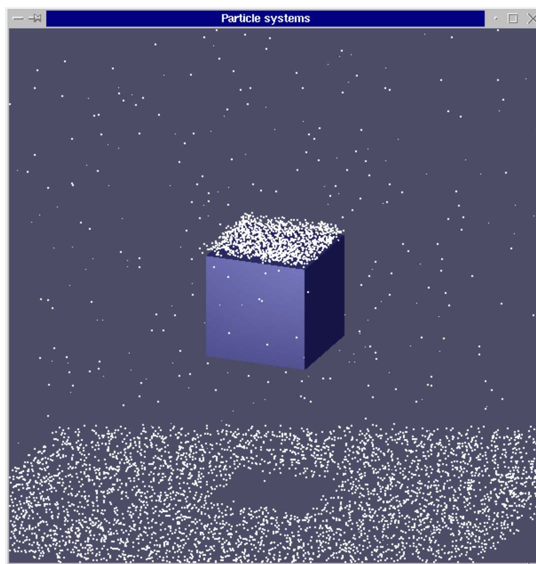
An Investigation of Snow Effects to Enhance the Graphics in Computer Games



Submitted in partial fulfilment of the requirements

for the Degree of

Bachelor of Science (Honours)

of Rhodes University

by

Deborah Michelle Patrick

November 2000

ABSTRACT

This project investigates how feasible it is to model and render snow in a three-dimensional environment, which could be added to a computer game to enhance its graphics. Falling snow and its collection are designed using particle systems, collision detection and surface modelling. We implement falling snow that appears relatively realistic using OpenGL and a particle system API. The feasibility of adding snow to a computer game is measured by testing the effect adding snow to a three-dimensional environment has on the frame rate. We show that it is possible to add falling snow to a computer game. However, we are not able to implement the collecting of snow so that it appears realistic, and therefore do not test whether is can be feasibly added to a computer game.

ACKNOWLEDGEMENTS

To Shaun Bangay, my supervisor - Thank you for all your support, patience and assistance.

To my friends in Calnet - Thank you for all the good times and many hours we spent in the lab.

To the staff in the Computer Science Department - Thank you for your help.

To my parents – Thank you for making a degree at Rhodes possible.

To all the people not mentioned – A big thank you!

CONTENTS

CHAPTER 1 - INTRODUCTION **7**

CHAPTER 2 - LITERATURE SURVEY **9**

2.1 Properties of Snow 9
2.2 Particle Systems 13
2.3 Collision Detection 18
2.1 Surface Modelling 23
2.2 Summary 27

CHAPTER 3 - DESIGN **28**

3.1 Falling Snow 28
2.3 The Collection of Snow 32
2.4 Testing the Frame Rate 37
2.5 Summary 39

CHAPTER 4 - IMPLEMENTATION **40**

2.6 Hardware and Software 40
2.7 Falling Snow 42
2.8 The Collection of Snow 44
2.9 Summary 44

CHAPTER 5 - TESTS AND RESULTS **45**

5.1 The Frame Rate 45
2.10 Falling Snow 47

2.11 Summary 51

CHAPTER 6 - CONCLUSION **52**

6.1 Future Work 52

CHAPTER 7 - REFERENCES **53**

APPENDIX A - SCREENSHOTS **55**

APPENDIX B - CODE **57**

LIST OF FIGURES

Figure 1: A design of falling snow 29
Figure 2: A design of the collection of snow 36
Figure 3: The frame rates for different window sizes 47
Figure 4: The frame rate for different point sizes 48
Figure 5: The frame rate for different rates of particle creation 49
Figure 6: The frame rates for different numbers of particle groups 50

CHAPTER 1

INTRODUCTION

The purpose of this project is to investigate the modelling and rendering of snow in a three-dimensional environment, which could be used to enhance the graphics in a computer game. Due to advances in information technology and computer graphics in recent years, there has been an increase in the popularity of computer games. Standard desktop computers are now powerful enough to produce impressive graphics and render virtual environments that appear almost realistic. The popularity of a computer game is largely influenced by the quality of its graphics and the convincing nature of its environment [10]. Although there are countless computer games available, which contain realistic scenes of different genre from space to underground subways, only a small percentage of these games contain fallen snow, and even less contain falling snow.

This project investigates snow, for two reasons. Firstly, snow alters the appearance and mood of a scene, and could benefit the appearance of a three-dimensional virtual environment. In many parts of the world snow is a common phenomenon during the winter months, and to those who do not experience it during winter, it is a fascinating, exciting novelty. Including a phenomenon that is so common or exciting into a computer game should make the game so much more realistic or enjoyable. Secondly, snow is an example of a complex natural phenomenon with a poorly defined surface. The traditional way of modelling objects in computer graphics is to describe the objects in terms of numerous polygons. Unfortunately, because of their irregular surfaces, this method is not appropriate for modelling snow or other natural phenomena. This project examines a method that makes use of particle systems to represent snow. If we can prove that particle systems are able to effectively and efficiently model snow that can be added to an environment in a computer game, then it should be possible to model and add other natural phenomena, such as rain, as well [7].

For a computer game to be successful, the players must feel as if they are immersed in the environment and part of the game. To achieve player immersion the environment should be

realistic (or convincing) and the frame rate high. For a realistic environment, the objects should not only look realistic, but also obey many of the laws of physics. For instance, snowflakes should not pass through solid objects. This is dealt with using collision detection, which can be computationally expensive. The rendering speed, measured by frame rate, should be high (approximately 20 frames per second (fps)). A game that runs at a frame rate that is too low will appear jerky and be unpleasant for the player. Unfortunately, visual quality, and rendering speed contradict one another, and so tradeoffs have to be made [10].

This project investigates the modelling and rendering of snow in a three-dimensional environment, by looking at how feasibly and realistically snow can be modelled, and what affect adding snow to an environment has on the frame rate. The remainder of this project is structured as follows. Chapter 2 looks at literature related to this project. The properties of snow are examined, as well as particle systems, collision detection, and surface modelling. Chapters 3 and 4 discuss the design and implementation of falling snow and its collection. Chapter 5 tests the frame rate and discusses the results obtained, and Chapter 6 gives a final conclusion with possible extensions and improvements.

CHAPTER 2

LITERATURE SURVEY

There are several important issues to consider when modelling snow. This chapter examines work related to these issues. The chapter is divided into four sections. The first section looks at the properties of snow. It is easier to investigate the modelling of realistic looking snow that obeys the laws of physics if the properties of snow are known. The next section examines research relating to particle systems. The use of particle systems is currently the most effective way to model natural phenomena such as snow. The last two chapters, before the summary, explore literature on collision detection and surface modelling, which are two essential, yet problematic, issues that need to be considered when modelling snow. When snowflakes fall, their collision with a solid surface needs to be detected. The collision of snowflakes with solid surfaces results in a build up of snow that forms a surface. This surface must somehow be modelled in computer graphics.

2.1 PROPERTIES OF SNOW

Previous work has been done on the modelling of snow. Most of this work focuses on modelling fallen snow as realistically as possible, in which case achieving a high frame rate is not of importance. In this project we are not only interested in fallen snow, but falling snow as well, and because this project is concerned with adding snow to a computer game, the frame rate is of large importance. However, because most of the previous work focuses on producing realistic and relatively accurate fallen snow, the properties of snow are highlighted. We shall therefore discuss the properties of snow by examining this previous work.

2.1.1 A NATURAL PHENOMENON

Snow is a natural phenomenon, and therefore possesses characteristics of objects created by nature. Most objects created by nature have irregular or poorly defined surfaces. They have a

dynamic form that may change between the states of gas, liquid and solid, and they are often unpredictable. In computer graphics, these characteristics make the modelling of natural phenomena a challenging task, and the modelling of snow is no exception. Snow can appear to have a smooth or rough surface, be hard or soft, and behave in a predictable or unpredictable manner. As Fearing states "When snow moves, it can travel like a single handful of flour or an acre of solid concrete" [7].

In nature, snow is made up of a large number of snowflakes. Each individual snowflake is a unique crystal of varying size and shape, which moves and bonds with other crystals to produce the overall effect of snow. The appearance of snow varies at different distances. From far away snow appears to be a smooth white layer. Its colour and texture are consistent, and appear to be relatively simple to model in computer graphics. From close up snow appears to have a complex texture and colour scheme, and is not as simple to model [11].

Law, Oh and Zalesky accomplish the modelling and rendering of snow by modifying a simple ray-tracing program. As in nature, their snow consists of a large number of snowflakes, and the surface snowflakes largely determine the snow's appearance. Each snowflake represents a random volume of the snow, which is modelled using an algorithm similar to Perlin's 3D noise function. The normal of each point on the surface of the snow is calculated by using the normal of the snowflake at that point. A parameter in the noise function determines whether or not a point on the surface contributes to the specular lighting of the snow. If it does, then the normal calculated at the point is used to determine the amount of specular lighting the point contributes to. In their model, Law *et al*, focus largely on three main factors that affect snow. These factors are wind, temperature, and the surface the snow falls onto [11].

2.1.2 WIND, TEMPERATURE, AND OTHER FACTORS THAT AFFECT SNOW

In nature, snowfall is affected by many factors. Some of these factors include: the amount of moisture in the air, the density of the clouds, and the time of year [11]. More obvious factors include: temperature, wind velocity, gravity and the surface the snow lands on. Because snow is a complex natural phenomenon, affected by many factors (some of which are complex natural phenomena themselves), it is almost impossible to model snow accurately, taking every factor into account. Therefore, simplified models of snow have to be produced. These models should appear realistic, and correctly implement the most obvious factors that affect snow.

Law *et al* implement a simplified model of snow on a terrain. Two of the factors they take into account are temperature and wind. The temperatures of the atmosphere and surface on which the snow falls affect the size and shape of individual snowflakes. They also affect the rate at which snow melts. Wind affects the path of a falling snowflake, and is the main factor affecting the transport of snow. In nature the altitude at which snow falls determines the temperatures of the atmosphere and surface. Law *et al* determine the rate of melting snow by inspecting the altitude of the terrain in their model. Unfortunately, they do not try to create wind (because it is a complex natural phenomenon that is not easy to describe). Instead, they use a function that helps to produce wind. This function includes some randomness, imitating the unpredictability of wind [11].

To produce fallen snow on a terrain, Law *et al* estimate the amount of snow that would collect at various points on the terrain. This is done taking into account some of the factors that affect snow. The amounts obtained are then linearly interpolated to produce a surface of snow [11].

2.1.3 THE COLLECTION OF SNOW ON A SURFACE

In nature, when snow falls it will collect on all parts of a surface that are directly exposed to the sky. Some snow will also collect under objects, on parts of surfaces that are not directly exposed to the sky. In computer graphics, when modelling the collection of snow on a surface, one must make sure that snow collects evenly on all parts of surfaces that should receive snow. Randomly falling snow particles must not only collide with large surfaces, but very small ones as well [7]. If the snow in the model falls in a vertically straight line, a technique must be devised that allows some snow to collect on surfaces under objects as well.

The modelling of fallen snow can be looked at from two different angles. You can construct the collection of snow by attaching randomly falling snow particles to the surface with which they collide. Here the falling snow and its collection are dependent. The collection of snow is automatic and depends on the falling snow and its collisions with surfaces. Otherwise, you can construct the collection of snow by examining all surfaces and their exposure to the environment's sky. A collection of snow is then modelled on all surfaces where snow in nature would collect. In this method the collection of snow is independent from the falling snow.

Fearing looks at modelling the collection of snow, by examining which surfaces falling snow could land on. A pattern of fallen snow is generated for each surface. This pattern is generated by sending up a series of particles from various points on the surfaces, towards a plane representing the sky. As the particles move upwards, tests are performed to check for intersections with an object. If a particle is found to intersect with an object, it does not contribute to the snow on the surface it started on. A particle that reaches the "sky" without intersecting with an object does contribute to the snow on the surface it started on. Fearing's reason for sending particles upwards rather than dropping particles down arises from "the need for control" [7]. The amount and shape of snow modelled on each surface can be controlled, and does not depend on randomly falling particles colliding with its surface. After certain criteria (such as the size of the sample and the amount of computing time available) have been met for generating the fallen snow pattern, the appropriate amount of snow for each surface is calculated, and three-dimensional snow is formed. This new snow (represented as a number of particles) then has to undergo a stability test. If the snow is unstable, it is moved. Moving snow may cause other snow that was once stable, to become unstable. Therefore stability tests have to be performed several times, causing computation time to become large. Once the stability tests have been performed the three-dimensional snow surface represented as a set of polygons can be generated [7].

