



Faculty of Science and Technology

BSc (Hons) Games Programming

May 2019

Simulating Snowfall and Accumulation Through the GPU

by

Alasdair Hitchen

## **DISSERTATION DECLARATION**

This Dissertation/Project Report is submitted in partial fulfilment of the requirements for an honours degree at Bournemouth University. I declare that this Dissertation/Project Report is my own work and that it does not contravene any academic offence as specified in the University's regulations.

### **Retention**

I agree that, should the University wish to retain it for reference purposes, a copy of my Dissertation/Project Report may be held by Bournemouth University normally for a period of 3 academic years. I understand that my Dissertation/Project Report may be destroyed once the retention period has expired. I am also aware that the University does not guarantee to retain this Dissertation/Project Report for any length of time (if at all) and that I have been advised to retain a copy for my future reference.

### **Confidentiality**

I confirm that this Dissertation/Project Report does not contain information of a commercial or confidential nature or include personal information other than that which would normally be in the public domain unless the relevant permissions have been obtained. In particular, any information which identifies a particular individual's religious or political beliefs, information relating to their health, ethnicity, criminal history or personal life has been anonymised unless permission for its publication has been granted from the person to whom it relates.

### **Copyright**

**The copyright for this dissertation remains with me.**

### **Requests for Information**

I agree that this Dissertation/Project Report may be made available as the result of a request for information under the Freedom of Information Act.

Signed: 

Name: Alasdair Hitchen

Date: 16/05/2019

Programme: BSc GP



# Abstract

When it comes to the creation of a believable 3D environments, visual effects and the physically based simulation of artefacts plays a highly important role in immersing the viewer. The focal point of this project will be the implementation of a snow simulation demo, replicating the dispersion of snow over a space under the influence of diverse environmental effects. Previous implementations and methods will be considered such as particle systems, screen-space solutions and occlusion-based snow build-up. Accumulation of snow on the surfaces of objects in the scene will be developed with a focus on physical realism while maintaining high performance in calculation and rendering through parallel processing. As maintaining real-time processing speeds is essential to game programming, performance will be profiled in comparison to various scene implementations and layouts with varying levels of detail and realism.

The software is developed in C++ and OpenGL 4 with the target platform being MS Windows. A demonstration program will be produced to allow users to view various effects on the scene through a user interface.

# Table of Contents

Abstract .....	4
Glossary of Terms.....	7
Chapter 1 – Introduction .....	8
1.0 – Project Concept and Justifications .....	8
1.1 – Aims and Objectives.....	8
1.1.1 – Aims.....	8
1.1.2 – Objectives .....	8
1.1.3 – Additional Features and Objectives .....	9
Chapter 2 – Literature Review .....	10
2.0 – Introduction.....	10
2.1 – Real World Weather.....	10
2.1.1 – Snowflake Shape .....	10
2.2 – Particle Systems .....	11
2.4 – GPU Computation .....	12
2.5 – Snow Accumulation .....	13
2.6 – Wind .....	15
2.7 – Collision Detection .....	16
2.8 – Snow Simulation Systems Summary .....	16
Chapter 3 – Methodology and Process .....	17
3.0 – Overall Design and Implementation .....	17
3.1 – System Design.....	17
3.1.1 – Particle Simulation .....	18
3.1.2 – Snow Accumulation.....	18
3.2 – System Implementation .....	19
3.2.1 – OpenGL Abstraction .....	19
3.2.2 – Particle Systems .....	19
3.2.3 – Collision Detection .....	21
3.2.4 – Accumulation System .....	22
3.2.5 – Wind Effects.....	23
3.2.6 – GUI.....	23
3.3 – Testing and Evaluation Strategy .....	24
Chapter 4 – Results and Reflection .....	25
4.0 – Realism .....	25
4.1 – Performance .....	28

4.1.1 – Single Frame Computation Time Distribution.....	28
4.1.2 – Particle Simulation .....	29
4.1.3 – Hardware Differences.....	31
4.2 – Summary .....	31
Chapter 5 – Conclusion and Future Work.....	33
5.0 – Future Work.....	33
5.1 – Conclusion .....	34
References .....	35
Appendices.....	37
Appendix A.....	37
Appendix B .....	39

# Glossary of Terms

API – Application Programming Interface

CPU – Central Processing Unit

FPS – Frames per Second

GDC – Game Developers Conference

GLSL – OpenGL Shading Language

GPGPU – General Purpose Graphics Processing Unit

GPU – Graphics Processing Unit

GUI – Graphical User Interface

HDR – High Dynamic Range

HBV – Hierarchical Bounding Volume

IBO – Index Buffer Object

RAM – Random Access Memory

SIMD – Single Instruction Multiple Data

SSBO – Shader Storage Buffer Object

UBO – Uniform Buffer Object

VRAM – Video Random Access Memory

VBO – Vertex Buffer Object

VAO – Vertex Array Object

# Chapter 1 – Introduction

## 1.0 – Project Concept and Justifications

In its current state, snow simulation in real-time (60fps) is primarily based on occlusion implementations, which leverage similar techniques as used in basic shadow mapping. Visual fidelity and realism play a huge role in the immersion of a scene and snow can completely alter the scape of an environment. As the common hardware available to end-users improves, more performance intensive solutions can be considered viable and this project aims to demonstrate a more physics-based approach, now possible due to advancements in hardware and graphics APIs.

The occlusion technique typically seen in recent and older implementations separates the two primary factors of a snow simulation: the snowfall and snow accumulation. This project intends to combine the two, linking the snow particles to the build-up of snow on a surface.

The project will be developed in C++ and supports a minimum of OpenGL 4.3. C++ was chosen due to its performance capabilities and the wealth of previous experience in the language. OpenGL was selected as it is a widely known with an extensive feature set and being a higher-level graphics API, development time is reasonable, fitting with the duration of this project. Newer features in OpenGL such as Shader Storage Buffer Objects will be essential to this project.

## 1.1 – Aims and Objectives

The project has three primary aims to be completed. These aims focus on the core elements of the project to ensure that the primary features are delivered on time.

### 1.1.1 – Aims

1. The primary aim of the project is to develop an implementation of a physics-based particle snow accumulation system.
2. The secondary aim is to develop a demonstration scene which shows the system working with visual effects to improve fidelity.
3. The tertiary aim is to allow the user to modify the settings allowing for different snow and environmental settings.

### 1.1.2 – Objectives

The objectives break down the aims into sub-tasks. The features will include:

- Snowfall and Accumulation
  - Transform feedback system for updating and rendering particles
  - Collision detection system to detect when a particle collides with an object in the scene
  - Spatial partition system for storing accumulated snow values in differing parts of the scene
- Demo Scene
  - Create several objects which can be added to the scene. This includes collision meshes and texturing.
  - Implement visual effects such as shadow mapping to improve visual fidelity.
  - Create a system for importing .obj files from the CPU side to the GPU.
  - Implement a lite abstraction of OpenGL to speed up development.
- Interaction
  - Integrate imGUI into the project to shorten development time



- Link the GUI into the particle system to allow modification. This will allow the user to modify the scene to show how the dynamic particles behave.
- Provide performance metrics to the user to allow for comparison of different settings. This should, at a minimum, provide counts for FPS, Delta Time, memory usage, GPU usage and VRAM usage.

### 1.1.3 – Additional Features and Objectives

If time permits, extra features will be considered for implementation into the project. These features are not core to the primary functionality but would improve the experience and provide further polish to the finished item.

- Real-time optimisation
  - Managing the scene and snow detail in real-time to ensure performance is not affected to a level which interrupts the user experience.
- Additional Lighting Effects
  - Implementation of OpenGL features such as Bloom, Light Scattering and HDR.
- Wind Map
  - Particles can be affected by wind and a pre-processed wind-map would allow for further realism in how the snow particles flow through the scene. Default behaviour would include a global value for wind, which would only provide a uniform adjustment to all particles in the scene. A more detailed approach would have the wind field reacting to objects in the scene and flowing around them in a fluid-like manner.

# Chapter 2 – Literature Review

## 2.0 – Introduction

Recent academic investigation into fully featured real-time snow simulations are few in number, but many examples of the different elements of the simulation exist, such as the particle simulations presented by White (2014) and Lv & Liu (2013). Scrolling texture approaches will also be considered such as that shown by Wang & Wade (2004).

There are many different snow accumulation techniques for 3D environments, including but not limited to occlusion, depth mapping and surface exposure calculation. These methods consist of a mixture of both online and offline solutions with one of the most well known being the implementation by Fearing (2000). This method is an offline render, which produced realistic results but ran on now outdated hardware. Foldes & Benes (2007) present an occlusion based implementation which also simulates snow melt on a large scale.







The literature discussed in this chapter will form the foundations on which to begin this project. Through merging different methods presented in previous research, as well as looking into new implementations previously not discussed, this project hopes to bridge a gap in snow simulation techniques. While offline rendering is not an aim of this application, older implementations that could theoretically now run on modern hardware will also be considered. General techniques will be researched due to the somewhat sparse nature of recent academic and professional research into snow simulation.

## 2.1 – Real World Weather

When designing a simulation following a real-world happening, considerations must be taken to how the real phenomenon occurs both visually and what causes it to behave the way it does.

### 2.1.1 – Snowflake Shape

Koh (1989) presents a classification for types of snowflake and the temperatures at which they form. The available paper is unfortunately of low visual quality and therefore it can be difficult to distinguish the shapes of photographed individual flakes. Table one below shows a selection of the types of snowflake seen in real life:

ID	Flake Shape	Name
1		Stellar crystal with sectorlike ends
2		Combination of needles
3		Plate with dendritic extensions
4		Hexagonal Plate
5		Minute Column
6		Plate with simple extensions




7		Ordinary dendritic crystal
8		Pyramid
9		Scroll

Table 1 - Real world snowflake types (Koh, 1989)

The table presented in Koh's paper is based on the classification system first put forward by Choji & Chung Woo (1966).

