecting Snow Particles in the Wind Field 35
3.4.1 Snow Packages and Snowflakes 35
3.4.2 Sliding 38
3.4.3 Collision Detection 39
3.5 Accumulating Snow 40
3.5.1 Valid Collisions 40
3.5.2 Stable Buildup 41
3.5.3 Updating the Snow Level Set 46

4 Results **49**
4.1 Varying Temperature 49
4.2 Wind-driven Accumulation 50
4.3 High-Resolution Boundaries 58
4.4 Adding Snowflakes 60

5 Discussion / Future Work **64**

6 Conclusion **68**

Bibliography **69**

A Snow Particle Classes **72**

List of Figures

2.1 Implicit circle with $r = \eta = 1$. 7

2.2 Gradients point in the direction of increasing ϕ and are perpendicular to the iso-surface. 8

2.3 Points inside the interface have negative distance and points on the outside have positive distances. 9

2.4 Points that are equidistant to more than one point on the interface have ill-defined gradients. 10

2.5 The red circles are the results of propagating the blue circles for two different uniform speed functions. a) shows the result of $F = -1$ and b) shows the result of $F = 1$. 11

2.6 Euclidian distance field sampled on a uniform grid. Numbers represent the two dimensional Euclidian distance field at the center of each square. The red circle shows the interface. The numbers in the image are approximations. 12

2.7 a) Voxels are only stored in a narrow band around the interface (red). Only a small part of the voxels are shown here for simplicity. b) The relationships between the different bands. 15

2.8 From the left: $A \bigcap B$ (Intersection), $A \bigcup B$ (Union), $A - B$ (Difference). . 16

2.9 The velocity at position $\vec{x}(t)$ at time t is set to the velocity $\vec{v}(\vec{x}(t-\Delta t), t-\Delta t)$ at a position $\vec{x}(t-\Delta t)$. 19

2.10 The Neumann boundary condition allows flow exchange between fluid cells only. 21

3.1 An overview of the data flow used in our method. 24

3.2 Stanford bunny discretized in the fluid solver domain. Solid grid cells rendered as blue spheres, non-solid cells are not shown but make up the rest of the fluid domain (the semi-transparent box) in this case. 26

3.3 Grid cell classification showing that it is crucial that the boundary cells are positioned inside the β-band as the normals are used to make sure the wind field does not flow into solids. 27

3.4 Cross-section of the fluid solver domain showing grid cell classification. Red grid cells are fluid cells, blue grid cells are inside a solid object (in this case a sphere), green grid cells have one or more solid neighbor cells, yellow cells are cells were there is wind flowing into the domain or cells with such neighbors, turquoise cells are outflow cells or cells with such neighbors and finally, black cells are solid cells on the edge on the solver domain. 28

3.5 Top, two dimensional flow around a circle. The grey lines represent the velocities on the fluid solver grid. Notice the vorticity behind the circle. Bottom, visualization of three dimensional flow around a Stanford bunny using streamlines. The semi-transparent box surrounding the bunny is the fluid solver domain. Wind direction is right-to-left in both images. 30

3.6 Class hierarchy for different types of snow particles. 31

3.7 Triangles distributed per layer with constraints to prevent free-floating triangles. Image courtesy of [AL04]. 33

3.8 Close-up of two out of 256 snowflake template meshes used for -3°C 34

3.9 Forces acting on a snowflake or snow package as it moves in the wind field. 37

3.10 Forces involved when moving slide packages. 38

3.11 Previous work defines the stability angle as β. With our method it is more convenient to define this angle as θ. 41

3.12 Angles and directions involved in guaranteeing stability when propagating the snow level set. For simplicity $C_{smooth} = 0$ is assumed here. 43

3.13 Left: Parts of the interface (marked with yellow) within the collision domain (red circle) are not valid collision points. Right: The collision domain has been moved so that the collision plane is lying totally beneath or on the surface. 45

3.14 Snow packages rendered as white spheres being advected in the wind field. Green spheres show slide packages living on the surface of the accumulated snow. Collisions already stored in the collision buffer shown as red spheres. 46

3.15 The interface is propagated so that it conforms to the shape of f_c, which is guaranteed to be adhere to the stability criteria setup earlier. 48

4.1 Snow buildup on a sphere. The settings were identical apart from temperature, which was -2°C for the left sphere and -8°C for the right sphere. . . 50

4.2 Front view of house scene with our method for two different wind speeds with identical direction. Left: wind speed 5 $\frac{m}{s}$. Right: wind speed 0.5 $\frac{m}{s}$. . 51

4.3 Side view of house scene for two different wind speeds, with identical direction. Left: wind speed 5 $\frac{m}{s}$. Right: wind speed 0.5 $\frac{m}{s}$. Notice the accumulation of snow in-between houses and behind the right-most house. The buildup behind the house is caused by the house sheltering the area from wind, making snow packages fall vertically to the ground instead of being carried by the wind out of the scene for this region. 52

4.4 Front view of house scene for two different wind speeds, with identical direction. Left: wind speed 5 $\frac{m}{s}$. Right: wind speed 0.5 $\frac{m}{s}$. A better overview of the accumulated snow. Notice that for small wind speeds wind-driven accumulation patterns are practically unnoticeable. 52

4.5 Feldman scene as presented in [FO02]. Wind-effects are clearly visible and the snow surface is smooth and the results look good. However, the method in [FO02] stores built-up snow in a height field so it can only handle very simple geometry. 53

4.6 Feldman scene as presented in [MMA+05]. Snow-cover is not very smooth. Sharp edges, uncharacteristic of snow, reveal the underlying triangulation. 54

4.7 Our version of the setup introduced in [FO02]. Our houses are represented as simple boxes as this will not have a noticeable impact on the wind flow around the houses at ground level. Our three compared scenes differ more than one would expect but this is because we have not been able to find the exact configurations of the two other methods we compare with. In our case the wind seems to have been stronger than in previous work and areas around the house furthest away are not snow-covered due to this. However, this shows realistic behaviour of our transport model and should not be considered an artifact. 55

4.8 Wind field in original scene. 56

4.9 Wind field after snow accumulation. 57

4.10 Snow on a buddha statue with effective resolution of 256^3. 59

4.11 Frame 50 from animation consisting of 300 frames. 61

4.12 Frame 100 from animation consisting of 300 frames. 61

4.13 Frame 150 from animation consisting of 300 frames. 62

4.14 Frame 200 from animation consisting of 300 frames. 62

4.15 Frame 250 from animation consisting of 300 frames. 63

4.16 Frame 300 from animation consisting of 300 frames. 63

List of Tables

2.1 CSG operations for distance volumes (discrete level sets) V_A and V_B assuming positive outside and negative inside values. 15

Acronyms & Abbreviations

AOR	Angle of Repose
CFD	Computational Fluid Dynamics
CSG	Constructive Solid Geometry
DT-Grid	Dynamic Tubular Grid
ENO	Essentially Non-Oscillatory
GUI	Graphical User Interface
NS equations	Navier-Stokes equations
SDK	Software Development Kit
TVD	Total Variation Diminishing
WENO	Weighted ENO

Notation

This section contains examples of the most common notations used in this thesis. Even though some of the notations are identical it will be clear from the context what is being referred to.

$\vec{x}$	Vector
$\hat{x}$	Normalized vector
$\bullet$	Scalar product, sometimes referred to as the *dot product*
∇	Vector of spatial partial derivatives, $(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial x})$, in three dimensions
∇^2	$\nabla \bullet \nabla$
$\nabla(\vec{u})$	$(\frac{\partial \vec{u}}{\partial x}, \frac{\partial \vec{u}}{\partial y}, \frac{\partial \vec{u}}{\partial z})$, in three dimensions
$\nabla \bullet \vec{u}$	$\frac{\partial \vec{u_x}}{\partial x} + \frac{\partial \vec{u_y}}{\partial y} + \frac{\partial \vec{u_z}}{\partial z}$, in three dimensions
$\mathbf{M}$	Matrix
$\mathbf{P}$	Operator
$f()$	Function
$[m/s]$	Unit, meter per second
$\forall$	"for all"
$:$	"such that"
$\in$	"in"
$\notin$	"not in"
$\equiv$	"defined as"
$\subset$	Subset, e.g. $D \subset \Re$, D is a subset of all real numbers
$\Re^n$	Euclidian n-space, the space of all n-tuples of real numbers, $(x_1, x_2, ..., x_n)$
$A \bigcap B$	CSG Intersection between distance volume A and B
$A \bigcup B$	CSG Union between distance volume A and B
$A - B$	CSG Difference between distance volume A and B

Chapter 1

Introduction

Snow is common during the winter season in many parts of the world. Heavy snowfall upon a scene will dramatically change its appearance in terms of geometry as well as illumination. The powdery nature of snow allows it to completely cover small objects and accumulate on sharp features giving the scene a smoother characteristic. Snow is arguably one of nature's most complex and fascinating substances, and finding accurate models for this phenomena has proven to be a great challenge to the scientific community for many years.

We strongly believe that the computer graphics industry would benefit from a robust method for snow distribution that is independent of the scene complexity and instead scales with resolution. Such techniques could be used to generate snow covered scenes for movies as well as interactive applications.

1.1 Problem Statement

The major issues to resolve in order to generate realistic snow scenes are: (1) Assuring that snow accumulates in the right locations by taking wind into account. (2) Guaranteeing that the snow accumulates in such a way that is physically plausible in the sense of granular stability. (3) Visualization of falling snowflakes.

We do not attempt to make our simulations interactive as this approach would require simplifications of our models that would heavily interfere with our goal for realism. The general opinion within the film industry is that a simulation should be sufficiently fast so that it may be run over-night (approx. 10–12 hours) so this is the time frame we aim for.

1.2 Related Work

Here we present previous work in the fields related to our work. Previous work has been divided into separate categories since these components are rather independent of each other.

1.2.1 Snow

Earlier work in the field of snow buildup can be roughly divided into physically and non-physically based approaches. Our method belongs to the first category. The second category mainly consists of attempts to generate snow covered scenes by using image analysis [PTS99, OS04], not dealing with volumetric buildup. Physically based snow buildup deals with accumulating volumes of snow on solid objects. Recent previous work in the physically based category uses polygon meshes [Fea00, MMA+05] or height fields [SOH98, FO02] to represent accumulated snow. Even though one could argue that fallen snow is by nature representable as a height field this is not true if the underlying geometry overlaps itself along the gravitational axis, in which case snow could possibly accumulate in several intervals along a height field column. First consider a horizontal plane with finite dimensions. The accumulated snow on this plane converges toward a pyramid-shape, perfectly representable as a height field. Now add a sphere hovering above this plane. Snow may accumulate both on the sphere and in the region on the plane directly beneath the sphere. Polygon representations avoid this problem but introduce the need for elaborate subdivision schemes in order to increase the level of detail in areas of complex occlusion and they also require special treatment for overlapping geometry. Particle tracing is used to estimate the occlusion for polygons in the scene and snow the "rises" of these polygons, which ignores the fact that snow buildup has a separate time-dependency for different areas of the scene. Furthermore, the "risen" snow may not be stable and global refinement steps must be used to guarantee stability. They also introduce the need for height sorting when transporting unstable snow to lower areas in the scene. An implicit approach that uses metaballs was proposed in [NID+97], however, the positioning of the metaballs is done manually and the snow surface lacks fine detail.

Early attempts at generating realistic snowfall were made by Reeves [Ree83] and Sims [Sim90]. Back in those days computer graphics was far from what it is now, mainly due to the difference in available computational power, why these approaches are over-simplified. More recent approaches include [LZK+04, MMA+05]. Of these two [MMA+05] seems the most promising since it is far more simple to implement and the results look much better. Langer et. al. [LZK+04] propose a method that uses spectral analysis to reduce the number of snow particles needed to visualize heavy snowfall. Although this sounds promising, results are not convincing at this stage. The technique proposed by Moeslund et. al. [MMA+05] shows much more convincing results. Their idea is to represent snowflakes as meshes, giving them a unique shape. When these meshes are rotated the silhouette of the snowflake changes giving a very dynamic impression.

1.2.2 Wind

Fearing [Fea00] introduces the rather non-intuitive idea to shoot snow particles from the ground upwards to generate occlusion patterns. This idea works well for constant wind fields, as these are not time-dependent. However, for a time-dependent wind field this approach does not make much sense, since at time $t = 0$ the snow particles should be at their initial locations and not their points of impact on the solids. Additionally it can be said that a global wind parameter is not suitable when visualizing falling snowflakes since they would effectively travel through solid objects and not exhibit the turbulent effects associated with falling snow. Moeslund et. al. [MMA+05] improve on this method by adding a Navier-Stokes solver to their snow accumulation method, generating more realistic accumulation patterns and also present an algorithm for snowflake generation and advection. To the best of our knowledge they do not use the Dirichlet boundary condition for their wind field, the reason possibly being that their mesh representation makes it difficult to compute normals outside the mesh. The Dirichlet boundary condition makes the wind field behave better around internal boundaries in the wind domain. For simplicity they have only used boxes and triangles. In contrast, our approach is general and handles any kind of geometry. Feldman and O'Brien [FO02] use a similar approach to calculate a dynamic wind field but do not visualize snowflakes.

The work by Stam [Sta99] provides a solid basis for calculating wind fields. By using a combination of semi-Lagrangian advection schemes and implicit solvers this method guarantees stability for arbitrarily large time-steps. This method was later improved upon by Fedkiw et. al. [FSJ01] by adding better interpolation schemes and vorticity confinement methods. Since we will not be advecting densities or free surfaces the high-quality vorticity produced by Fedkiw et. al. is not required in our case, as we are more interested in the general properties of the wind field, rather than the fine features.

1.2.3 Level sets

Level set methods [OS88] have been very successful in interface tracking applications such as computer graphics, computer vision and computational physics. The common factor in these applications is that they represent the interface (i.e. surface) as a time-dependent Euclidian signed distance function discretized on a computational grid. For computational speed-up the level set equation can be solved in a narrow band around the interface, as this is sufficient to track it [PMO+99]. However, storing the distance values on the full grid together with additional data-structures to keep track of the narrow band results in huge memory footprints. Where the need for large grids arises the memory requirements very soon become unmanageable. To address this, work has been done to store the distance values in octrees [LGF04], increasing the resolution in a narrow band around the interface.

The main drawback of this approach is that the non-uniform discretization makes it non-trivial to use higher-order finite difference schemes, such as ENO[1] [OS91] or WENO[2] [LOC94]. Instead semi-Lagrangian methods [Str99] are often used, with the drawback that these methods are only applicable on *hyperbolic* problems like advection in external velocity fields (the basic problem of computational fluid dynamics). Unfortunately semi-Lagrangian methods require interpolation which is non-trivial on multi-resolution grids since neighboring grid cells may not be at the same level in the octree which gives rise to so-called *T-junctions* [Min03].

The level set method we will be using for our implementation is the *Dynamic Tubular grid* (DT-grid) [NM06] which combines the computational efficiency of narrow band methods with the low memory footprints of hierarchical data structures. Only grid points in a tube (in 3D) surrounding the narrow band are kept stored in memory keeping memory usage at a minimum. This means that memory consumption scales with the area (in 3D) of the interface instead of the bounding box volume, a highly desirable feature. The DT-grid is also unbound in the sense that the narrow band is rebuilt after the level set is propagated, which means that the effective size of the bounding box is completely dynamic.

1.3 Our Proposed Method

We present an implicit approach to modeling and animation of progressive accumulation of wind-driven snow on static surfaces. Our contributions can be summarized as follows: (1) *Dual compact level sets*, [NM06], are used to represent the surfaces of the dynamic snow and the static boundaries. This allows for topologically and geometrically complex modeling of snow and our method is general in the sense that the shapes of the solids do not effect computation times or require special cases to be introduced. Additionally, the signed distance representation of the compact level sets serve as acceleration data structures for voxel classification as well as closest-point and normal computation. (2) Our snow model is based on a steady-state description where each frame represents a snapshot of the physically based buildup. This effectively allows us to animate the progressive snow buildup. (3) The transport model includes sliding in an incompressible stable wind field. In contrast, the most recent previous approaches [Fea00, FO02, MMA+05] employ explicit surface representations and require an iterative global refinement step to finalize a stable distribution of snow.

Our general approach is based on the concept of so-called *snow packages* which we define as Lagrangian tracker particles representing discrete volumes of snow. Typically these snow packages represent substantially larger volume than a single snowflake, but the size can be

[1] Essentially Non-Oscillating
[2] Weighted ENO

varied arbitrarily. We advect these snow packages in a dynamic wind-field produced from a fairly standard Navier-Stokes solver, based on the techniques described in [Sta99], that accurately handles the deforming boundaries of the accumulating snow. We represented all the surfaces using the efficient DT-Grid level set data structure of [NM06] that essentially stores distance values in a narrow band around the zero crossing (i.e. the surface). This results in very low memory footprints and allows for high-resolution surfaces. The snow surface is initialized as the static boundary surface and the snow volume is then defined as the CSG difference.

Two different mechanisms for snow modeling are assumed: *Buildup* from snow package collisions and *sliding* from snow impacting very steep or slippery surfaces. Using the snow packages and level sets introduced above we model the snow transport as follows. First we note that collisions between snow packages and the level sets are efficiently computed using the signed distance transform of the latter. When such collisions are detected we use a physically based stacking test (depending on surface normals, material, package sizes and temperature) to determine how much of the impacting snow can be deposited without becoming unstable. The corresponding snow volume is then "added" to the snow surface by a local level set operation. The remaining (unstable) volume of the snow package is then converted into one or more *slide packages* that essentially represent a mini-avalanche. Slide packages are advected separately along the projection of the gravity vector onto the surface tangent plane. During this advection new stability tests are evaluated to account for buildup by local sliding. Should a slide package get air-born it is simply converted into a snow package and advected in the flow field. For visualization of snowflakes (as opposed to snow packages) we use a slight modification of the movement model described in [MMA+05].

1.4 Structure of this thesis

Chapter 2 will give the reader an overview of the mathematical background for this thesis. In Chapter 3 we describe the details of our proposed method. The results of our method are presented in chapter 4, followed by a discussion in chapter 5. We summarize the results and discussion in chapter 6.

Chapter 2

Theory

This section will give the reader an overview of the mathematical techniques used. The equations presented here are essential to our approach and will be referred to later on together with the numerical techniques we used to solve them. We explain the basics of the level set equation and give an introduction to the incompressible Navier-Stokes equations and a possible means of solving it. The level set section is based on the book on level set methods by Osher and Fedkiw from 2003 [OF03].

2.1 Level Sets

Given a function

$$\phi : D \rightarrow \eta, D \subset \Re^n, \eta \in \Re \tag{2.1}$$

we can define the level set

$$L(\eta) \equiv \{\xi \in D : \phi(\xi) = \eta\} \tag{2.2}$$

or in other words the set of points that solve the equation

$$\phi(\xi) = \eta \tag{2.3}$$

Consider the level set $\phi(x, y) = x^2 + y^2$. This equation implicitly describes all possible origo-centered circles. In order to find the circle's radius, r, we must solve equation (2.3) with $\eta = r^2$. η is also known as the iso-value. Solving this equation will give us the set containing all (x,y)-coordinate pairs with distance r from the origin. Consider the case were $\eta = 1$. Equation (2.3) is then $x^2 + y^2 = 1$. It is rather obvious that there exist such tuples of (x, y) that satisfy this condition, but they are not explicitly given from this equation. The solution is an $(n - 1)$ dimensional subspace to the n dimensional level set, known as the iso-surface, and is directly associated with the η used to solve equation 2.3.

For every tuple (x, y) we can obtain a value for $\phi(x, y)$ as defined above. There are three principal cases to consider, as shown in figure 2.1.

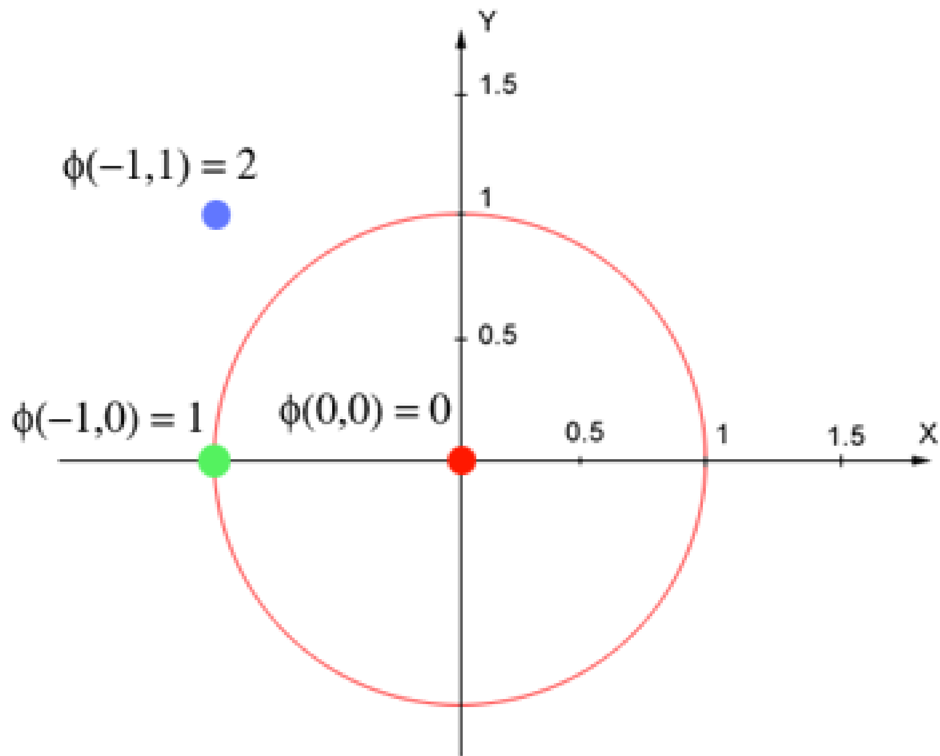


Figure 2.1: Implicit circle with $r = \eta = 1$.

The first case is the blue marking, where $\phi(x, y) > \eta$. This means this point lies outside the circle. The second case is the green marker, which shows a point where $\phi(x, y) = \eta$, hence this coordinate pair satisfies the condition for lying on the circle. The third case is marked in red an shows a point inside the circle, $\phi(x, y) < \eta$. As we shall soon see it is crucial that we can classify any point in space as inside, outside or on the *interface*. Where the interface S is defined as

$$S \equiv \{\vec{x} : \phi(\vec{x}) = \eta\} \tag{2.4}$$

or in other words, all the n-space tuples with the property that if they are used as arguments to the implicit function this will return the iso-value. For the interior (inside) region Ω bounded by S we get

$$\begin{aligned}
\phi(\vec{x}) &< \eta \rightarrow \vec{x} \in \Omega \\
\phi(\vec{x}) &> \eta \rightarrow \vec{x} \notin \Omega \\
\phi(\vec{x}) &= \eta \rightarrow \vec{x} \in \partial\Omega \equiv S
\end{aligned}$$

The gradient of an n dimensional level set is calculated as

$$\nabla\phi \equiv (\frac{\partial\phi}{\partial x_1}, \frac{\partial\phi}{\partial x_2}, ..., \frac{\partial\phi}{\partial x_n}) \tag{2.5}$$

In three dimensions we would write this as

$$\nabla\phi = (\frac{\partial\phi}{\partial x}, \frac{\partial\phi}{\partial y}, \frac{\partial\phi}{\partial z}) \tag{2.6}$$

For a coordinate system with three base vectors $\vec{e}_x$, $\vec{e}_y$ and $\vec{e}_z$ the gradient operator is defined as:

$$\nabla = [\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}] \equiv \vec{e}_x\frac{\partial}{\partial x} + \vec{e}_y\frac{\partial}{\partial y} + \vec{e}_z\frac{\partial}{\partial z} \tag{2.7}$$

Consider the coordinate system with base vectors in the tangent, $\vec{e}_1$, binormal, $\vec{e}_2$, and normal, $\vec{n}$, directions. The gradient is then

$$\vec{e}_1\frac{\partial}{\partial 1} + \vec{e}_2\frac{\partial}{\partial 2} + \vec{n}\frac{\partial}{\partial n} = \vec{e}_1(\vec{e}_1 \bullet \nabla) + \vec{e}_2(\vec{e}_2 \bullet \nabla) + \vec{n}(\vec{n} \bullet \nabla) \tag{2.8}$$

Since the surfaces we are dealing with are isosurfaces $\frac{\partial\phi}{\partial e_1} = \frac{\partial\phi}{\partial e_2} = 0$, leading us to:

$$\nabla\phi = \vec{n}(\vec{n} \bullet \nabla) \parallel \vec{n} \Rightarrow \vec{n} = \pm\frac{\nabla\phi}{|\nabla\phi|} \tag{2.9}$$

The normal, $\hat{n}$, is always perpendicular to the iso-surface and points in the direction of increasing ϕ, as seen in figure 2.2. As we shall see when discussing Euclidian distance fields the direction of the normal is dependent on how we define the distance to the interface.