Although Fearing's method produces realistic looking snow, it would not be practical to use it in a computer game. It is too slow, due to all the tests that have to be performed to check whether points on a surface should receive snow, and whether snow calculated is stable. It also does not take falling snow into consideration. However it does use two important properties of fallen snow, which should be discussed. These are "flake flutter" and "flake dusting" [7].

2.1.3.1 Flake flutter

Flake flutter occurs when snow does not fall in a straight line and lands underneath an object on a surface that is not directly exposed to the sky. Wind and the crystal shape of the snow are properties that cause flake flutter. The flake flutter effect produces a smooth curve of snow between surfaces that are directly exposed to the sky, and those that not. The amount of flake flutter that occurs, and the shape of the curve it produces depends on the size and shape of the object it falls underneath, how close the object is to the surface the snow falls on, the amount of falling snow, and the properties of wind and crystal shape that cause the flake flutter [7].

2.1.3.2 Flake Dusting

Flake dusting occurs when the snow does not cover the entire surface it has landed on. Instead the snow appears as thin powder of snowflakes on the surface. Modelling this powder of snow as three-dimensional objects is not practical. Fearing represents this powder as partially transparent textured polygons positioned slightly away from the surface they appear on [7].

2.1.4 INTERACTION AND OTHER EFFECTS

Although this project is only concerned with falling snow and its collection, other properties of snow that would enhance a computer game, are formations of avalanches and ice crystals. It would also be interesting if characters and objects in a game could interact with the snow, such as pick up the snow for a snow fight. Some interesting effects have been implemented in current games. For instance footprints appear in the snow in Commandos (but no snow falls in this game) [4].

2.2 PARTICLE SYSTEMS

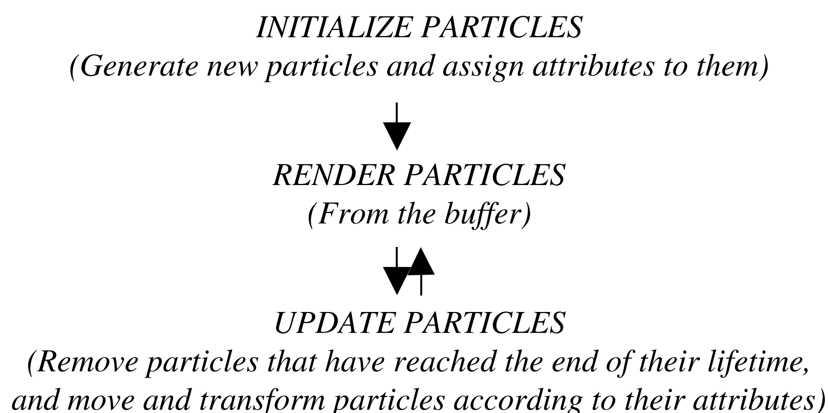
Particle systems first came about in 1983, when William T. Reeves published his paper "Particle Systems - A Technique for Modeling a Class of Fuzzy Objects." [15] Until then, the only way of modelling objects in computer graphics was to represent the objects as a list of three-dimensional points, or vertices, grouped as polygons. In this section we discuss particle systems, and why they are the best available method for modelling snow.

2.2.1 WHAT IS A PARTICLE SYSTEM?

A particle system is a collection of many small particles that model an object in computer graphics [14]. A particle system differs from the traditional way of modelling objects, in that the object's volume rather than its surface is modelled [15]. Each particle contains certain attributes, which describe the position and characteristics of the particle. These attributes are chosen depending on the object the particle is modelling. Particles were first used in the modelling of fire, in the Genesis sequence from the film Star Trek II: The Wrath of Khan. In the film the particles are represented as independent volumes of light moving and changing in three-dimensional space [15]. Since then, particle systems have been used to model many other objects. Reeves has used particle systems to model trees and grass. The particles used to model trees and grass differ from those used to model fire. Apart from the differences in appearance and movement, trees and grass are more structured than fire; consequently the particles used are not independent of one another [15]. Particles used to model falling snow could be independent points like the particles used to model fire, while particles used to model a surface of fallen snow might need to be dependent.

2.2.1.1 The basic steps of a particle system

A particle system program has these basic steps:



[13]

2.2.1.2 Particle attributes

A particle's attributes are properties such as colour, position, velocity and age. These properties can be changed over time, making the particle dynamic [15]. Because the particles are dynamic they are able to model dynamic objects such as liquids, whose form changes and moves over time. Using particles to represent snowflakes we can model snow with realistic properties, such as decrease a snowflake in size or allow it to totally disappear due to melting from the sun [13].

Often particles behave in a similar manner, and are then generated and transformed as a group [13]. These particles may have attributes that are the same, or that are different. The generation of particles and the assigning of their attributes are done using stochastic methods. Stochastic methods are used as they have a certain amount of randomness associated with them, imitating the randomness of nature [14]. The equation usually used to calculate the attributes of individual particles is:

$$ParticleAttribute = AttributeMean + Rand() * AttributeVariance$$

Where *ParticleAttribute* is the attribute of the individual particle, *AttributeMean* and *AttributeVariance* are the mean and variance of the attribute for particles of a certain group and *Rand()* is some random element [9].

2.2.2 ADVANTAGES AND DISADVANTAGES OF PARTICLE SYSTEMS

By examining the advantages and disadvantages of particle systems, we can decide whether particle systems are more suitable, than traditional polygon based methods, for modelling snow.

2.2.2.1 The advantages of using particle systems

The two main advantages of particle systems are that they can generate large amounts of detail and model objects with a dynamic form. According to Reeves a particle is a lot simpler than most geometric primitives[1]. If this is so (in other words the particles are represented as points), then it is possible to render more particles, than geometric primitives, in a given amount of time. Thus, producing a more detailed representation of an object [15]. Particle systems are also "procedural" and stochastic, so very little effort is needed to model and render complex objects

[1] Geometric primitives are points, lines, and polygons

[14]. The characteristics and position of particles in a particle system can be changed to allow it to represent dynamic objects, and the level of detail can be easily adjusted. Therefore moving snow (for example snow that is falling, avalanching, or melting) can be modelled, and snow in the distance can be modelled in less detail than snow close by [14].

Since particle systems are able to produce a large amount of irregular three-dimensional detail with very little effort, they are the more appealing method for modelling natural phenomena such as snow. Unfortunately, because they produce so much irregular detail, exact "visible surface" and shading calculations become infeasible [15].

2.2.2.2 The disadvantages of using particle systems

Reeves' initial paper on particle systems ignores visible surface and shading problems. The particles in the paper represent fire, modelled as individual light sources [15]. When particles overlap in a pixel, their colours are simply added together. Shading is easy, due to each particle being an independent light source. Unfortunately, particle systems are sometimes used to model objects that are more complex than fire. The colours of the particles representing these objects cannot simply be added together, and the objects require a more complex shading model [15].

Later research by Reeves attempts to fix the problem of shading. Trees and grass are generated using a probabilistic shading model. Because a tree consists of millions of independent particles, it is difficult and time consuming to shade each individual particle accurately, calculating whether it is in shadow. Instead Reeves determines the probability of a particle being in shadow, and then uses a random number to decide whether or not to render the particle if it were in shadow [15].

When modelling snow, the visible surface problem is of no concern. The snow particles are white, and white added to white produces white. Unfortunately shading is a problem, especially when modelling the collection of snow as a surface. Producing a white surface with no shadows does not look realistic. It might therefore be advantageous to model falling snow and the collection of snow separately - using a particle system for the falling snow and a traditional method for the surface.

2.2.3. MODELLING SNOW

Several techniques have been applied to model snow with particle systems. Each particle in a particle system represents a snowflake in snow, and has certain attributes that cause it to behave like a snowflake.

Sims uses white particles, spirals and vortices to generate a snowstorm. The particles are dropped from what appears to be the top of the window. Each particle has an initial velocity and slightly random vertical spiral axis. To obtain the appearance of collecting snow, particles are attached to a plane after colliding with it. This is achieved by bouncing the particles off the plane with zero bounce and high friction. Sims does not consider gravity or air friction, because they cancel one another out [18]. Guan and He also use white particles to produce snow. To generate continuously falling snow, they re-initialise the velocity and position of a particle (to the top of the window) after it has collided with a surface. In their model they produce wind using a sine wave [9].

When modelling snow for a computer game, as few snowflake particles as possible should be rendered. Performance is important in computer games, and according to Crawford, Juliano, Larsen and Lok, one way of increasing the performance is to reduce the rendering workload by decreasing the number of particles rendered. The number of particles can be reduced using view frustum culling [5], or more complex particles [13]. Using view frustum culling, not all the particles in the entire environment are rendered. Instead only the first few layers of particles close to the viewer are rendered [5]. More complex particles may allow you to use fewer particles, which achieve the same visual effect. Motion blurring, fog and lighting are also suggested to make the falling snow more convincing [13].

Although view frustum culling may be used, it is important that when producing snow, the entire world is considered, and not just a section [13]. It would not be very realistic if a player moved across the world in a game, and there was suddenly no snow.

2.2.4 PARALLEL COMPUTING

Usually thousands, and sometimes millions, of particles are used in a particle system. To render these particles as efficiently as possible, Sims uses parallel possessing techniques. Since a particle system usually consists of numerous particles that are very similar, particle systems lend themselves well to parallel processing. Using parallel processing, it is possible to control all the particles in a group with one or two commands. This is more efficient than issuing the same commands to each particle individually [18].

To implement parallel processing, Sims represents each particle by a virtual processor. This virtual processor contains a data structure with the particle's state variables. When particles are created their state variables can be assigned new values or copies of state values from existing particles. When a particle's lifetime is over, it is removed from its processor, to make space for new particles. The movement of each particle is controlled by the position, velocity and acceleration of the particle. Acceleration operations include damping, spiral, and bouncing off planes or spheres [18].

Sims focuses on generating realistic images rather than physically accurate models. He uses Euler's method of integration to update the state of the particles. This method is simple and fast but not extremely accurate. Fortunately, because there are so many particles, the small inaccuracies that occur are not noticed, and the overall appearance is convincing [18].

2.3 COLLISION DETECTION

The majority of computer games today, require a high amount of realism in order to be successful. It is therefore important that the collisions between solid objects be accurately detected (and responded to). Players will not be happy if their characters move through walls, or die when a bullet does not actually hit them [1]. When adding snow to a game, it is important that the snowflakes collect on the surface they land, and that they do not pass through it.

Most computer games require collision detection and much research has been done on producing fast, accurate collision detection algorithms. Not only does the collision detection algorithm need to accurately detect a collision, but it also needs to be extremely fast. Collision detection can be a

slow process, and an algorithm that takes too long will decrease the frame rate, making a game unpleasant for a player.

2.3.1 COLLISION DETECTION METHODS

There are several available methods for detecting collisions. Most of these methods are, as Dave Roberts declares, "either pixel or boundary based" [16].

2.3.1.1 Pixel based collision detection methods

To detect a collision between two objects, tests are performed to see if the objects intersect (or are about to intersect). If an intersection is detected, appropriate action is taken[2]. If a pixel based collision detection method is used, the tests performed check for an intersection, by testing whether any vertices of the two objects themselves overlap. In other words there is a pixel (or pixels) representing at least one point from both of the colliding objects. Pixel-based methods are very accurate, but require a large amount of testing. They are therefore slow [16].

2.3.1.2 Boundary based collision detection methods

Boundary methods solve the speed problem by only requiring a few simple tests to be performed when checking if two objects intersect (or are about to intersect). This is done by enclosing complex objects in simple geometric shapes, and then testing for collisions between the shapes. These shapes are known as bounding volumes and are usually spheres or rectangular boxes, which are easy to use when testing for collisions. Unfortunately, this method is not always accurate. Unless the shape of the object is almost the same as its bounding volume, there will be parts of the bounding volume that are not part of the object. If these parts of two bounding volumes intersect, a collision of the objects will be indicated, even though the objects themselves do not collide [16].

The shape of the bounding volume enclosing an object depends on the shape of the object. Spheres are easy to use, but boxes usually fit the object better. There are two kinds of bounding

[2] This action depends on the objects colliding. They could explode, attach themselves to one another, or disappear for instance.

boxes. These are axis-aligned bounding boxes (AABBs) and oriented bounding boxes (OBBs) [2].