Koh states that snowflakes grow according to ambient temperature until they become large enough to fall in the form of precipitation. Assuming ambient temperature sits below 0°C, the precipitation will fall in the form of snow. Through collision and other forms of breakup, snow particles many vary further in shape.

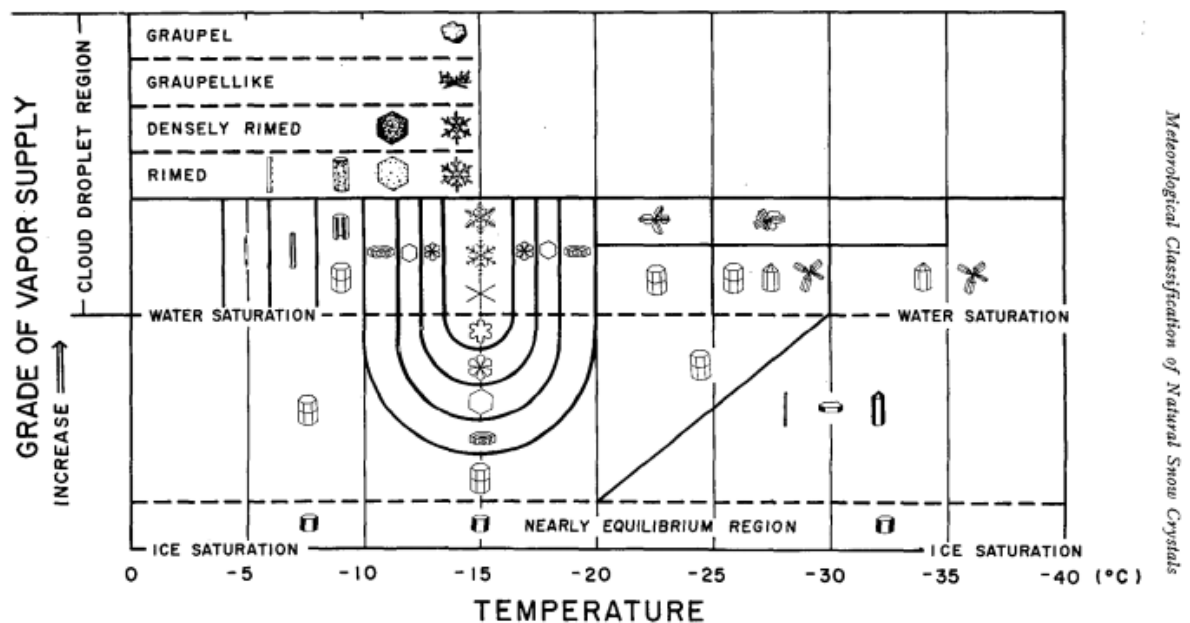


Fig. 2. Temperature and humidity conditions for the growth of natural snow crystals of various types

Figure 1 - A diagram showing the different snowflakes that form depending on ambient temperature and conditions (Magono & Lee, 1966, p. 327)

Figure 1 shows how as ambient temperature differs, the primary shape of falling snow particles changes. The plane shape is the most common, generally forming at temperatures of around -15°C. More obscure shapes form at lower temperatures of -30°C and below.

## 2.2 – Particle Systems

White (2014) presents a particle based system that covers rain and snow. The author discusses the primary differences between a CPU and GPU based approach to updating and drawing each particle with advantages and disadvantages for each. A GPU implementation is chosen for the final design for the sake of the significantly improved performance characteristics when compared to the CPU. The final results include a blended texture for each particle including transparency and while the visuals are satisfactory the simulation fails to provide complex mesh collision detection, relying on bounding boxes or bounding spheres.

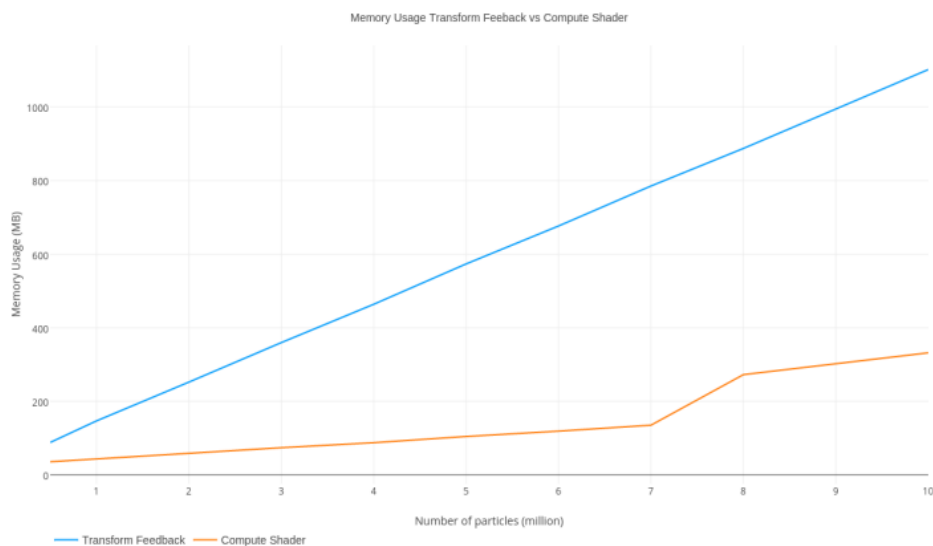
Lv & Liu (2013) proposes a system of simulating particles while also covering the separate visual effect that heavy snowfall causes which is to apply a fog and blur when viewing an object at a distance through the particles. The author also discusses the implementation of a wind field to apply a gustiness to the environment, giving the particles a non-uniform movement path, which is more accurate to the real life effects.

Kipfer et al. (2004) discusses the creation of a completely GPU based particle system in a time when GPGPU computation was in its infancy. The method described is what is known to today as transform feedback. The author covers the differences in how implementations vary between the GPU and CPU, discussing how the benefits of parallel processing and high memory bandwidth favour particle simulation on the GPU over the CPU.

## 2.4 – GPU Computation

Eidissen (2009) presents a method of simulating snow in a wind field with the particles all processed on the graphics card. This implementation utilises the Nvidia CUDA programming language extensions to manage the graphics card memory and calculations. The CUDA approach allows for the greatest potential performance due to how the memory and workers on the GPU can be manually assigned and managed. However, this has the downside of requiring significantly increased development time, stemming from having to learn the syntax of CUDA as well as having an in-depth knowledge of graphics card architecture. This implementation also features collision detection but is limited to a terrain heightmap and not arbitrary objects within the scene.

Rice (2016) covers a library abstracting the two primary methods of GPGPU programming: Transform Feedback and Compute Shaders. Benchmarks are taken of the two different implementations and it is found that Compute Shaders offer only a slightly improved frame rate but significantly improved memory footprint. This improvement in memory usage does not however appear to follow a linear pattern over 7 million particles but for the purposes of this project, particles counts that high are not necessary or feasible.



*Chart 1 - Comparison of Transform Feedback and Compute Shader memory usage (Rice, 2016)*

With this project aiming to simulate particles numbering in the hundreds of thousands and not millions, the memory usage factor between both the compute shader and the transform feedback method are negligible. With modern hardware hosting typically over 4GB of VRAM, 200MB is not a

considerable footprint. In terms of performance, Rice (2016) shows that under particle counts of one million, both transform feedback and compute shaders both retain the real-time frame rate of 60FPS (frames per second).

Owens et al. (2008) discusses the advantages of parallel GPU computing when compared to the CPU. The author discusses how the increase in GPU programmability and the increases in memory bandwidth mean that for many tasks which highly favour this style of architecture, performance has significantly surpassed that of a similar CPU implementation. A particle system follows the GPUs SIMD (Single Instruction Multiple Data) architecture and thus is highly parallelisable.

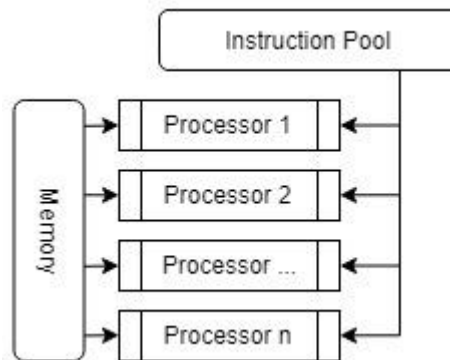


Figure 2 - The SIMD architecture

## 2.5 – Snow Accumulation

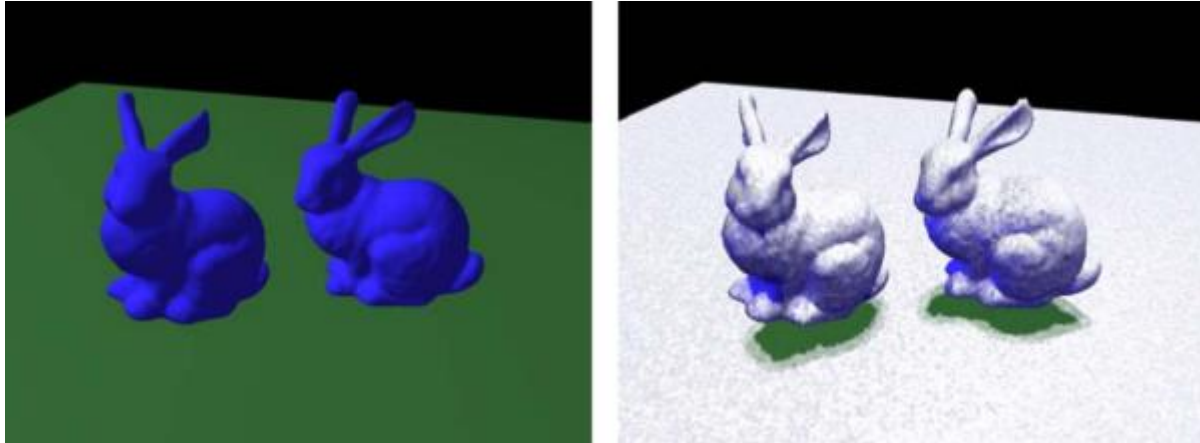
Fearing (2000) describes a method of snow accumulation for offline rendering. This implementation resulted in realistic results including dusting, flutter and wind-blown snow but at the time of design, the hardware could not calculate and render these effects in real-time. The primary difference with this model as opposed to others, was the technique of firing particles from the surface towards the sky, as opposed to the direction that would be expected. This is to increase the control, the idea being that each surface could dictate its resolution by altering the number of particles each site would fire. The implementation also includes the simulation of snow stability, resulting in build-ups where small avalanches may arise from over-accumulation on certain surfaces. This method of snow accumulation renders highly realistic results but is unable to run in real-time due to the heavy computation required to simulate all the different elements of the phenomenon.