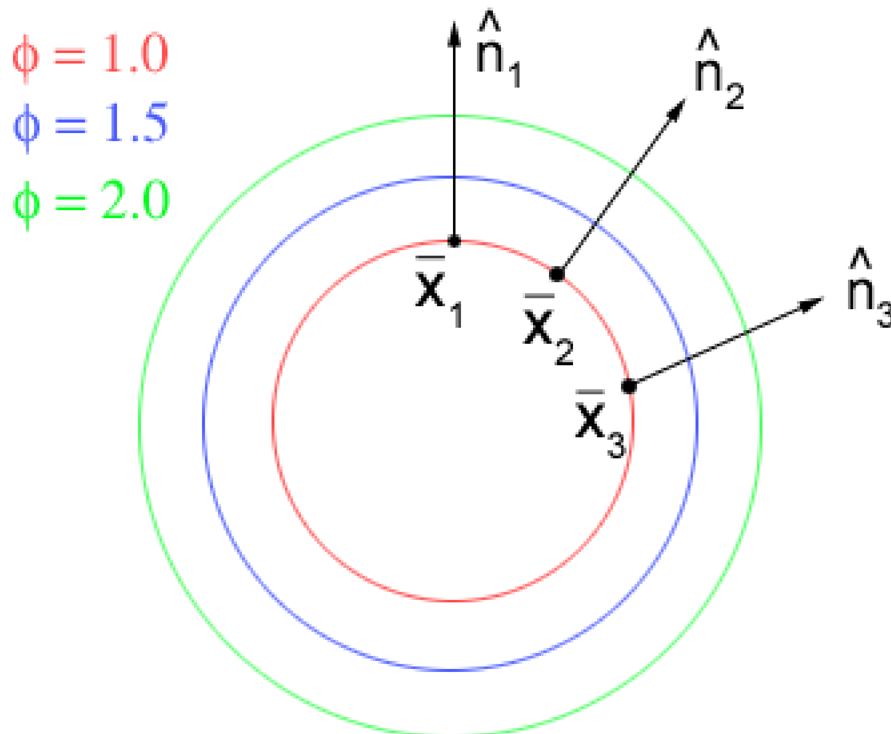


Figure 2.2: Gradients point in the direction of increasing ϕ and are perpendicular to the iso-surface.

2.1.1 Euclidian Distance Fields

A special kind of implicit function often used is the Euclidian distance function, defined as

$$\phi(\vec{x}) = min(|\vec{x} - \vec{x_s}|) \ \forall \ \vec{x_s} \in \partial\Omega \tag{2.10}$$

$$|\nabla\phi| = 1 \tag{2.11}$$

$$\vec{x} \in \Re^{dim} \tag{2.12}$$

These equations state that $\phi(\vec{x})$ returns a scalar value representing the closest Euclidian distance to the interface (i.e. the surface) for every point in space. The level sets we are interested in are in fact signed Euclidian distance functions. We define points inside the interface, $\phi(\vec{x}) < \eta$, to have negative distances to the interface and points outside the interface, $\phi(\vec{x}) > \eta$, to have positive distances to the interface. Points lying on the interface have distance zero. This is illustrated in figure 2.3.

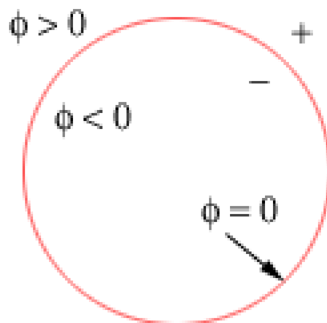


Figure 2.3: Points inside the interface have negative distance and points on the outside have positive distances.

The major weakness of this representation for our purposes is the fact that points that are equidistant to more than one point on the interface have ill-defined gradients. Since the gradient is the direction in which $\phi(\vec{x})$ increases the fastest (equation 2.6), there may be conflicts when this direction is not uniquely defined. Figure 2.4 illustrates this problem for a rectangle. Points lying on the red lines are equidistant to more than one point on the interface, $d_1 = d_2$. In conclusion, care has to be taken when handling equations where $\nabla\phi$ is present.

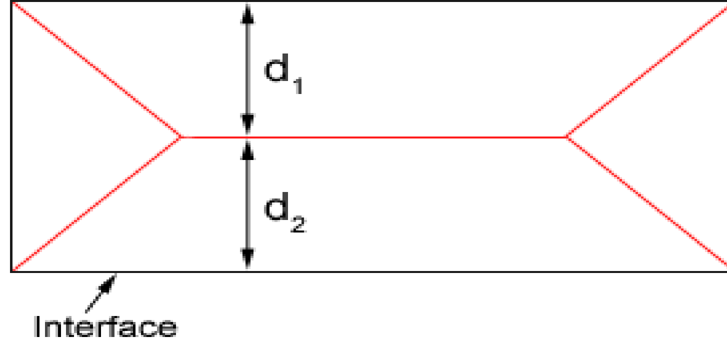


Figure 2.4: Points that are equidistant to more than one point on the interface have ill-defined gradients.

2.1.2 Dynamic Level Sets

The level sets we will de dealing with will not be static, i.e. the interface will move, making ϕ time-dependent. As snow builds up the snow level set will be propagated outwards to represent the accumulated snow. To represent this we add time-dependency to the interface definition given in equation 2.4. We now redefine the interface as

$$S(t) \equiv \{\vec{x}(t) : \phi(\vec{x}(t), t) = \eta\} \tag{2.13}$$

Introducing time-dependency allows us to move the interface over time, making it possible for our interfaces to completely change shape. Initially we setup our interfaces to correspond to the geometry we are interested in. This means we can set them up in such a way that our desired isosurface corresponds to $\eta = 0$. This is convenient, but not a requirement.

If we differentiate equation 2.13 with respect to time we find some interesting properties.

$$\begin{aligned}
\frac{d}{dt}\phi(x(\vec{t}), t) &= \frac{d}{dt}\eta \\
\frac{\partial\phi}{\partial t} + \frac{\partial\phi}{\partial x_1}\frac{\partial x_1}{\partial t} + \frac{\partial\phi}{\partial x_2}\frac{\partial x_2}{\partial t} + \ldots + \frac{\partial\phi}{\partial x_n}\frac{\partial x_n}{\partial t} &= 0 \\
\frac{\partial\phi}{\partial t} &= -\nabla\phi \bullet \frac{d\vec{x}}{dt} \\
\frac{\partial\phi}{\partial t} &= -|\nabla\phi| \cdot \frac{\nabla\phi}{|\nabla\phi|} \bullet \frac{d\vec{x}}{dt} \\
\frac{\partial\phi}{\partial t} &= -|\nabla\phi| \cdot \hat{n} \bullet \vec{v} \qquad (2.14) \\
\frac{\partial\phi}{\partial t} &= -|\nabla\phi| \cdot F(\vec{x}, \hat{n}, \ldots) \qquad (2.15)
\end{aligned}$$

From equation 2.14 we see that if we have values for $\vec{v}$ defined in the domain of ϕ we can move the surface in the normal direction, since the left-hand side of this equation expresses the rate of change of the distance to the interface. This can be more conveniently expressed in a so-called *speed function*, equation 2.15, allowing the possibility to let any number of factors influence the movement of the interface. Moving the interface is often referred to as *propagating* the interface. For example, setting $F(\vec{x}) = -1$ for all $\vec{x}$ will move the interface *inwards*, eroding it, while $F(\vec{x}) = 1$ for all $\vec{x}$ will dilate the interface. This is illustrated in figure 2.5.

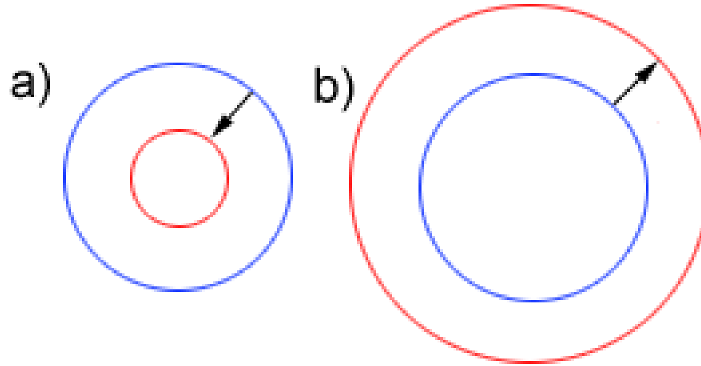


Figure 2.5: The red circles are the results of propagating the blue circles for two different uniform speed functions. a) shows the result of $F = -1$ and b) shows the result of $F = 1$.

Once the interface has been propagated the distance information from the previous time-step may no longer be correct. This may to some extent be caused by moving different parts of the interfaces with different velocities, but in general most kinds of propagation will invalidate the Euclidian distance field. In order to restore the distance field we make sure the *Eikonal* equation $|\nabla\phi| = 1$ is satisfied. This will guarantee that our distance field is in fact Euclidian. An iterative method is used to propagate information from the interface outwards by solving the equation (τ is used instead of t since these iterations have no interpretation in what we know as time):

$$\frac{\partial\phi}{\partial\tau} + (|\nabla\phi| - 1) = 0 \tag{2.16}$$

Upwind differentiation using a so-called *Godunov* scheme [RT92] is used when calculating $\nabla\phi$ in such a way that information from the direction of the interface is prioritized, meaning that we assume the distances on and close to the interface to be correct. The reason for doing this is that it is desired to have the interface remain in place during the reinitialization and propagate this information outwards. As the distance field converges towards an Euclidian distance field $\frac{\partial\phi}{\partial\tau}$ approaches zero which means a steady-state has been reached were no further changes are necessary.

2.1.3 Discrete Representation of Level Sets

When dealing with level sets in computers sampling is required because of the way memory is represented. In three dimensions space is divided into voxels (cubes) of equal size where each voxel is assigned a value corresponding to the Euclidian distance field value at the center of the voxel. This value is constant within the voxel. In two dimensions space is divided into pixels (squares). An example of a discretized level set representation is shown in figure 2.6. The numbers inside the squares represent the Euclidian distance function at a point in the center of each square.

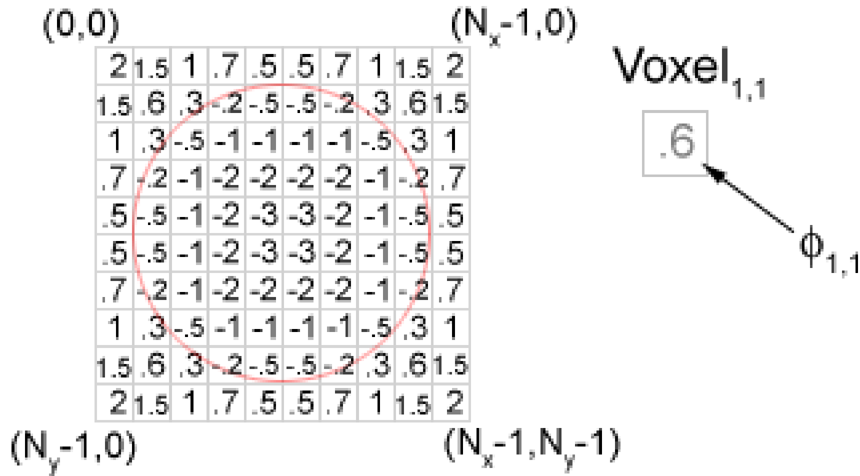


Figure 2.6: Euclidian distance field sampled on a uniform grid. Numbers represent the two dimensional Euclidian distance field at the center of each square. The red circle shows the interface. The numbers in the image are approximations.

Distance values for non-integer coordinates are retrieved by using trilinear interpolation. We will refer to the voxels by their indices i, j, k (in three dimensions).

We define the discrete spatial derivatives in the x-direction as:

$$D^{+} = \frac{\partial \phi}{\partial x} \approx= \frac{\phi_{i+1} - \phi_i}{\Delta x} \tag{2.17}$$

$$D^{-} = \frac{\partial \phi}{\partial x} \approx= \frac{\phi_{i} - \phi_{i-1}}{\Delta x} \tag{2.18}$$

$$D^{0} = \frac{\partial \phi}{\partial x} \approx= \frac{\phi_{i+1} - \phi_{i-1}}{2\Delta x} \tag{2.19}$$

the other directions follow intuitively from this. D^+ and D^- are $O(\Delta x)$ accurate while D^0 is $O(\Delta x^2)$ accurate. In our implementation $\Delta x = \Delta y = \Delta z = 1$ is used to avoid the computationally costly divisions. These first order differences will be used for such things as calculating normals, where accuracy can be traded for speed. However, some situations

require better accuracy, e.g. reinitialization, which means higher order difference schemes have to be used. WENO is fifth order accurate under ideal conditions and third order accurate otherwise. The WENO [LOC94] scheme is defined as:

$$D^{\pm} = w_1(\frac{v_1}{3} - \frac{7v_2}{6} + \frac{11v_3}{6}) + w_2(-\frac{v_2}{6} + \frac{5v_3}{6} + \frac{v_4}{3}) + w_3(\frac{v_3}{3} + \frac{5v_4}{6} - \frac{v_5}{6})$$

For D^+:

$$\begin{aligned} v_1 &= \frac{\phi_{i+3} - \phi_{i+2}}{\Delta x}, v_2 = \frac{\phi_{i+2} - \phi_{i+1}}{\Delta x} \\ v_3 &= \frac{\phi_{i+1} - \phi_i}{\Delta x}, v_4 = \frac{\phi_i - \phi_{i-1}}{\Delta x} \\ v_5 &= \frac{\phi_{i-1} - \phi_{i-2}}{\Delta x} \end{aligned}$$

For D^+:

$$\begin{aligned} v_1 &= \frac{\phi_{i-2} - \phi_{i-3}}{\Delta x}, v_2 = \frac{\phi_{i-1} - \phi_{i-2}}{\Delta x} \\ v_3 &= \frac{\phi_i - \phi_{i-1}}{\Delta x}, v_4 = \frac{\phi_{i+1} - \phi_i}{\Delta x} \\ v_5 &= \frac{\phi_{i+2} - \phi_{i+1}}{\Delta x} \end{aligned}$$

The weights, w_1, w_2, w_3 are given by

$$\begin{aligned} w_1 &= \frac{a_1}{a_1 + a_2 + a_3}, a_1 = \frac{1}{10}\frac{1}{(\varepsilon + S_1)^2} \\ w_2 &= \frac{a_2}{a_1 + a_2 + a_3}, a_2 = \frac{6}{10}\frac{1}{(\varepsilon + S_2)^2} \\ w_3 &= \frac{a_3}{a_1 + a_2 + a_3}, a_3 = \frac{3}{10}\frac{1}{(\varepsilon + S_3)^2} \end{aligned}$$

where $\varepsilon = 10^{-6}$ and S is given by

$$\begin{aligned} S_1 &= \frac{13}{12}(v_1 - 2v_2 + v_3)^2 + \frac{1}{4}(v_1 - 4v_2 + 3v_3)^2 \\ S_2 &= \frac{13}{12}(v_2 - 2v_3 + v_4)^2 + \frac{1}{4}(v_2 - v_4)^2 \\ S_3 &= \frac{13}{12}(v_3 - 2v_4 + v_5)^2 + \frac{1}{4}(3v_3 - 4v_4 + v_5)^2 \end{aligned}$$

2.1.4 Dynamic Tubular Grids

Storing the full grid, as in figure 2.6, is very inefficient, since the region of interest is often limited to the interface. The work by Nielsen and Museth [NM06] describes an efficient data-structure for storing the distance field in a tube around the interface. Their data-structure is called a *Dynamic Tubular Grid* (DT-grid). This approach hugely reduces memory footprints while at the same time allowing for computational efficiency in the narrow band around the interface. Because of this, much higher effective grid resolutions can be used, something that is crucial in our case and has been one of the weaknesses of the level set representation in the past. Furthermore, the DT-grid is not limited to a bounding box in the sense that the narrow-band is rebuilt after propagating the interface. This is extremely useful in our case since snow will rise from an initial surface meaning that the effective grid size will increase as the simulation progresses.

The width of the tubular grid is divided into two bands, the β-band and the γ-band. Narrow bands around the interface allow for higher order operations than just retrieving the interface, such as propagation and reinitialization to mention a few. The following relation must be satisfied for the narrow bands of the DT-grid: $dx \leq \beta \leq \gamma$, where dx is the voxel side in world coordinates. If γ is chosen such that $\gamma - \beta > dx$ this will guarantee that $\nabla\phi$ is defined in the β-band, provided the γ-band is sufficiently wide to include enough information for the finite difference scheme used to calculate $\nabla\phi$. This is easily accomplished by setting γ and β to integer multiples of dx, with $\gamma > \beta$. For first order schemes it is sufficient if $\gamma - \beta \geq 1$. Higher order schemes, such as WENO, require $\gamma - \beta \geq 2$. Figure 2.7 shows the relationships between the bands and illustrates how the Euclidian distance function is sampled in a narrow band around the interface as opposed to the full sampling in figure 2.6.

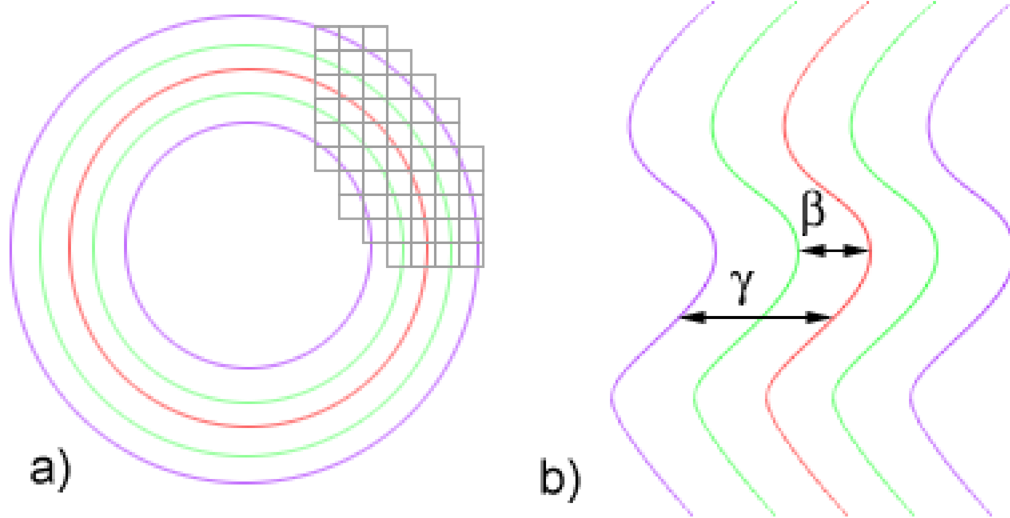


Figure 2.7: a) Voxels are only stored in a narrow band around the interface (red). Only a small part of the voxels are shown here for simplicity. b) The relationships between the different bands.

Although the DT-grid introduces complex data-structures in order to store the distance values efficiently sequential and access is $O(1)$. Even though distance values do not exist outside the narrow band a query to such a location will return $\pm\gamma$ depending on whether the location is inside or outside the interface. This is useful when classifying fluid solver voxels as solid or fluid and for particle-interface collision detection.

2.1.5 Level Set CSG Operations

Constructive Solid Geometry (CSG) is a technique for constructing surfaces from separate building blocks by using boolean operators. These boolean operators can be applied in an elegant fashion to level sets [MBW+02] and are very intuitive to understand. The three basic operations are listed in table 2.1.

Table 2.1: CSG operations for distance volumes (discrete level sets) V_A and V_B assuming positive outside and negative inside values.

CSG Operation	Implementation
Intersection, $A \bigcap B$	$Max(V_A, V_B)$
Union, $A \bigcup B$	$Min(V_A, V_B)$
Difference, $A - B$	$Max(V_A, -V_B)$

The geometric interpretation of table 2.1 is shown in figure 2.8

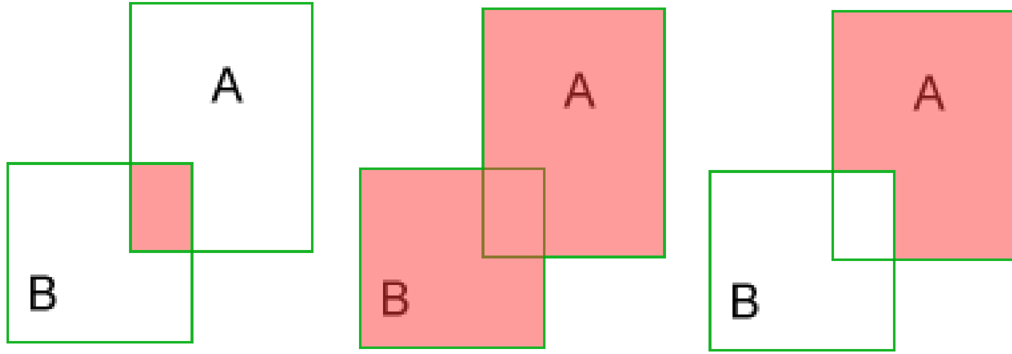


Figure 2.8: From the left: $A \bigcap B$ (Intersection), $A \bigcup B$ (Union), $A - B$ (Difference).

2.2 Computational Fluid Dynamics

This sections outlines the complex field of computational fluid dynamics (CFD) in computer graphics. First we present the equations governing fluid behaviour and proceed to explain the stable fluids [Sta99] method for solving these equations. The reason for diving into this field is that these methods can be used to compute wind fields, as air can be treated as a low viscosity fluid.

2.2.1 The Navier-Stokes equations

The Navier-Stokes equations (NS equations) have been known for over a century [Nav1827, Sto1851]. This model is derived from the conservation of mass and momentum and is a set of non-linear partial differential equations. As no analytical solution to these equations exists we are forced to solve them numerically. Doing this was not possible until recently when the invention of micro-processors provided the necessary computational power.

The compressible effect in a gas, such as air, can be neglected when the velocity is below the speed of sound [FSJ01]. Therefore we use the NS equations for viscous incompressible flow, which in differential form can be written as

$$\frac{\partial \vec{v}}{\partial t} = \vec{F} + \nu \cdot \nabla^2 \vec{v} - (\vec{v} \bullet \nabla)\vec{v} - \frac{\nabla p}{\rho} \quad (2.20)$$

$$\nabla \bullet \vec{v} = 0 \quad (2.21)$$

$$(2.22)$$

under the assumption that the fluid's temperature and density are constant and that the velocity and pressure are known for some initial time $t = 0$. $\vec{F}$ denotes external forces (such as gravity), ν is the kinematic viscosity of the fluid and ρ its density, p and $\vec{v}$ are pressure and velocity respectively. Equation 2.20 ensures that momentum is conserved. We will not derive this here and instead refer the reader to any standard text on CFD for a complete derivation. The second equation states that the velocity field must be divergence free, i.e. there must be no sinks or sources. Velocity, $\vec{v}$, is the derivative of position, $\vec{x}$, with respect to time for all positions.

The NS equations also have to be supplemented by additional boundary conditions, controlling what happens at the boundaries between fluids and solids. This will be further described in chapter 3 when we describe the implementation of our wind field.