2.3.1.2.1 *Axis-aligned bounding boxes (AABBs)*

All AABBs in an environment are aligned with a coordinate system, so that each of the boxes' faces is perpendicular to one of the coordinate system's axes [2]. When objects rotate, their AABBs do not rotate with them, instead the AABB is recalculated to enclose the object in its new orientation. According to an article by Nick Bobic, in the May 1999 issue of Game Developer, this computation is not slow or difficult, however this method may often cause inaccurate collision detections, since the boxes do not fit the objects as tightly as they can [2].

2.3.1.2.2 *Oriented bounding boxes (OBBs)*

OBBs are boxes that do rotate with the object they are enclosing. These boxes produce more accurate collision detection results, but according to Nick Bobic, are a lot more difficult to implement, slow, and unsuitable for objects whose shape changes. Choosing whether to use AABBs or OBBs therefore depends on whether speed and easy implementation, or accuracy is more important [2].

2.3.1.3 Snowflake collision detections

Adding falling snow to a computer game requires detecting when each snowflake hits the surface it lands on. Enclosing the snowflakes in bounding volumes is pointless, since the snowflakes are usually represented as simple points. The only way of performing collision detection is to check the collision of each individual snowflake, represented as a point. Snowflakes are therefore pixel based. Since most computer games require collision detection for the characters or objects in their scenes, most computer games have some sort of collision detection implemented. Usually the characters or objects are enclosed with bounding volumes. To test collisions of snow particles with an object, we have to test whether or not the snowflake point lies in the bounding volume. How accurately a bounding volume is fitted to an object and what we do with the snowflake after a collision has been detected is important. We can either allow the falling snow and its collection to be modelled dependently, attaching the snowflake to the surface it collides

with. Otherwise we can model the falling snow and its collection independently, destroying a snowflake when it collides with a surface, and independently building up snow on all possible surfaces the snowflakes might land. If the bounding volumes do not enclose the objects accurately, attaching a particle to the bounding volume it lands on may result in a snowflake that appears to have stopped in the middle of the air. One way of solving this problem it to use a hierarchy and subdivide bounding volumes into smaller bounding volumes.

Although bounding volumes decrease the complexity of tests, and therefore the time, they do not decrease the number of objects tested against each other. In our snow model we do not want to test every snowflake against every other snowflake. Methods must therefore be devised to eliminate unnecessary collision tests between objects [6].

2.3.2 ELIMINATING COLLISION TESTS

In many computer games today, each scene contains several moving objects. The easiest way to check for collisions is to test each object against every other object [6]. Unfortunately, the more elements a scene contains, the more inefficient this method becomes. This is because the number of tests performed is proportional to the number of objects, squared [3]. The formula for the number of tests performed for n objects is:

$$(n^2 - n)/2$$

As n increases, the number of tests performed, and therefore the amount of time needed for testing collisions, increase in approximately the same way as n^2. In other words collision detection between objects has $O(n^2)$. The more objects you have the worse the efficiency of the collision detection gets [16].

One way of making collision detection more efficient is to decrease the number of objects tested with one another for collisions [6]. Techniques for reducing the number of collision tests include rule elimination and eliminations based on spatial position [16].

2.3.2.1 Rule elimination

Sometimes objects in an environment cannot collide with each other, and it is therefore unnecessary to test them for a collision [1]. For instance, two objects that are stationary will never collide with each other. Testing whether two falling snowflakes in a computer game collide with each other is also unnecessary. In a computer game, the rules of the game usually determine which objects in the game are allowed to collide. Therefore, only testing those objects that according to the game rules are allowed to collide can reduce the number of collision tests. In some games, the rules cannot decrease the number of collision tests performed. In these games spatial-test elimination techniques should be used [16].

2.3.2.2 Spatial-test elimination

Using spatial-test elimination methods, only objects close to one another are tested for collisions. Spatial-test methods can be divided into two groups. The first group sorts and compares objects according to their position in the environment, while the second group subdivides the environment up into regions, and assigns one or more regions to each object. Objects in the same (and sometimes neighbouring) regions are then tested for collisions. Both groups require additional time and space to sort, compare, or assign regions to the objects in the environment. They should therefore only be used if the amount of time they save eliminating collision tests is more than the amount of time they spend sorting, comparing or assigning. According to Roberts a spatial-test elimination method is more appropriate for games that contain many objects. In games that only contain a few objects, this method uses more time than it saves, as there are not many collision tests that can be eliminated [16].

Three common methods for sub-dividing an environment up into regions are: Octrees, BSP trees, and Uniform Spatial Division [6].

2.3.2.2.1 ***Octrees***

This method uses planes parallel to the x-, y- and z-axis, to divide the environment up into cubic regions known as voxels. Regions that contain many objects are subdivided further into smaller regions [6].

2.3.2.2.2 *Binary Space Partitioning (BSP) trees*

In this method an n-dimensional environment is divided into two regions by an n-1 dimensional plane. Each object in the environment is then assigned to one of the two regions depending on its position in the environment [6]. This method of sub-dividing the environment has been used in various computer games. Doom was the first commercial game to use BSP trees. Other games that use BSP trees are Quake II, Unreal, and Lithtech [2].

2.3.2.2.3 *Uniform Spatial Division*

Using Uniform Spatial Division, the environment is divided into a finite number of equally sized blocks, called a grid [17]. The block that each object in the environment is located in is then determined, and collision tests are carried out among objects in the same block [16]. Sometimes objects in neighbouring blocks are also tested for collisions. Testing neighbouring blocks is usually done if the blocks in the grid are very small [3]. If the objects in the environment are relatively evenly spaced, each block will contain only a few objects, or none at all [16]. If the objects are not evenly spaced and moving around, dividing the environment into a useful grid may be a difficult task [3].

If an object is located in more than one block, it is usually included in all the blocks it is located in. It is therefore better to use blocks that are bigger than the largest object in the environment. Objects smaller than a block can only be located in one, two, or four blocks (at a time), while objects bigger than a block require extra calculations to determine the blocks they are located in [16].

The blocks can be stored in two ways: using an array or a linked list. Arrays are faster than linked lists, but allocate space for a fixed number of objects. Therefore, if there is no way of knowing the maximum number of objects that may occur in a block, linked lists are better to use [3].

Dividing a three-dimensional environment into an effective grid is difficult, and requires keeping track of a large number of blocks (many of which don't contain objects). Uniform Spatial Division is therefore more appropriate for two-dimensional environments [3].

2.4 SURFACE MODELLING

When snowflakes fall, they collect to form a layer of snow on the surface on which they fall. Various methods can be used to model this layer in computer graphics. One method represents the layer of snow as a three-dimensional surface. This section discusses various ways of modelling the collection of snow, focusing mainly on the different ways the collection of snow can be modelled using a three-dimensional surface.

2.4.1 USING THE FALLING SNOWFLAKES

The most natural way to model the collection of snow would be to collect the falling snowflakes as they land on a surface. Two ways of doing this are to assign a zero velocity to the snowflake particles as they collide with the surface [13], or to bounce the snowflake particles off the surface with a high friction, so that they stick to the surface when they collide with it [18]. The problem with these methods is that the number of particles rendered each frame gets infinitely large. Increasing the number of particles decreases the frame rate. There is also a limit to the number of particles a computer can keep track of. A way of solving this problem is to generate the surfaces that the snowflakes land on as textured surfaces [13].

2.4.2 TEXTURED SURFACES

Using textured surfaces to produce fallen snow, keeps the number of particles in a particle system relatively constant. When snowflake particles collide with a surface, they are removed from the particle system and included in a texture map. This texture map is then used to render the surface. Because snowflake particles collide with surfaces each frame, the texture map must be updated each frame. To do this, the particles that need to be added to the texture map at the end of a frame are orthographically projected onto a clear textured surface. This textured surface together with the current texture map, produce the texture map for the next frame [13].

Using a texture map for collided snow particles is an efficient and effective way of rendering the initial collection of snow on a surface where snow has not yet fallen. Unfortunately it is not an effective method for rendering the continued collection of snow, as it only produces one layer of snow. To generate a continued collection of snow, this method must be improved [13].

2.4.3 THREE-DIMENSIONAL SURFACES

Three-dimensional surfaces could be used to represent the collection of snow. There are many ways to produce a three-dimensional surface, and in this section we examine some of these ways. A surface that is not only texture mapped, but whose form also changes each frame, could improve the textured surfaces method for modelling collecting snow.

2.4.3.1 Polygon Meshes

A surface can be represented as a set of flat polygons that are joined together. This representation is known as a polygon mesh. It is relatively simple to produce, but is only an approximation of the surface it represents. The more polygons the mesh consists of, the more accurately it represents the surface, but the more space it requires and the more time that is needed to render the mesh, therefore decreasing the frame rate [8]. This project investigates the use of a mesh for the creation of a surface of fallen snow. As the snow falls the polygons in the mesh rise imitating the collecting of snow.

There are several ways of representing a mesh. Time and space are the deciding factors as to which representation to use. Foley, van Dam, Feiner and Hughes look at three of these ways, namely "explicit", "pointers to a vertex list", and "pointers to an edge list" [8].

2.4.3.1.1 *Explicit*

In the explicit representation, the coordinates of the vertices that represent each polygon are stored in a list (in the order in which they would be encountered when moving around the edge of each polygon). This method does not make efficient use of space, because the coordinates of vertices that represent more than one polygon are listed more than once. Also, the edges that

represent more than one polygon have to be drawn twice when the mesh is rendered. This makes the drawing of the mesh slower [8].

2.4.3.1.2 *Pointers to a Vertex List*

Using pointers to a vertex list, each polygon is represented by a list of pointers to a "vertex list" (or a list of indices to a vertex array). Here the coordinates of each vertex in the mesh are stored only once. Because the coordinates of the vertices are stored only once, this method is a better choice than the explicit method for two reasons. Firstly the method uses less space, and secondly it is easier to change the values of coordinates because they only have to be changed once in the list. Unfortunately, when the mesh is rendered, edges that represent more than one polygon still have to be drawn twice [8].

2.4.3.1.3 *Pointers to an Edge List*

Using pointers to an edge list solves the problem of drawing some edges more than once. Each polygon in the mesh is represented by a list of pointers to an "edge list". Each edge in the edge list points to two vertices in a vertex list that represent it, as well as to the one or two polygons to which it belongs. The polygon mesh is then rendered using the edges, rather than the polygons. This method is faster than the pointers to a vertex list and explicit methods, but is more complex, as it requires more lists. Because each edge is only drawn and listed once, according to Foley *et al*, "redundant clipping, transformations, and scan conversion are avoided" [8].

Although polygon meshes are useful for representing objects with flat surfaces (or objects that are made up of many flat pieces), they are not suitable for representing objects with curved surfaces. To create a reasonably realistic curved surface using polygons requires thousands of polygons and coordinates to be created, stored and maintained [8]. A more appropriate method for modelling curved surfaces is to use splines. Although the surface is still only approximated, splines require less space and provide an easier way to produce a more accurate description of the surface [20].

2.4.3.2 Splines

Splines are mathematical descriptions of curved surfaces. Instead of representing curved surfaces with many small polygons, it is possible to describe many surfaces using mathematical functions and a few "control points" [20]. The control points are able to accurately define a surface, which results in splines having two advantages over polygons. They require less space, as only a few control points need to be stored, and they are a more accurate approximation of a surface, because the control points accurately define the surface [20]. By moving the control points, the shape of the surface can be changed [19].

In OpenGL, "evaluators" can be used to describe a surface (of any degree) with control points. The accuracy of the curve rendered can be specified, and the normal vectors for the surface are calculated automatically. Because evaluators are a "low-level description of the points on a surface", OpenGL provides a "higher-level interface". This interface is the "NURBS" facility [20].

One disadvantage of using splines is that the shape of an object must be known and specified (using a complicated mathematical function), before the surface can be generated. This decreases the uses of splines. Szeliski and Tonnesen investigate the use of particle systems to overcome the disadvantages of splines [19].

2.4.3.3 Particle Systems

Szeliski *et al* model surfaces using an "oriented" particle system, which unlike splines can "split, join, or extend" surfaces automatically. Their particle system is also able to model surfaces by interpolating a set of three-dimensional points. This is useful for producing surfaces where the mathematical shape of the surface is not known in advance. Surfaces with or without holes can also be created. To create or transform a surface in the above-mentioned ways, individual particles are added, deleted or moved in the particle system (sometimes automatically) [19].