Figure 3 - A comparison between a real-world photo and a rendering from Fearing's (2000) implementation of a snow accumulation system.

Ohlsson & Seipel (2004) present a method of rendering accumulated snow on complex 3D models through use of a snow prediction function. This function takes into account factors such as the exposure a position has to the environment and the surface inclination. This method has the

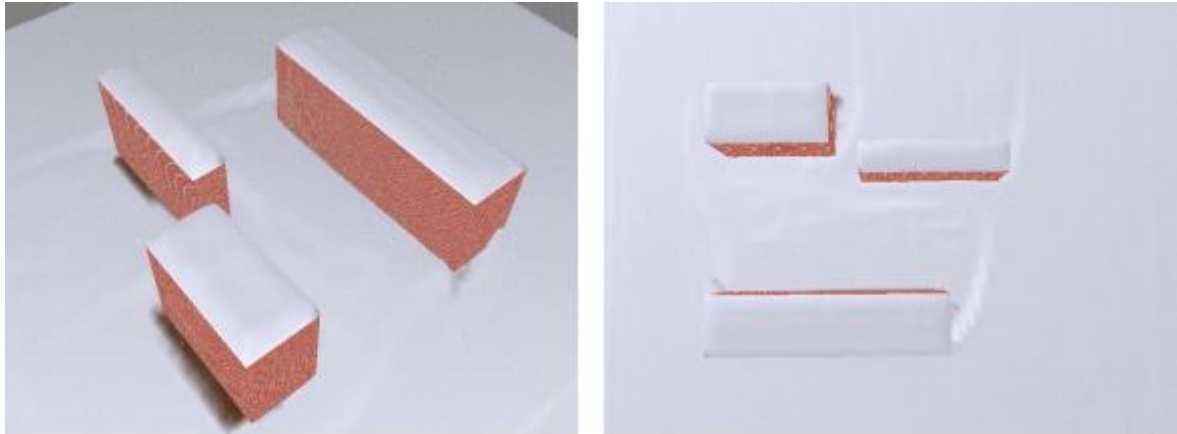
advantage of yielding realistic results for simple scenes with a single direction of snowfall but falls short when considering the physics of how snow falls throughout a 3D space. Snow accumulates predictably and without any kind of variance due to wind or other environmental factors. This method also does not run at a stable 60FPS on the hardware used at the time of testing. It can however be assumed that running on modern hardware, this method would be suitable for real-time rendering.



*Figure 4 - The Ohlsson and Seipel (2004) method of rendering accumulated snow*

Haglund et al. (2002) explain a real-time implementation based on a 2D matrix which stores data on snow depth over areas in which snow can fall. This method has the advantage of being scalable to the hardware which is running the simulation through increasing the density and resolution but has limitations when simulating complex meshes. This implementation combines the simulation of particles with the accumulation through detecting collisions with the surface and applying a displacement to the mesh at the closest vertex to the point of collision. The simulation yields satisfactory results on large, flat surfaces but is not suited to more intricate scenes.

Saltvik et al. (2006) covers a parallel method of simulating both snowfall and accumulation in real time, with a focus on parallelisation. The implementation used for accumulation by this author is similar to that of Haglund et al. (2002) in that surfaces which can be accumulated on are all designated with height matrices to store snow depth. This implementation has the advantages of being capable of running in real-time, with processing being distributed over multiple CPU threads to make the best use of the hardware. However, this simulation lacks the ability to simulate more than 40,000 particles without slowing. This can be linked to the hardware standards of the time, being considerably slower when compared to today's standards. A realistic wind field is also implemented, using a Navier-Stokes fluid dynamics method for simulating the movement of the air throughout the scene.



*Figure 5 - The results produced through the method implemented by Saltvik et al. (2006).*

Stomakhin et al. (2013) present an offline method of simulating accumulated snow dynamics. The authors discuss how particles are used in a MPM (Material Point Method) to simulate snow as a granular material. This is due to the changeability of snow's interaction characteristics such as stiffness, ranging from almost a solid to being close to liquid. This method has the advantage of being incredibly realistic, simulating the distribution of snow as it moves and collides but this comes at the cost of performance. The simulation runs offline, at this time being far from achievable in a real-time application. While this level of dynamics is outside the scope of this implementation, it is important to consider different techniques to rendering and simulating how snow interacts with an environment.

At GDC (Game Developers Conference) 2014, Barre-Brisebois (2014) presented the methodology behind the snow system in a mainstream game. This technique uses tessellation and displacement height maps on accumulation surfaces, applied to the world on a per-need basis. The displacement technique works through detecting an actor or object within the bounds of an orthogonal view frustum set to a depth from the surface up to roughly ankle height. This implementation is similar to that by Haglund et al. (2002) in that surfaces which accumulate snow are designated by hand through a height map. This method performs in real-time with the computation run on the GPU using DirectX 11. The tessellation model adds detail to the depth of the snow, giving it significantly improved visual fidelity. However, this method does not include dynamic accumulation of snow, instead it slowly returns to a default state over time to simulate new snow being added as snowfall is a constant throughout the game world.

## 2.6 – Wind

Moeslund et al. (2005) present a model of snowfall and accumulation which comprehensively simulates the effects of wind through a grid-style wind field. This method uses backwards-integration and interpolation to calculate how a movement of air will affect the air cells surrounding it.



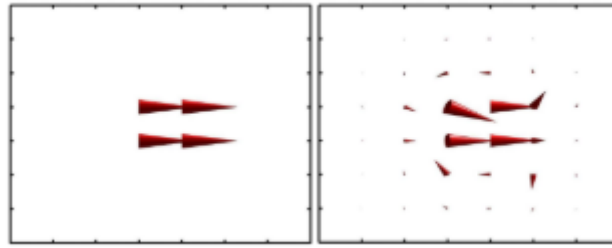


Figure 6 - Before and after of the wind projection step in the Moeslund et al. (2005) implementation

This method provides very realistic results when it comes to snow build-up but is computationally expensive to calculate however in a static scene this could be done once left for the remainder of the simulation. The author's implementation is also not run in real-time but was run on hardware that can be considered outdated by today's standards.

## 2.7 – Collision Detection

Robert Bruce (2006) discusses different approaches to broad-phase collision detection, with their associated advantages and disadvantages. As the intended implementation for this project will involve significant collision detection, it is important to consider how this will be approached. The author discusses spatial partitioning data structures such as KD-Trees and BSP-Trees and how they accelerate the process of collision detection. Performance results are also included showing the time taken to process different implementations with varying levels of complexity in the scene.

## 2.8 – Snow Simulation Systems Summary

The majority methods reviewed do not cover the full range of effects that this project wishes to encompass but provides an effective look into each element of the simulation process.

Author	Real-time	CPU/GPU Based	Accumulation	Particle Simulation
Eidissen (2009)	Yes	<b>GPU</b>	Yes (Limited)	Yes (CUDA)
Fearing (2000)	No	CPU	Yes (Complete)	No
Haglund et al. (2002)	Yes	CPU	Yes (Limited)	Yes
Ohlsson & Seipel (2004)	Yes	<b>GPU</b>	Yes	No
Lv & Liu (2013)	Yes	CPU	No	Yes
White (2014)	Yes	<b>GPU</b>	No	Yes
Moeslund et al. (2005)	No	CPU	Yes	Yes
Saltvik et al. (2006)	Yes	CPU	Yes	Yes

Table 2 - Comparison of Snow Simulation methods

As can be seen from table 2 very few implementations cover both particle simulation and accumulation on the GPU and those that do are narrow in scope. The limiting factor in these approaches seems to lie in the hardware which when compared to today's standards do not come close to the performance standards now expected. The approach used by White (2014) combined with a similar approach to accumulation used by Haglund et al. (2002) would be best suited for the implementation of this project's intended scope.



# Chapter 3 – Methodology and Process

## 3.0 – Overall Design and Implementation

The software design and implementation method that was followed throughout development was based on the Agile development cycle. This method proved successful when dealing with complicated systems, implementing and testing features iteratively to improve operation and simulation quality. Differing methods of implementation were considered with their advantages and disadvantages analysed for each primary and secondly feature to ensure the final project was created to a satisfactory level.

## 3.1 – System Design

The system design was a critical stage of development due to the number of interacting systems involved in the simulation. The simulation was originally intended to run on both the CPU and GPU before it was discovered that synchronisation issues and data transfer bottlenecking would mitigate any performance gains. It was then decided to move the simulation completely over to the GPU. The design will follow and take elements from various implementations discussed previously in the literature review stage, especially those focused on a GPU-based particle system such as those by Eidissen (2009).