2.2.2 Stable Fluids

The field of computer graphics differs from that of computational physics in that computer graphics strives to produce impressive images while the field of computational physics aims to produce exact numbers. This means that computer graphics can allow for short-cuts that produce good-looking results at the expense of physical correctness. A method that takes advantage of this fact is the stable fluids method proposed by Stam in 1999 [Sta99]. Approaches to solve the NS equations with finite differencing and explicit time integration, such as the one proposed by Foster and Metaxas in 1996 [FM96], are unstable for large time-steps and may cause simulations to *blow up*, resulting in the simulation having to be restarted from the beginning with a smaller time-step. Unstable applications are not well suited for interactivity and can also make over-night simulations a total waste of time.

What makes the stable fluids approach attractive to the computer graphics community is that it is stable regardless of the time-step used. The fact that it is not as accurate as more complex models does not matter in this case. The following equations are solved on a Cartesian voxel grid.

The foundation of the stable fluids approach is a mathematical theory known as the *Helmholtz-Hedge Decomposition* [Bet98]. This theory states that any vector field can be uniquely decomposed into the form

$$\vec{w} = \vec{v} + \nabla q \tag{2.23}$$

where $\vec{v}$ has zero divergence ($\nabla \bullet \vec{v} = 0$) and ∇q is an irrotational gradient field. Recall from the NS equations that they require the velocity field to be divergence free. This insight allows us to define an operator $\mathbf{P}$ that projects the vector field $\vec{w}$ onto its divergence free part $\vec{v} = \mathbf{P}\vec{w}$. If we rearrange equation 2.23 as follows

$$\vec{v} = \vec{w} - \nabla q \tag{2.24}$$

This means that if we can find the scalar field q we can make our velocity field divergence free. This is done by multiplying both sides of equation 2.23 by ∇. Recall that the term $\nabla \bullet \vec{v} = 0$ and will disappear, leaving us with

$$\nabla \bullet \vec{w} = \nabla^2 q \tag{2.25}$$

This is a so called *Poisson* equation for the scalar field q with Neumann boundary condition $\frac{\partial \vec{v}}{\partial \vec{n}} = 0$ on the interface, where $\vec{n}$ is the interface normal. Once we find q it is trivial to calculate $\vec{v}$ from equation 2.23. This is the relationship we end up with:

$$\vec{v} = \mathbf{P}\vec{w} = \vec{w} - \nabla q \tag{2.26}$$

Applying operator $\mathbf{P}$ to both sides of equation 2.20 gives us a single equation for velocity, since $\mathbf{P}$ guarantees that equation 2.21 is satisfied:

$$\frac{\partial \vec{v}}{\partial t} = \mathbf{P}(\vec{F} + \nu \cdot \nabla^2 \vec{v} - (\vec{v} \bullet \nabla)\vec{v}) \tag{2.27}$$

using $\mathbf{P}\vec{v} = \vec{v}$. Notice that the term $\mathbf{P}(-\frac{\nabla p}{\rho}) = 0$ disappears since it is a scalar field and as such is only divergence free if it has a constant value, which in turn is the assumption made for incompressible flow.

Finding the velocity field $\vec{v}$ is now divided into steps where each of the terms is solved individually. In the case of air the viscosity term $\nu \cdot \nabla^2 \vec{v}$ can be neglected since it is very small, $\nu_{air} = 1.73 \cdot 10^-5\ Ns/m^2$, according to NASA. When solving for such fluids as water gravity is the force that drives the simulation. A common setup for water simulations is to have solid objects, air and water regions within the simulation domain. The water will obey the laws of gravity and interact with the solid objects to create such effects as splashing and swirling. In the case of wind there is only solids and air. Although wind could be driven by external forces we have chosen to use another approach, further described in chapter 3, where we set the velocities on the boundaries of the simulation domain since this is a more intuitive way of modeling wind sources and in our case provides us with a global wind direction outside the fluid solver domain. The actual fluid equation we are solving is this:

$$\frac{\partial \vec{v}}{\partial t} = \mathbf{P}(-(\vec{v} \bullet \nabla)\vec{v}) \tag{2.28}$$

Provided the velocity field is known at a time $\vec{v}_t$ the following time step $\vec{v}_{t+\Delta t}$ is solved for as follows according to the original method proposed by Stam:

$$\vec{v}_t(\vec{x}) \overbrace{\rightarrow}^{add\ force} \vec{v}_0(\vec{x}) \overbrace{\rightarrow}^{advect} \vec{v}_1(\vec{x}) \overbrace{\rightarrow}^{diffuse} \vec{v}_2(\vec{x}) \overbrace{\rightarrow}^{project} \vec{v}_{t+\Delta t}(\vec{x}) \tag{2.29}$$

For our purpose diffusion and forces will not be considered, since the viscosity for air is very small and instead of using forces as wind-sources we set velocities on the boundary of the fluid solver domain. Our method of solution is then

$$\vec{v}_t(\vec{x}) \overbrace{\rightarrow}^{advect} \vec{v}_0(\vec{x}) \overbrace{\rightarrow}^{project} \vec{v}_{t+\Delta t}(\vec{x}) \tag{2.30}$$

Only the advection and projection step will be explained here since only these will be used for calculating the wind field. We refer the reader to [Sta99] for a full description of the original method.

The advection term $(\vec{v} \bullet \nabla)\vec{v}$ is the non-linear term in the NS equations. This can be interpreted as moving the velocity field along itself, hence this term is sometimes referred to as the *self-advection* term. To obtain the velocity at a point $\vec{x}$ at a time $t + \Delta t$ the point $\vec{x}$ is traced backwards over a time Δt in the velocity field $\vec{v}_t$. The new velocity at point $\vec{x}$ is then set to the velocity it had Δt time ago at its previous position. This is illustrated in figure 2.9.

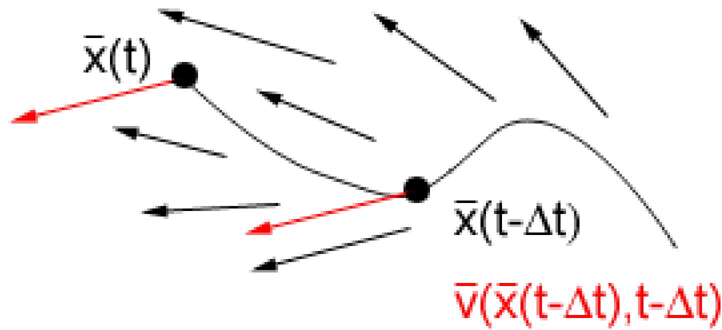


Figure 2.9: The velocity at position $\vec{x}(t)$ at time t is set to the velocity $\vec{v}(\vec{x}(t-\Delta t), t-\Delta t)$ at a position $\vec{x}(t-\Delta t)$.

This method of solving the self-advection term is based on a technique for solving partial differential equations known as the *method of characteristics*. This technique can be used to solve advection equations of the type

$$\begin{aligned} \frac{\partial a(\vec{x},t)}{\partial t} &= -\vec{v}(\vec{x}) \bullet \nabla a(\vec{x},t) \\ a(\vec{x},0) &= a_0(\vec{x}) \end{aligned}$$

where a is a scalar field, $\vec{v}$ is a steady vector field and a_0 is the field at time $t = 0$. Let $\vec{p}(\vec{x}_0, t)$ denote the *characteristics* of the vector field $\vec{v}$ which flow through the point $\vec{x}_0$ at $t = 0$:

$$\begin{aligned} \frac{\partial}{\partial t}\vec{p}(\vec{x}_0,t) &= \vec{v}(\vec{x}_0,t) \\ \vec{p}(\vec{x}_0,t) &= \vec{x}_0 \end{aligned}$$

Now let $\vec{a}(\vec{x}_0, t) = a(\vec{p}(\vec{x}_0, t), t)$ be the value of the field along the characteristic passing through the point $\vec{x}_0$ at $t = 0$. The variation of this quantity over time can be computed using the chain rule of differentiation:

$$\frac{d\vec{a}}{dt} = \frac{\partial a}{\partial t} + \vec{v} \bullet \nabla a = 0$$

This shows that the value of the scalar does not change along the streamlines. In particular, we have $\vec{a}(\vec{x}_0, t) = \vec{a}(\vec{x}_0, 0) = a_0(\vec{x}_0)$. Therefore, the initial field and the characteristics entirely define the solution to the advection problem. The field for a given time t and location $\vec{x}$ is computed by first tracing the location $\vec{x}$ back in time along the characteristics to get the point $\vec{x}_0$, and then evaluating the initial field at that point:

$$a(\vec{p}(\vec{x}_0, t), t) = a_0(\vec{x}_0)$$

This method is used to solve the advection equation over a time interval $[t; t + \Delta t]$ for the fluid. In this case, $\vec{v} = \vec{v}_{fluid}(\vec{x}, t)$ and a_0 is any of the components of the fluid's velocity at time t.

In order to guarantee that $\vec{v}_{t+\Delta t}(\vec{x})$ is divergence free we carry out the projection step that projects $\vec{v}_0(\vec{x})$ onto its divergence free part:

$$\vec{v}_{t+\Delta t}(\vec{x}) = \vec{v}_0(\vec{x}) - \nabla q \tag{2.31}$$

where q is the scalar field representing the divergence of $\vec{v}_0(\vec{x})$.

The Laplace operator ∇^2 acting on a scalar field q can be discretized as

$$\nabla^2 q_{i,j,k} = \frac{q_{i+1,j,k} + q_{i-1,j,k} - 2q_{i,j,k}}{(\Delta x)^2} + \frac{q_{i,j+1,k} + q_{i,j-1,k} - 2q_{i,j,k}}{(\Delta y)^2} + \frac{q_{i,j,k+1} + q_{i,j,k-1} - 2q_{i,j,k}}{(\Delta z)^2} \tag{2.32}$$

where $\Delta x, \Delta y, \Delta z$ represent the length of a side of a voxel in each spatial dimension respectively and $q_{i,j,k}$ is the voxel identification. In most cases (including ours) $\Delta x = \Delta y = \Delta z = 1$ is used to avoid having to divide the right-hand side terms. Solving equation 2.25 for $\vec{q}$ for all voxels gives the following linear system:

$$\mathbf{A}\vec{q} = \nabla\vec{v}_0 \tag{2.33}$$

where $\mathbf{A}$ is a $N \times N$ ($N = N_x \cdot N_y \cdot N_z$, where N_{dim} is the number of voxels in each dimension respectively) and $\vec{q}$ and $\vec{v}_0$ are N-dimensional vectors representing the divergence of $\vec{v}_0$ and the intermediate velocity field respectively. $\mathbf{A}$ is a sparse positive definite matrix with non-zero elements concentrated around the diagonal. To solve equation 2.33 the matrix $\mathbf{A}$ must be inverted. As this matrix is typically very large[1] an efficient method of doing this is required. As in [FSJ01, FF01] a preconditioned conjugate gradient solver is used for this purpose with the termination criteria that the largest residual should be smaller than some user-defined value, allowing the user to control the trade-off between speed and precision. The matrix is efficiently stored in memory using dynamic arrays to represent only the non-zero elements.

The matrix $\mathbf{A}$ is sometimes referred to as the *connectivity matrix*. This is because it describes the connectivity between voxels to satisfy the Neumann boundary condition,

[1] A $64 \times 64 \times 64$ voxel grid results in a matrix with dimensions 262144×262144.

$\frac{\partial \vec{v}}{\partial \vec{n}} = 0$. This means that there should be no change in flow along the the normal of a boundary interface. To enforce this we break the connection between solid voxels and any air voxels adjacent to them, disallowing any flow exchange. Figure 2.10 illustrates this. Chapter 3 contains a description of how the cell classifications is done.

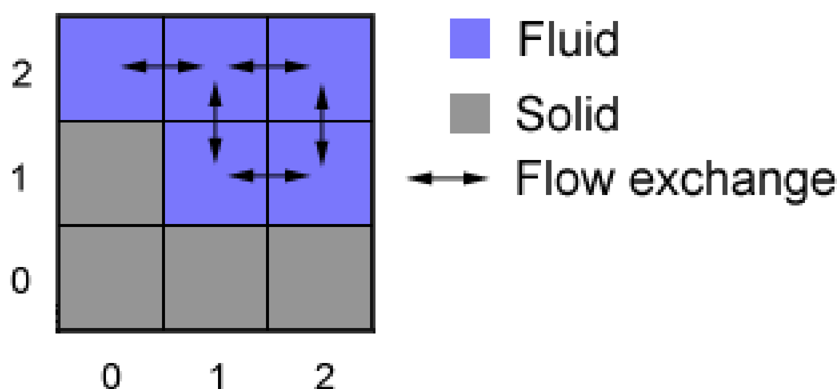


Figure 2.10: The Neumann boundary condition allows flow exchange between fluid cells only.

The connectivity matrix for the pixels (voxels in 3D) in figure 2.10, assuming all boundaries are closed (i.e. no flow exchange with the area outside the simulation domain), would be:

$$
\mathbf{A} = \begin{array}{c} \\ \end{array}
\begin{array}{ccccccccc}
(0,0) & (1,0) & (2,0) & (0,1) & (1,1) & (2,1) & (0,2) & (1,2) & (2,2)
\end{array}
$$

$$
\mathbf{A} = \begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -2 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & -2 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & -3 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -2
\end{pmatrix}
\begin{array}{c}
(0,0) \\ (1,0) \\ (2,0) \\ (0,1) \\ (1,1) \\ (2,1) \\ (0,2) \\ (1,2) \\ (2,2)
\end{array}
$$

Rows representing solid grid cells have all elements set to zero since now flow exchange

takes place here. The diagonal element is the negated number of fluid neighbors of the grid cell. A one at element $\mathbf{A}_{i,j}$ with $(i \neq j)$ allows flow exchange between the two corresponding grid cells.

Chapter 3

Implementation

This section describes the different components we have implemented. The program was written in C++, taking advantage of the speed and object-orientation that this language offers. However, the following methods could also be implemented other languages if this is more convenient.

Figure 3.1 shows an overview of the data flow in our method. It begins with a hole-free mesh generated by one of many commercial software packages available. The mesh needs to be hole-free for scan converter [Mau03] to be able to convert it into a level set. However, many modeling programs have built-in tools for filling holes in meshes.

The scan converter will produce a DT-grid containing the Euclidian distances for the level set. Our program assumes two DT-grids for each solid object in the scene. The first is a static level set, serving as boundary condition for the snow buildup. The second is an advectable level set which will represent accumulated snow. Initially these are identical, but this is not required by the program. The reason for taking two DT-grids as input instead of just one that could easily be duplicated is that by taking two the program offers the user the option of continuing a simulation after it was interrupted. Since the snow-grid is output to file when the program terminates the user may pass this file as the second grid, in which case the two level set surfaces are not identical to begin with.

Our program proceeds by simulating snow buildup in the scene and may be interrupted at any time. A preview of the scene, rendered with OpenGL, is offered as a guideline for when to interrupt the simulation. Alternatively the user may specify a set number of frames for the program to run or a certain amount of particles to be spawned before termination.

The program outputs a file for each object in the scene containing geometric information about the snow surface as triangles that can be read by commercial rendering tools. Conversion from implicit surfaces to triangles is done by applying the *Marching Cubes* algorithm [LC87] on the snow level sets in the scene. A snow level set is defined as the CSG difference between the snow level set and the corresponding solid. This keeps the triangle count low and avoids problems with depth buffering caused by the interface moving slightly when reinitializing. The program also outputs a script file that describes the state of the

particle system for the entire simulation. If desired, the snow surface information can be output for every frame, which is useful for animations, but this generates a huge amount of data so it is optional.

Finally, commercial rendering software outputs ray-traced images or a sequence of images of high-quality. Rendering techniques such as global illumination and/or sub-surface scattering may be used to achieve very convincing results. A sequence of images may be encoded into an animation if desired.

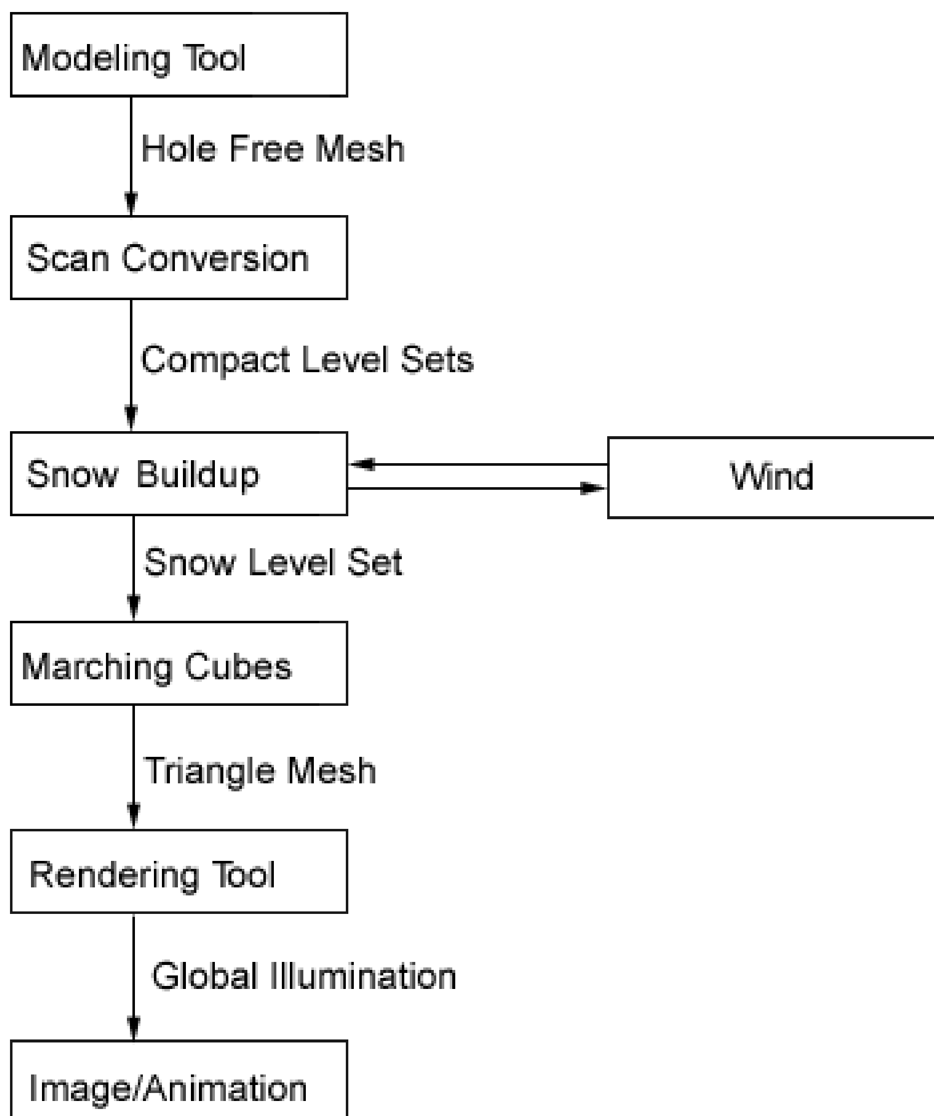


Figure 3.1: An overview of the data flow used in our method.

3.1 Dual Level Set Representation

Every solid in a scene is represented by two level sets, one that is static and one that is advectable. The static level set, or *solid level set*, represents the actual object, like a bunny or a sphere, that snow builds up on and remains constant for the duration of the simulation. The advectable level set, which will sometimes be referred to as the *snow level set*, on the other hand represents the accumulated snow. For a scene where no snow has yet fallen these two level sets are identical. Typically, this is the case at the beginning of a simulation, but the program does not assume this. The solid level set and the snow level set have the same effective resolution.

Different situations require different properties for the snow level set. For instance, when outputting the results of the simulation to a triangle mesh only the accumulated snow is of relevance, as the solid given as input can and should be reused[1]. However, in the cases of package sliding, collision detection and fluid grid cell classification the snow level set must incorporate the underlying solid level set. The solution to this is to use the CSG operators defined in chapter 2.

In the cases where the snow level set needs to incorporate the underlying solid we define the snow level set as the union between the solid and the accumulated snow. For output and rendering the snow level set is defined as the difference between the the accumulated snow and the underlying solid.

3.2 The Wind Field

As mentioned previously our fluid solver is based on the work by Stam [Sta99]. Recall from chapter 2 that the basic setup for the stable fluids approach is a Cartesian grid with pressures and velocities stored at the centers of the voxels. In this section we will refer to the voxels as *grid cells*. The grid cells in this grid are initially classified as solid or non-solid (fluid). This is illustrated in figure 3.2.

[1] The reason why it should be reused is that converting a mesh into a level set and then extracting a new mesh by using marching cubes will in most cases result in a loss of detail and a greater number of triangles.

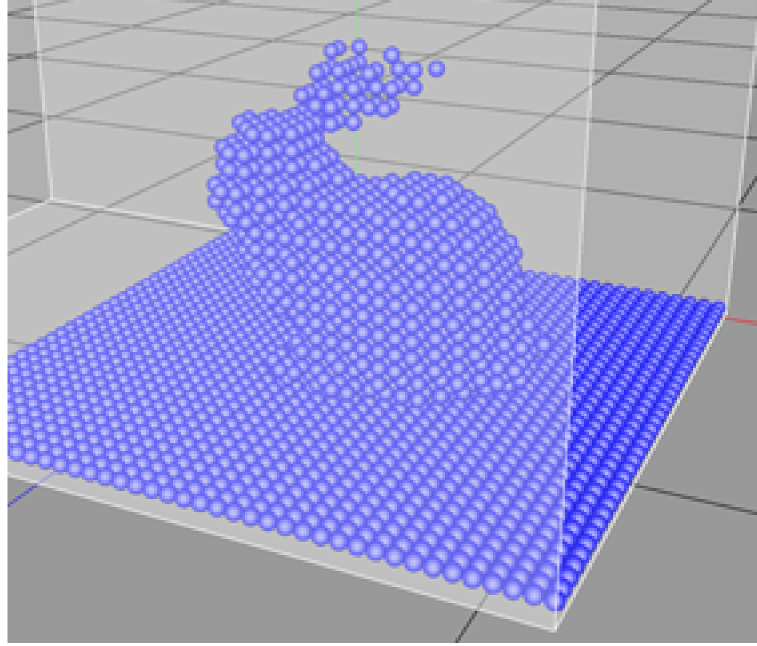


Figure 3.2: Stanford bunny discretized in the fluid solver domain. Solid grid cells rendered as blue spheres, non-solid cells are not shown but make up the rest of the fluid domain (the semi-transparent box) in this case.

3.2.1 Grid Cell Classification

Grid cell classification is done in two passes. First all solid grid cells are identified. A fluid grid cell is classified as solid if ϕ at that point is smaller than $0.5\Delta\delta$, where $\Delta\delta$ is the side of a voxel. Recall from chapter 2 that $\Delta\delta = 1$ to avoid divisions when calculating partial derivatives. This translates to checking if the interface passes through a sphere with radius half the grid cell size inscribed in the grid cell. However, to efficiently handle internal boundaries, in-flow and out-flow we extend the cell classifications somewhat. In the second pass grid cells with solid neighbors, referred to as *boundary cells* (figure 3.3), are found and assigned a normal direction by using the distance field of the boundary level set. For this reason it is important that the normal is defined at the locations of the boundary cells. This is achieved by setting the *beta*-band width of the boundary level set to a value larger or equal to the spacing between two voxels in the fluid grid, since we have already established that the gradient of the distance function (the normal) is defined within the β-band.