Particles in a particle system tend to arrange themselves as a volume rather than a surface. To force the particles in their system to arrange themselves as a surface, Szeliski *et al* use a set of "potential" functions. The particles in their system have forces that cause them to attract one another when they are far away, and then repel one another as soon as they get too close. In order

to model the particles as surface elements rather than small volumes, each particle' sstate contains an orientation, and thus the particles are called "oriented particles" [19].

To render oriented particles, Szeliski *et al* use simple icons such as axes or flat discs. These icons represent the location and orientation of each particle. To produce a more realistic surface, the surface should be "triangulated". A triangulated surface can be represented as a wireframe or as a shaded surface. Szeliski *et al* use the Delaunay triangulation, which is a common triangulation technique used for two dimensions and three dimensions [19].

Unfortunately, there are disadvantages when using particle systems to model surfaces. The modelling of particle systems requires more computation than splines, and it is more difficult to achieve accurate control over the mathematical shape of a surface if a particle system is used [19].

2.5 SUMMARY

In this chapter, important issues regarding the modelling of snow were discussed and related work was examined. It was stated that it is almost impossible to model snow taking all of the factors that affect it into account. A simplified model that implements the most obvious factors must therefore be used. Two ways of modelling snow in computer graphics are to use polygons or particles. The use of particles seems to be the more appropriate method to use. The collisions of falling snow particles with surfaces must be detected using a collision detection method, the most common being the use of bounding volumes. To model the collection of snow, surface modelling can be used.

CHAPTER 3

DESIGN

As stated in the introduction, this project investigates the modelling and rendering of snow in a three-dimensional environment. To investigate snow, the authors examine the feasibly of generating snow in computer graphics, and the realism of the snow produced. The effect that adding snow to an environment has on the frame rate is also looked at. In this chapter the design decisions made to produce falling and collecting snow are discussed, as well as the tests performed to examine the effect that adding snow to an environment has on the frame rate. The first two sections look at the design of the snow. The next section looks at the different tests performed and the method used to examine the frame rate, and the last section gives a summary of the chapter.

3.1 FALLING SNOW

In this section we describe a design for falling snow. The design is based on the issues regarding the modelling of snow discussed in Chapter 2. To describe the design of falling snow, a description of the requirements of the falling snow model are given, together with a diagram and description of the design.

3.1.1 REQUIREMENTS

The falling snow must be modelled and rendered as realistically and efficiently as possible. Since it was found in Chapter 2.1 that it is almost impossible to realistically model snow taking into account all the factors that affect it, we only model falling snow with the factors that we consider to be important. These include wind, temperature, gravity, and the density of the snow clouds. In Chapter 2.2, it was found that particle systems are more appropriate than the traditional polygon method for modelling snow. We therefore model the falling snow as individual snowflake

particles using a particle system. The particle system used should implement Sims parallel processing ideas, as discussed in Chapter 2.2.4 [18].

3.1.2 DESIGN DIAGRAM AND DESCRIPTION

Each individual snowflake particle is assigned its own attributes and functions. The attributes define its position and characteristics, and the functions define how it reacts when affected by the important factors implemented. The attributes and functions should be assigned to new snowflake particles using a stochastic method, and updated using Euler's method of integration [18]. Similar snowflake particles are processed as a group. The different groups together make up the falling snow.

3.1.2.1 Diagram

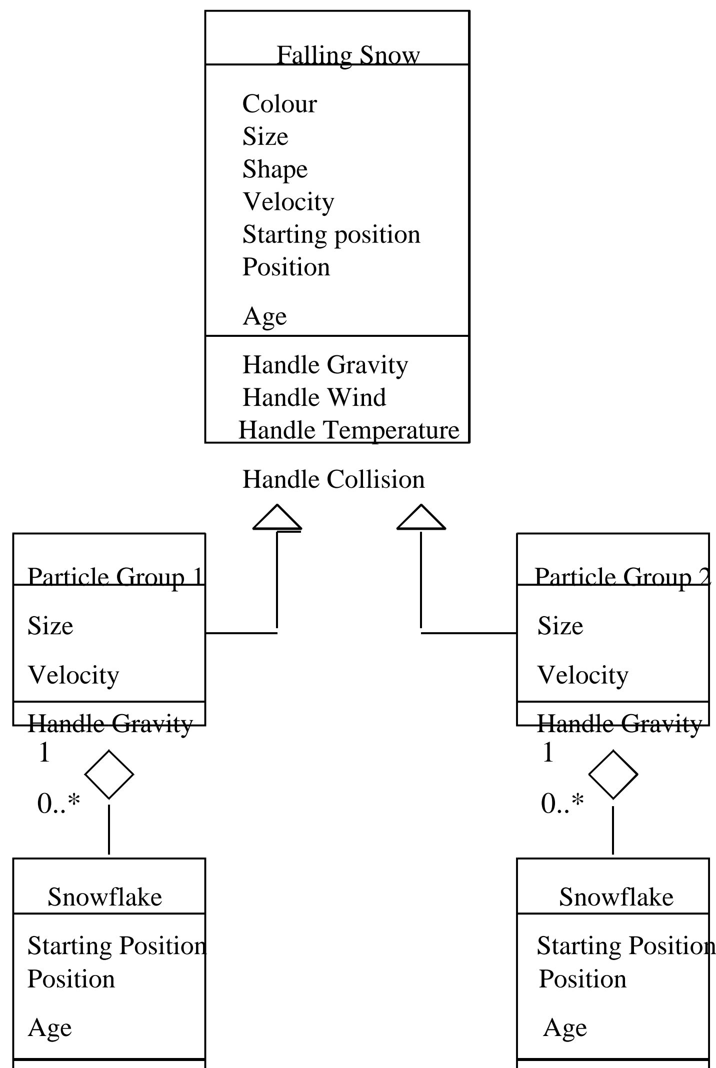


Figure 1: A design of falling snow

3.1.2.2 Description

The falling snow model is made up of particles, which represent snowflakes. These particles are divided into two groups. The sizes of the particles in the two groups are different. Therefore the particles in each group are not affected by gravity in the same way, and travel at different velocities. Each particle group knows about many snowflake particles, which make up the group. The number of particles generated represents the density of the clouds in nature. The attributes and functions chosen for each particle, and the reason for their choice, are discussed below.

3.1.2.2.1 *Attributes*

In Chapter 2.2, which discussed particle systems, is was noted that both Guan *et al* and Sims model snowflake particles using the colour white [9]. Since snowflakes appear white, the obvious colour to assign to a particle must be white.

The size of a snowflake particle should be in proportion to the other objects in the environment. Unfortunately, accurately representing the size and shape of a snowflake in computer graphics is not usually possible. In nature, snowflakes are extremely small, but when modelling them in computer graphics they need to be made bigger than they should be, in order to be seen. Modelling snowflakes that cannot be seen is pointless. Also, a more significant problem is that a pixel is the smallest picture element that can be drawn. Particles can therefore not be drawn any smaller than the size of one pixel, and although each individual snowflake in nature has a unique crystal shape, this is not possible to model using small pixels. Each snowflake particle must therefore be represented as a point, with a size of one or two pixels.

The velocity of each particle should be accurately represented. Unfortunately this attribute may be difficult to implement. Some computers are faster than others, and so a snowflake may appear to fall too quickly on one machine, and with the same velocity, too slowly on another. Therefore, for this project, we will implement the velocity of the snowflake particles so that they appear to fall at a realistic speed on our machine, a Celeron 500.

Particles should appear to fall continuously from above the top of the window. These particles should fall at random intervals, from random places at the top of the window. To implement this we will assign each particle a random starting position, and because the particles are moving, we keep track of their current position. By assigning each particle an age, we are able to keep track of how long a particle has existed in the environment.

3.1.2.2.2 ***Functions***

In Chapter 2.2.3 it was stated that Sims does not consider gravity or air friction as they cancel each other out [18]. Although gravity may not affect the speed of falling snow (due to air friction), it still has an effect on falling snowflakes. It is the forces of gravity that cause a snowflake to fall to the ground. In our model we will therefore consider gravity, by moving particles from the top of the window towards the bottom, so that they appear to fall to the "ground".

For snowflake particles to appear to be affected by wind they must not move vertically from the top of the screen to the bottom. Instead they must move in a random fashion, but as a group, across the screen. Wind is a complex nature phenomenon itself, and can affect snowflakes in many ways. In this project we are concerned with producing snow that appears realistic and does not decrease the frame rate considerably. We are not concerned with the different affects wind could produce. We therefore do not consider wind in this project, and assume that there is no wind in the environment.

Temperature affects the size of a snowflake, by causing it to melt. In this project the snowflakes are represented as points, only one or two pixels wide. It is therefore not possible to decrease their size, although it is possible to remove them. In our model, a falling snow particle will either remain the same size or disappear before reaching a surface. The collection of snow can appear to melt by removing some of the particles from a group. To represent the effect of melting caused by temperature we therefore decrease the size of the group, rather than the size of each individual particle.

To produce falling snow, some sort of collision detection must be used. In the design of falling snow, we are interested in the falling snow and not its collection. What happens to a snowflake particle when it collides with a surface is not important, however some sort of collision detection is needed, or the number of particles in the particle system will increase infinitely. In this design we therefore implement a basic collision detection method. When the particles reach an imaginary plane at the bottom of the window they are removed. To implement this, each particle is tested to see if it passes through the plane. A more advanced use of collision detection is discussed in the design of the collection of snow.

3.2 THE COLLECTION OF SNOW

In this section we design the collection of snow. There are many ways to model collecting snow. This section examines some of these ways, deciding on the best design for certain circumstances. Like section 3.1, this section bases the design of the collection of snow on the issues discussed in Chapter 2. A description of the requirements for the collecting snow model are given, followed by a discussion on advantages and disadvantages of modelling collecting snow dependently or independently of the falling snow. A design diagram and description of a collecting snow model are presented.

3.2.1 REQUIREMENTS

The snow modelled must appear realistic. To achieve realism the laws of physics must be obeyed. Therefore the falling snow particles must collide with objects, and subsequently collect to form a surface. As noted in Chapter 2.1.1, this surface should appear as a smooth white layer in the distance, and have a complex texture when it is close to the viewer [11]. Fearing' s properties of flake flutter and flake dusting, discussed in Chapter 2.1.3 should also be implemented [7]. The rate at which the snow collects should depend on the rate and density at which it falls.

3.2.2 DEPENDENT VERSUS INDEPENDENT ON FALLING SNOW

The collection of snow can be modelled dependently, or independently of the falling snow. Both methods have advantages and disadvantages.

3.2.2.1 Dependent on falling snow

In this project, modelling the collection of snow dependently of the falling snow means that for every snowflake particle that collides with a surface, a particle remains on the point on the surface where that particle collided. This can be achieved, as discussed in Chapter 2.4.1, by attaching the particle to the surface as it collides; or by removing the particle as it collides, replacing it with a texture map, or another particle that has no velocity [13].

3.2.2.1.1 *Advantages*

Advantages of modelling collecting snow dependently of falling snow are that the collecting snow appears realistic, and the rate at which the snow collects is equal to the rate and density at which it falls. The surface of snow is formed automatically without the user having to specify its form, and it should be easy to implement Fearing's properties of flake flutter and flake dusting, discussed in Chapter 2.1.3 [7].

3.2.2.1.2 *Disadvantages*

A disadvantage is that only one layer of snow is formed, and even if a method is devised to model more than one layer, it is not reasonable to model layers of snow using individual snowflake particles. These particles are only one or two pixels in size, and therefore thousands are needed to produce each layer.

Another disadvantage is that if bounding volumes are used for collision detection, they must enclose objects relatively accurately. In Chapter 2.3.1.3, which discusses snowflake collision detections, it is pointed out that if bounding volumes do not enclose the objects accurately,

attaching a particle to the bounding volume it collides with may result in a snowflake particle that appears to have stopped in the middle of the air.

Particle systems are used to produce our snow, and as stated in Chapter 2.2.2.2, surfaces created by particles cannot be shaded. To produce a shaded surface, the surface must be modelled using the traditional polygon method [15]. To model a shaded surface of snow, the collection of snow must be modelled as a surface independently of the falling snowflake particles.

3.2.2.2 Independently of falling snow

Modelling the collection of snow independently of the falling snow allows the collection of snow to be modelled as a surface using traditional polygon methods. Modelling the collection of snow independently requires all the surfaces in the environment to be examined. A surface of snow is then modelled on all parts of surfaces on which snow in nature would fall. Modelling the collection of snow in this way can be very difficult if there are many moving objects in the environment. This is because the parts of surfaces that require a surface of snow keep changing. If the collection of snow is modelled independently of the falling snow, snowflake particles that collide with a surface are destroyed.