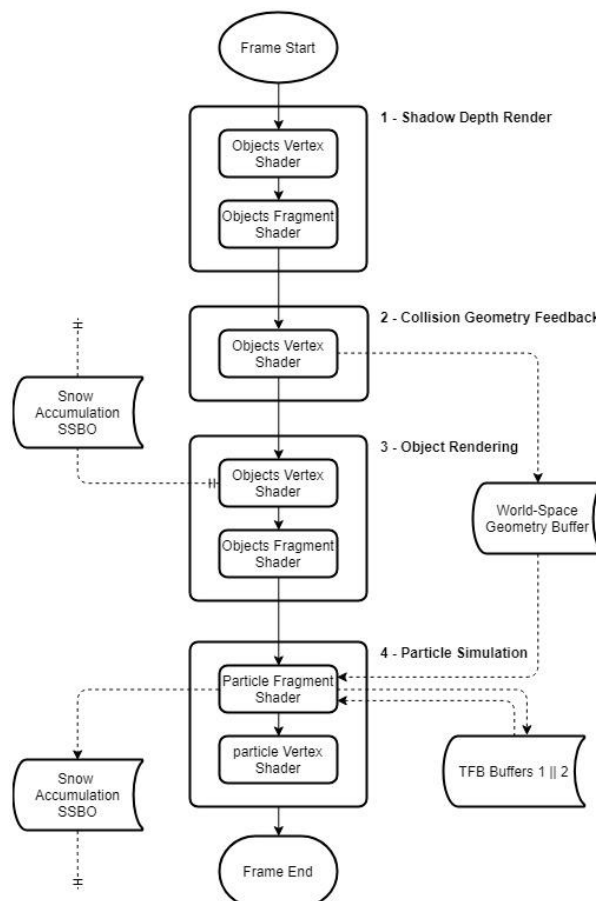


Figure 7 - Simulation Flowchart

### 3.1.1 – Particle Simulation

Considerations must be taken when deciding on the number of particles to simulate. The simulation must run in real-time and therefore dynamic optimisations may have to be made. The particle system will be designed to be highly customisable in both visuals and in the settings controlling the background performance critical data.

The simulation steps must also be performance friendly to how a graphics card processes the shader code, for example avoiding branching statements which causes divergence issues in SIMD architecture. Branch divergence causes processors on the GPU to go through code without executing it to remain in sync with other processors in the same warp which are executing it. If 50% of threads evaluate a condition differently to the other 50%, the processing speed is halved (Han & S. Abdelrahman, 2011). This behaviour can be avoided by reducing the use of branching such as that caused by the “if-else” statement.

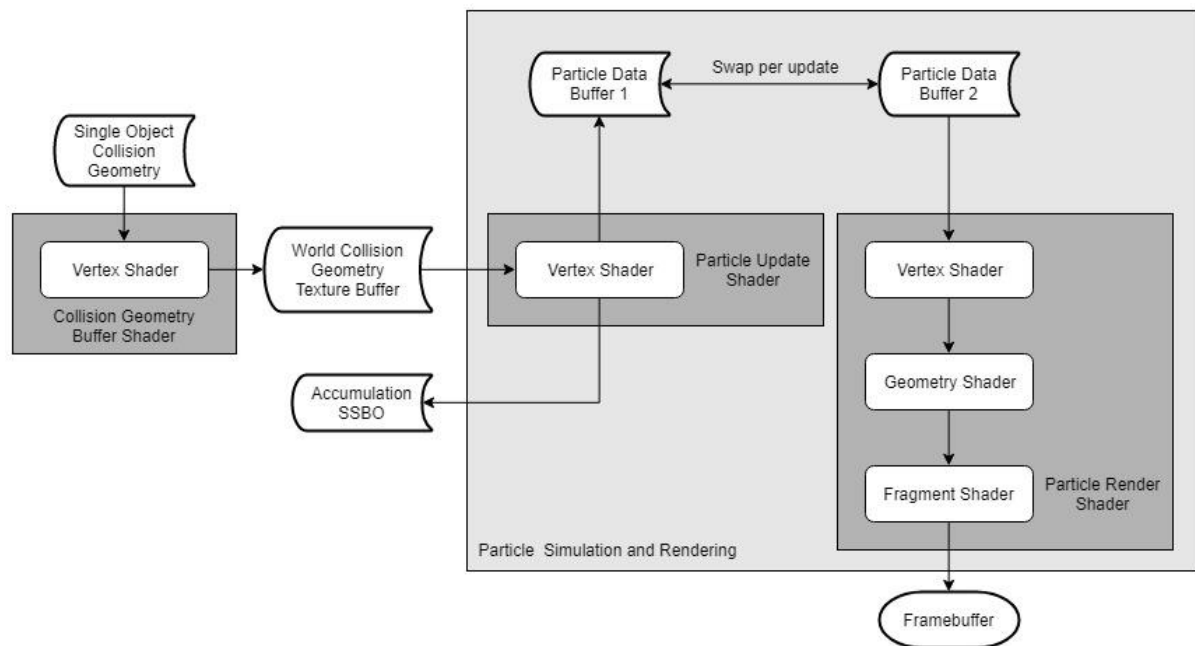


Figure 8 - The particle simulation structure

The transform feedback implementation was chosen due to the simpler structure and the background knowledge that can be immediately applied in implementing the system. The compute shader alternative would require significant reading and learning in order to create a reliable system which is outside the bounds of available time.

### 3.1.2 – Snow Accumulation

The snow accumulation system must be dynamic enough to support arbitrary objects in a static scene and not rely on low resolution bounding boxes. Similar to how the particle simulation is implemented, the accumulation system must be customisable to ensure it can run on a greater range of modern hardware and if implemented into other applications, it can run smoothly.

Storing the data for the snow accumulation in a SSBO on the GPU allows access from various shaders throughout the simulation pipeline as and when it is needed. As this buffer is required in both the particle update and the accumulation stage of the simulation, this is a perfect use case for this data storage method.

## 3.2 – System Implementation

The simulation system can be broken down into a number of key stages. These four stages are shown in Figure 7 below as:

- System 1 – Shadow Depth Render
  - This step is not essential to the simulation but provides further visual fidelity to the scene.
- System 2 – Collision Geometry Feedback (see 3.2.3)
  - The collision geometry feedback stage loads all the world-space geometry into a buffer from the vertex shader through transform feedback.
- System 3 – Object Rendering (see 3.2.4)
  - The object rendering state takes the data from the spatial partition which holds all the accumulation data and uses the values for the current vertex position to calculate how much to offset the vertex by.
- System 4 – Particle Simulation (see 3.2.2)
  - The particle simulation step calculates the new position for each particle and detects collisions with the world space geometry from system 2. On a collision, a value is added to the accumulation spatial partition in the bin which surrounds the collision position.

### 3.2.1 – OpenGL Abstraction

Initial implementation of the project focused on the abstraction of the core OpenGL functionality. This covered the basics of Vertex Array/Object Buffers, Shaders and Textures. Having a solid foundation of abstraction allowed for development ahead of this implementation stage to be streamlined and be considerably quicker when compared to using individual OpenGL calls. Effort was also made to limit driver overhead, techniques discussed by Everitt et al. (2014) at Game Developers Conference 2014 where implemented to improve performance by reducing the number of draw calls and OpenGL state changes.

### 3.2.2 – Particle Systems

From the planning stages of the project, it was evident that CPU-side particles would not give the performance required for the simulation. Two common methods of simulating particles on the GPU exist, being through Transform Feedback or via a Compute Shader. The former was chosen because of a simpler structure to fit within time constraints but a Compute Shader implementation should be considered for future work.

#### 3.2.2.1 – Transform Feedback

The transform feedback approach utilises two buffers for the simulation and rendering of particles. These buffers are interchanged with every update tick of the particle system with the primary active buffer covering the updating of particle positions and other attributes while the second handles the rendering. This is commonly known as “ping-ponging” and is a method of circumventing the issues which arise from reading from and writing to a buffer in the same frame.

Due to the parallel nature of GPU computing, this provides a significant increase in performance over a CPU-side implementation. Particle systems are a perfect candidate for a parallelised approach primarily due to there being no interdependence between the data, meaning each thread can process its particle without others slowing it down in any way.

As can be seen in figure 8, the transform feedback particle system in this implementation works through ping-ponging between two buffers, both containing the particle data. The currently active buffer is bound to the GL\_TRANSFORM\_FEEDBACK\_BUFFER binding point, after which transform feedback is started and the particle update shader is run by a typical `glDrawArrays` call. The transform feedback is then ended and will have filled the bound buffer with the collected data.

### 3.2.2.2 – Geometry Shader Billboarding

In order to move away from OpenGL point rendering and give the particles a defined shape, a method called billboarding was used to render a 2D textured quad in an orientation which always faces the active camera. This is a performance friendly method of drawing hundreds of thousands of snow particles in the scene. These are also known as “Point Sprites”.

To simplify and accelerate the drawing of these particle quads, a geometry shader was created to generate the quad mesh from a single point. As can be seen in Figure 99, the generated vertices are placed around the particle’s position to form a quad which is output from the geometry shader as a triangle strip.

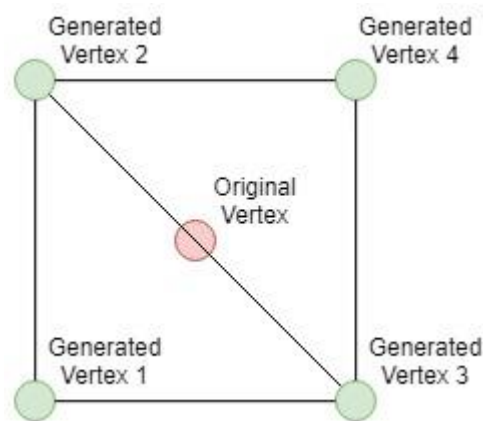


Figure 9 - Geometry Shader Output

The geometry shader also handles the orientation of the particle which as mentioned will follow the camera's rotation to always be facing it. This is achieved through passing the positional data from the vertex shader to the geometry shader in view space, allowing for a simple translation using 2D vectors for each generated vertex's correct position. This move to view space is done in the vertex shader by multiplying the vertex position by the model and view matrices. Once the generated vertex is in the correct position in view space, it is then multiplied by the projection matrix to draw it correctly for the active camera. When the newly generated geometry is then passed onto the fragment shader, a snowflake texture is applied to the quad including the transparent alpha channel. This is then drawn to the framebuffer last to ensure the correct ordering for transparency, with the scene shown behind the snowflakes.