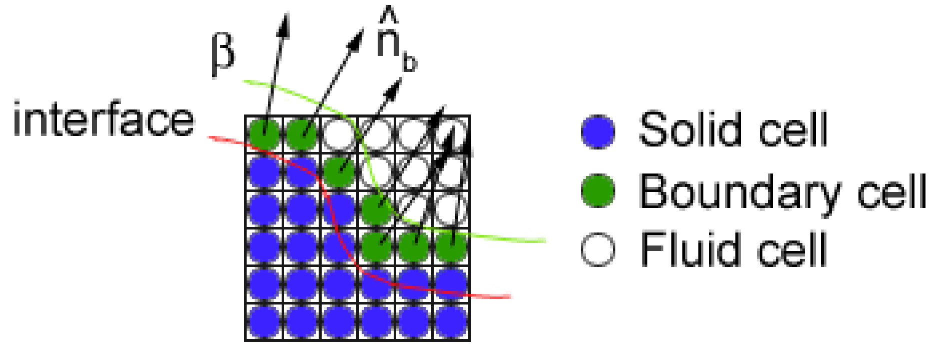


Figure 3.3: Grid cell classification showing that it is crucial that the boundary cells are positioned inside the β-band as the normals are used to make sure the wind field does not flow into solids.

The boundary cells are used with the Dirichlet boundary condition to assure that $\hat{n}_b \bullet \vec{v} = 0$ in these cells, where $\hat{n}_b$ is the normal stored in the cell and $\vec{v}$ is the wind velocity. This prevents wind from flowing into objects by projecting the velocity onto the tangent plane of the surface. Cells on any of the six walls of the solver domain are classified as either open or closed by the user and are not solver for. Closed cells will act as walls preventing the wind from flowing out of the domain. Open cells are classified as either in-flow or out-flow cells, depending on the global wind direction (which is user-defined). If the inward facing normal of an open cell points in the same direction as the global wind direction, $\hat{n}_i \bullet \vec{v}_{global} > 0$, the cell is classified as an in-flow cell and its velocity is set to the global wind direction. If on the other hand $\hat{n}_i \bullet \vec{v}_{global} < 0$ the cell is classified as an out-flow cell. The velocity for out-flow cells is not set and never solved for. Instead we simply allow grid cells neighboring out-flow cells to calculate partial derivatives in that direction as if the out-flow cell were an ordinary fluid cell. Using a global wind direction to drive the fluid simulation means that snow particles outside the solver domain can use the global wind direction in the movement model.

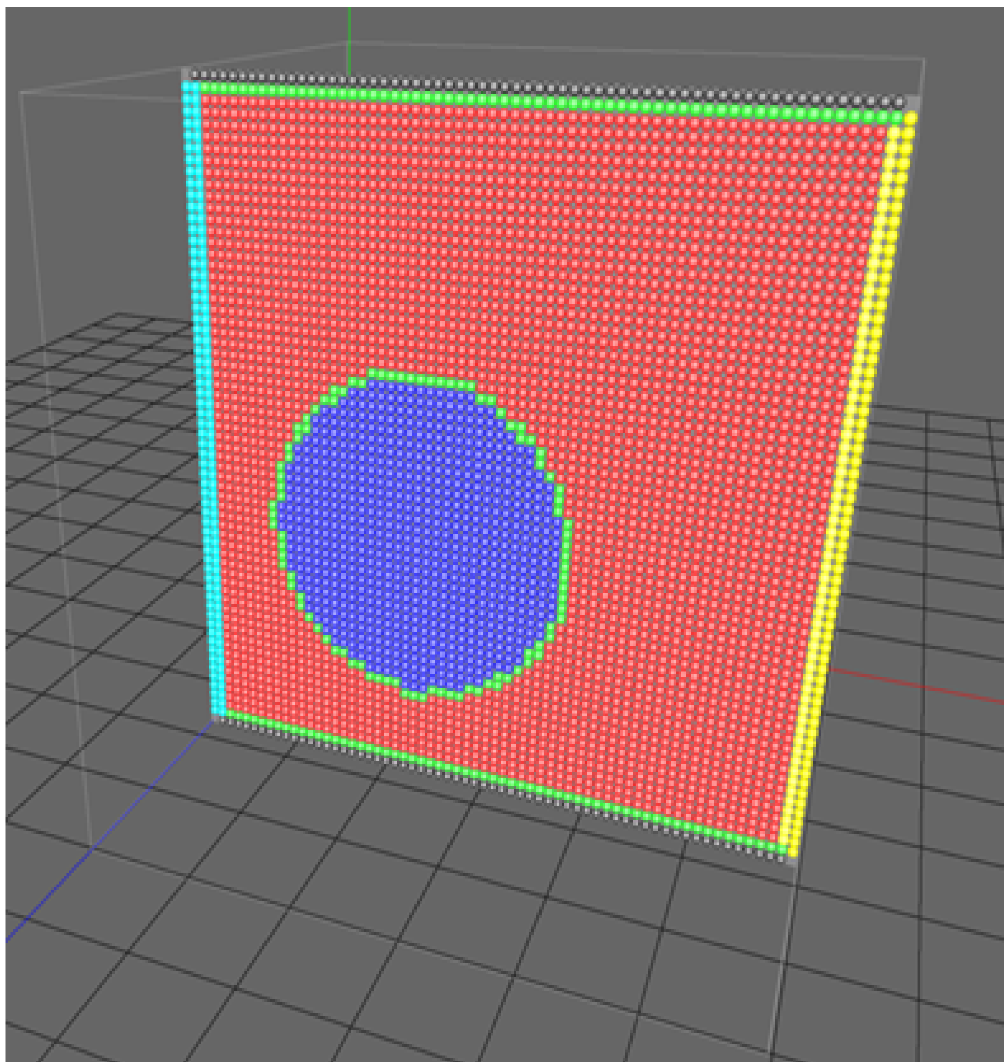


Figure 3.4: Cross-section of the fluid solver domain showing grid cell classification. Red grid cells are fluid cells, blue grid cells are inside a solid object (in this case a sphere), green grid cells have one or more solid neighbor cells, yellow cells are cells were there is wind flowing into the domain or cells with such neighbors, turquoise cells are outflow cells or cells with such neighbors and finally, black cells are solid cells on the edge on the solver domain.

3.2.2 Update Cycle

Updating the wind field is done in a series of operations. As these operations have already been described in chapter 2 we will only present the order in which they are applied here.

1. First of all grid cells are classified. This is done for every time-step in the simulation. If found that the number of solid cells has changed since the last time-step the velocity field is updated, otherwise this is not done to save simulation time. Should

the number of solid cells be the same as for the previous time-step the update cycle is aborted here.

2. Next, the connectivity matrix **A** is rebuilt.

3. The first step in the actual solving for the new velocities is the self-advection step, which was explained in chapter 2. The time-step used is based on the maximum velocity in the fluid domain.

4. As the velocity field is not guaranteed to be divergence free after the self-advection step the projection operator is applied on this velocity field in order to extract its divergence free part. This involves inverting the connectivity matrix by using the conjugate gradient method and is the most time-consuming part of the wind field calculations.

Figure 3.5 shows some results. These are presented here as they will not be discussed together with the results for the snow accumulation. Worth noting is the vorticity behind the circle in the top image. Vorticity is arguably one of the most interesting features of a wind field, especially when moving snowflakes in it as these will swirl and move chaotically in these turbulent areas. However, for determining accumulation patterns vorticity does not make a huge difference and a semi-static wind field (as described above) is used for this, as the computation time is better spent in other areas, such as updating the snow level set. It should also be noted that in nature snow accumulation requires an enormous amount of snowflakes, more than we could ever hope to simulate. For the limited number of snowflakes used in the short animations made for this thesis the accumulation from these would have had little or no impact at all on the built up snow. This allows us to use a dynamic wind field for advecting snowflakes since these will not trigger computationally expensive level set updates. Furthermore, this means that we add snowflakes moving in a dynamic wind field to a scene that is already snow covered in a separate step. Screenshots from snowflake animations will be presented in the Results chapter.

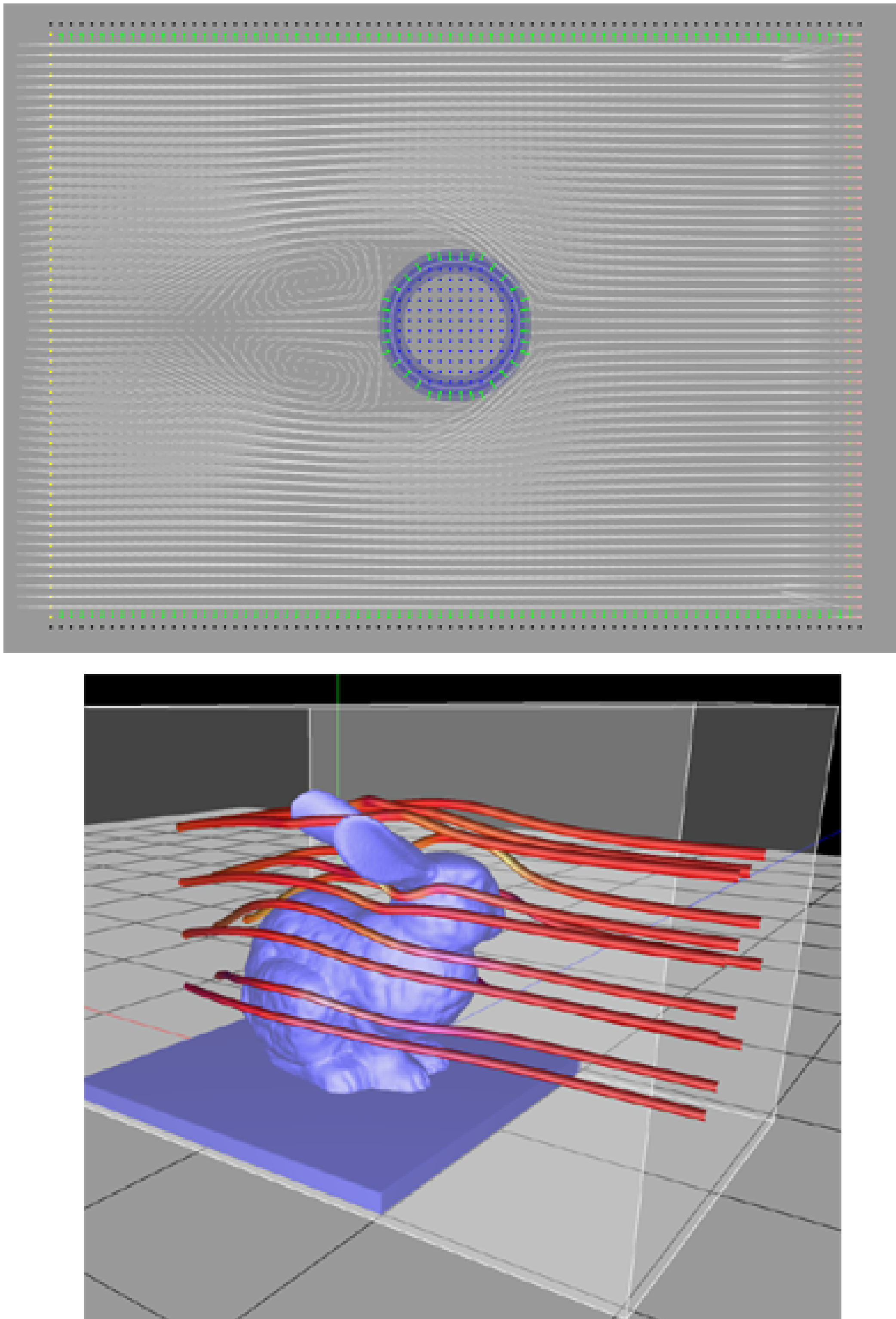


Figure 3.5: Top, two dimensional flow around a circle. The grey lines represent the velocities on the fluid solver grid. Notice the vorticity behind the circle. Bottom, visualization of three dimensional flow around a Stanford bunny using streamlines. The semi-transparent box surrounding the bunny is the fluid solver domain. Wind direction is right-to-left in both images.

3.3 Snow Particles

We define three different types of snow particles. They are closely related but used for different purposes. Snow packages and slide packages are used to determine snow accumulation patterns, while snowflakes are added to the final scene to visualize snowfall. Since the three types of snow particles are closely related we can define a base class `SnowParticle` from which the others can be derived (figure 3.6). More detailed information about these classes is available in Appendix A.

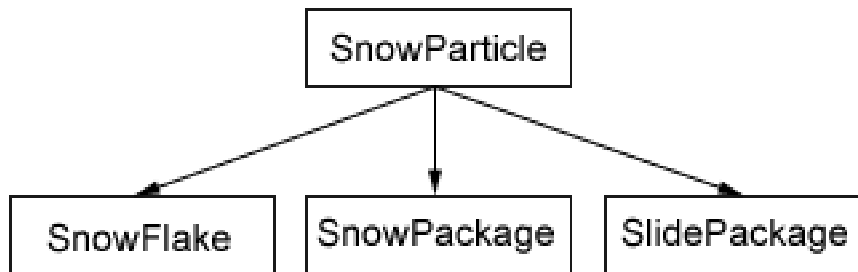


Figure 3.6: Class hierarchy for different types of snow particles.

Snow particles are stored in a list for efficient insertion and deletion. We will refer to this list as *the particle system* [Ree83]. Particles are *spawned* (initiated) in a random location on a *source plane* of arbitrary orientation and dimensions. The source plane should be setup so that the snow particles interact with the objects in the scene. Therefore the obvious approach of positioning it directly above the scene does not always work, since a strong wind would then transport the particles away from the scene before they had any chance of colliding with the objects in the scene.

3.3.1 Snow Packages

Snow packages are volume carrying Lagrangian tracker particles. The interaction between snow packages and the snow level sets is what causes snow to accumulate and drives the buildup. The collisions with the snow level sets are tested and represent locations were snow will accumulate, if the locations are valid collision points, as explained below. The size of individual snow packages can be set arbitrarily. As we shall see using smaller snow packages is more accurate, but this also means that more snow packages will have to collide with the snow level sets in order to build up an equivalent amount of snow. Also, the resolution of the advectable level sets limits the minimum size allowed for snow packages, as explained in section 3.5.

3.3.2 Slide Packages

As we wish to ensure that built-up snow is stable in the sense that it looks physically plausible snow packages may not always be able to deposit their total volume at their points of impact with the advectable level sets. In fact, if a snow package collides with a snow level set at a location that cannot support any accumulation at all the whole volume is split into slide particles. The remaining volume is then split into an arbitrary, user-defined number of slide packages that move along the surface, depositing volume if their current locations allow it. If a slide package should slide off the surface and become air-born it is converted into a snow package, which is then moved in the wind field.

The motivation for using slide packages is that they will make snow accumulate more realistically as well as smooth the snow surface. Because they move in the gravity field they will tend to deposit their volumes at local minima points, effectively making the surface more smooth. Furthermore, slide package sizes will automatically adapt to the level of detail in a scene. If a snow package collides with a detail much smaller than its radius the collision will not be valid and the package will be split. The resulting slide package may then be small enough to cover the detail in snow. This means the original size of the snow packages, set by the user, is actually less important than the minimum size allowed for slide particles, also set by the user, a feature that makes it much easier to fine-tune simulations. The reason for that being that lowering the minimum allowed size will always give better results, at the cost of the simulation being slower of course.

3.3.3 Snowflakes

As mentioned earlier snowflakes do not contribute to snow buildup. A snowflake is removed from the particle system if it collides with an advectable level set. A new snowflake is spawned at a random location on the source plane to keep the particle count constant.

Snow packages and slide packages are typically not included in final production renderings so little effort has been made in order enhance the visualization of them. However, snowflakes will greatly increase the visual quality of a winter scene. Moeslund et. al. [MMA+05] propose an interesting method for generating realistic snowflakes. Instead of using the naive approach with a texture on a so-called *billboard*[2] they represent each snowflake as a rotating triangle mesh. The advantage is that the silhouettes of the snowflakes will change over time which more accurately mimics the behaviour of real snowflakes. The snowflakes are divided into layers, each layer containing the same amount of triangles, meaning that the triangles are more tightly packed at the center of the snowflake. For a full description of this method we refer the reader to [AL04].

[2]This is basically a quad oriented so that it is always facing the camera.

To save memory we have chosen to create 256 template snowflake meshes and assign an index into this mesh database for each snowflake. Results show that even though two snowflakes have the same mesh, their different rotations are enough to give them a unique appearance.

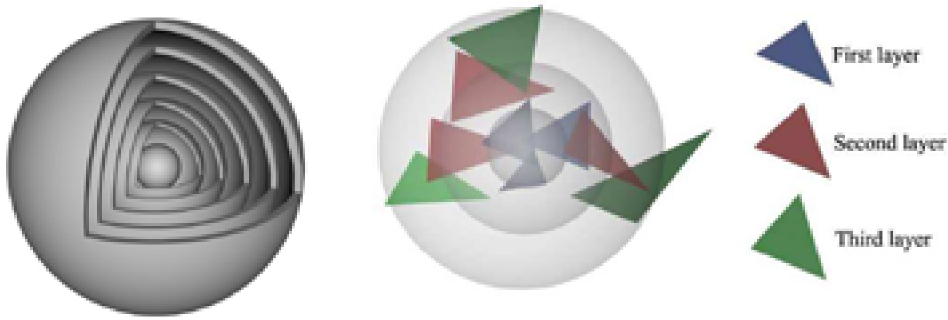


Figure 3.7: Triangles distributed per layer with constraints to prevent free-floating triangles. Image courtesy of [AL04].

Snowflake size and density are controlled by a global temperature parameter. Following the results in [Jun00] an equation for the snowflake radius is proposed in [AL04]:

$$r = \begin{cases} 0.0075 \cdot |T|^{-0.35} & T \leq -0.061 \\ 0.02 & T > -0.061 \end{cases} \tag{3.1}$$

where r is the radius in meters and T is the temperature in degrees Celsius[3]. Note that this equation is C^0 continuous. The absolute value of T is used in order to avoid complex numbers. The radius from equation 3.1 is used as an average and is further modified randomly within the interval $\pm 50\%$ for each snowflake. Snowflakes remain constant after they are spawned.

The number of triangles per layer in a snowflake mesh is determined by the density of the snowflake, where a more dense snowflake will have more triangles per layer. According to [RVC+98] the density of a snowflake is inversely proportional to the radius. The following relationship is given:

$$\begin{aligned} \rho_{snowflake} &= \frac{C_{humidity}}{r} \\ C_{dry} &= 0.085 \\ C_{wet} &= 0.362 \end{aligned} \tag{3.2}$$

[3]We assume temperatures to be below zero. This is a reasonable simplification since we will not be modeling melting snow.

where $C_{humidity}$ is measured in $[\frac{kg}{m^2}]$. The distinction between dry and wet snow is set to minus one degree Celsius. For temperatures below minus one we use C_{dry} and for temperatures above we use C_{wet}.

To make the snowflakes appear a little more "fuzzy" we use two meshes per snowflake. The meshes are created separately to capture the randomness involved for both meshes. However, when rendering the snowflake both meshes are considered and one is rendered with a semi-transparent material giving the snowflake a softer appearance.

In conclusion, the steps required to create a snowflake mesh are listed [AL04]:

1. Find the snowflake radius from equation 3.1.
2. Calculate the number of layers necessary (same for both meshes): $n_{layers} = \lceil \frac{r_{snowflake}}{r_{layer}} \rceil$, where the integer n_{layers} is the number of layers and r_{layer} is the radius of a single layer, which is constant for all layers.
3. For each layer (in both meshes), add a number of triangles based on the density of the snowflake. For dry snowflakes we use 10 triangles per layer and for wet snowflakes 40 triangles per layer.

Close-up of two out of the 256 snowflake template meshes used for -3℃ are shown in figure 3.8. Template meshes are temperature dependent and must be generated separately for each temperature.

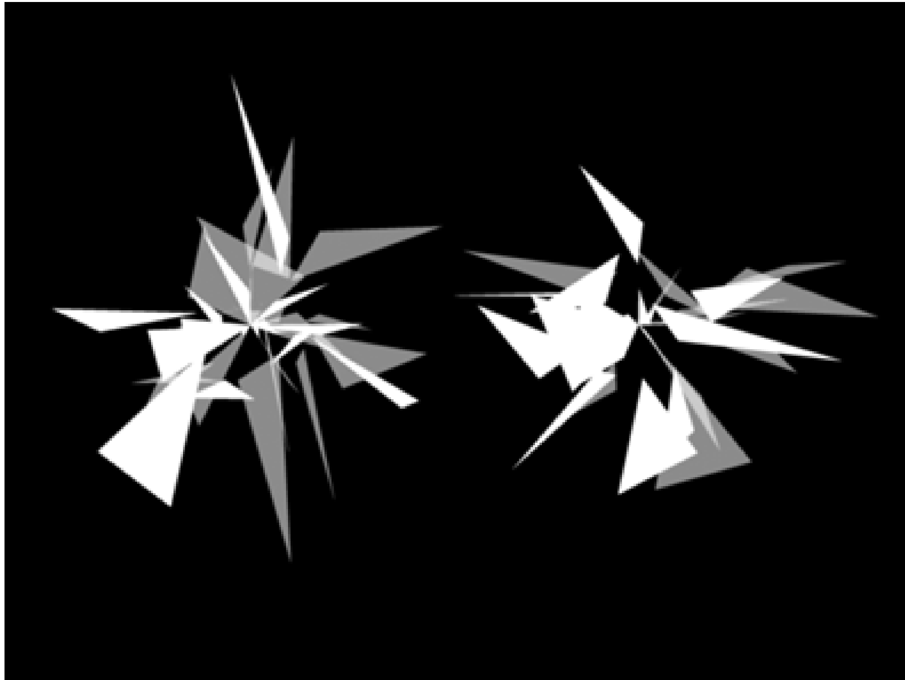


Figure 3.8: Close-up of two out of 256 snowflake template meshes used for -3℃ .

3.4 Advecting Snow Particles in the Wind Field

Snowflakes do not contribute to snow accumulation but play an important role in final productions of winter scenes. A good model for their movement together with a consistent rendering technique (as described above) will achieve this. Snow packages represent falling snow, albeit in larger quantities than single snowflakes. To generate realistic accumulation patterns snow packages use the same movement model as snowflakes. This is a realistic approach since it means that the points of impact will correspond well to those of snowflakes. As opposed to snowflakes and snow packages, slide packages are not advected in the wind field. Instead they are moved along the surface of the object on which they are sliding. When we describe the model in the following section we will refer to snowflakes being moved. Snow packages are treated as snowflakes in this regard and have all the properties that snowflakes have. In addition they also have information about the package radius, which is not to be confused with the radius in equation 3.1 as it is completely separate and used *only* when setting the speed function of the advectable level set.

3.4.1 Snow Packages and Snowflakes

We adopt the method described in [MMA+05, AL04] for our movement model. Four forces are considered to act upon snowflakes:

- Gravity ($\vec{F}_g$): The force that pulls snowflakes downward, modelled as:

  $$\vec{F}_g = V_{snowflake} \cdot \rho_{snow} \cdot \vec{g} \tag{3.3}$$

  where $V_{snowflake}$ is the volume of the snowflake, found by assuming the snowflake is sphere-shaped with a known radius (equation 3.1), ρ_{snow} is given by equation 3.2 and $\vec{g}$ is the gravity vector. We use the standard value of $\vec{g} = (0, -9.82, 0) \quad [\frac{m}{s^2}]$. The gravity force is constant during the life-time of a snowflake.

- Buoyancy ($\vec{F}_{buoy}$): This is the force that makes wood float in water. The underlying principle of this force is known as Archimedes' law and states that a body submerged in a fluid will experience a buoyant force equal to the weight of the displaced fluid. It acts in the opposite direction of the gravitational force. We model it as:

  $$\vec{F}_{buoy} = -V_{snowflake} \cdot \rho_{air} \cdot \vec{g} \tag{3.4}$$

  As none of these terms change during the life-time of a snowflake this force is also constant. We assume ρ_{air} to be constant, although in reality it varies with altitude. The altitude changes are very small in our scenes so this can be neglected.