Modelling the collection of snow as a surface using traditional polygon methods does not always look realistic, although it does allow the surface to be shaded. To produce a more realistic looking collection of snow, the surface can be texture mapped [13].

The different ways the surface can be modelled are discussed in Chapter 2.4. Splines can produce a smooth, but slightly wavy surface of snow, which looks more realistic than a polygon mesh [20]. Unfortunately, using splines makes implementing Fearing' sflake flutter almost impossible [7]. Using a mesh, the flake flutter can be approximated. Oriented particles produce a more accurate surface, but require more computation than splines, and are therefore not practical for computer games where the frame rate is important (and the surfaces are only seen for a few seconds) [19].

3.2.2.2.1 *Advantages*

Advantages of modelling the collection of snow independently of falling snow are that the surfaces of snow can be shaded; you do not have to worry about particles stopping in the middle of the air (due to inaccurate bounding volumes), and you have more control over the surfaces of snow that are produced [7].

3.2.2.2.2 *Disadvantages*

Disadvantages are that the form of the surface has to be calculated in advance, Fearing's flake flutter is not easy to implement [7], and the surfaces of snow produced do not always look very realistic. Snow does not necessarily collect in the same places that it falls.

3.2.2.3 Summary

ADVANTAGES	DEPENDENT	INDEPENDENT
Realistic	Yes	No
Surface automatically generated	Yes	No
Flake flutter	Yes	No
Flake dusting	Yes	Yes
Shading	No	Yes
Many layers	No	Yes
Particle system	Yes	Yes
Traditional polygons	No	Yes

Table 1: Modelling the collection of snow independently versus dependently of falling snow

Table 1 shows that neither method is better. If the snow is to be used in a computer game, such as a skiing or snowboarding game, which requires a large amount of realism, and moves through scenes too quickly to worry about the collection of snow, then it is better to model the collection of snow dependently of falling snow. If a large collection of snow is needed, or the surface must be shaded, then it is better to model the collection of snow independently of falling snow.

3.2.3 DESIGN DESCRIPTION AND DIAGRAM

In order to model a collection of snow, and detect collisions between snowflake particles and objects in the environment, each surface or bounding volume in the environment must be specified.

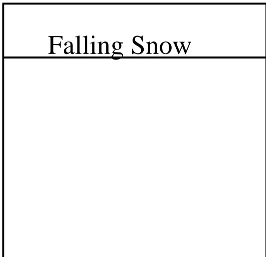
3.2.3.1 Description of the design

Each snowflake particle knows about a surface list. This list consists of all the surfaces or bounding volumes with which the snowflake particle could collide. If a collision is detected between a snowflake particle and a surface (or bounding volume), the particle is either attached to the surface, or removed from the particle group, depending on the environment the snow is being modelled for.

If the collection of snow is modelled independently of the falling snow, a polygon mesh is used to represent the surface of snow. The heights at the vertices of the polygons that make up the mesh are stored in an array. As the snow falls, these heights increase. The polygons are represented as triangles rather than rectangles, so that all the vertices of each polygon always lie on one plane.

Because we assume there is no wind in the environment, there are no flake flutter or flake dusting effects. To produce a surface of snow that appears to have a complex texture, the surface of snow is texture mapped.

3.2.3.2 Diagram



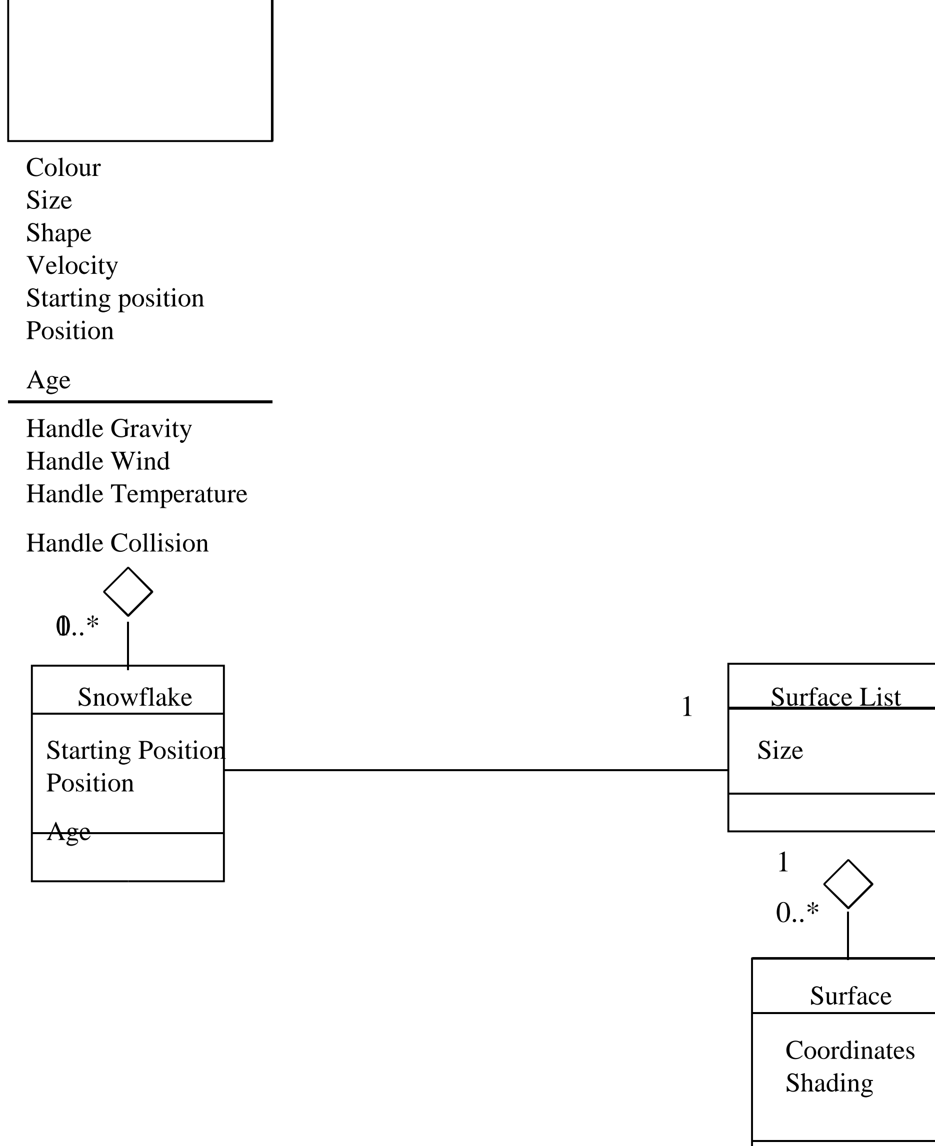


Figure 2: A design of the collection of snow

3.3 TESTING THE FRAME RATE

In order for a model of snow to enhance the graphics in a computer game (or for falling snow to be of any use in a three-dimensional environment), it must not have a drastic effect on the frame rate when added to the computer game (or environment). This section discusses the method we use to test the effect that our model of falling snow has on the frame rate. We describe the timer used, the process of testing the frame rate, and the different factors that are tested for their effect on the frame rate.

3.3.1 TIMER

To test the frame rate, a simple timer is used. This timer can be used in two ways. The first way tests the frame rate by timing only the rendering of objects each frame, while the second method times the rendering of objects, together with the time it takes to set states.

3.3.1.1 Rendering only

Testing the frame rate by timing only the rendering of objects each frame, tests the number of frames that can be rendered in a second. A timer is started just before the colour buffer is cleared (before beginning the drawing of the next frame), and ended when all the objects have been drawn to the screen:

```
Draw function ()
 {
   Start the clock;
   Render objects;
   End the clock;
 }
```

3.3.1.2 Rendering and setting states

Testing the frame rate by timing the rendering of objects, together with the time it takes to set states, tests the time taken from the start of one frame to the start of the next one. A timer is started after the objects have been drawn to the screen in a frame, and ended after the objects are drawn to the screen in the next frame:

```
Draw function ()
 {
   Settings;
   Render objects;
   End the clock;
   Start the clock;
 }
```

3.3.2 PROCESS OF TESTING THE FRAME RATE

To test the frame rate of a program we run the program 60 times. Each time the program is run for at least 10 seconds. A counter is increased by one each time a frame is rendered. The totals for the sixty counters are added together and divided by the total time (which is approximately, but not less than, 60 seconds). The amount obtained is then the approximate number of frames rendered each second.

3.3.3 FACTORS TESTED

To test the effect that adding falling snow to a computer game or environment has on the frame rate, we test different factors and make comparisons. By doing this we can see if it is feasible to add falling snow to a computer game or environment, and if it is what must be done to produce the most optimal model. We do not test the collection of snow for two reasons. Firstly we find the collection of snow difficult to implement realistically. Secondly it is usually pointless adding collecting snow to a computer game, because the characters are not in a scene (portion of the world) long enough to see the snow collect.

3.3.3.1 The two different timing methods

Firstly, the two different methods of timing discussed in Chapter 3.3.1 are compared. To compare them, the average frame rate of a window with no objects is found using each method. The results obtained are then compared. If the results obtained, which will be discussed in Chapter 5 are similar, only one method is used to perform the rest of the tests.

3.3.3.2 Window size

We found (while implementing the timer) that the size of the window had a large effect on the frame rate. We therefore test the frame rate for different window sizes. One size is chosen as the best. This size is then used for the rest of the tests. Also, by testing the frame rate of a window

with no objects, we can find the maximum frame rate that is possible on our machine for that window size. This is useful to know, because it is gives us a way of analysing the frame rates we obtain. A program that is run on different machines will probably run at two different frame rates.

3.3.3.3 Polygons

Another useful frame rate to have is that of the rendering of one or two polygons. The traditional way of modelling objects is to use polygons. Most objects in computer games are therefore modelled with polygons. If we can show that (on the same machine) a program producing snow has a similar, or higher, frame rate than a program producing a few polygons, then we can prove that it is feasible to add snow to a computer game.

3.3.3.4 Factors used to model snow

Several factors in the falling snow model that influence the characteristics of the snow produced are investigated for their effect on the frame rate. Three of these factors include the size of the snowflake particles, the rate at which they fall, and the number of particle groups. To test each factor, all other factors are kept constant (if possible). The factors remain the same for all tests, except for when they are being tested themselves.

3.4 SUMMARY

In this chapter design decisions were made. The advantages and disadvantages of modelling collecting snow dependently or independently of falling snow were examined, and designs of falling and collecting snow were described. The method used to test the effect that adding falling snow to an environment has on the frame rate was looked at.

CHAPTER 4

IMPLEMENTATION

This chapter discusses the implementation of the design issues examined in Chapter 3. The chapter is divided into four sections. The first section looks at the hardware and software used. The frame rate of a program varies when run on different machines, thus a description of the machine we use to test the frame rate on is given, as well as a description of the interfaces used and why they are chosen. The next two sections discuss the implementation of falling and collecting snow, and the problems encountered. The last section gives a summary of the chapter.

4.1 HARDWARE AND SOFTWARE

To implement the designs of falling and collecting snow that are described in Chapter 3, we use OpenGL as an interface to the graphics hardware and the GLUT toolkit [20]. Dave McAllister' s Particle System Application Programmer Interface (API) is used to assist in implementing the snow design using particle systems [12]. The programming language used is C++ and the operating system is Linux. The frame rate is tested on a Celeron 500 PC without a graphics accelerator. Reasons for these choices are discussed below.

4.1.1 OpenGL

In order to create a program (that produces snow in a three-dimensional environment), without directly dealing with the hardware (which can be complicated), an interface to the hardware is needed. Since OpenGL (from Silicon Graphics, Inc) is a well-known interface that is platform and hardware independent, we choose to use it [20].

Unfortunately, to be hardware independent, OpenGL does not deal with windowing issues. To deal with windowing issues a toolkit is needed [20].

4.1.2 OpenGL Utility Toolkit (GLUT)

Qt and GLUT are two toolkits used for dealing with windowing tasks. GLUT (written by Mark Kilgard) is chosen because it can be used for both the Windows (from Microsoft) and Linux operating systems. Qt is Linux dependent. Also, GLUT appears to be the standard windowing system used with OpenGL [20].