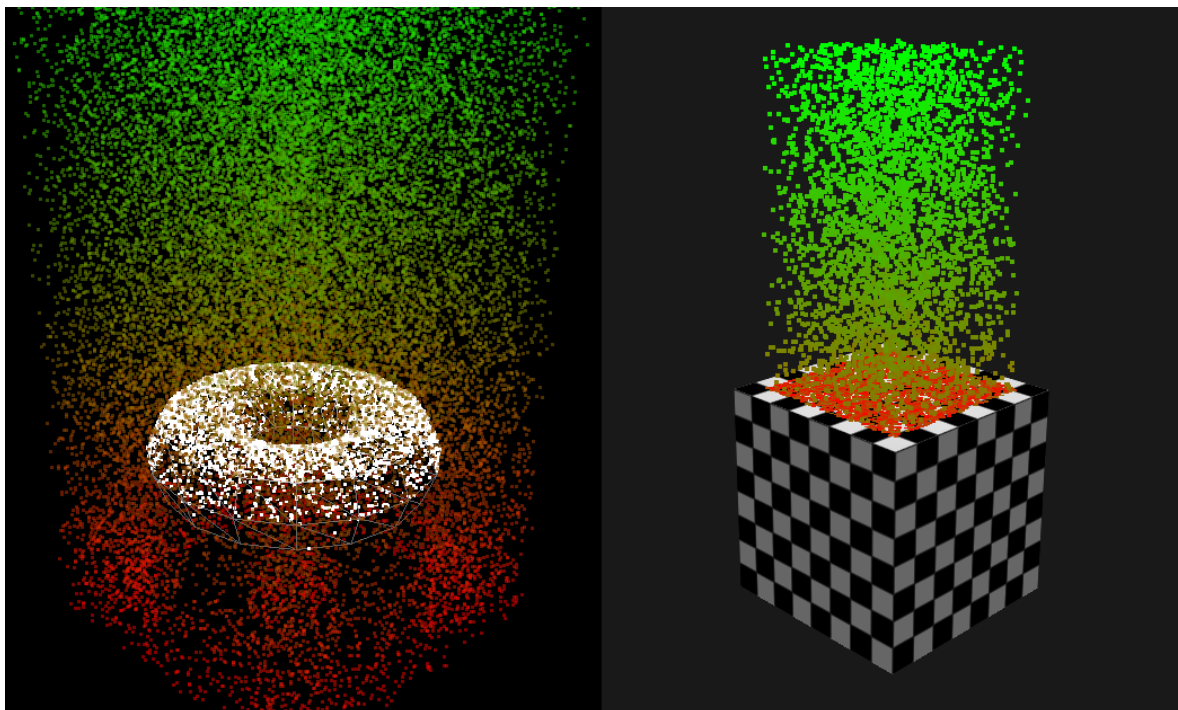


*Figure 10 - Billboarded snowflake particles*

### 3.2.3 – Collision Detection

Because of the physics-based nature of the simulation, a robust and accurate collision detection system was required. It quickly became apparent that having particles and mesh data on the GPU meant that the collision detection had to also stay GPU-side to avoid bottlenecks between the CPU and GPU. A similar solution to that described in The OpenGL Programming Guide (Kessenich, et al., 2017) was implemented using Transform Feedback.

This system worked through capturing the world space geometry calculated when drawing a mesh through OpenGL transform feedback. The geometry was stored in a Texture Buffer, formatted as RGBA (XYZW) with each texel representing a 3-dimensional vertex and 3 sequential texels representing a triangle which could be tested for collision.



*Figure 11 - Early stage collision tests*

This captured world space geometry could then be used by the particle simulation to detect collisions by looping through the texture buffer and reading three texels at a time to form a complete triangle. A line segment would then be formed between the particle's original position and



its newly calculated position to test against the triangle. This method of collision detection can be seen shown in Figure 12.

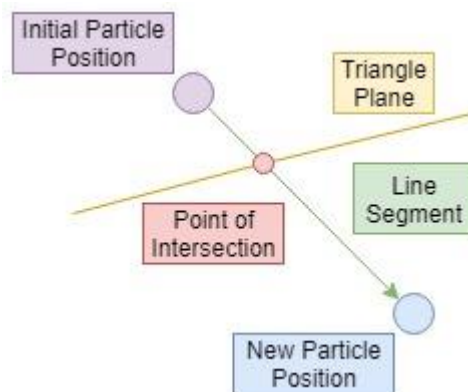


Figure 12 - Collision Detection Method

Each particle tests for collisions against the world-space geometry in parallel on the GPU, significantly improving performance over a typical single line of execution. This method also allows for all other stages of the simulation to remain on the graphics card and removes the need to move data backwards and forward from the GPU and the CPU.

As the implementation currently stands, each particle will test against every triangle of the world space geometry each frame. This process contains no optimisation, therefore particles which are far from any objects will still be checking for collisions unnecessarily. As was discussed previously in the literature review chapter, this process could be accelerated through use of a spatial partitioning data structure. Common methods include KD-Trees and BSP-Trees, both being highly effective in minimising unnecessary computation. In a CPU implementation of a particle system, these data structures are simple to implement but due to data being kept on the GPU for this project, a GPU based tree would be necessary. This introduces challenges as shader code is significantly more limited when compared to C++ due to the hardware limitations that the GPU applies on what is possible. While examples of KD-Trees being run on a GPU do exist, implementation for this project would have taken too long a time to be considered.

### 3.2.4 – Accumulation System

The accumulation system is based around a Shader Storage Buffer Object which is accessible from both the object rendering shader and the particle system shader. This method of using a SSBO provides the functionality of being arbitrarily writeable and readable from a GLSL shader which is required to get data from the particle simulation shader to the object rendering shader.

```
11 layout (std430, binding = 1) buffer buffer_accumulation
12 {
13     vec4 ac_dimensions;           // the size of the spatial partition
14     vec4 ac_resolution;          // the number of partitions in the width, height and depth
15     vec4 ac_position;            // the position of the spatial partition
16     // pre compute
17     vec4 ac_positionBL;          // the bottom left position of the spatial partition (used to offset for always positive values)
18     vec4 ac_binSize;             // the size of an individual bin
19     // data
20     int bin[];                   // the array of bins
21 };
```

Figure 13 - The accumulation SSBO

The system works through dividing the scene up into a spatial partition with each block of partition given a corresponding single integer in an array, representing the level of snow in the area it covers. When the vertex shader is running through the vertices of an object in the scene, each vertex

calculates the partition it is located within and checks for the level of snow it contains. This level of snow is then used to add a displacement to the vertex before being passed onto the fragment shader to define the ratio at which to mix the snow and base textures.

Snow is added to partitions through the collision detection system. When a collision is detected, the partition that contains the position of the collision is calculated and a small amount of snow is atomically added to its total snow level. One of the major considerations was data-race conditions arising from multiple writes to the same memory locations. OpenGL's atomic operations are a solution to this issue, providing sequenced access to memory for parallel writes to the buffer.

This method performs quickly as calculating which partition a 3D position lies within is a  $O(1)$  time complexity operation, easily done in parallel on the GPU. The atomic operations on the GPU are also hardware supported and therefore do not impact performance to any considerable degree.

Another method that was tested for this purpose was storing the data in the form of a 2D texture buffer, similar to how the world space geometry is treated. However, this is restricted in the data it can contain with all the memory simply in one large chunk addressed in the same format as is expected for a texture. SSBOs allow for arbitrary data structures before an array without an allocated size. This array is used for the spatial partition data as can be seen in Figure 13.

### 3.2.5 – Wind Effects

Wind is a huge part of snow simulation. With snow particles being so lightweight, any kind of atmospheric wind will have a heavy effect on the movement of the particle as it falls towards the ground. The wind field implemented by Eidissen inspired the effects used in the project but on a simpler level due to time constraints.

The method used in the project follows a similar structure to the accumulation system in that it works through a spatial partition. The difference being that the bins store a vector modelling a wind direction as opposed to a value representing an amount of snow. As a particle is updated, it tests the partition to find the wind value at its current location and applies this to the velocity.

The implementation of the wind field is basic with wind values simply being randomised at initialisation. Further work would have this wind field interacting and circumventing objects as it does in reality.

### 3.2.6 – GUI

The user interface provides control to the user of different factors which effect the simulation. This allows the user to see different interactions in the technical demonstration and see how the particle simulation differs with changing parameters. The GUI also shows performance metrics which assists in assessing what the simulations limits are.

The GUI for this project was created using the Dear ImGui library, developed by Omar Cornut. ImGui is a fully featured immediate mode UI library which utilises OpenGL for drawing. This gives it high performance while still maintaining the benefits of quick implementation times through the immediate mode style.

The GUI is controlled through a dedicated class which is called per update and links to the particle system through its settings. All the logic for updating and rendering the user interface is held here to segment it from other unrelated sections of the project.

### 3.3 – Testing and Evaluation Strategy

The primary aims of this project were to create a real-time simulation and thus performance is a critical factor. It is an important feature that performance is able to be profiled accurately so further improvements can be made through optimisations.

The testing for this application was performed on the following hardware:

Hardware	Specification
CPU	Intel Core i7-4790k – 4.0GHz
GPU	Nvidia GTX970
RAM	16.0GB
OS	Windows 10 x64

*Table 3 - Test hardware*

Alongside the basic profiler shown on the GUI (see 3.2.6), further in-depth tools were developed in order to output data from the CPU and also the GPU. Using the in-built OpenGL Elapsed Time Query functions, accurate timings can be done on GPU calls at nanosecond level. This data was written to a class which held the performance data for a single frame and could be output in the form of an entry to a CSV (Comma Separated Values) file. This data includes:

- The frame number
- Shadow mapping computation time
- The time taken for the transform feedback of collision geometry
- The render time for the visuals of the scene, including offsetting for snow values
- Particle simulation time including collision detection and physics

These timer queries are wrapped around the primary elements of the simulation in order to see where optimisation efforts can be focused. It can also show where processing can be reduced through lessening the impact of a particular system, for example through reducing the number of particles in the particle simulation system. All GPU profiling data was taken from between frame 1000 and 3000. This ensures the data is not affected by the application start-up methods slowing execution.

The data was visualised using a Python script with the Matplotlib library to render the charts and the Numpy Library to assist in formatting the data correctly. All values received from the OpenGL timer queries were converted from nanoseconds into milliseconds to make the values more human readable in instances where timers ran for longer periods of time.