- Drag ($\vec{F}_{drag}$): Represents the drag force exerted on the snowflake by the surrounding air. This makes the snowflake follow the wind field. The drag force is modelled as:

$$
\begin{aligned}
F_{drag} &= \hat{u}_{fluid} \cdot \frac{|\vec{u}_{fluid}|^2 \cdot V_{snowflake} \cdot \rho_{snow} \cdot |\vec{g}|}{u_{max}^2} \\
\vec{u}_{fluid} &= \vec{u}_{wind} - \vec{u}_{snowflake}
\end{aligned}
\tag{3.5}
$$

  where $\vec{u}_{wind}$ and $\vec{u}_{snowflake}$ are the velocities of the wind and the snowflake respectively. u_{max} is the terminal velocity in $[\frac{m}{s}]$ of the snowflake in the gravitational direction. For dry snow this is in the interval [0.5;1.5], and for wet snow it is in the interval [1.0;2.0], as observed by Hanesch in [Han99]. When a snowflake is born it is assigned a u_{max} randomly from one of the intervals, according to temperature, that it will keep for the duration of its existence. For a complete motivation of how this force is modelled we refer to [AL04].

- Lift ($\vec{F}_{lift}$): The lift force is caused by the irregular shapes of snowflakes. It is perpendicular to the drag force and makes the snowflakes move in spiraling patterns. These circular movements are modelled as a uniform circular motion in the XZ-plane[4]. This is a rather crude approximation to a complex phenomena but it will suffice in this case. The influence of $\vec{F}_{lift}$ is modelled as a velocity, $\vec{u}_{circular}$, as follows:

$$
\begin{aligned}
\vec{u}_{circular} &= C_{vel} \cdot \omega \cdot R \cdot \vec{u}_{rotation} \\
\vec{u}_{rotation} &= (-sin(\omega t), 0, cos(\omega t)) \\
C_{vel} &= \frac{|\vec{u}_{wind} - \vec{u}_{snowflake}|}{|\vec{u}_{snowflake}|}
\end{aligned}
\tag{3.6}
$$

  where R and ω are individual snowflake properties controlling the radius and angular velocity of the circular motion respectively. When a snowflake is born R is chosen randomly from the interval [0.0;0.2] and ω is randomly chosen from the interval $[\frac{\pi}{4};\frac{\pi}{3}]$ and is measured in $[\frac{rad}{s}]$, motivated by experimental data in [AL04].

[4] assuming $\hat{F}_{gravity} = (0, \pm 1, 0)$

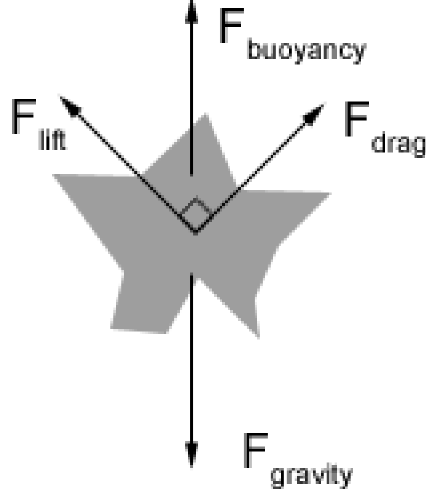


Figure 3.9: Forces acting on a snowflake or snow package as it moves in the wind field.

Figure 3.9 shows the relationships between the forces mentioned above. Since $\vec{F}_g$ and $\vec{F}_{buoy}$ always act in opposite directions we can simplify their contributions into a single force $\vec{F}_{down}$, defined as

$$\vec{F}_{down} = \vec{F}_g + \vec{F}_{buoy} = (\rho_{snow} - \rho_{air}) \cdot V_{snowflake} \cdot \vec{g} \tag{3.7}$$

The overall movement model is:

$$\begin{aligned}
\vec{x}(t+\Delta t) &= \vec{x}(t) + (\vec{u}(t)_{snowflake} + \vec{u}(t)_{circular}) \cdot \Delta t + 0.5 \cdot \vec{a} \cdot \Delta t^2 && (3.8)\\
\vec{a} &= \frac{3 \cdot (\vec{F}_{down} + \vec{F}_{drag})}{4 \cdot \pi \cdot r^3_{snowflake} \cdot \rho_{snow}} \\
\vec{u}(t+\Delta t)_{snowflake} &= \vec{u}(t)_{snowflake} + \vec{a} \cdot \Delta t && (3.9)\\
\theta(t+\Delta t) &= \theta(t) + \omega \cdot \Delta t && (3.10)
\end{aligned}$$

where $\vec{x}(t)$ is the snowflake position at time t and Δt is the time-step. As mentioned earlier snowflakes are rotated around their center of gravity in order to give them a more dynamic behaviour. The variable θ represents the current angle. Snow packages are not rotated since they are typically rendered as spheres, if at all. Simple forward Euler integration is used as higher accuracy has not been considered necessary.

The update-loop for a single snowflake or snow package consists of the following steps:

1. Find $\vec{u}_{wind}$ at the current position by using trilinear interpolation.
2. Use $\vec{u}_{wind}$ to calculate $\vec{F}_{drag}$.
3. Calculate $\vec{a}$ by combining $\vec{F}_{drag}$ and $\vec{F}_{down}$.

4. Calculate $\vec{u}_{circular}$.
5. Update snowflake position by using equation 3.8.
6. Update snowflake velocity by using equation 3.9
7. If the snow particle is a snowflake update θ according to equation 3.10.

3.4.2 Sliding

Slide packages reside on snow surfaces and are created from rest volumes of snow package collisions where the whole snow package volume could not be added to the snow buildup. Slide package movement is not dependent on wind, instead slide packages are moved in the direction of the gravity direction projected onto the tangent plane, as illustrated in figure 3.10.

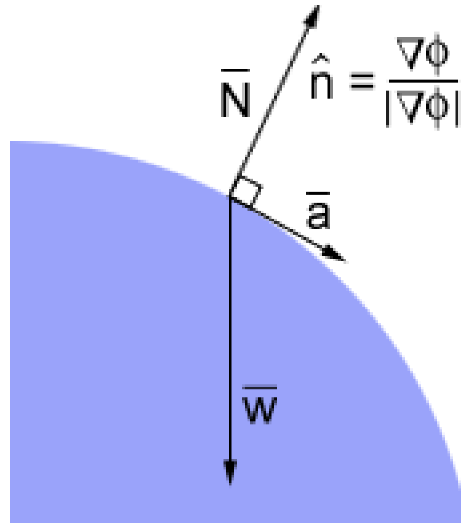


Figure 3.10: Forces involved when moving slide packages.

The forces are found as follows:

$$\vec{W} = m_{sp} \cdot \vec{g} \tag{3.11}$$

$$\vec{N} = (\hat{n} \bullet (-\vec{W})) \cdot \hat{n} \tag{3.12}$$

$$\vec{a} = \frac{\alpha \cdot (\vec{W} + \vec{N})}{m_{sp}} \tag{3.13}$$

where m_{sp} is the mass of the slide package and α is a scaling factor used to avoid high velocities, as this would require us to use smaller time-steps. Before a slide package is

moved we use the distance field to make the slide package "stick" to the positive side of the interface:

$$\vec{x} = \begin{cases} \vec{x} + \frac{1.1 \cdot \nabla\phi}{|\nabla\phi|} & \phi(\vec{x}) < 0 \\ \vec{x} - \frac{0.9 \cdot \nabla\phi}{|\nabla\phi|} & \phi(\vec{x}) > 0 \end{cases} \tag{3.14}$$

Where $\vec{x}$ is the position of the slide package. This process can be repeated an arbitrary number of times for higher precision but results have shown that one iteration is enough. The constants on front of the gradient terms are there to ensure that the slide particle ends up outside the interface.

The time integration for slide packages is very similar to that of snowflakes (using equivalent notation):

$$\vec{x}(t + \Delta t) = \vec{x}(t) + \vec{u}(t) \cdot \Delta t + 0.5 \cdot \vec{a} \cdot \Delta t^2 \tag{3.15}$$

$$\vec{u}(t + \Delta t) = \vec{u}(t) + \vec{a} \cdot \Delta t \tag{3.16}$$

If at some point $\vec{N} \bullet \vec{g} > 0$ the slide package is released from the surface and converted into a snow package of equal volume.

3.4.3 Collision Detection

Collision detection is important for two reasons. The first is that this the decisive factor in determining where snow should build up. The second is that it would look strange if snowflakes travelled through solid objects. Representing geometry as distance fields provides several advantages over explicit representations in terms of collision detection. First of all their is no need for complex acceleration structures such as octrees or kd-trees, as the distance field in itself contains enough information to determine whether a collision has occurred or not. The task of querying for a collision simplifies to checking the distance value at the location of a particle. Before a particle is assigned a new position the distance value at the new position is checked to see if it is valid, i.e. not inside a solid ($\phi < 0$). By solid we here mean the advectable snow level sets unionized with the corresponding static surface.

In practice the method described above must be further elaborated on. Since the interface is infinitely thin we cannot expect snow particles to collide with it. Furthermore, we do not have access to the distance field at all points in space since the DT-grid only stores this information in a narrow band around the interface. This means we have to limit the maximum distance travelled by snow particles for a single time-step. We are guaranteed to have valid gradient in the β-band and so we define this maximum distance to be $0.9 \cdot 2 \cdot \beta$. What this means is that the snow particle cannot move through the β-band without being

detected. If $|\phi(\vec{x})| < \beta$ and $\vec{u} \bullet \nabla\phi < 0$ the particle is inside the β-band and its velocity is pointing in the direction of the interface. This is recorded as a collision and equation 3.14 is used to "suck" the particle onto the interface.

Collision detection is not done for slide packages, as they per definition live on the interfaces of our snow level sets. Instead they are queried to find out if their new locations are valid collision points, as described in the next section. If so, a collision is stored and the possible rest-volume is further split into new, smaller slide packages.

Now that the largest distance a snow particle may travel per time-step has been established the largest possible time-step can be calculated. Assuming $\frac{\vec{u}_t}{|\vec{u}_t|} = \frac{\vec{a}_t}{|\vec{a}_t|}$, the worst-case scenario when the velocity and acceleration are pointing in the same direction, we find Δt to be used in equation 3.8 by solving the following second degree polynomial:

$$\overbrace{|\vec{x}(t+\Delta t) - \vec{x}(t)|}^{distance=d} = \vec{u}(t)\cdot\Delta t + 0.5\cdot\vec{a}(t)\cdot\Delta t^2 \tag{3.17}$$

$$d_{max} = 0.9\cdot 2\cdot\beta$$

$$\downarrow$$

$$\Delta t \le \begin{cases} -\frac{|\vec{u}(t)|}{|\vec{a}(t)|} + \sqrt{(\frac{|\vec{u}(t)|}{|\vec{a}(t)|})^2 + \frac{2\cdot d_{max}}{|\vec{a}(t)|}} & |\vec{a}(t)| > 0 \\ \frac{d_{max}}{|\vec{u}(t)|} & |\vec{a}(t)| = 0 \end{cases} \tag{3.18}$$

3.5 Accumulating Snow

Our means of accumulating snow is by propagating a level set with certain criteria. Snow package interaction with snow level sets will provide us with information about where snow is likely to accumulate. However, not all such interaction will result in snow buildup. Consider for instance the case of a snow package colliding with a vertical surface. Intuition tells us that smooth vertical surfaces cannot hold snow. We proceed to define valid locations for snow buildup and an explanation of how the propagation ensures stable buildup. Finally, a word on when to update the snow level sets. As this is a costly operation it should be avoided if not absolutely necessary.

3.5.1 Valid Collisions

The two main factors governing the stability of snow are temperature and the shapes of the ice crystals. As our simulation does not operate at resolutions of single ice crystals we will use experimental data for our stability model, which will be dependent on temperature alone. Previous work by Fearing [Fea00] uses an angle as a criteria for stability and we will adopt this approach. However, in our case it is more convenient to use a different angle, as shown in figure 3.11. Fearing [Fea00] defines $\beta = \arctan(\frac{h_1 - h_0}{b})$ as a measurement of

stability, h_1 and h_0 representing heights at neighboring triangles and b being the horizontal distance between them. Our criteria uses the normal to measure stability. By projecting the normal onto the plane described by the gravity direction and the point $\vec{x}$ we obtain the vector $\vec{n}_{xz}$. It follows that $\hat{n} \bullet \hat{n}_{xz} = cos(\theta)$ It can be shown that $\theta = \frac{\pi}{2} - \beta$.

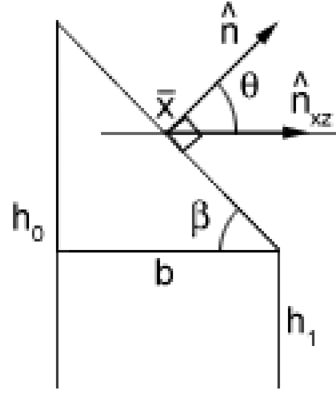


Figure 3.11: Previous work defines the stability angle as β. With our method it is more convenient to define this angle as θ.

Once a measurement of stability is setup a critical angle, θ_{AOR} is used to prevent snow from building up in unstable ways. This critical angle is dependent on temperature only and is motivated by experimental results from [Fea00] and converted to our angle, θ, instead of the angle β.

$$\theta_{AOR} = \begin{cases} \frac{\pi \cdot (50 - 30 \cdot |T+6|^{-0.25})}{180} & T \leq -8.5 \\ \frac{26.1418 \cdot \pi \cdot T}{8.5 \cdot 180} & T > -8.5 \end{cases} \tag{3.19}$$

When a collision occurs $\theta_{collision}$ is computed for that point. If $cos(\theta_{collision}) < cos(\theta_{AOR})$ and $\hat{n}_{collision} \bullet \vec{F}_{gravity} < 0$ the interface at that point can hold snow, if not the colliding snow package is split into slide packages. After this has been checked one final criteria needs to be verified. If there are no grid points within the radius of the snow package at the point of impact it would be meaningless to allow this collision as it would never contribute to our speed function (as described in the last section). For this reason the resolution of the snow level set limits the size that should be used for snow packages.

3.5.2 Stable Buildup

At this point we have described a robust method for moving snow particles, how collision detection is done and what a valid collision is. This means that when snow packages collide with snow level sets we have enough information to determine whether to add volume to this region and possibly split any rest-volume. We now move on to describe

how volume is added in such a way that it does not violate the stability criteria mentioned above, equation 3.19, for any part of the interface.

We start of by defining a collision function, $f_c(x)$, that is cheap to evaluate and offers some control of the shape of the curve. The collision function is defined as:

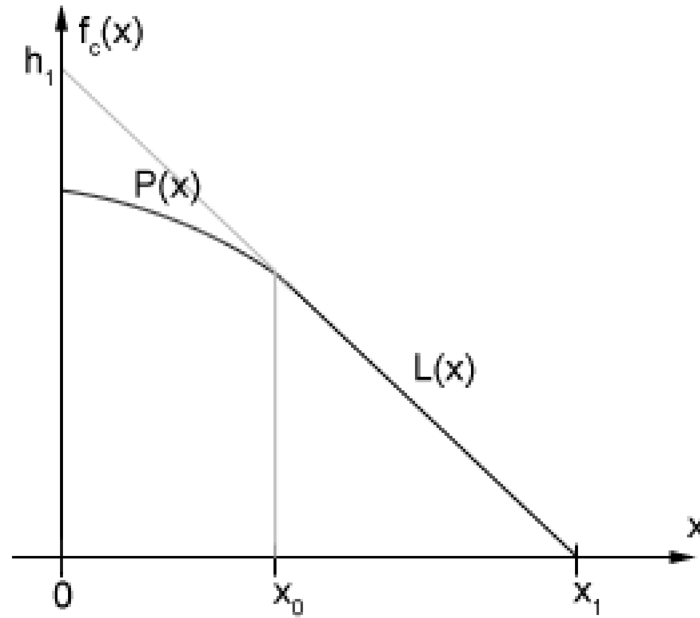
$$f_c(x) = \begin{cases} P(x) & 0 \leq x \leq x_0 \\ L(x) & x_0 < x \leq x_1 \\ 0 & x > x_1 \end{cases} \tag{3.20}$$

$$P(x) = -\left(\frac{h_1}{2 \cdot x_1^2 \cdot C_{smooth}}\right)^2 \cdot x + h_1 \cdot (1 - 0.5 \cdot C_{smooth}) \tag{3.21}$$

$$L(x) = -\frac{h_1}{x_1} \cdot x + h_1 \tag{3.22}$$

$$x_0 = C_{smooth} \cdot x_1$$

$$C_{smooth} \in [0; 1]$$



As x in this case is a distance it is always positive. The function $f_c(x)$ is monotonically decreasing and C^1 continuous in the domain $[0; x_1]$. C_{smooth} is used to control the smoothness of the curve, where a value of zero gives a cone-like shape as the contribution from $P(x)$ vanishes and a value of one means that $f_c(x)$ is a second degree polynomial for the whole domain. It will be useful to know the solid of revolution, $V_{collision}$, for $f_c(x)$ in order to calculate rest-volumes after the stable volumes have been deposited on the interface.

The solid of revolution is found by solving two integrals as shown next[5].

$$V_{collision}(h_1, x_1, C_{smooth}) = 2\pi \int_0^{x_0} P(x) \cdot x\,dx + 2\pi \int_{x_0}^{x_1} L(x) \cdot x\,dx = ... = \quad (3.23)$$
$$= \pi \cdot h_1 \cdot x_1^2 \cdot [C_{smooth}^2 \cdot (1 - \frac{3 \cdot C_{smooth}}{4}) + \frac{1}{3} \cdot (1 + C_{smooth}^2(2 \cdot C_{smooth} - 3))]$$

As expected we find that if C_{smooth} is set to zero $V_{collision}$ evaluates to the volume formula for a cone.

Our application uses individual C_{smooth} per object in the scene. There are many alternatives to this. For instance C_{smooth} might depend on curvature or the speed of the colliding snow package. Figure 3.12 illustrates how the collision function is used. In short the interface will "expand" in such a way that it adds the new volume to itself. The details surrounding this will be discussed further on.

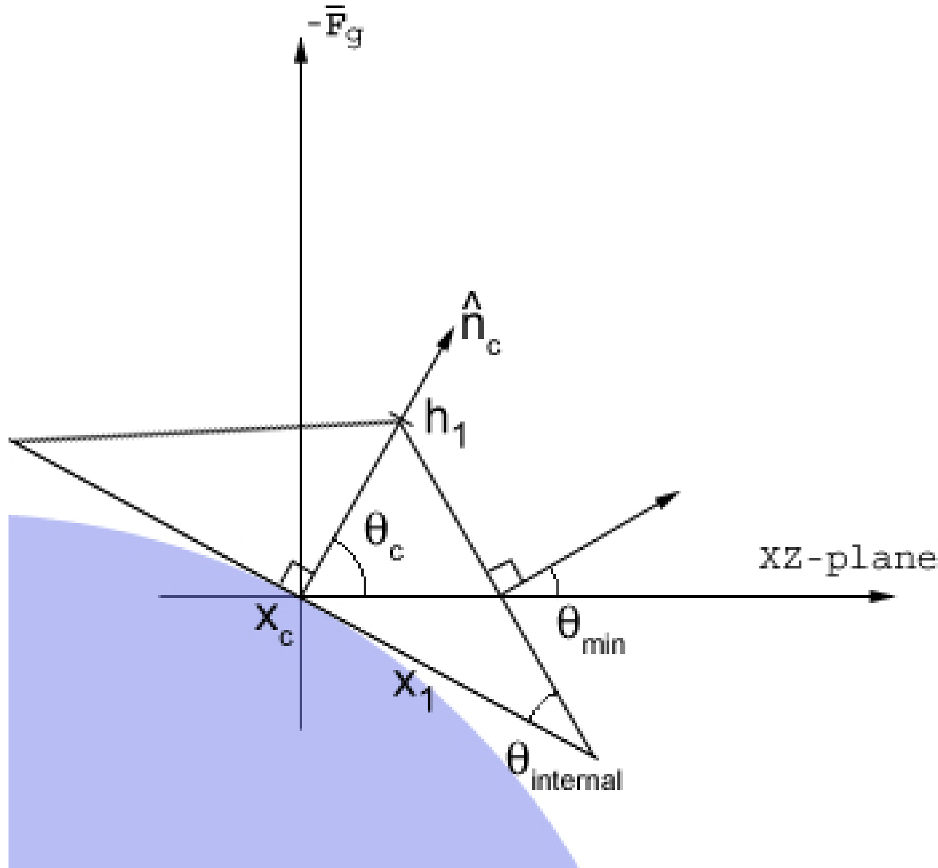


Figure 3.12: Angles and directions involved in guaranteeing stability when propagating the snow level set. For simplicity $C_{smooth} = 0$ is assumed here.

After $cos(\theta_c) < cos(\theta_{AOR})$ and $\hat{n}_c \bullet \vec{F}_{gravity} < 0$ have been verified h_1 must have a value that guarantees that $\theta_{min} > \theta_{AOR}$, assuring that the built up snow is stable. We set

[5]The math involved is simple but the expressions become rather large so they are not presented here.

$x_0 = r_{snowpackage}$ as we will need to fixate one of the variables h_1 and x_0 in order to solve the resulting equations. We also calculate a volume conserving height, h_{vc}, to avoid adding more volume than that carried by the colliding snow package. If we define h_{max} to be the maximum height that does not violate $\theta_{min} > \theta_{AOR}$, the height to be used is simply $min(h_{max}, h_{vc})$. Basic trigonometry leads us to the following expressions:

$$\begin{aligned}
h_{max} &= x_1 \cdot tan(\theta_c - \theta_{AOR}) \\
\frac{4 \cdot \pi \cdot r^3_{snowpackage}}{3} &= V_{collision}(h_{vc}, r_{snowpackage}, C_{smooth}) \\
&\Updownarrow \\
h_{vc} &= \frac{4 \cdot r_{snowpackage}}{3 \cdot [...]}
\end{aligned}$$

Where [...] comes from equation 3.23.

The collision domain is defined as all the level set grid points with distance less than or equal to x_1 from $\vec{x}_c$. So far we can determine whether the point of impact is a valid collision or not. From this we continue to find a maximum allowed height for our collision function that ensures that the added snow is stable. However, we cannot guarantee that every point in the collision domain is a valid collision point, and thus we can not guarantee stability yet. The collision function assumes that the vicinity of the collision point is flat and this is one of the key assumptions when calculating the maximum stable height. As snow package sizes approach those of real snowflakes this assumption becomes more and more valid.

Our solution to the problem of a entirely stable domain is simple and intuitive. We define an error metric, ε, to measure how far the collision domain is from being stable. Every level set grid point in the collision domain with $\nabla\phi \bullet \hat{n} > 0$ is assigned an ε-value, based on the following criteria:

$$\varepsilon = \begin{cases} |d_p| + \phi(\vec{x}) & d_p < 0 \\ 0 & d_p \geq 0 \end{cases} \tag{3.24}$$

Where d_p is the distance from the current grid point to the plane defined by the point of impact and $\nabla\phi$ at this point. The interpretation of ε is illustrated in figure 3.13. The maximum ε-value for the domain is found and the entire collision domain is moved in the opposite $\hat{n}$-direction by this amount. Since we know that the shape of the snow to be added will be stable under the assumption that the underlying geometry is flat, the same must be true if the collision plane is entirely beneath the snow surface. If ε is greater than $f_c(0)$ the collision is cancelled and the whole snow package volume is split into slide packages instead.

