Snow Rendering for Interactive Snowplow Simulation—Supporting Safety in Snowplow Design

Final Report

Prepared by:

Peter Willemsen

Department of Computer Science
University of Minnesota Duluth

Northland Advanced Transportation Systems Research Laboratory
University of Minnesota Duluth

CTS 11-04

Technical Report Documentation Page

1. Report No. CTS 11-04	2.	3. Recipients Accession No.
4. Title and Subtitle Snow Rendering for Interactive Snowplow Simulation—Supporting Safety in Snowplow Design		5. Report Date February 2011 6.
7. Author(s) Peter Willemsen		8. Performing Organization Report No.
9. Performing Organization Name and Address Department of Computer Science University of Minnesota Duluth 1114 Kirby Drive Duluth, MN 55812		10. Project/Task/Work Unit No. CTS Project #2008014 11. Contract (C) or Grant (G) No.
12. Sponsoring Organization Name and Address Intelligent Transportation Systems Institute Center for Transportation Studies 200 Transportation & Safety Building 511 Washington Ave. SE Minneapolis, MN 55455		13. Type of Report and Period Covered Final Report, FY 2008 14. Sponsoring Agency Code
15. Supplementary Notes http://www.its.umn.edu/Publications/ResearchReports/		

16. Abstract (Limit: 200 words)

During a snowfall, following a snowplow can be extremely dangerous. This danger comes from the human visual system's inability to accurately perceive the speed and motion of the snowplow, often resulting in rear-end collisions. For this project, the researchers' goal is to use their understanding of how the human visual system processes optical motion under the conditions created by blowing snow to create a simulation framework that could be used to test emergency lighting configurations that reduce rear-end collisions with snowplows. Reaction times for detecting the motion of the snowplow will be measured empirically for a variety of color set-ups on a simulated snowplow that slows down while driving on a virtual road with curves and hills. The simulated driving environment will utilize a head-mounted, virtual reality display to render an improved snow cloud model behind the snowplow. This driving simulator environment will serve as the basis for testing the effects of color and lighting alternatives on snowplows. The results of this work will move the researchers closer to determining optimal color and lighting configurations on actual snowplows.

17. Document Analysis/Descriptors Snow rendering, Snowplows, Safety, Snowplow safety, Simulation		18. Availability Statement No restrictions. Document available from: National Technical Information Services, Springfield, Virginia 22161	
19. Security Class (this report) Unclassified	20. Security Class (this page) Unclassified	21. No. of Pages 38	22. Price

Snow Rendering for Interactive Snowplow Simulation—Supporting Safety in Snowplow Design

Final Report

Prepared by:

Peter Willemsen

Department of Computer Science
University of Minnesota Duluth

February 2011

Published by:

Center for Transportation Studies
University of Minnesota
200 Transportation and Safety Building
511 Washington Ave. S.E.
Minneapolis, MN 55455

Acknowledgements

The authors(s) wish to acknowledge those who made this research possible. The study was funded by the Intelligent Transportation Systems (ITS) Institute, a program of the University of Minnesota's Center for Transportation Studies (CTS). Financial support was provided by the United States Department of Transportation's Research and Innovative Technologies Administration (RITA).

The project was also supported by the Northland Advanced Transportation Systems Research Laboratories (NATSRL), a cooperative research program of the Minnesota Department of Transportation, the ITS Institute, and the University of Minnesota Duluth College of Science and Engineering.

The author(s) would like to thank the students at University of Minnesota Duluth who have contributed to this project: Siddharth Deokar, Michele Olsen, and Daniel Schobert.

Also, thanks to Dr. Albert Yonas, Dr. Lee Zimmerman, and Dr. Peter Shirley.

Table of Contents

List of Tables

List of Figures

Executive Summary

In snowing conditions, following a snowplow creates an extremely dangerous situation. The danger comes from the human visual system's inability to accurately perceive speed and the motion of the snowplow, often resulting in rear-end collisions. The overall objective in our research is to use our understanding of how the human visual system processes optical motion, under the conditions created by blowing snow, to create a simulation framework that can be used to test emergency lighting configurations on snowplows. We feel that such a simulator has the potential to reduce rear-end collisions with snowplows as it could be used to test a variety of dangerous driving scenarios. In our simulation framework, we plan to measure reaction times for detecting motion of the snowplow to achieve empirical measurements for a variety of warning lighting set-ups on a simulated snowplow that slows down (and speeds up) while driving on a virtual, curving hilly road. This driving simulator snow environment will serve as the basis for testing the effects of color and lighting alternatives on snowplows. The results of this work will move us closer to determining optimal color and lighting configurations on real snowplows.

During the previous year of work, our efforts have been on developing a simulation framework capable of rendering snow in a real-time virtual driving environment. In particular, we have created the base simulation framework that will be used in future years for representing snow in a real-time driving simulation. Our current snow rendering system is capable of rendering a few million snow particles at interactive rates. These snow particles are subject to an average wind field and reflect off of simple geometric structures in the simulation environment. This allows us to move the snow particles during a simulation to mimic the effects of blowing snow in a road-based setting. We also developed a 3D model of a snowplow to use in our simulations. The snowplow is quite realistic and has been constructed to easily allow interchangeable back ends with different lighting configurations.