Differences in hardware will also be considered for a test case to show how advances in modern GPU hardware impact the speed of the simulation.



# Chapter 4 – Results and Reflection

## 4.0 – Realism

When considering the visuals of the scene, real world effects must be considered due to the realism focused nature of the project. Figure 14 shows a comparison between a screenshot from the application and a real-world photo of a picnic bench. In this rendering, there was no wind affecting the particles, ensuring they all fell straight down towards the ground.



*Figure 14 - Comparison between a render from the application and a real-world photo (Dunn, 2014)*

As can be seen in the rendering, there is still considerable room for improvement in terms of visual fidelity. When considering the lighting, the render lacks the global illumination of the real world and background scenery. When looking at the snow build-up, the blocky nature of the spatial partition is noticeable and the extrusions from the table lack detail in their triangle count. Something that could solve this issue would be the introduction of tessellation to add detail to the model. This was however outside the limits of time for this implementation.

Another obvious difference is in how the snow accumulates on the surface of the table. The snow in the real-life image is on a completely flat surface and lacks the gaps between the planks but otherwise it appears considerably more rounded than the accumulated snow in the rendered image. This is another issue which could be solved through tessellation as it stems from a lack of geometry.

The final textures in the rendered snow simulation comes through mixing the original colour with a snow texture and this creates issues on boundaries. As the accumulated snow adds no additional geometry and simply extends what already exists, there is slight blurring as the texture changes. In the real-world image, the snow and bench are clearly separate but in the rendered image they appear to be joined as one.





What the rendering does well however is how the table blocks the snow from accumulating below it. As can be seen below the table, small amounts of snow make it through the cracks in the surface between the wooden planks but overall the clear majority settles above. This becomes more apparent when a global wind is applied to every particle as can be seen in Figure 15. To the right of

the table and the bin, a space where snow is unable to reach is left untouched and remains green from the grass underneath.



*Figure 15 - Effects of wind on snow accumulation*

What can also be noticed in Figure 15 is the variance in particle size. This is linked to the mass of the particle with heavier particles rendered larger. The shape does not vary but this could be easily implemented in a future iteration of the project though applying a different texture depending on snowflake mass. This is modelled after real life where snowflake size varies significantly. While a procedural snowflake generation system could provide the complete uniqueness seen in real-world snow, such a system is outside the scope of this project.

	<p>Stage 1</p>
	<p>Stage 2</p>
	<p>Stage 3</p>
	<p>Stage 4</p>

*Table 4 - Snow progression over time*

## 4.1 – Performance

### 4.1.1 – Single Frame Computation Time Distribution

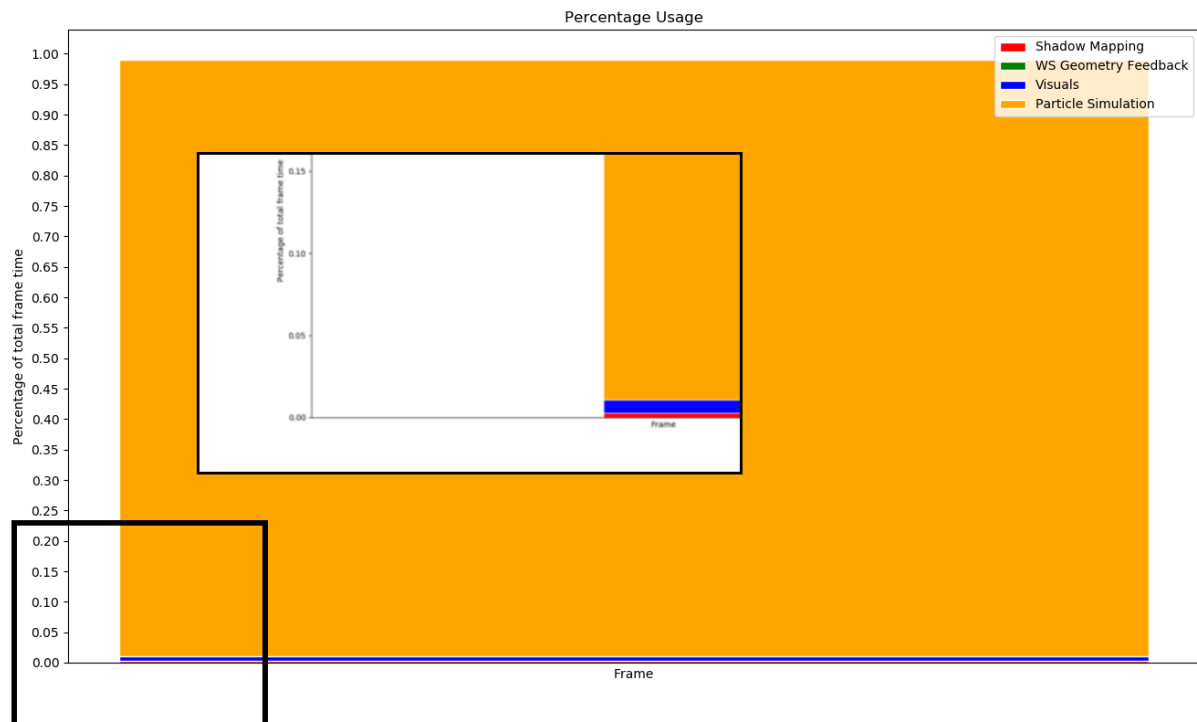
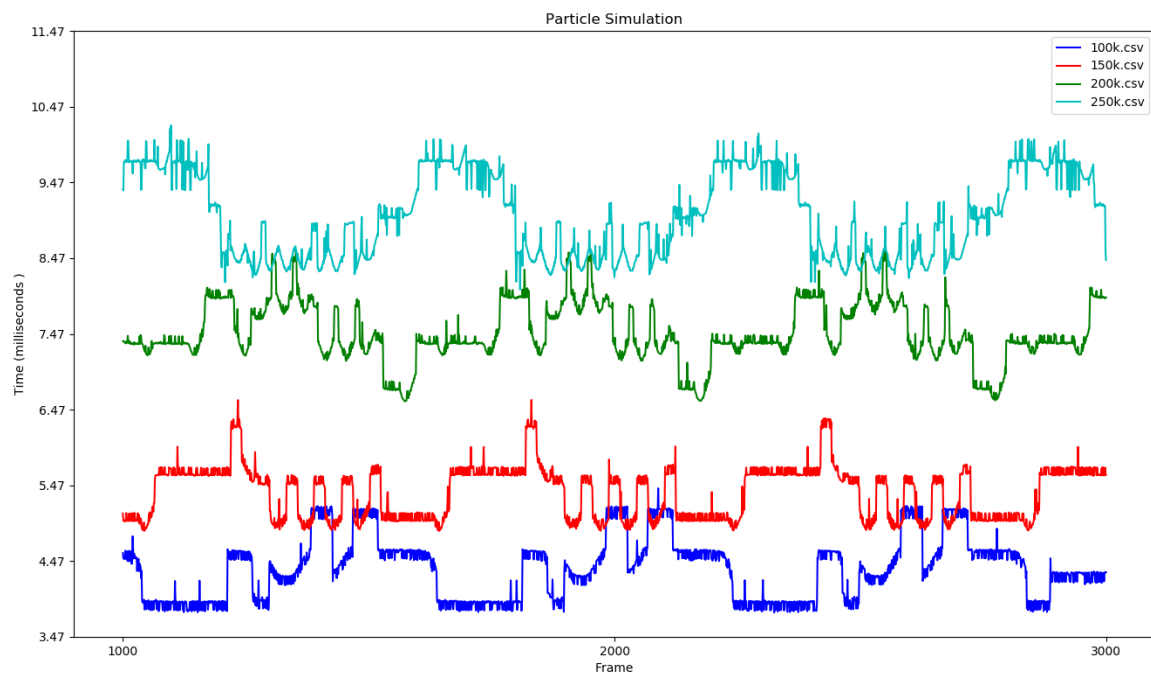


Chart 2 - Single frame processing distribution

Chart 2 shows the percentage distribution of the total time taken to render an average single frame. As is apparent from the data, the particle simulation step takes up a considerable amount of time when compared to other operations. The zoomed-in segment of the chart shows that the other elements of the simulation (including rendering the visuals) takes up less than 2% of the total time taken. This shows that the first steps in future optimisation of the simulation should start at the particle simulation stage. As other systems within the snow simulation do not in any way significantly impact performance, the particle simulation step will be the primary focus of benchmarking analysis.

### 4.1.2 – Particle Simulation



*Chart 3 - Particle Simulation time with differing numbers of particles*

Chart 3 shows how as the particle count is increased, the time taken for the GPU to execute the particle simulation step also increases. This change follows an  $O(n)$  complexity. While the shader which handles updating of particles runs in parallel over many different snowflakes at one time, complexity is introduced when comparing particle position to mesh geometry for collision detection. Updating and rendering of particles in large numbers is not computationally expensive, without the collision detection method the simulation can handle around a million particles in real-time.