4.1.3 Linux AND C++

Because OpenGL and GLUT are used, both Windows and Linux are appropriate operating systems. Due to personal preferences, Linux is chosen. Also due to personal preferences, C++ is chosen as the programming language.

4.1.4 PARTICLE SYSTEM API

To model snow using particle systems, Dave McAllister's Particle System API is used. This API is used for three reasons:

1. The style of the API is similar to the style used in OpenGL. States are set, and remain in effect until another command is given to change them.
2. The API allows C++ programs to generate and maintain particles.
3. The API implements Sims' parallel processing ideas mentioned in Chapter 2.2.4 [12].

The Particle System API consists of commands that allow C++ programs to implement particles. There are four types of commands, namely "State setting", "Action", Particle group", and "Action list". State setting commands allow you to change the current state of attributes in the particle system. Action commands control a group of particles, causing effects such as bouncing and explosions. Particle group commands manage a group of particles, and Action list commands manage action lists (groups of action commands). To implement particles in a program you create and initialise the particles, apply actions to them, and then drawn them to the screen [12].

4.1.5 MACHINE

A Celeron 500 PC without a graphics accelerator is used.

4.2 FALLING SNOW

In this section we describe the implementation of falling snow using OpenGL and the Particle System API. Two problems encountered are looked at, and code for a working solution that looks realistic is given.

4.2.1 IMPLEMENTATION

In chapter 3.1 we discussed the design of falling snow. This design is used in the implementation (For the design description refer to Section 3.1.2.2).

4.2.1.1 Creating two particle groups

Two create two particle groups of snowflakes use:

```
firstGroupNumber = pGenParticleGroups (2, 150);
```

The groups are numbered sequentially beginning with `firstGroupNumber`, and each group can consist of a maximum of `150` snowflake particles.

4.2.1.2 Setting the state of the attributes

To draw a particle group, `pDrawGroupp` is used. Before the particle group is drawn, the state of attributes, for each particle in the group, can be changed. To change the state of attributes of a group, the group must be the current group to which all state changes apply. To make a group (represented by the number `particleGroup`) the current group, use:

```
pCurrentGroup (particleGroup);
```

In our falling snow model, the particles in each group have different sizes and velocities. Each particle is represented as an OpenGL point. The size of each particle is set using:

```
glPointSize (pixelsize);
```

Each particle (represented by the number `particleGroup`) is given a velocity using:

```
pVelocity (0.0, -0.6 - (particleGroup * 0.1), 0.0);
```

This velocity causes the particles to move down the "negative y-axis", imitating the effect of gravity.

4.2.1.3 Creating and destroying snowflake particles

To produce continuously falling snow, generate particles at the top of the window, and remove them from the environment (and their particle group) when they collide with an imaginary plane at the bottom of the window, using:

```
pSource(r, PDRectangle, -50.0, 50.0, -50.0, 100.0, 0.0, 0.0, 0.0, 0.0,
                              100.0);
```
and
```
pSink (false,PDPlane, 0.0, -40.0, 0.0, 0.0, 1.0, 0.0);
```

The particles are created at a rate of `r` each frame (This is used to control the density of the snowfall). [12]

4.2.2 PROBLEMS

The techniques discussed above, in Chapter 4.2.1, produce falling snow. However, due to a few problems the falling snow may not look realistic. Two of these problems are discussed below.

4.2.2.1 The maximum number of particles in a particle group

Depending on certain factors, if the maximum number of particles a group can consist of (specified using `pGenParticles`) is too low, an unrealistic gap appears in the middle of the falling snow, as seen in Figure A1 (Appendix A). The factors that affect this gap are: the velocity of each particle, the distance the particles have to travel (from the top of the window to the bottom), and the rate of particles created each frame.

This problem can be avoided by calculating the maximum number of particles a group needs, using the formula:

$$\text{Number of particles needed} = \frac{\text{Height}}{|\text{Velocity}|} * \text{Rate}$$

4.2.2.2 The rate at which snow falls

During the initial few seconds when the snow starts to fall, it does not look realistic if particles are created every frame. This can be seen in Figure A2 (Appendix A). The start of falling snow can be made more realistic by allowing snow to fall less densely at first. Since only one or two particles are created each frame, we cannot decrease the number of particles created each frame to decrease the density. Instead, we initially only create one particle every few frames. This can be seen in Figure A3 (Appendix A), and the code is shown in lines 32 to 48 in Appendix B.

4.2.3 THE IMPLEMENTATION

The full code for the implementation of falling snow is found in Appendix B. Figure A4 (Appendix A) is a snapshot of the falling snow.

4.3 THE COLLECTION OF SNOW

Due to time constraints and the complexity of snow, we are not able to provide a useful implementation of collecting snow.

We feel that generating collecting snow in a computer game would not enhance the graphics of the game for two reasons. Firstly, simple representations of surfaces of snow produced using polygons, splines or particle systems do not look realistic, and more complex representations require too many computations that decrease the frame rate. Polygon meshes are only an approximation of the surface; splines cannot produce flake flutter; and surfaces created with particle systems cannot be shaded. Secondly, in most computer games a character is not in a scene long enough to see the collecting of snow. Therefore implementing it just decreases the frame rate.

4.4 SUMMARY

In this chapter we discussed the hardware and software used, and why it was chosen. Falling snow is implemented using OpenGL and a particle system API. We state that we are not able to produce collecting snow efficiently.

CHAPTER 5

TESTS AND RESULTS

To test the effectiveness of the falling snow implemented in Chapter 4.2, and the feasibility of adding it to a computer game, we test the effect that it has on the frame rate of a program. In Chapter 3.3 the method used to test the frame rate was described. In this Chapter, we implement the tests and discuss the results obtained. Because the frame rate of a program differs according to the hardware used, we need a way of telling whether the frame rate we obtain is a good frame rate or not. The first section of this chapter examines the frame rate of a program that generates a window with no objects. The second section examines the frame rate of a program that generates falling snow, and the third section gives a summary of the chapter.

5.1 THE FRAME RATE

As stated in Chapter 3.3.2, we test the frame rate of a program by running the program for approximately 10 seconds, 60 times, increasing a counter each time a frame is rendered. In this section we compare the results obtained when using two slightly different timing methods. We also investigate the effect that the window size has on the frame rate of a program. The tests are performed on a program that generates a window with no objects, and the frame rate obtained represents the maximum frame rate obtainable on our machine at that window size. These results can be compared to the results obtained when running a program that produces falling snow. This is useful in deciding whether the frame rates obtained are good or bad.

5.1.1 TWO TIMING METHODS

Chapter 3.3.1 discussed two slightly different ways to test the frame rate. The first method tests the time it takes to render a frame. It does not consider the time it takes to change state settings or call functions that are not part of the rendering process. The second method times the actual

number of frames produced each second. It tests the time taken from the start of one frame to the start of the next one. The average frame rate obtained for each test is shown below, in Table 2.

PROGRAM	METHOD 1 (Rendering only)	METHOD 2
Window with no objects	9.857	9.862
Falling snow	9.561	9.578

Table 2: Approximate frame rates (Frames per second)

The results shown in Table 2 show that there is no significant difference between the two tests. Method 2 produces a slightly higher frame rate (which means that the time taken to change settings and call functions is less than the time it takes to render a frame). Because there is no significant difference between the two tests, only one method will be used to measure the frame rate for the rest of the tests. The second method (that tests the time taken from the start of one frame to the start of the next one) is used because the authors feel that it is a more accurate description of the average number of frames produced each second. The code for timing the frame rate using the second method can be seen in lines 81 to 105 in Appendix B.

The two programs tested (in Table 2) have a window size of 500 x 500 (in pixels). The results in the table show that the average frame rate for a program that generates a window with no objects (and therefore the maximum frame rate obtainable on our machine at a window size of 500 x 500) is approximately 10 frames per second. In the Introduction we stated that the average frame rate for a computer game should be approximately 20 frames per second [10]. On our machine, without a graphics accelerator, we are not able to produce programs (with a window size of 500 x 500) that run at a frame rate higher than 10 frames per second. This means that a program that runs at less than 20 frames per second (or even less than 10 frames per second) could still be suitable for a computer game, but would not be enjoyable played on a Celeron 500 PC without a graphics accelerator. Therefore to test the effectiveness of our falling snow in a program, we compare the frame rate obtained to the frame rate of other programs, instead of comparing the frame rate value to an amount such as "20 frames per second".

5.1.2 WINDOW SIZES

The size of the window produced greatly affects the frame rate. To prove this, the frame rate of a program that draws a window is tested several times. Each time, the size of the window it draws is changed. The frame rates obtained are shown in the Figure 3 below:

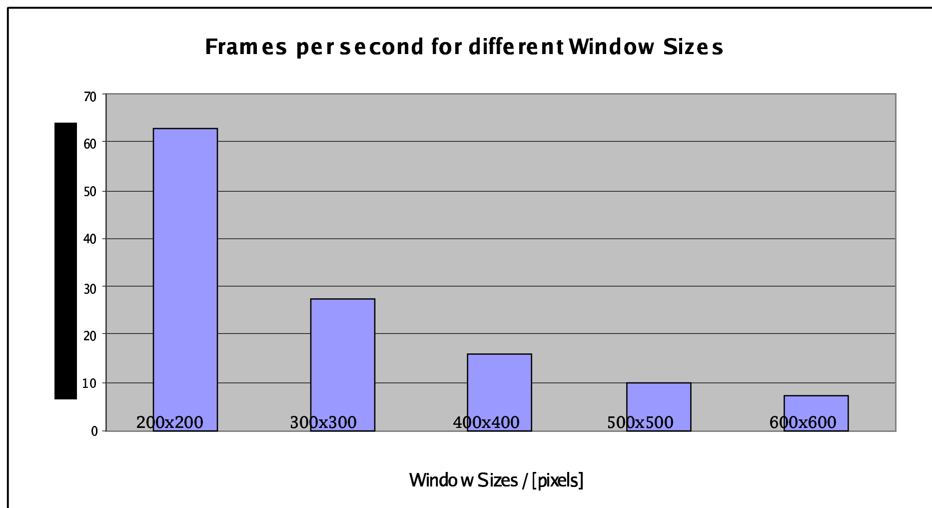


Figure 3: The frame rates for different window sizes

Because the size of the window produced greatly affects the frame rate, we choose the most appropriate window size and keep it constant for all other tests. Figure 3 shows that there is a trade off between window size and frame rate. A window size of 500 x 500 is chosen, because it has a size that is not too small, and a frame rate that is tolerable.

5.2 FALLING SNOW

In Chapter 4.2 we showed that it is possible to produce falling snow that looks realistic. In this section we investigate the effect that this snow has on the frame rate of a program. Three factors that influence the characteristics of the snow are investigated for their effect on the frame rate. These factors are: the size of the snowflake particles, the rate at which they fall, and the number of particle groups. To test each factor, all other factors are kept constant (if possible).

5.2.1 POINT SIZE OF PARTICLES

The individual snowflake particles that make up the snow are represented as OpenGL points. The size of these points can be specified using:

```
glPointSize (pixelsize);
```

Where `pixelsize` represents the width of the points in pixels.

We perform some tests to see if the size of the points has any effect on the frame rate. All factors other than size remain the same for each test performed. Figure 4 shows the results obtained.

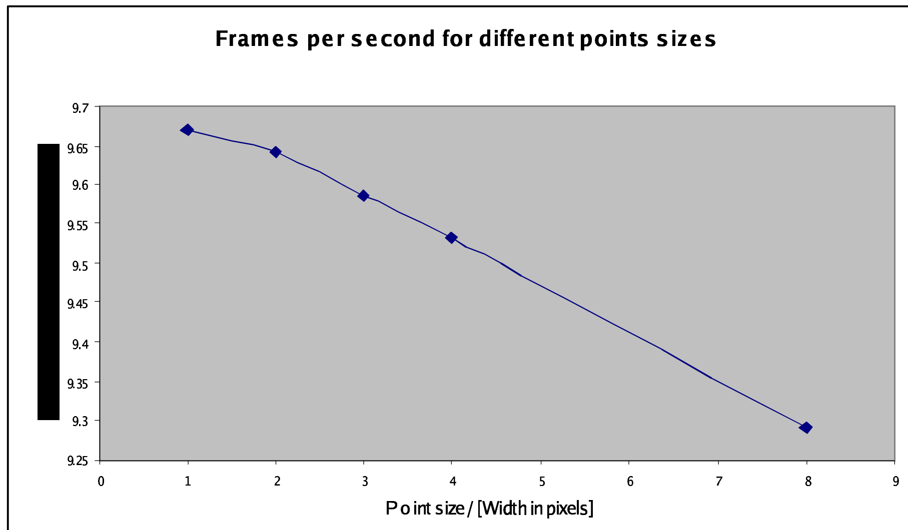


Figure 4: The frame rates for different point sizes

The results shown in Figure 4 show that the size of a particle in a particle group does affect the frame rate. However, it does not have a significant effect on the frame rate (There is a difference of only 0.4 frame, between a point size of 1 and a point size of 8). The frame rate of a program decreases as the size of the particles in the program increase. This is due to the fact that more pixels are needed to represent larger points. Point sizes of 1 and 2 pixel widths are chosen for our snow particles. These sizes are chosen because they have the least effect on the frame rate and look the most realistic. Point sizes of 4 or greater should not be used to represent snowflakes because the points appear square rather than round. This is due to the rectangular nature of pixels. The problem can be solved using an expensive technique called "antialiasing", but this is not worth using for thousands of small snowflakes, especially when a high frame rate is important. Falling snow with point sizes of 3 and 8 can be seen in Figures A5 and A6 (Appendix A). Using two different sizes (with two different velocities) for the snowflakes looks more realistic than using just one size.