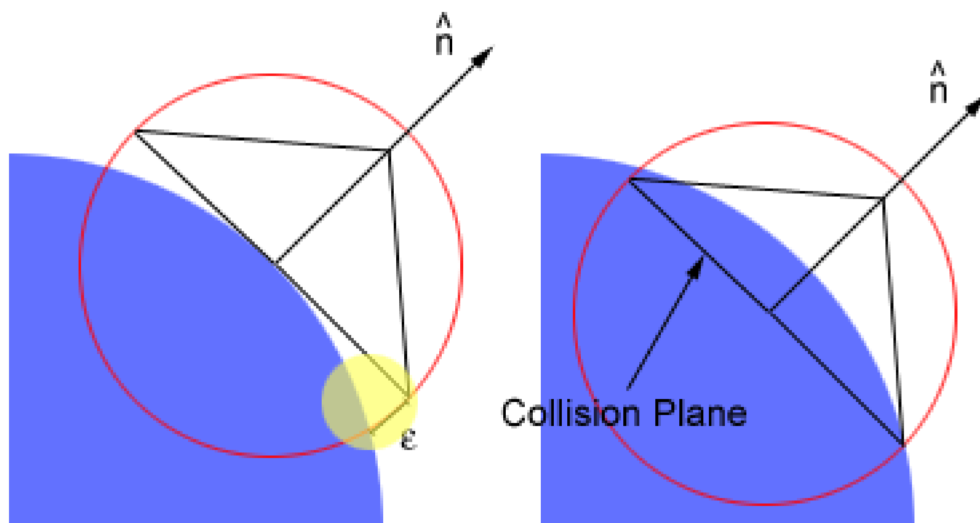


Figure 3.13: Left: Parts of the interface (marked with yellow) within the collision domain (red circle) are not valid collision points. Right: The collision domain has been moved so that the collision plane is lying totally beneath or on the surface.

After the collision domain has been moved and $\varepsilon < f_c(0)$ the collision is stored in the so-called *collision buffer* until next time the snow level set is updated. Slide packages are spawned if the rest-volume is large enough to make any slide packages larger than a user-defined size. Slide packages are spawned at a random location on the collision plane with distance x_1 to the point of impact. The rest-volume is found simply as we know how much volume was added and the volume of the snow package that collided with the snow level set.

Figure 3.14 shows how snow packages and slide packages work. The green spheres represent slide packages and snow packages are rendered as white spheres. Notice that slide packages are often smaller than snow packages. The reason for this is that slide packages may represent rest-volumes of snow that could not be deposited by snow packages.

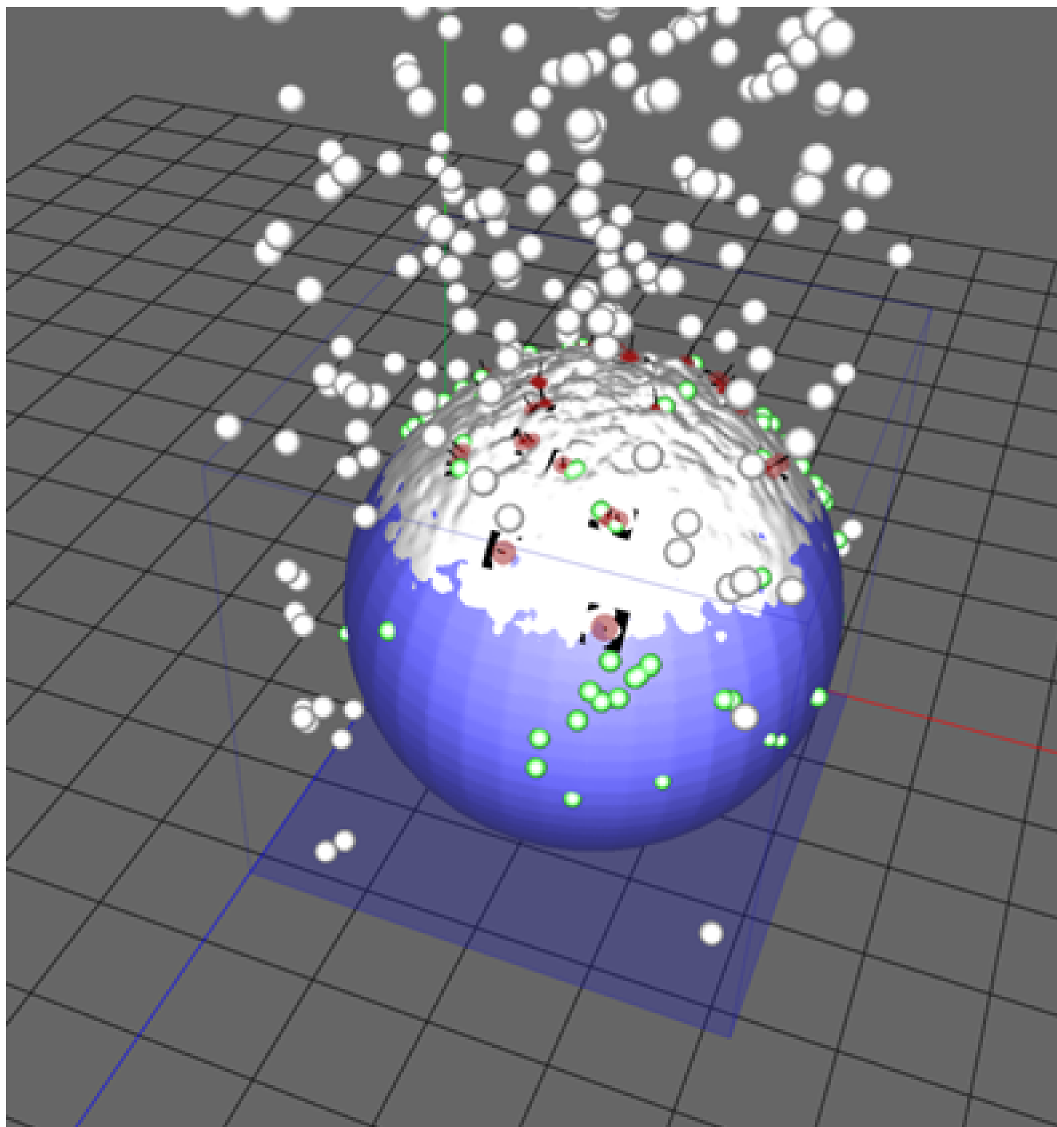


Figure 3.14: Snow packages rendered as white spheres being advected in the wind field. Green spheres show slide packages living on the surface of the accumulated snow. Collisions already stored in the collision buffer shown as red spheres.

3.5.3 Updating the Snow Level Set

It would be very inefficient to update the snow level set after every collision, since the update operation involves reinitializing the snow level set. Instead collisions are stored in a buffer until the next update. The time-complexity for this operation is $O(M \cdot N)$, where M is the number of grid points in the β-band and N is the number of collisions in the buffer. It is not optimal to loop over all collisions for every grid point so an acceleration structure would help here. However the major bottle-neck is the reinitialization that is performed after the propagation is finished. We have chosen to update the snow level set every n^{th} valid collision, where n is user defined. There is a trade-off between updating often, which means that each update will add only a little snow, and updating more

conservatively, which means the collision buffer will contain more snow volume to be added. In the case were there are many collisions in the buffer some of them are likely to over-lap. A smaller volume inside a larger one will have no visible effect after the update, and furthermore, since collision detection and the wind field are based on the snow level set, updating this more often will make the simulation more accurate, as volume will then be added to already accumulated snow, instead of becoming an over-lapping collision in the buffer.

Accumulated snow is added to the scene by propagating the snow level set outward. The speed function used is dependent on the collisions in the collision buffer. When setting the speed for each grid point inside the β-band the following algorithm is used:

1. For every collision domain containing the grid point repeat step 2-3.
2. Project the grid point's world coordinates onto the collision plane and calculate the distance, d_c, from this point to the center of the collision domain (the point of impact). Also remember the distance from the grid point to the collision plane, d_p while projecting.
3. Estimate the distance to the interface from the projected grid point position, d_s, as $max(d_p - \phi, 0)$.
4. The speed is set to the largest value of $(f_c(d_c) - ds)$ for all the collisions checked.

The goal is to propagate the snow interface so that it conforms to the shapes of f_c for the collisions in the collision buffer, as illustrated in figure 3.15. The propagation of the snow level set is carried out in several iterations, were the number of iterations, n, is found as $n = \lceil \frac{max(f_c(0))}{dx} \rceil$. We divide the "highest" collision by the grid spacing of the snow level set, since the rebuilding of the narrow-band requires that the interface does not move more than one voxel. Rebuilding of the narrow-band must be done after each iteration. Reinitialization is done only once, after all propagation and rebuilding is done, and the full snow level set, the union between accumulated snow and the underlying solid, is reinitialized. Reinitialization uses WENO schemes to compute partial derivatives as good accuracy is required in this case. After each level set update the collision buffer is cleared.

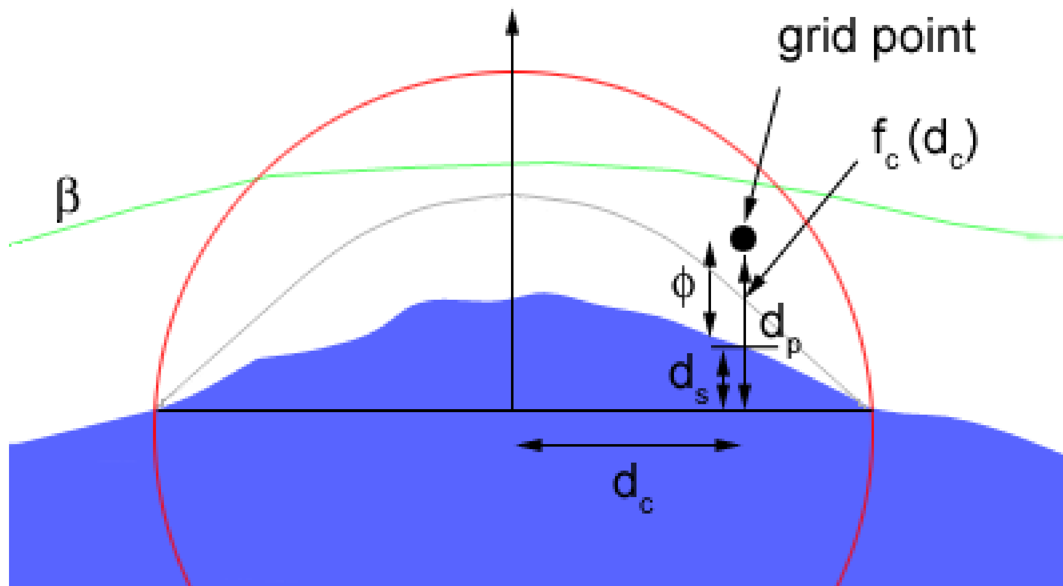


Figure 3.15: The interface is propagated so that it conforms to the shape of f_c, which is guaranteed to be adhere to the stability criteria setup earlier.

Chapter 4

Results

In this chapter we will present some results that our method has produced. All the simulations were run on a 2 GHz machine with 512 MB RAM. Unfortunately at this point we have not been able to go higher than 256^3 effective level set resolution due to some missing optimizations in the implementation regarding setting the speed function and reinitializing the snow level set. The simulations simply take too long. However, the results show that our method does what we set out to do. Several suggestions for optimization are presented in the Discussion/Future Work chapter.

4.1 Varying Temperature

The level sets for the spheres in figure 4.1 are stored in DT-grids with an effective resolution of 128^3. The simulation was done to illustrate how temperature effects snow buildup. For the left sphere the lower temperature used allows snow to buildup at steeper angles than for the right sphere, resulting in very different accumulation profiles.

The radius of the sphere (same in both cases) is 1 meter. Snow package radius was set to 5 cm and slide packages with a minimum radius of 3 cm were allowed. The smoothness, C_{smooth} was set to 0.8 in both simulations. The right sphere took three hours to simulate and has reached the point where no more snow can buildup. The left sphere took five hours to simulate and there is possibly room for a little more snow to buildup at the top. Since it carries a larger volume of snow more updates were required for the left sphere. No wind was used in this simulation and images were ray-traced with a commercial software package.

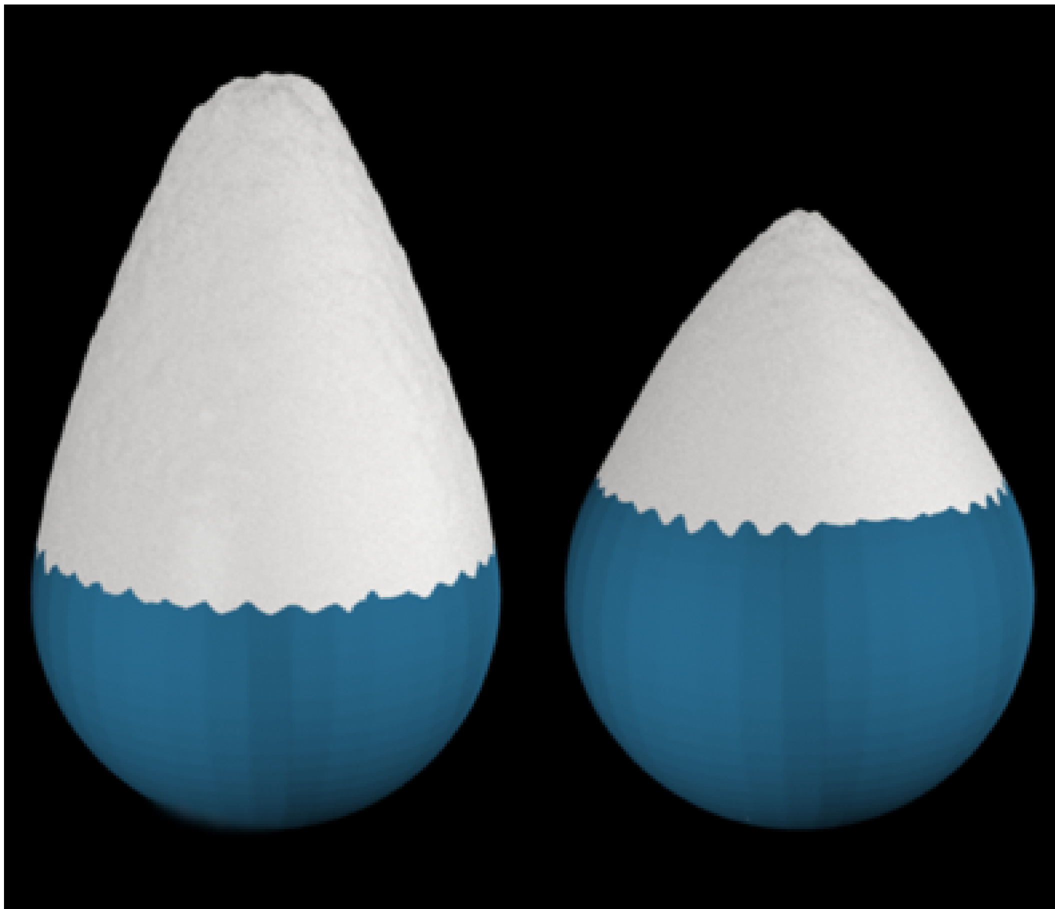


Figure 4.1: Snow buildup on a sphere. The settings were identical apart from temperature, which was -2℃ for the left sphere and -8℃ for the right sphere.

4.2 Wind-driven Accumulation

The following discussion is based on figures 4.5-4.7. For testing wind-driven accumulation we have chosen to use a setup presented in [FO02], also used by [MMA+05]. The houses and ground are a single level set with effective resolution of 256^3. Due to lack of information regarding the geometry and configurations used in [FO02] and [MMA+05] we have had to do our best to try to reproduce their scenes. However, the comparisons are still fair since characteristics of the different methods are still measurable. We note that the accumulation patterns in the figures are quite similar, differences possibly depending on different wind-speeds and different transportation models. The accumulated snow in figure 4.5 is smooth and the accumulation patterns are plausible. However, as mentioned earlier, this method only works for geometry that is representable as a height-field. More complicated architecture that over-laps itself in the gravitational direction is not possible with their model. Figure 4.6 shows the same scene simulated with the model presented in [MMA+05]. Here the snow exhibits sharp features uncharacteristic of granular materials. However, the accumulation pattern is plausible and generalizes to other polygon meshes.

Our method, figure 4.7, produces smoother surfaces but also makes the need for a redistribution phase apparent, since the snow that has accumulated on top of the houses (boxes) should be redistributed by the wind field. In our simulation wind speed on the inward facing boundaries of the fluid solver domain were set to 5 $\frac{m}{s}$ (wind directions obvious from the images) and the snow temperature was set to -3°C . Fluid solver resolution was 64^3 and the same snow package and slide package settings were used: 10 cm radius for snow packages, 3 cm minimum radius for slide packages. For this type of snow accumulation it is important to allow for a lot of slide packages since these will accumulate snow around the houses as snow packages collide with the walls. Our scene was generated over-night and the results from [FO02] took three hours to produce (no information available from [MMA+05]).

Along with comparing our results with previous work we have also investigated the results of varying wind speeds. The right-hand side of figures 4.2-4.4 show snow buildup driven by a wind field with boundaries set to 5 $\frac{m}{s}$, and the left-hand side show virtually the same scene where the wind at the boundaries was set to 0.5 $\frac{m}{s}$. Both scenes have an effective resolution of 256^3. It is clear that the wind effect is hard to identify in the right-hand version. However, it is very noticeable in the left-hand version. We also note that the snow surface in the left-hand version is a lot smoother than that in the right-hand version. A likely explanation for this is that more of the buildup in the left-hand version was done by slide packages, which are typically a lot smaller than snow packages.

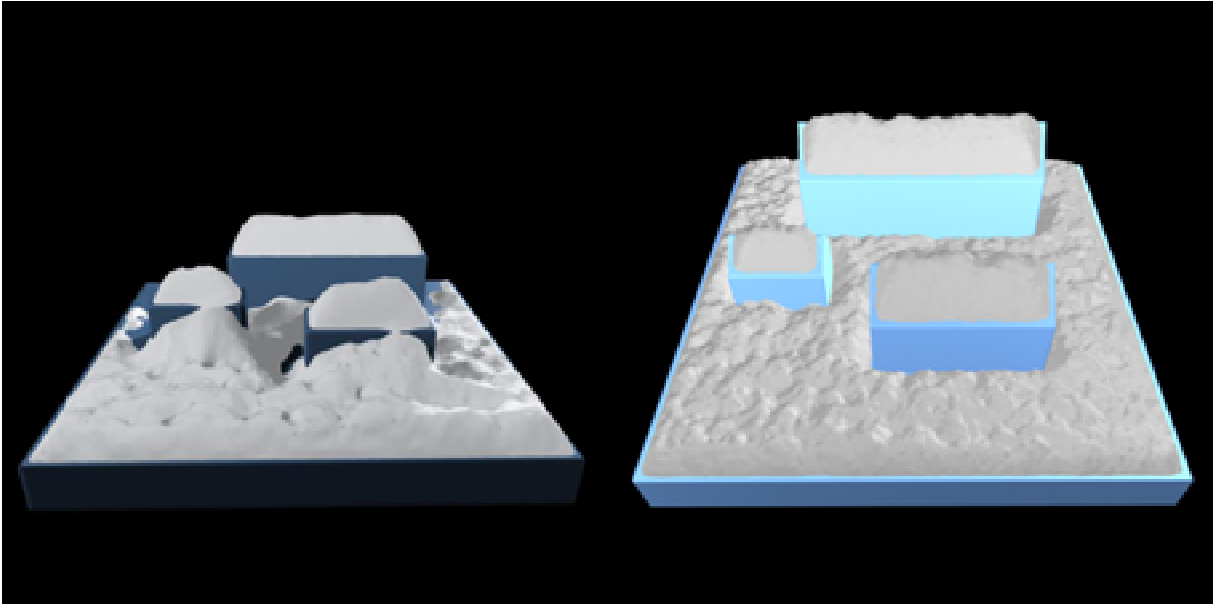


Figure 4.2: Front view of house scene with our method for two different wind speeds with identical direction. Left: wind speed 5 $\frac{m}{s}$. Right: wind speed 0.5 $\frac{m}{s}$.

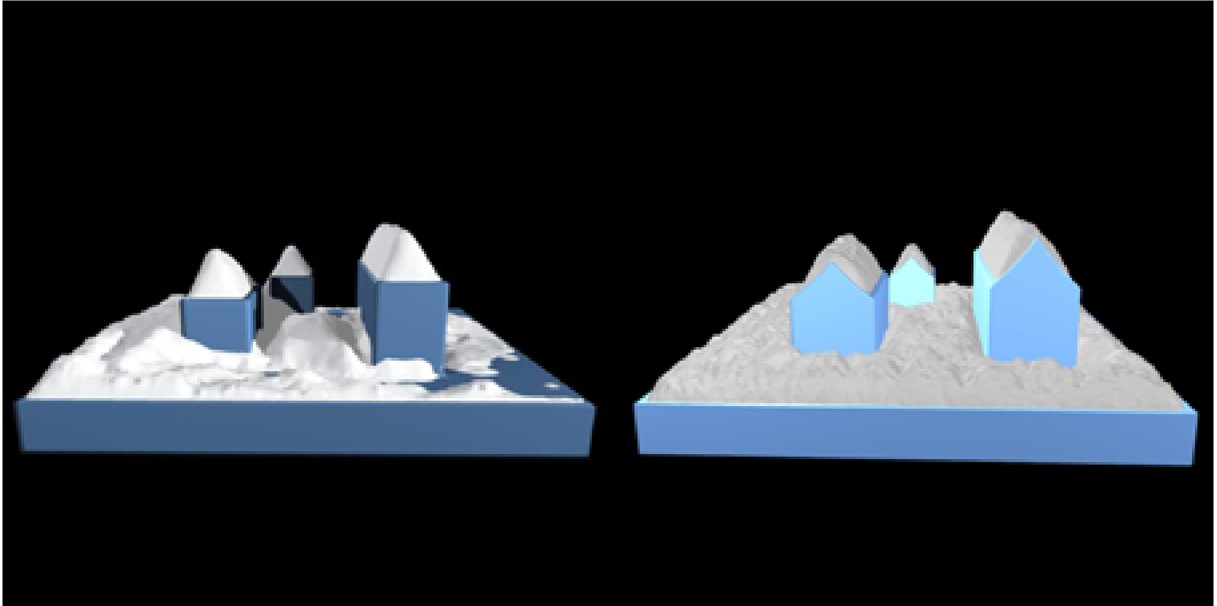


Figure 4.3: Side view of house scene for two different wind speeds, with identical direction. Left: wind speed 5 $\frac{m}{s}$. Right: wind speed 0.5 $\frac{m}{s}$. Notice the accumulation of snow in-between houses and behind the right-most house. The buildup behind the house is caused by the house sheltering the area from wind, making snow packages fall vertically to the ground instead of being carried by the wind out of the scene for this region.

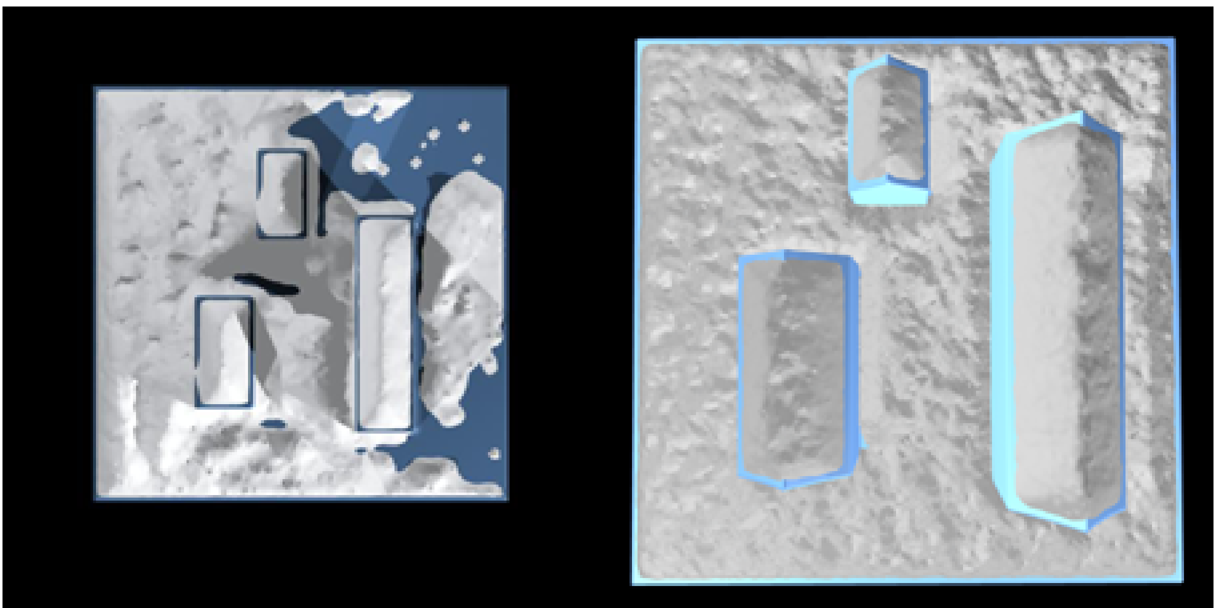


Figure 4.4: Front view of house scene for two different wind speeds, with identical direction. Left: wind speed 5 $\frac{m}{s}$. Right: wind speed 0.5 $\frac{m}{s}$. A better overview of the accumulated snow. Notice that for small wind speeds wind-driven accumulation patterns are practically unnoticeable.