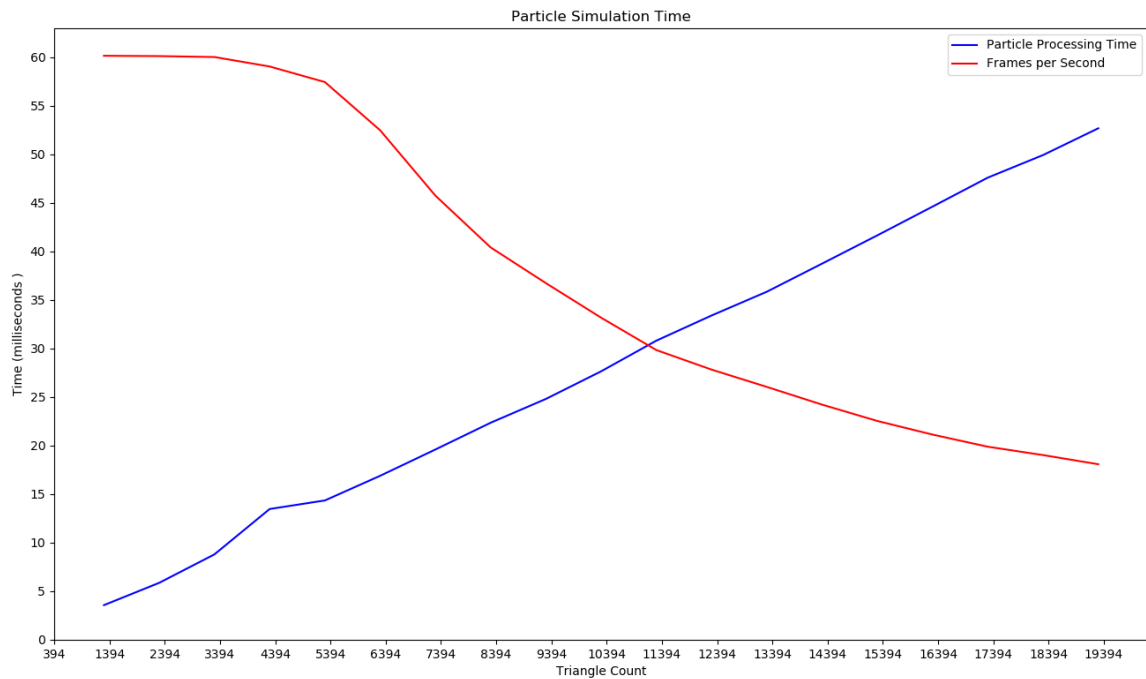


Chart 4 - Particle Simulation time with differing scene collision geometry at 100,000 particles

Chart 4 shows how the frame rate and particle processing time change as the scene collision geometry triangle count increases. As was deduced from Chart 4, the change in processing time increases at an  $O(n)$  complexity with the triangle count as more collisions must be checked. It can be seen that once the scene collision geometry passes roughly 3500 triangles, the frame rate begins to decline from the maximum of 60fps, no longer becoming real-time. It would be recommended that basic collision meshes be used whenever available to maximise performance.

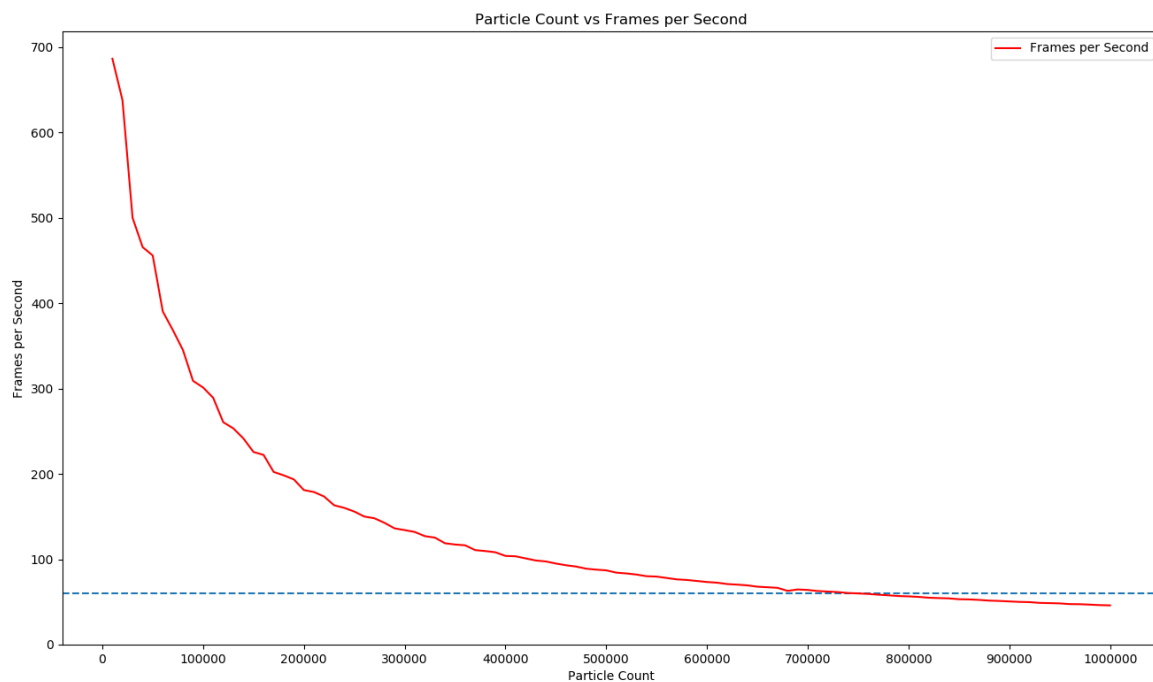
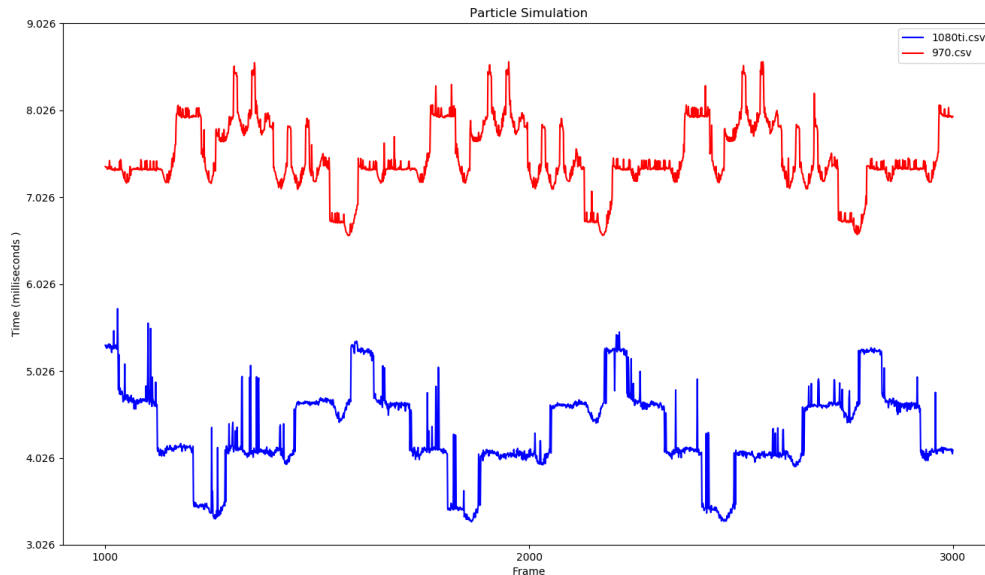


Chart 5 - The average frames per second at differing numbers of particles. The scene contains 794 collision mesh triangles.



Chart 5 shows the decrease in average FPS as the particle count is increased. The scene was sparsely populated with collision meshes, containing just 794 triangles. The dashed blue line acts as a reference to show the level at which the framerate passes below 60FPS, therefore no longer simulating in real-time. It was expected that the FPS would decrease linearly with the increase in particle count, discovering the reasoning behind this would require further investigation.

### 4.1.3 – Hardware Differences



*Chart 6 - Comparison between a Nvidia GTX-970 and a Nvidia GTX-1080ti, both simulating 200,000 particles*

Chart 6 shows how a difference in hardware affects the performance of the simulation. The data shows that a Nvidia GTX-1080ti gives an improvement of an average of 3ms a frame over a Nvidia GTX-970. With the 1080ti hosting 10.6TFLOPS over the 970s 3.494TFLOPS, this change is to be expected and shows that this simulation will run considerably better on modern hardware.

## 4.2 – Summary

The snowfall and accumulation simulate in real-time and produces results that are recognisable to the real-world equivalent. The technical side to the simulation is however significantly more refined than the visual elements. Due to the lack of tessellation, detail in the snow is limited to that of the underlying mesh, requiring these meshes to be modelled in unnecessarily high polygon counts on flat surfaces. The colouring of the snow textures could also be improved with the addition of multi-texturing with bump-mapping and specular highlighting to give a greater sense of roughness and depth to the surface.

Addition of wind to the simulation shows how the snow accumulation system works through true particle collision detection as opposed to approximation through occlusion. As can be seen in Figure 14 and Figure 15, the effects are realistic when compared to the real world phenomenon. To improve this further, implementation of a wind field which considers objects in the scene as obstacles which the air flows around, would significantly improve the realism of the distribution of the snow. This method was used by Moeslund et al. (2005) and the results proved to be more accurate than this implementation can attain in its current state.

Performance overall is satisfactory for a simple scene, as well as in more complex scenes with effective collision meshes. Two of the greatest impacts to performance are the number of particles

being simulated and the number of collision detection checks to perform. Limiting the number of triangles in the collision meshes and reducing particle count to only what is necessary is the ideal way to optimise a scene for use in this application. As this implementation requires heavy use of the GPU, use of the latest hardware is also recommended for the best experience.

Lighting and visual effects could be improved through addition of a skybox for background imagery. Snow also has a fogging effect on objects both inside and behind it when falling in large quantities. This effect is not simulated in this implementation but it's addition would improve visual realism and fidelity.

The features required for the simulation to run take on average 8ms to run with 200,000 particles and a scene complexity of roughly 3500 triangles for collision detection. While this seems small, it makes up 50% of the total time budget for a 60fps experience of 12ms per frame. For some applications, this may prove to be too much unless the user is running the latest hardware, reducing that 8ms considerably.



# Chapter 5 – Conclusion and Future Work

## 5.0 – Future Work

While this project met its primary aims, additional feature targets were unfortunately not met. Features which could not be implemented included real-time optimisation of the simulation to maintain a stable 60fps and a wind field which flows around objects in the scene. The reason many of the extra features were not implemented was a lack of available time, as more significant sections of the project required further attention and development.

Initial improvements to the application if more time was available would focus on the more basic visual elements in lighting and texturing. The renderer for this technical demo does not support multi-texturing at this time, only rendering in diffuse colour and missing bump-mapping and specular highlighting. Snow accumulated on the ground often appears flat and dull which would be significantly improved through added roughness.