5.2.2 RATE OF PARTICLES CREATED

Particles are created at the top of the window using:

```
pSource(r, PDRectangle, -50.0, 50.0, -50.0, 100.0, 0.0, 0.0, 0.0, 0.0,
                                    100.0);
```

The rate at which these particles are created can be controlled. This rate represents the density of the snowfall. Tests are performed to see if the rate of particle generation (density of snowfall) has any effect on the frame rate. The results are shown in Figure 5 below. All factors other than the rate of particle generation and the maximum number of particles a group may consist of, are kept constant.

The maximum number of particles a group may consist of is changed. More particles are required in particle groups with a high particle generation, than in those with a low particle generation (See section 4.2.2.1 for more details).

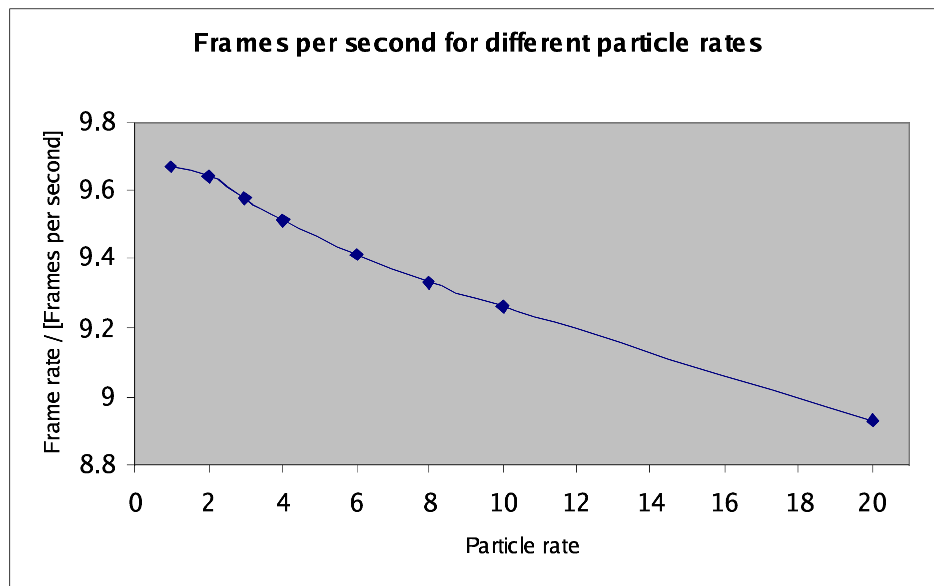


Figure 5: The frame rates for different rates of particle creation

The results shown in Figure 5 show that the rate at which particles are created does affect the frame rate. A program, in which 20 new particles are created each frame, produces almost one less frame each second than a program in which only one new particle is created each frame. We choose to produce two new particles each frame. Producing two new particles each frame produces a realistic density of snow. This can be seen in Figure A7 (Appendix A). Figure A8

(Appendix A) shows a snapshot of falling snow, where twenty new particles are created each frame.

5.2.3 NUMBER OF PARTICLE GROUPS VERSUS RATE OF PARTICLES CREATED

The falling snow is represented by individual snowflake particles. These particles can be controlled as a group rather than individually. Tests are performed to compare the frame rates of programs with one particle group against programs that produce the same result using more than one particle group.

For instance, suppose you want to produce falling snow where two new particles are created each frame. Is it better to have one particle group that creates two particles, or two particle groups, which each create one particle?

The results obtained for particle systems that create two, three, and four new particles each frame, using different numbers of particle groups are shown in Figure 6. All factors are kept constant except the number of particle groups, the rate at which particles are created in each group, and the maximum number of particles each group can consist of. The tests are performed on programs that produce falling snow.

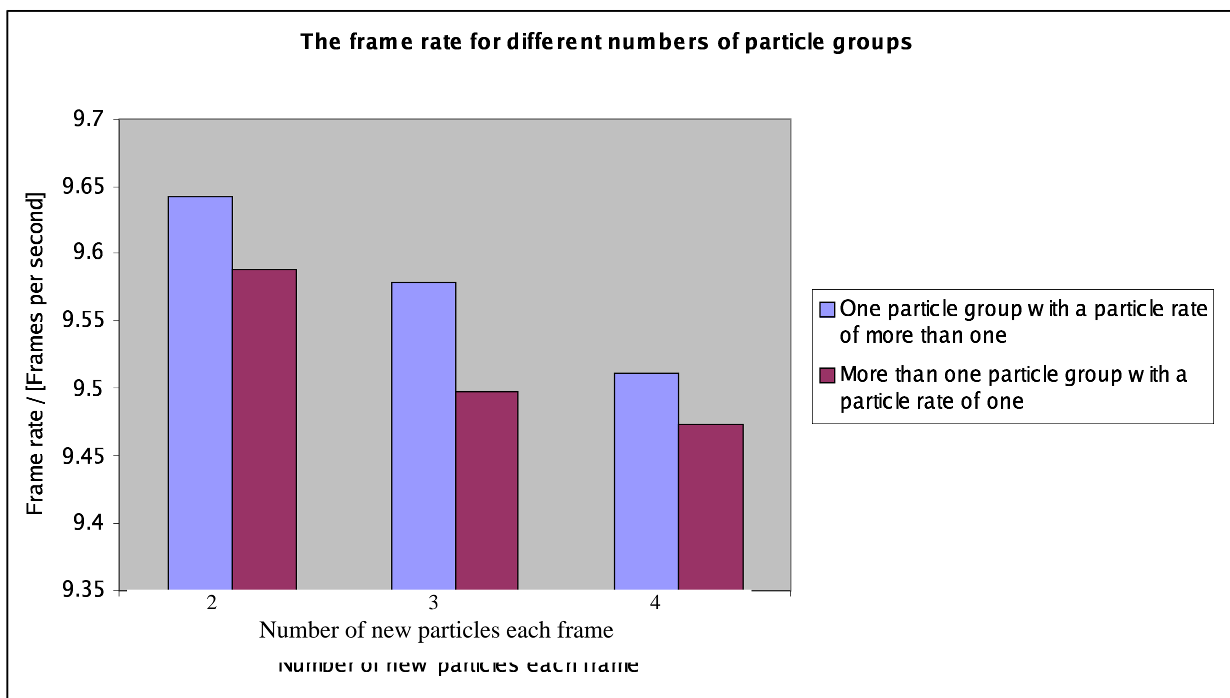


Figure 6: The frame rates for different numbers of particle groups

The results in Figure 6 show that for the programs tested, it is much better to have one particle group than many particle groups. Therefore, when designing a program using particle systems, you should usually use as few particle groups as possible. We use two particle groups in our system, one for each of the two different sizes of particles.

5.2.4 FINAL IMLEMENTATION

Considering all factors discussed in this project our final implementation of falling snow is drawn in a window that is 500 x 500 (in pixels). Two particle groups represent the snow. The size and velocity of the particles in each group are different, and one new particle is generated in each group every frame, except in the beginning when the snow begins to fall.

The average frame rate of the program that generates this snow is:

≈ 9.5808 frames per second

The frame rate of a cube drawn on a window of size 500 x 500 is:

≈ 9.41865 frames per second

The frame rate of the program that generates our falling snow is higher than that of a program that draws a cube (six polygons). Since most objects in computer games are drawn using polygons like the cube, we conclude that it is possible to add falling snow to a computer game, because our falling snow has less effect on the frame rate than a cube.

5.3 SUMMARY

In this chapter we tested the frame rate of various programs to investigate whether the falling snow implemented in chapter 4 would be successful in a computer game. Two methods of testing the frame rate were compared. We found the size of the window had a large effect on the frame rate. Three factors that influence the characteristics of snow namely, particle size, particle creation rate, and the number of particle groups, were investigated for their effect on the frame rate. We found that they all affect the frame rate. The final falling snow model is given and

compared to a cube. Because the frame rate of the program generating the falling snow was higher than that of the program drawing the cube, we concluded that it is possible to add realistic falling snow to a computer game.

CHAPTER 6

CONCLUSION

This project demonstrates that it is possible to model relatively realistic falling snow in a three-dimensional environment, which does not decrease the frame rate significantly. Therefore we are able to add falling snow to a computer game to enhance its graphics. However, we are not able to shown that this is true for the collection of snow.

Using OpenGL and a particle system API, this project demonstrates that particle systems can effectively and efficiently produce falling snow.

In Chapter 5 we show that the size of particles, rate of particles generated each frame, and number of particle groups in a particle system, affect the frame rate of a program. Therefore these factors must be chosen carefully if the most optimal model of falling snow is required.

6.1 FUTURE WORK

Since particle systems can effectively and efficiently produce falling snow. They should also be able to effectively produce other natural phenomena such as rain. Investigating the implementation of these other natural phenomena, using particle systems is possible future work. More work could also be done in investigating the collection of snow and the interaction of objects with snow.

CHAPTER 7

REFERENCES

[1] Bobic, N., "Advanced Collision Detection Techniques", available via the WWW at http://www.gamasutra.com/features/20000330/bobic_01.htm, 2000

[2] Bobic, N., "Advanced Collision Detection Techniques", available via the WWW at http://www.gamasutra.com/features/20000330/bobic_02.htm, 2000

[3] "Collision Detection and Particle Interaction", available via the WWW at http://freespace.virgin.net/hugo.elias/models/m_colide.htm

[4] "Commandos: Behind Enemy Lines Game Guide", available via the WWW at http://www.gamespot.co.uk/pc.gamespot/features/commandos_sg/

[5] Crawford, J., Juliano, J., Larsen, E., and Lok, B., "Presence, Precipitation, and The Old Well", available via the WWW at http://www.cs.unc.edu/~lok/classes/comp236/

[6] Fang, W., "Collision Detection", available via the WWW at http://www.csse.monash.edu.au/hons/projects/1995/William.Fang/report.html, 1995

[7] Fearing, "Computer Modelling of Fallen Snow", available via the WWW at http://www.cs.ubc.ca/nest/imager/contributions/fearing/snow/snow.html, 2000

[8] Foley, J., van Dam, A., Feiner, S. K., Hughes, J. F, *Computer Graphics: Principles and Practice*, Addison Wesley, Massachusetts, 1990, 471-478

[9] Guan, T., and He, Zhu, "Approximating Interaction Between Particle Systems", available via the WWW at http://www.cs.unc.edu/~guan/c236/report/report.html

[10] Hadwiger, M., "PARSEC: Enhancing Realism of Real-Time Graphics Through Multiple Layer Rendering and Particle Systems", Institute of Computer Graphics, Vienna University of Technology, available via the WWW at `http://perun.cg.tuwien.ac.at/cescg/`

[11] Law, S., Oh, B., and Zalesky, J. "The Synthesis of Snowcovered Terrains", Massachusetts Institute of Technology, available via the WWW at `http://graphics.lcs.mit.edu/~boh/Projects/snowGenFinalWrite.html`, 1996

[12] McAllister, D. K., "Particle System API", available via the WWW at `http://www.cs.unc.edu/~davemc/Particle/`, 2000

[13] "Natural Phenomena", available via the WWW at `http://www.sgi.com/software/opengl/advanced98/notes/node150.html`