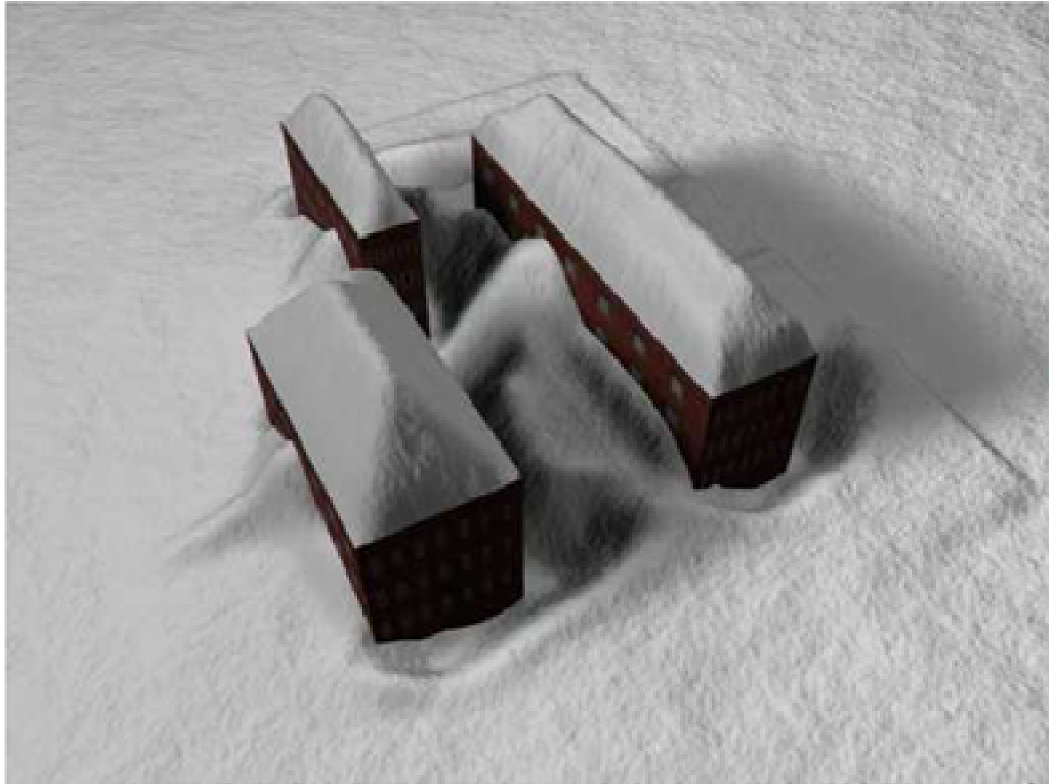


Figure 4.5: Feldman scene as presented in [FO02]. Wind-effects are clearly visible and the snow surface is smooth and the results look good. However, the method in [FO02] stores built-up snow in a height field so it can only handle very simple geometry.

Figure 4.6: Feldman scene as presented in [MMA+05]. Snow-cover is not very smooth. Sharp edges, uncharacteristic of snow, reveal the underlying triangulation.

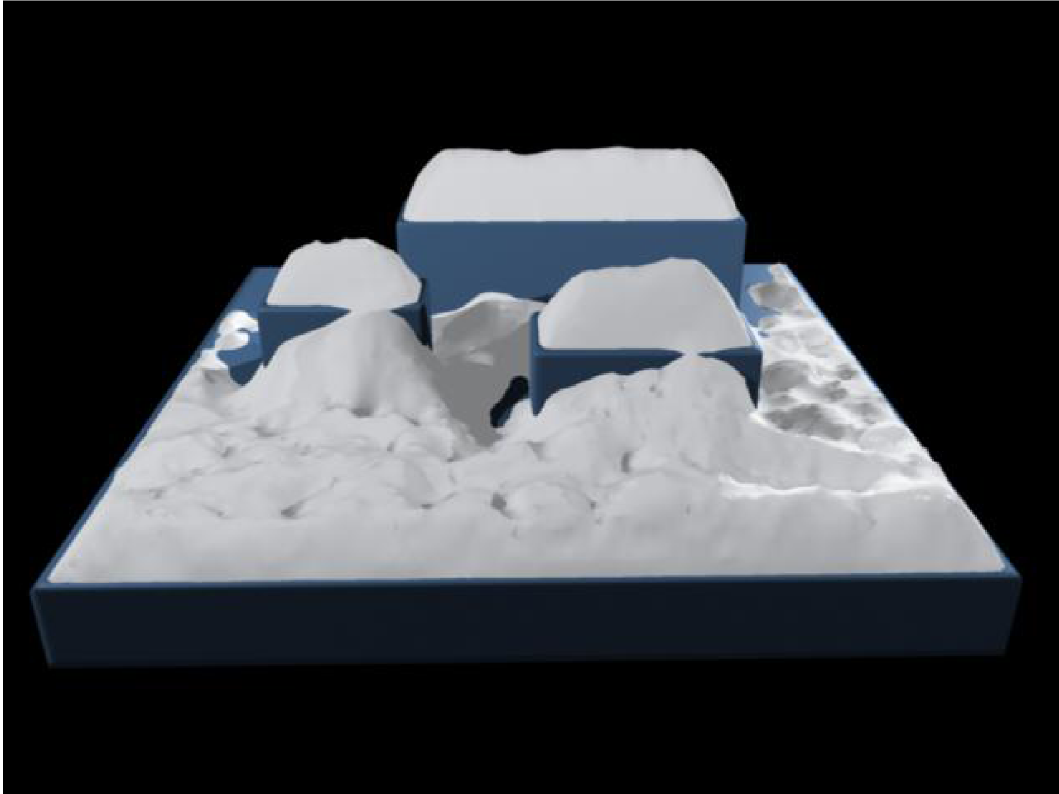


Figure 4.7: Our version of the setup introduced in [FO02]. Our houses are represented as simple boxes as this will not have a noticeable impact on the wind flow around the houses at ground level. Our three compared scenes differ more than one would expect but this is because we have not been able to find the exact configurations of the two other methods we compare with. In our case the wind seems to have been stronger than in previous work and areas around the house furthest away are not snow-covered due to this. However, this shows realistic behaviour of our transport model and should not be considered an artifact.

Returning to the right-hand version in figures 4.2-4.4 we will present the changes in the wind field caused by the built up snow. This is illustrated by using streamlines and is presented in figures 4.8 and 4.9. The effects are most noticeable around the roof tops, where the wind has had to make a steeper ascent to flow past built up snow. Wind velocity in this case was 0.5 $\frac{m}{s}$ from right to left in the image.

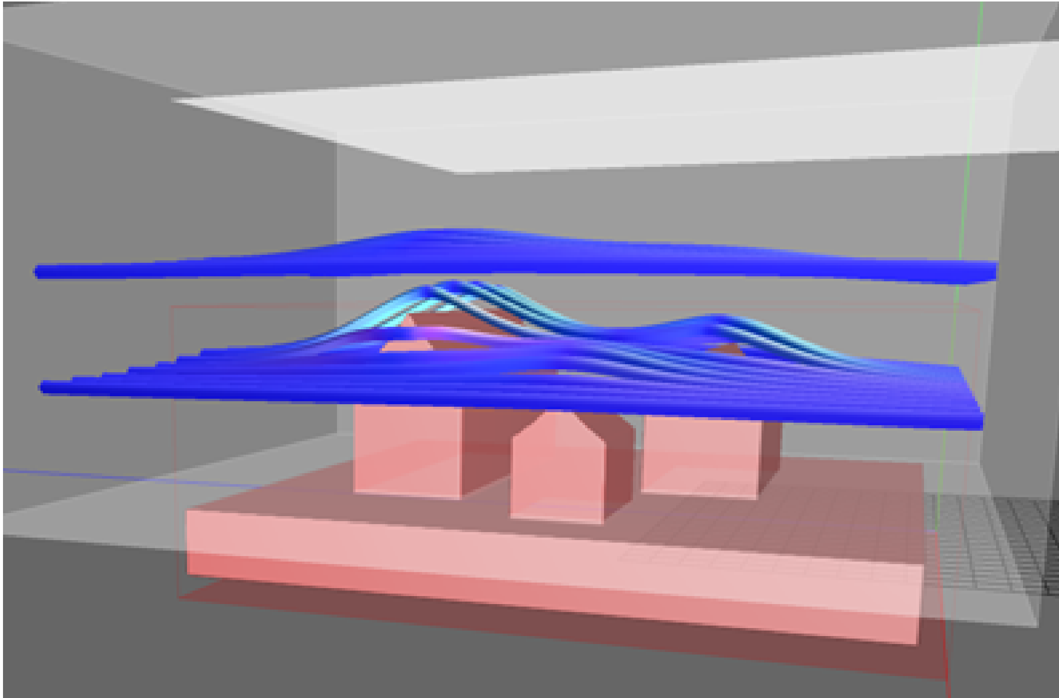


Figure 4.8: Wind field in original scene.

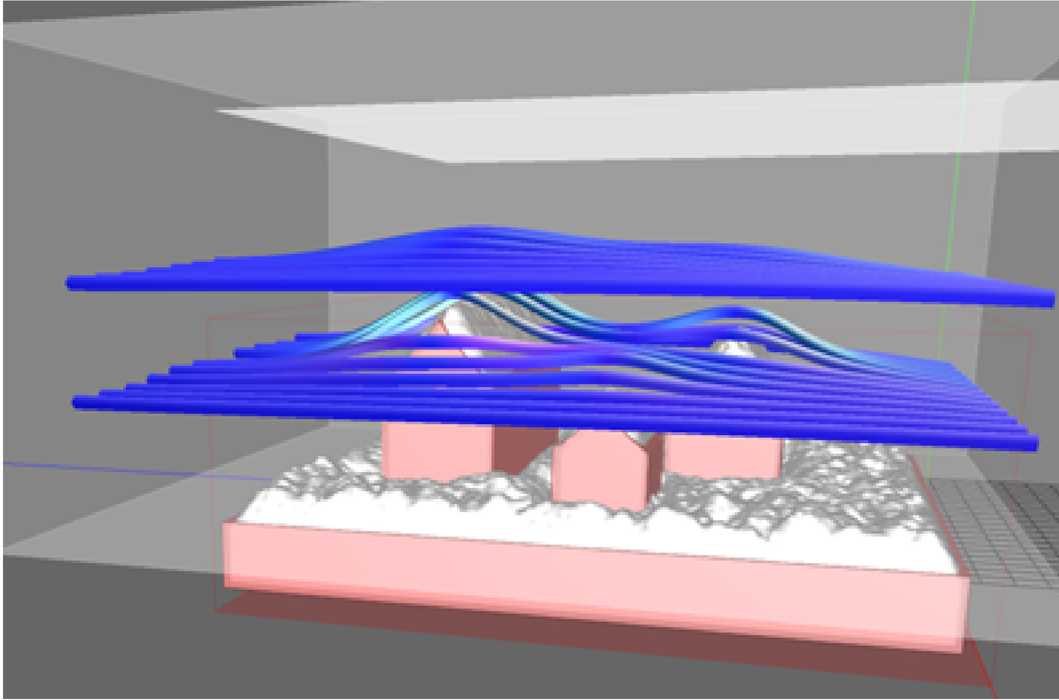


Figure 4.9: Wind field after snow accumulation.

4.3 High-Resolution Boundaries

Figure 4.10 shows the result of using a fairly high-resolution model. The buddha statue and the ground are 256^3 effective resolution. This simulation was run over-night and no wind was used. In general the effects of wind are more interesting for scenes with several objects. For this scene heavy use of slide packages is crucial as the buddha over-laps itself along the gravitational direction in many places. Our method handles this well and snow accumulates in places that we would expect it to. The statue is approximately 0.2 meters high and snow package radii were 2 cm and slide package minimum radii 0.5 cm. The small sizes allow snow to build up on small, thin surfaces like the creasing in the buddha's clothing.

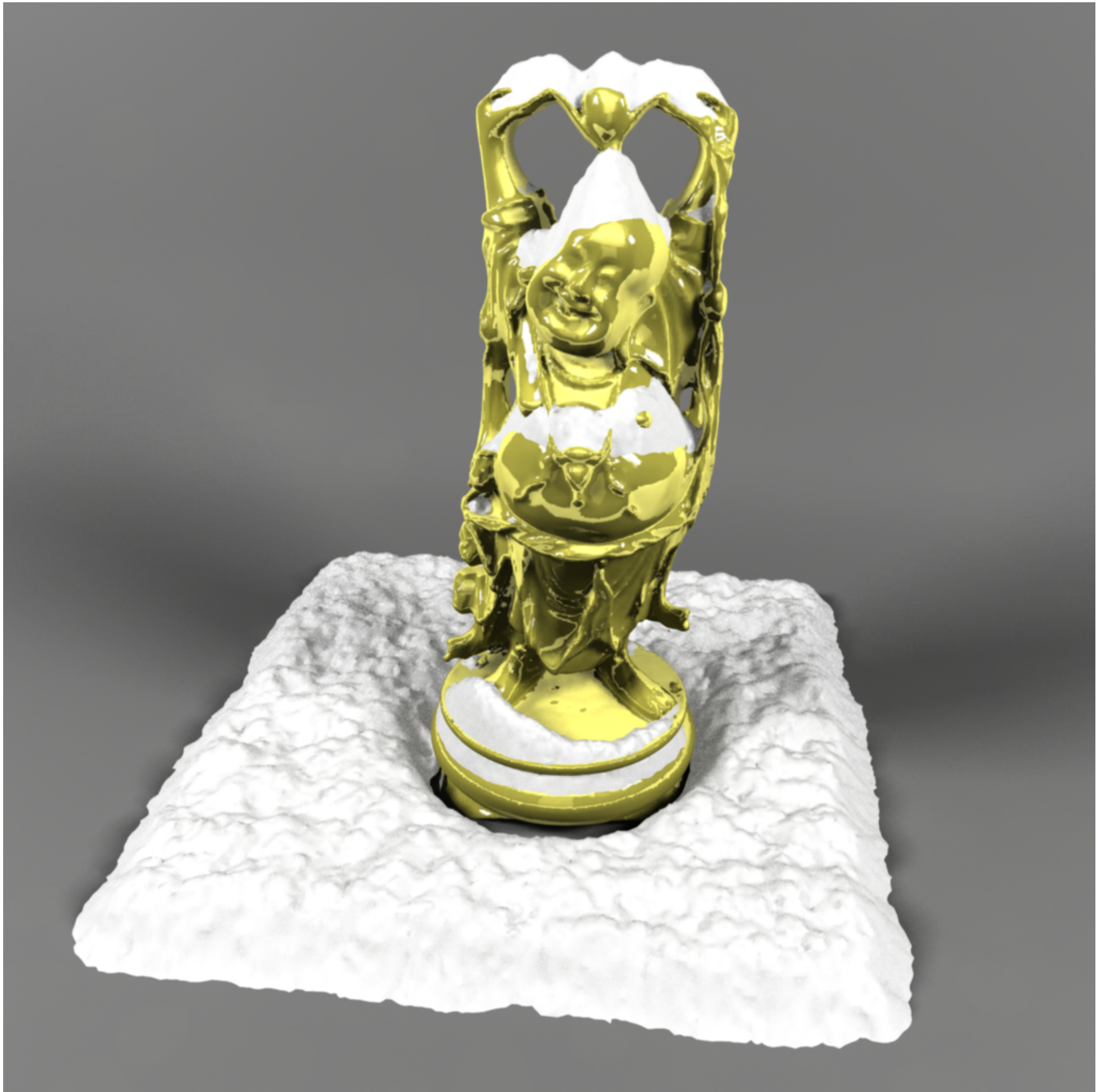


Figure 4.10: Snow on a buddha statue with effective resolution of 256^3.

4.4 Adding Snowflakes

We took one of the house scenes and added snowflakes to it as a post-processing step. This means that no level set updating was done but instead time was spent on updating the wind field each frame. Swirling and turbulence are difficult phenomena to capture in still images but we present a few images from this simulation. The number of snowflakes was limited to a maximum of 512. The reason being that exporting data for each snowflakes position for each frame to the rendering software is a slow process. Actually simulating the snowflake movement took approximately 45 minutes, while rendering an image sequence of 300 images took approximately 24 hours. A solution to this, further discussed in the next chapter, would be to make our program a plug-in to the rendering software, effectively removing the need to import and export data. The images are shown in figures 4.11-4.16.

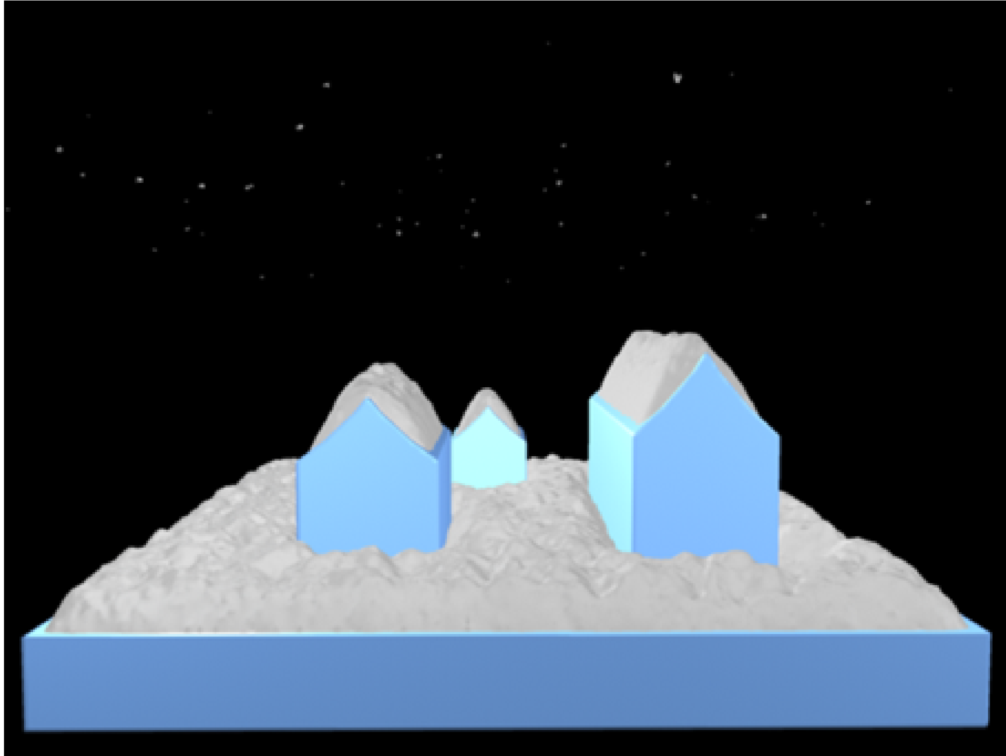


Figure 4.11: Frame 50 from animation consisting of 300 frames.

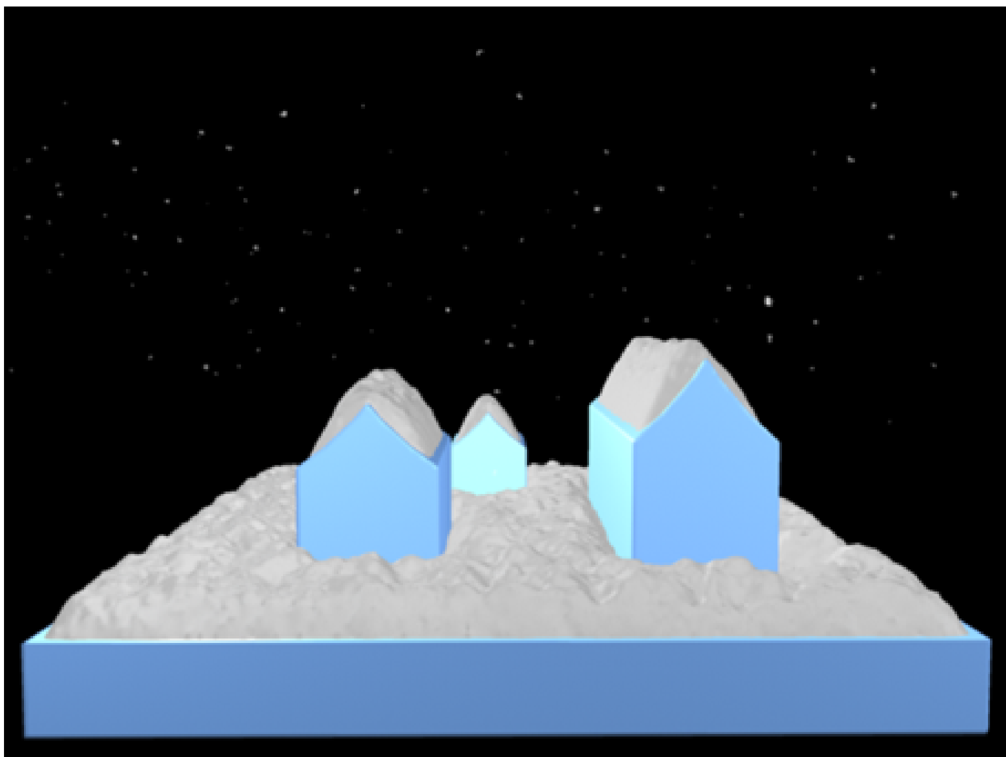


Figure 4.12: Frame 100 from animation consisting of 300 frames.

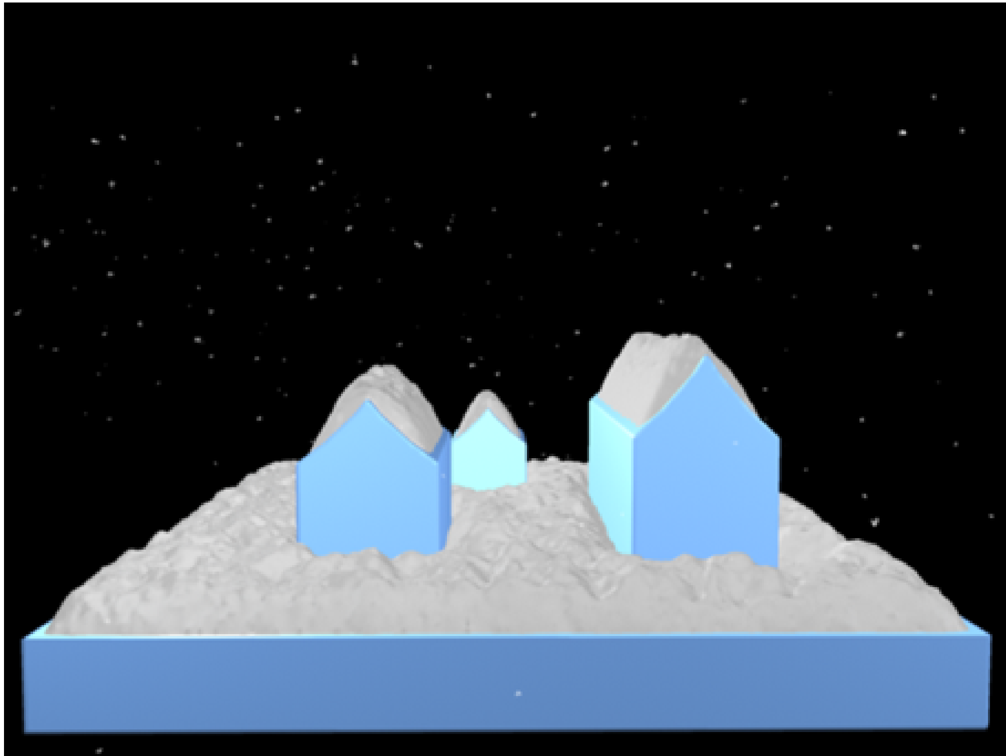


Figure 4.13: Frame 150 from animation consisting of 300 frames.