Real-time optimisation could not be included due to time constraints. This feature would have proved useful in regulating the particle count and balancing snow accumulation speed to compensate, allowing for a linear increase of the snow accumulation level with less particles being simulated to improve performance when necessary.

Further optimisation could also come in the form of a data structure to hold the collision geometry more effectively. Data structures such as a KD-tree can significantly improve performance when dealing with large amounts of collision mesh geometry through culling unnecessary checks. However, implementation of this method proves difficult on the GPU with limited shader functionality. HBVs (Hierarchical Bounding Volumes) would be a simpler approach with more immediate results through testing particles against a bounding box surrounding an object to decide if further collision detection should continue against the mesh triangles it contains. In more complex scenes, this would significantly increase performance.

The addition of tessellation would be a significant improvement for this implementation, removing the limitations the underlying meshes place upon the snow accumulation system. As it stands, the system requires the mesh to be of a high polygon count to extrude vertices that have snow covering them. Tessellation would allow this process to be placed on the GPU and give the snow undoubtedly more detail as it sits on objects. This was another feature that was cut due to time constraints as it would involve learning OpenGL's tessellation system and implementing it into the already formed rendering pipeline of the project. It would be recommended on future development to resign the renderer to support more advanced features.

Additionally, the current implementation has the limitation of not performing correctly with dynamic objects moving throughout the scene. Collisions can be detected with dynamic object without issue due to the transform feedback system, but the snow accumulation does not follow the object it sits upon. The location of accumulated snow is tied to the scene's spatial partition and not the objects themselves. In a future version, this could in theory instead be tied to the objects themselves and make the system dynamic. It is anticipated however this implementation would require improved hardware to run in real-time.

## 5.1 – Conclusion

Overall, the implemented systems for the simulation of snowfall and accumulation succeeded in meeting the primary aims of the project. Visual and technical results are consistent in their realism when compared to real world effects to a satisfactory standard. The snowfall and accumulation add a great sense of mood and realism into a scene, completely transforming the environment into something different in a natural manner. When compared to the real-world phenomenon the results are easily recognisable and it can be noticed where the physics-based accumulation deprives covered areas of snow.

The simulation however could use improvement in various areas such as in the detail at which accumulated snow is presented. As the accumulated snow relies on the mesh density of the underlying object it is resting on, sparse meshes will present with blocky and unrealistic results. Addition of these features would have been possible with extra time; however, the primary aims took priority. Further tweaks could also be made to the existing settings to improve the rates of accumulation to more realistic values.

Through testing of the simulation's different aspects, it can be found the implementation is suitable for real-time simulation and could theoretically be implemented into a simple game or other visual piece. Performance is found to be tightly coupled to hardware as the scene complexity increases and therefore it would be recommended that the simulation be run on as modern hardware as is available. Memory usage is well within acceptable limits for implementation into other applications. Use of modern C++ features such as smart pointers ensure the program runs with no memory leaks cutting the simulation short.

The scope of this project was ambitious in how it looked to achieve aims that very few other implementations of similar systems were able to achieve. The primary aims were all met, and it can be said that overall the project was a success. During the planning and implementation of the project, a great deal was learned about not only technical methods and practices, but also about the natural world and how snow is formed in an environment. This subject was incredibly interesting to work on and it is hoped that this project will act as a reference to how future real-time snow simulations are thought through and implemented.

# References

- Barre-Brisebois, C., 2014. *Deformable Snow Rendering in Batman: Arkham Origins*. San Francisco(California): Game Developers Conference.
- Dunn, M., 2014. *Picnic table with snow*. [Art] (Roads End Naturalist).
- Eidissen, R., 2009. *Utilizing GPUs for Real-Time Visualization of Snow*, Trondheim: Norwegian University of Science and Technology.
- Everitt, C., Foley, T., McDonald, J. & Sellers, G., 2014. *Approaching Zero Driver Overhead in OpenGL*. San Francisco(California): Game Developers Conference.
- Fearing, P., 2000. Computer Modelling Of Fallen Snow. *SIGGRAPH '00*, 1(27), pp. 37-46.
- Foldes, D. & Benes, B., 2007. Proceedings of the Fourth Workshop on Virtual Reality Interactions and Physical Simulations. *Occlusion-Based Snow Accumulation Simulation*, 1(4), pp. 35-41.
- Haglund, H., Andersson, M. & Hast, A., 2002. *Snow Accumulation in Real-Time*. Gävle, SIGRAD, pp. 11-16.
- Han, T. D. & S. Abdelrahman, T., 2011. *Reducing branch divergence in GPU programs*. Newport Beach, ACM International Conference.
- JEGX, 2014. *Particle Billboarding with the Geometry Shader (GLSL)*. [Online] Available at: <https://www.geeks3d.com/20140815/particle-billboarding-with-the-geometry-shader-glsl/> [Accessed 12 04 2019].
- Kessenich, J., Shreiner, D. & Sellers, G., 2017. *The OpenGL Programming Guide*. 9th ed. Boston: Addison-Wesley.
- Kipfer, P., Segal, M. & Westermann, R., 2004. *UberFlow: A GPU-Based Particle Engine*. Grenoble, Eurographics Association, pp. 115-122.
- Koh, G., 1989. *Physical and optical properties of falling snow*, s.l.: Defence Technical Information Center.
- Lv, H. & Liu, F., 2013. Real-Times Snowfall Simulation Based on Particle System and Pulverization. *Applied Mechanics and Materials*, Volume 373-375, pp. 1168-1171.
- Magono, C. & Lee, C. W., 1966. Meteorological Classification of Natural Snow Crystals. *Journal of the Faculty of Science*, 1(7), pp. 321 - 335.
- Moeslund, T., Madsen, C., M., A. & Lerche, D., 2005. *Modeling Falling and Accumulating Snow*, Bath: Vision, Video, and Graphics.
- Ohlsson, P. & Seipel, S., 2004. *Real-time Rendering of Accumulated Snow*. Gävle, SIGRAD 2004.
- Owens, J. D. et al., 2008. *GPU Computing*. s.l., IEEE, pp. 879 - 899.
- Rice, E., 2016. *OpenGL GPU Features and SPH Fluid*, Bournemouth: Bournemouth University.
- Robert Bruce, J., 2006. *Real-Time Motion Planning and Safe Navigation in Dynamic Multi-Robot Environments*, Pittsburgh: School of Computer Science, Carnegie Mellon University.

Saltvik, I., Cathrine Elster, A. & Nagel, R. N., 2006. *Parallel Methods for Real-Time Visualization of Snow*. Trondheim, Norwegian University of Science and Technology.

Stomakhin, A. et al., 2013. A material point method for snow simulation. *ACM Transactions on Graphics*, 32(4), pp. 102:1 - 102:9.

Wang, N. & Wade, B., 2004. Rendering falling rain and snow. *SIGGRAPH '04*, Issue 2004, p. 14.

White, G., 2014. *Particle Systems for Weather Simulation*, Dublin: Undergraduate Library.

# Appendices

## Appendix A

Table data showing average frames per second as well as average particle simulation time when changing the number of particles in the scene.

Particle Count	Avg Particle Sim Time (ms)	Avg Frames Per Second
10000	0	686.487
20000	0	637.819
30000	0.0496	500.089
40000	0.299827	465.717
50000	1	455.919
60000	1.00307	390.298
70000	1.01142	368.49
80000	1.05475	345.122
90000	1.88644	309.034
100000	2	301.247
110000	2	289.197
120000	2.04058	260.57
130000	2.12717	253.357
140000	2.42869	241.55
150000	3	225.696
160000	3	222.28
170000	3.10661	202.337
180000	3.31149	198.32
190000	3.44099	193.615
200000	4	181.103
210000	4	178.789
220000	4.03691	173.597
230000	4.30025	163.273
240000	4.41927	160.251
250000	5	155.924
260000	5.02663	150.053
270000	5.07297	148.048
280000	5.12062	142.695
290000	5.8695	136.213
300000	6	134.194
310000	6	132.018
320000	6.04252	127.11
330000	6.13259	125.377
340000	6.92424	118.77
350000	7	117.287
360000	7	116.378

370000	7.11011	110.732
380000	7.39599	109.638
390000	7.61111	108.2
400000	8	103.94
410000	8	103.5
420000	8.02772	101.032
430000	8.44828	98.6041
440000	8.58932	97.4698
450000	9	95.042
460000	9.00645	93.0264
470000	9.06565	91.5056
480000	9.53258	89.0081
490000	9.8246	87.8702
500000	10	87.0325
510000	10.0095	84.3523
520000	10.0911	83.4129
530000	10.3601	82.1208
540000	10.9501	80.1622
550000	11	79.7408
560000	11	78.0951
570000	11.0602	76.463
580000	11.3931	75.7659
590000	11.8043	74.5924
600000	12	73.3694
610000	12	72.5899
620000	12.262	70.9853
630000	12.6553	70.3056
640000	12.8127	69.4322
650000	13	67.8879
660000	13.006	67.2411
670000	13.0721	66.5414
680000	13.9713	62.9798
690000	13.8978	64.6484
700000	14	64.0603
710000	14.0382	62.7822
720000	14.1543	62.184
730000	14.7143	61.5595
740000	14.9868	60.4675
750000	15	60.0224
760000	15	59.4243
770000	15.099	58.5295
780000	15.6678	57.7857
790000	15.986	56.9418
800000	16	56.5405

810000	16	55.9218
820000	16.4218	54.9518
830000	16.7656	54.5481
840000	16.9852	54.2069
850000	17	53.2026
860000	17	52.9988
870000	17.0496	52.4
880000	17.9109	51.5835
890000	17.965	51.1929
900000	18	50.6199
910000	18.004	50.0821
920000	18.1526	49.8006
930000	18.9795	48.9075
940000	19	48.6464
950000	19	48.3145
960000	19	47.4984
970000	19.1392	47.3666
980000	19.9615	46.8638
990000	20	46.2802
1000000	20	46.0061

## Appendix B

Doxygen Documentation

<https://alasdairh.github.io/SnopenGL/>