[14] Owen, G. S., "Particle Systems", available via the WWW at `http://old.cs.gsu.edu/materials/HyperGraph/animation/particle.htm`, 2000

[15] Reeves, W. T., "Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems" In *Proceedings of SIGGRAPH '85*, 1985, 313-322

[16] Roberts, D., "Collision Detection. Getting the most out of your collision tests", available via the WWW at `http://www.ddj.com/articles/1995/9513/9513a/9513a.htm`, 1995

[17] Schinner, A., "Collision Detection", available via the WWW at `http://itp.nat.uni-magdeburg.de/~schinner/algorithm/node2.html`, 1999

[18] Sims, K., "Particle Animation and Rendering Using Data Parallel Computation" In *Computer Graphics (SIGGRAPH '90 Conference Proceedings)*, August 1990, 405-412

[19] Szeliski, R., Tonnesen, D., "Surface Modeling with Oriented Particle Systems" In *Computer Graphics (SIGGRAPH '92 Conference Proceedings)*, July 1992, 185-194

[20] Woo, M., Neider, J., and Davis, T., *OpenGL Programming Guide*, Addison Wesley Developers Press, Massachusetts, 1997

APPENDIX A

SNAPSHOTS

Figure A1: Too few particles in a particle group

(Section 4.2.2.1)

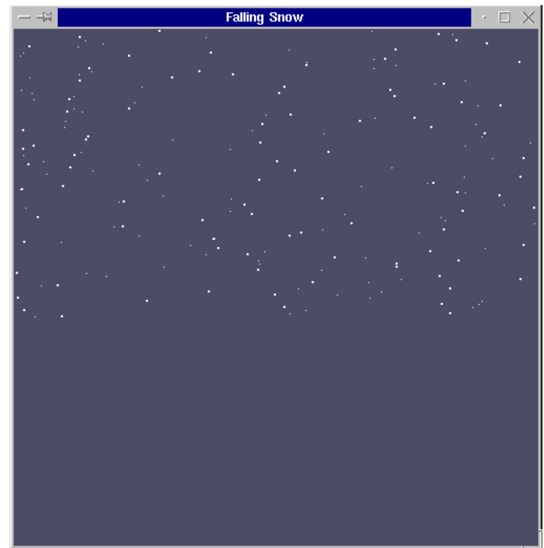


Figure A2: Constant rate of creation

(Section 4.2.2.2)

Figure A3: Less dense at first

(Section 4.2.2.2)

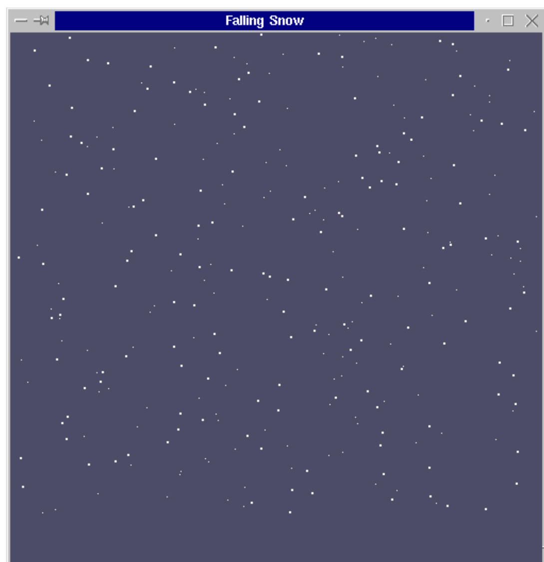


Figure A4: Falling snow

(Section 4.2.3)

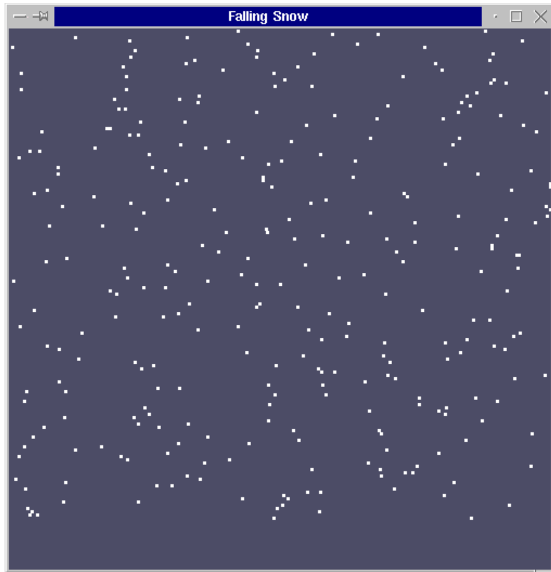


Figure A5: Point size 3 (pixels wide)

(Section 5.2.1)

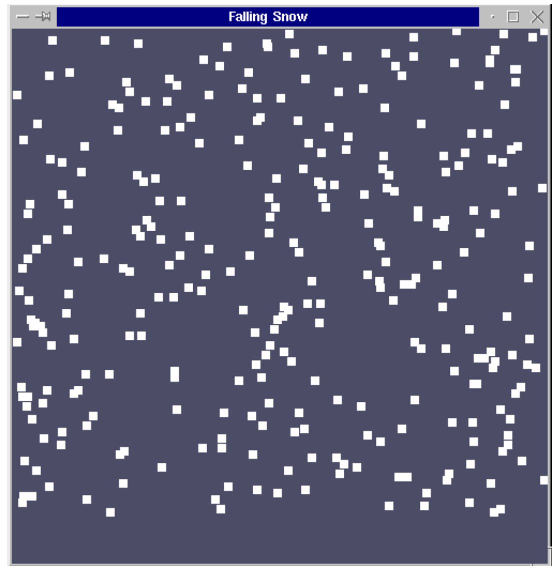


Figure A6: Point size 8 (pixels wide)

(Section 5.2.1)

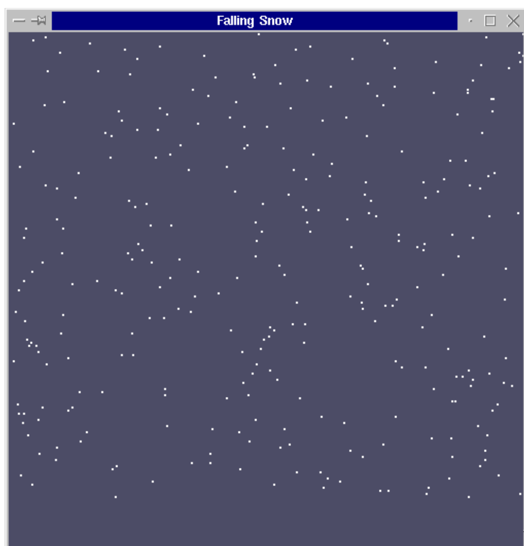


Figure A7: Particle rate of 2

(Section 5.2.2)

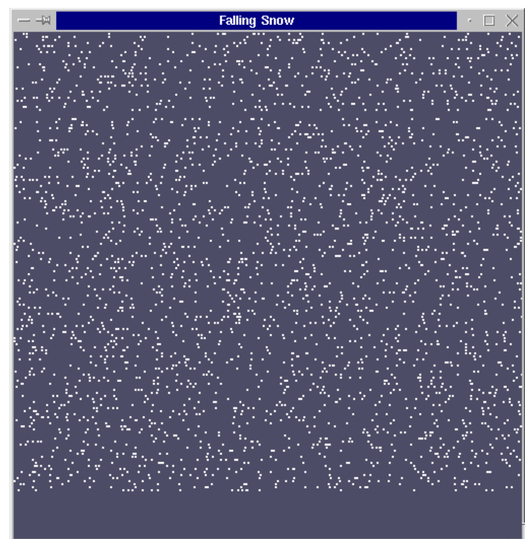


Figure A8: Particle rate of 20

(Section 5.2.2)

APPENDIX B

CODE

```
1.  /* Falling Snow */
2.
3.  #include <GL/glut.h>
4.  #include "particle/papi.h"
5.  #include <iostream.h>
6.  #include <sys/time.h>
7.  #include <unistd.h>
8.
9.  // declaring global variable -----------------------------------------
10. // ----------------------------------------------------------------------
11.
12. int firstGroupNumber;
13.
14. // realclock() ---------------------------------------------------------
15. // ----------------------------------------------------------------------
16.
17. double realclock ()
18.
19. {
20.   struct timeval currtime;
21.   gettimeofday (&currtime, NULL);
22.
23.   double ct = currtime.tv_sec * 1000000.0 + currtime.tv_usec;
24.   return ct;
25. }
26.
27. // Settings for falling snow -----------------------------------------
28. // ----------------------------------------------------------------------
29.
30. void FallingSnow (int particleGroup)
31.   {
32.     static int count = 0;
33.     static int number = 15;
34.     int rate;
35.     if (count < 240)
36.       {
37.         rate = 0;
38.         if ((count % number) == 0)
39.           rate = 1;
40.         if (((count + 1) % 40) == 0)
41.           number -= 2;
42.         count ++;
43.       }
44.     else
45.       rate = 1;
46.     pCurrentGroup (particleGroup);
47.     pVelocity (0.0, -0.6 - (particleGroup * 0.1), 0.0);
```

```
48.     pSource(rate, PDRectangle, -50.0, 50.0, -50.0, 100.0, 0.0, 0.0, 0.0,
49.     0.0, 100.0);
50.     pSink (false,PDPlane, 0.0, -40.0, 0.0, 0.0, 1.0, 0.0);
51.     pMove ();
52.   }
53.
54. // draw() -------------------------------------------------------------
55. // ---------------------------------------------------------------------
56.
57. void draw (void)
58.   {
59.
60. // Settings -----------------------------------------------------------
61.
62.     glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
63.     glMatrixMode (GL_MODELVIEW);
64.     glLoadIdentity ();
65.
66. // Draw falling snow --------------------------------------------------
67.
68.     int ps = 1;
69.
70.     for (int i = firstGroupNumber; i < firstGroupNumber + 2; i++)
71.       {
72.         glPointSize (ps);
73.         FallingSnow (i);
74.         pDrawGroupp (GL_POINTS, false, false);
75.         ps++;
76.       }
77.
78.     glFlush ();
79.     glutSwapBuffers ();
80.
81. // Test the time taken between the rendering of frames -----------------
82. // (This includes the time to render a frame) --------------------------
83.
84.     static double totalTime = 0;
85.     static int countFrames = 0;
86.     static int flag = 1;
87.     static int flag2 = 1;
88.     static double startTime;
89.     double endTime;
90.
91.     if (flag2 == 0)
92.       {
93.         endTime = realclock();
94.         double timeTaken = (endTime - startTime)/(double)CLOCKS_PER_SEC;
95.         totalTime += timeTaken;
96.         countFrames++;
97.         if ((totalTime > 10.0) && (flag == 1))
98.           {
99.             cout << "There were " << countFrames << " frames rendered in
100.            " << totalTime << " seconds.\n";
101.            flag = 0;
102.          }
103.      }
104.    flag2 = 0;
105.    startTime = realclock();
106.
107.// reshape() -----------------------------------------------------------
108.// ---------------------------------------------------------------------
```

```
109.
110.void reshape (int w, int h)
111.  {
112.    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
113.    glMatrixMode (GL_PROJECTION);
114.    glLoadIdentity ();
115.    glOrtho (-50.0, 50.0, -50.0, 50.0, -50.0, 50.0);
116.  }
117.
118.// init() ---------------------------------------------------------
119.// ----------------------------------------------------------------
120.
121.void init (void)
122.  {
123.    glClearColor (0.3, 0.3, 0.4, 0.0);
124.    glEnable (GL_DEPTH_TEST);
125.
126.    glMatrixMode (GL_PROJECTION);
127.    glLoadIdentity ();
128.    glOrtho (-50.0, 50.0, -50.0, 50.0, -50.0, 50.0);
129.
130.// Create 2 particle groups -----------------------------------------
131.
132.    firstGroupNumber = pGenParticleGroups (2, 150);
133.  }
134.
135.// main() ---------------------------------------------------------
136.// ----------------------------------------------------------------
137.
138.int main (int argc, char** argv)
139.  {
140.    glutInit (&argc, argv);
141.    glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
142.    glutInitWindowSize (500, 500);
143.    glutInitWindowPosition (100, 100);
144.    glutCreateWindow ("Falling Snow");
145.    init ();
146.    glutDisplayFunc (draw);
147.    glutIdleFunc (draw);
148.    glutReshapeFunc (reshape);
149.    glutMainLoop ();
150.    return 0;
151.  }
```