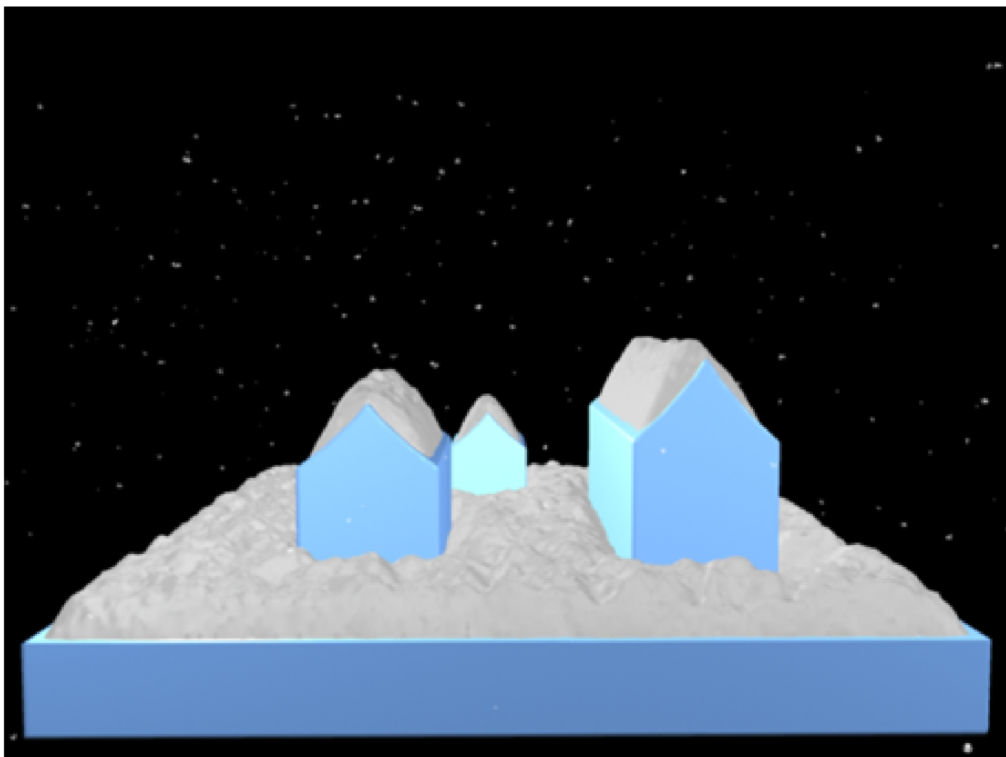


Figure 4.14: Frame 200 from animation consisting of 300 frames.

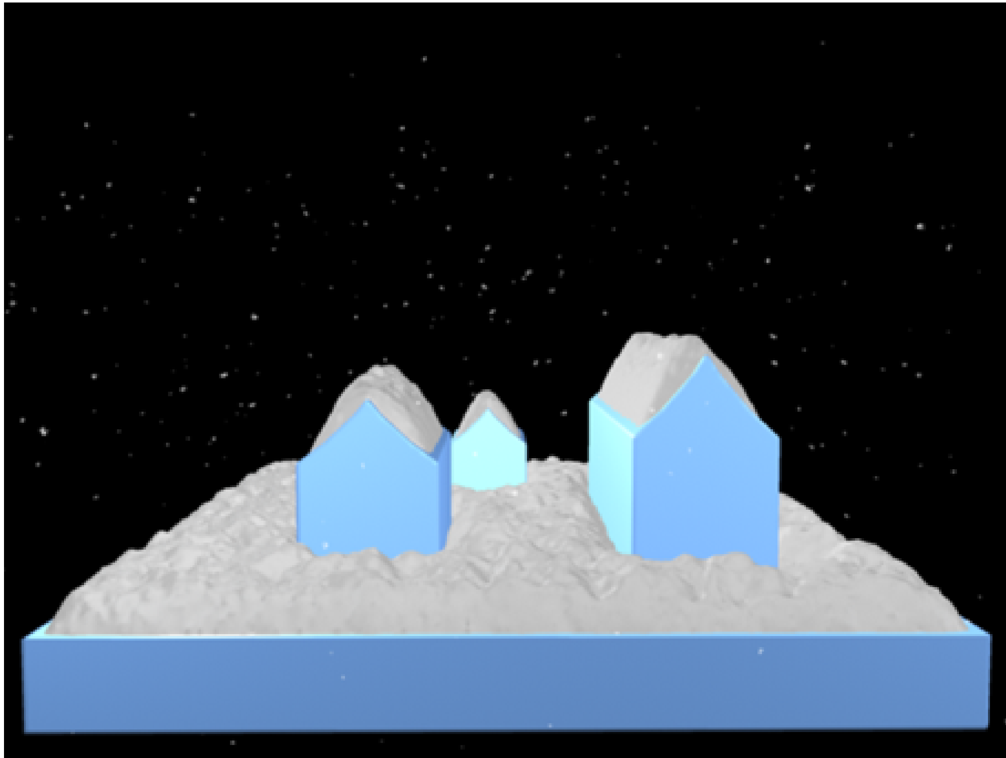


Figure 4.15: Frame 250 from animation consisting of 300 frames.

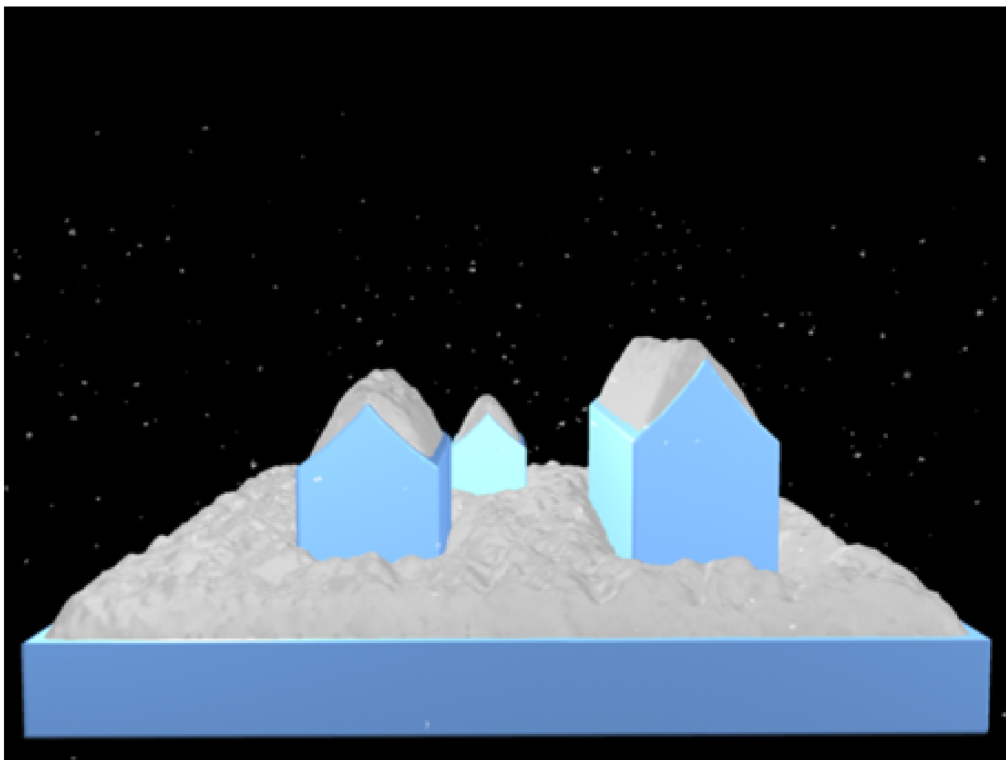


Figure 4.16: Frame 300 from animation consisting of 300 frames.

Chapter 5

Discussion / Future Work

After examining our results and comparing them to the goals setup initially we find that most of the goals have been achieved. Our model handles complex geometry without introducing special cases and our results compare well to previous work. However, there is always room for improvements and this chapter will present ideas on how to solve existing problems and how to extend the current model.

The biggest problem is the fact that simulations for high-resolution ($512+^3$) level sets take too long. We were aiming for over-night simulations (approx. 10-12 hours) but as level set resolutions increase this time limit becomes hard to meet. The main cause of this is that the time it takes to reinitialize the snow level set increases dramatically with the number of grid points in the DT-grid. In fact, a lot of the reinitialization is done for areas that have not been propagated at all. The current implementation reinitializes a level set consisting of the snow and the underlying solid. An idea to speed this up would be to extract only the built up snow by using CSG difference and then reinitializing only this part. The newly reinitialized built up snow could then be added back to the solid by a CSG union operation, for use with collision detection and solver grid cell classification. We could also speed up the propagation methods, which at the moment iterate over the whole collision buffer for every grid point in the level set. A much more efficient approach would be to iterate over the collisions in the buffer and write the speed values to the grid points inside the collision domain.

In most of the scenes produced for this thesis a scene has consisted of a single object. Sometimes it is more practical to separate objects from each other and store them as individual level sets. The buddha-scene is actually two separate objects, each with its own snow level set representing snow buildup on that particular object. In this case the statue is one object and the ground another. The reason for doing this split is that it can save time since we can choose to update the snow level sets separately. The ground is likely to accumulate snow faster than the statue since the valid collision area of the plane is much larger than the same area for the statue. Hence, the ground snow level set will have to be updated more frequently and avoiding to include all the statue's grid points in this operation saves a reasonable amount of time. It also allows us to use different smooth

settings for the two objects. However, there is a drawback with this method. Consider the snow that builds up on the ground. At some point this snow should start to interact with the foot of the statue, but it will not do so. Hence, if we want to split a scene into separate objects, which will be absolutely necessary for larger scenes, some kind of scene description will be necessary, telling the program which objects' snow level sets could possibly interact with each other. Recall the buddha-scene. Add a sphere hovering above the ground some distance from the statue. If the snow on the ground rose enough it might collide with the sphere, but the snow on the sphere will never interact with the snow on the statue, and vice versa. If such connections could be established it would be possible to use the union of all connected objects each time snow was accumulated on one of the connected objects.

As well as having multiple object in a scene one could also imagine having multiple fluid solver domains within the same scene. These could be over-lapping or free floating. The point would be that for large scenes where the interesting details are far apart it would be very inefficient to have one large fluid domain covering all these details. Since the resolution is constant very large empty areas where nothing happens would have to be solved for, leading to slower simulations. If we instead create separate fluid domains around the objects of interest and use the global wind direction in-between this would be much more efficient. Furthermore, we could allow fluid domains to over-lap each other where fluid domains with in-flow cells positioned inside other fluid domains would simple have these velocities interpolated from the other fluid domains instead of using the global wind direction.

Continuing the discussion on improving wind fields it should be noted that as level set resolution increases the width of the β-band in world space decreases. This means that in order to guarantee that fluid solver boundary grid cells are inside the β-band the immediate options are to either increase the tube-width of the level set, resulting in a larger memory foot-print and more expensive level set operations, or increasing the resolution of the wind field. High-resolution level sets are capable of representing very fine details. However, these details do not have a huge impact on the wind field and it is hard to motivate spending more time on solving for the wind field just because finer details are present, since they will not make much difference. Widening the tube is not a good option either since the whole idea of the DT-grid is to store the distance function in a narrow band kept as tight as possible. A possible solution would be to use a down-sampled version of the high-resolution model as fluid solver grid cell classification. This down-sampled version would have to be updated every time the high-resolution level set is updated but this operation is relatively fast, especially when considering the alternatives.

Even though the built up snow is guaranteed to be stable this assumes that wind does not redistribute snow. In this sense advecting snow package in a wind field alone is not a totally realistic approach to simulating the accumulation of wind-driven snow. A level

set is a robust representation for an eroding surface, since it is insensitive to changes in topology. Our method for calculating the wind field is also capable of adapting to changes within the fluid domain, which means that a simple approach to redistributing built up snow could use wind magnitude and direction to convert parts of built up snow into snow packages. The volume corresponding to the newly formed snow packages would then be eroded from the location at which they were spawned. This would make our model much more flexible since it would mean that we could change the wind direction during a simulation and guarantee that not only the following snow packages deposit snow in plausible locations, but also that already accumulated snow adapts to the new wind direction. A more elaborate solution compared to creating new snow packages by wind erosion is the idea of advecting snow as a density in the wind field. This would realistically mimic the phenomena that can be seen around snow covered roof-tops in strong winds. The roofs of the houses force the wind to change direction creating strong gusts of wind. When the wind carries the snow of the roofs the appearance is very smoke-like. Either the densities would interact with the snow in the scene or they would simply be carried out of the scene by the wind. The latter approach is easier to implement but would in fact not deal with redistribution. We strongly believe that redistribution is important. Even though our method cannot handle it yet we believe that implicit surfaces would make implementing redistribution much easier compared to polygon meshes (height field are also known to work well with erosion but are extremely limiting by nature).

Using large snow packages will add more volume per collision and level set update. However, as we do not want the underlying shapes of individual collisions to be visible in the final results this approach cannot always be used. Another approach is to use smaller snow packages but less smoothing, giving individual collision shapes a more cone-like appearance, but sharp edges are not a characteristic of snow, or any other granular material for that matter. The use of small snow packages leads to a lot of collisions and hence a lot of level set updates. An idea would be to use larger packages initially and the "sprinkle" the scene with small package before extracting the final result. Another idea would be to allow the smoothness parameter to vary with time, making collision shapes increasingly less cone-like as the simulation progresses. At the moment every solid object in a scene has its own smoothness parameter, but it would be possible to take this one step further and say that every point on the interface has its own smoothness parameter, somehow dependent on the curvature at that point. Low curvature would mean that cone-like shapes would tend to stick out more and so these areas require more smoothing.

A nice effect that Fearing [Fea00] uses is *flake dusting*. This is used to represent extremely thin layers of snow or snow that sticks to horizontal surfaces with just the right temperature. The technique he presents uses semi-transparent textures on certain polygons to give the appearance that these are in fact holding very thin layers of snow. The need for such a technique arises at the boundaries between snow covered areas and uncovered

areas. Our model produces quite distinct boundaries between such areas which in some cases looks bad, as seen in figure 4.1. However we have found that unfortunately such techniques do not translate well to our implicit approach. One possibility would be to advect noisy white material on the object manifolds in a gravity field and try to somehow place this noisiness at the boundary where snow build up ends. This is far from a concrete solution and this problem remains unsolved.

Our simulation system relies heavily on commercial rendering software for producing final results. Many of the more advanced rendering softwares offer users the chance to write their own plug-ins that use the full power of the software package for user-specific needs. Converting our program to a plug-in would mean we could incorporate positioning of objects and the wind domain with the GUI available in the plug-in SDK. More importantly it would remove the need for importing/exporting rather large amounts of data from our program to the rendering engine. It would also mean that the user could use fancy menu systems to set options and easily save and load settings for different scenes. Furthermore, experience tells us that film artists are seldom content after running a physically based simulation. There is always tweaking to do and details to add. With a nice GUI an artist could easily be allowed to manually position collisions and specify their collision domains as a post-processing option. It is often hard to exactly anticipate the behaviour of the program since results accumulate into an ever more complicated scene. For this reason post-processing is absolutely necessary for professional users. The advantages of a plug-in are many but this has been out of scope for this thesis.

The mathematical model we have developed is fairly generalizable since it measures stability as a single angle, θ_{AOR}. For instance if this angle were know for sand our program could easily simulate the buildup of sand. We have not investigated this further as we found it best to focus on one area and try to get good results there. Setting $\theta_{AOR} = \frac{\pi}{2}$ means that the only place where accumulation can occur is at local height minima in the scene. This is exactly the behaviour of water and so our model could be used to estimate where water accumulates.

Chapter 6

Conclusion

Our method has proven to generate realistic buildup of snow. Most importantly our method generates stable snow surfaces based on local propagations, removing the need for global smoothing operations. Complex geometry is handled without special cases making it easy for the user to setup interesting scenes. However, previous methods are more efficient when it comes to quickly covering large flat areas since the occlusion for these areas is very simple and not much global smoothing needs to be done. As for very complex geometry with a lot of over-lapping in the gravitational direction our method is probably faster (run-times not given in previous work), and more accurate. The accuracy comes from the ability to use high-resolution level sets that are equivalent in this case to having millions of polygons but in a more memory efficient structure. Some features of snow distribution are still missing, such as redistribution by wind and melting. We strongly believe this work serves as a good platform for exploring such alternatives in the future. Many of the ideas presented in the previous chapter seem promising and the prototype presented here can be optimized for much better run-time performance. Finally, the level set representation has proven to be well-suited for wind field calculations.

Bibliography

[AL04] Aagaard M., Lerche D.: Realistic Modelling of Falling and Accumulating Snow. *Master Thesis*, Lab of Computer Vision and Media Technology, Aalborg University, Denmark (2004).

[Bet98] David Betounes: Partial Differential Equations for Computational Science. *Springer-Verlag*, ISBN:0-387-98300-7 (1998).

[Fea00] Fearing P.: Computer modeling of fallen snow. In *Proc. of SIGGRAPH* (2000), 37–46.

[FF01] Fedkiw R., Foster N.: Practical Animations of Liquids. In *Proc. of SIGGRAPH* (2001), 23–30.

[FM96] Foster N., Metaxas D.: Modeling the Motion of a Hot Turbulent gas. In *Computer Graphics Proceedings, Annual Conference Series* (1996), 181–188.

[FO02] Feldman B., O'Brien J.: Modeling the Accumulation of Wind-Driven Snow. In *Conference Abstracts and Applications SIGGRAPH* (2002).

[FSJ01] Fedkiw R., Stam J., Jensen H.: Visual Simulation of Smoke. In *Proc. of SIGGRAPH* (2001).

[Han99] Hanesch M.: Fall Velocity and Shape of Snowflakes. *Ph.D. theis*, Ludwig-Maximillians-Universittet, Mnchen (1999).

[Jun00] Junker Norman: Winter Weather Forecasting.
http://www.hpc.ncep.noaa.gov/research/snow2a/index.htm
Accessed 2006-05-24.

[LC87] Lorensen W., Cline H.: Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In *Proc. of SIGGRAPH*, (1987), 163-169.

[LGF04] Losasso F., Gibou F., Fedkiw R.: Simulating water and smoke with an octree data structure. In *ACM SIGGRAPH '04*, 457–462. ACM Press (2004).

[LOC94] Liu X., Osher S., Chan T.: Weighted essentially nonoscillatory schemes. *Journal of Computational Physics*, 115:200–212 (1994).

[LZK+04] Langer M., Zhang L., Klein A., Bhatia A., Pereira J., Rekhi D.: A spectral-particle hybrid method for rendering falling snow. In *Eurographics Symposium on Rendering* (2004).

[Mau03] Mauch S.: Efficient Algorithms for Solving Static Hamilton-Jacobi Equations. *PhD thesis*, California Institute of Technology, Pasadena, California (2003).

[Min03] Min C.: Local level set method in high dimension and codimension. Technical Report CAM 03–67, UCLA, November (2003).

[MBW+02] Museth K., Breen D., Whitaker R., Barr A.: Level Set Surface Editing Operators In *Proc. of SIGGRAPH* (2002).

[MMA+05] Moeslund T. B., Madsen C. B., Aagaard M., Lerche D.: Modeling falling and accumulating snow. *VISION, VIDEO AND GRAPHICS* (2005).

[MN00] K. Muraoka, C. Norishige: Visual Simulation of Snowfall, Snow Cover and Snowmelt. *Seventh International Conference on Parallel and Distributed Systems Workshops (ICPADS'00 Workshops)* (2000).

[Nav1827] Navier L.: Memoire sur les lois du mouvements des fluides. *Memoir. de l'Academ. Royale des Sci. 6* (1827).

[NID+97] Nishita T., Iwasaki H., Dobashi H., Nakamei E.: A Modeling And Rendering Method For Snow By Using Metaballs. In *Proc. of EUROGRAPHICS*, vol. 16, European Association for Computer Graphics (1997).

[NM06] Nielsen M. B., Museth K.: Dynamic tubular grid: An efficient data structure and algorithms for high resolution level sets. *Journal of Scientific Computing* (2006).

[OF03] Osher S., Fedkiw R.: Level Set Methods and Dynamic Implicit Surfaces. *Springer-Verlag* (2003).

[OS88] Osher S., Sethian J.: Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulations. *Journal of Computational Physics*, 79:12–49 (1988).

[OS91] Osher S., Shu W.: High-order essentially nonoscillatory schemes for hamilton-jacobi equations. *SIAM J. Num. Anal.*, 28:907–922 (1991).

[OS04] Ohlsson P., Seipel S.: Real-time Rendering of Accumulated Snow. *Linkping Electronic Conference Proceedings* (2004).

[PMO+99] Peng R., Merriman B., Osher S., Zhao H., Kang M.: A pde-based fast local level set method. *Journal of Computational Physics*, 155(2):410–438 (1999).

[PTS99] Premoze S., Thompson W., Shirley P.: Geospecific rendering of alpine terrain. *Eurographics Workshop on Rendering* (1999), 107–118.

[Ree83] Reeves W.: Particle system - a technique for modeling a class of fuzzy objects. *ACM Transactions on Graphics* 2 (1983), 91–108.

[RVC+98] Rasmussen R., Vivekanandan J., Cole J., Karplus E.: Theoretical Considerations in the Estimation of Snowfall Rate Using Visibility. Technical Report, The National Center for Atmospheric Research (1998).

[RT92] Rouy E., Tourin A.: A viscosity solutions approach to shape-fromshading. *SIAM Journal on Numerical Analysis* 29 (1992), 867-884.

[Sim90] Sims K.: Particle animation and rendering using data parallel computation. In *Proc. of SIGGRAPH* (1990).

[Sta99] Stam J.: Stable Fluids. In *Proc. of SIGGRAPH* (1999).

[Sto1851] Stokes G.: On the theories of the internal friction of fluids in motion, and of the equilibrium and motion of elastics solids. *Transact. Cambridge Philos. Soc. 9* (1851).

[Str99] Strain A.: Tree methods for moving interfaces. *Journal of Computational Physics*, 151 (1999).

[SO88] Shu C., Osher S.: Effcient implementation of essentially nonoscillatory shock capturing schemes. *Journal of Computational Physics*, 77:439–471 (1988).

[SOH98] Sumner R., O'Brien J., Hodgins J.: Animating Sand, Mud and Snow. In *Proc. of Graphics Interface* (1999), 125–132.

Appendix A

Snow Particle Classes

Member variables for the class `SnowParticle` and the classes derived from it.

```
class Particle {
   protected:
      Vec3f mVel;                 //!< [m/s]
      Vec3f mPos;                 //!< [m]
}
class SnowParticle : public Particle {
   protected:
      float mOmega;               //!< [rad/s]
      float mRadius;              //!< [m]
      float mMass;                //!< [kg]
      float mTermVel;             //!< [m/s]
      float mRotationRadius;      //!< [m]
      Vec3f mDownForce;           //!< [N]
      float mTheta;               //!< Rotation around (0,1,0)
}
class Snowflake : public SnowParticle {
   protected:
      unsigned short int mMeshIndex;          //!< Store index into mesh database.
}
class SnowPackage : public SnowParticle {
   protected:
      float mPackageRadius;       // Determines package volume
}
class SlidePackage : public SnowParticle {
   protected:
      unsigned int mLVSIndex;     // Information about which level set the package
      float mPackageRadius;       // Determines package volume
}
```