Our approach is unique and powerful because we compute the effect of snow falling between an observer and the background on a per-snow particle basis. This is an important feature because we achieve critical environmental influences of the snow from this model, such as blurring of elements in the background and a lowering of the contrast between background and foreground elements. The latter is important for addressing the perceptual issues associated with driving in snowy conditions [1-3]. Equally important is the fact that each particle moves under the influence of a physics-based wind dispersion model. The lighting and motion combine to produce more realistic circumstances. Other research has incorporated spectral methods to represent the effect of falling snow [4]. However, our proposed approach can result in more realistic looking scenes because we explicitly model the falling snow and its effect on the environment. Our approach is also in contrast to methods that treat participating media (fog, haze, or air molecules) as an aggregate [5, 6], deriving functions for how light attenuates across distance rather than dealing with individual particles. Aggregate methods, however, may be important for simulating the influence of snow at distances far from the observer since our snow rendering framework is somewhat limited by the number of snow particles that can be rendered at real-time rates.

In our snow-rendering framework, we compute a lighting equation for each snow particle. This is

feasible due to the computational power and parallel processing available in modern graphics hardware. Current graphics cards, typically used to accelerate video game computations, contain Graphics Processing Units (GPU) that are highly parallel vector processors. Typically, these processors are used strictly for computer graphics rendering equations. However, the literature on graphics hardware programming contains many examples of GPUs being used to accelerate scientific, engineering, and graphics applications. In many of the cases, GPU-based implementations greatly outperform their Central Processing Unit (CPU)-counterparts, sometimes by several orders of magnitude. This increase in performance relates to the GPU's SIMD (single-instruction, multiple data) style of processing.

The lighting calculation applied to each snow particle attempts to characterize the absorption, emission, and scattering of incident light on the snow particle. These components are related to the optical properties of snow. For packed or aggregate snow, researchers have attempted to determine the optical properties for the light-snow interaction [7,8]. The optics of falling snow is different, but few studies applicable to graphics have been conducted to directly determine the optical properties of falling snow [9]. Yonas and Zimmerman [1] have attempted to make some preliminary measurements for light propagation in snowy conditions and we intend to integrate that information as best we can into our model. It may be important to attempt new measurements at some point over this project.

Our snow simulation runs at interactive rates performing basic forward scattering from the lights defined in the scene. While we have not yet integrated the snowplow with the snow simulation, we are able to show the effects of the forward scattering in our test scenes. With the work conducted over the past year, we are able to begin work on a more integrated simulation framework that affords experimental testing of alternative lighting configurations on the back snowplows.

Chapter 1. Introduction

Our ability to perceive motion in general and optical expansion in particular is crucial for safe driving. Expansion indicates that we are approaching the car ahead of us. In previous work [1,2], two situations were found that interfere directly with our perception of the expansion motion that alerts drivers that collision is imminent: fog and blowing snow combined with the color of the vehicle and the color of the surrounding road can create a dangerous equiluminant situation. In an equiluminant situation the brightness of the vehicle and the background are equal. This can also be described as a low luminance contrast situation since the contrast between foreground and background luminance is minimized. Luminance can be thought of as the amount of light intensity, but not color, that comes from a surface. When equiluminant, or low luminance contrast situations are present, our ability to detect motion is reduced as well as our ability to locate objects in space [3]. In particular, these situations present themselves under snowing or foggy conditions. Also of note, flashing lights, such as those used to improve detection of snowplows in poor visibility conditions, interfere with our ability to sense approach [2]. Our past data indicate that daytime driving behind an amber colored snowplow with amber flashing warning lights strongly reduces our ability to sense approach, increasing the potential for rear-end collisions with snowplows.

The research issue addressed in this report is on developing a blowing snow visual simulation framework that can be used to investigate how blowing snow (and even foggy) conditions can affect perception of exocentric vehicle speed, motion, and general detection. Such a simulator must be capable of providing visual information in real-time based on changing conditions. Our efforts over this first year have been on developing the infrastructure to support the rendering of falling snow. The objective of this system is to create equiluminant, or low-luminance contrast conditions, in the visual simulation that can be used to better understand our behavior under these adverse situations. Physical measurements were acquired in previous research to gauge how falling or blowing snow filters the color components of the light that reflects from the painted surfaces of a snowplow [1]. Computer-based psychophysical studies of the effects of luminance contrast and flashing displays on our ability to detect approach were also been previously conducted [2]. The information obtained from these past measurements have prompted and influenced the implementation of our current snow simulation and rendering algorithm. Our objective is to provide a more realistic simulation of blowing or falling snow that can be used to advance this prior work, ideally creating safer winter driving conditions by applying our knowledge to modification of the snowplow fleet.

The current snow rendering system can display approximately three million opaque snow particles at real-time rates. The speed and computational power of modern graphics cards is increasing regularly so it is expected that the number of snow particles that can be rendered to the screen with our system will increase over time. A turbulent wind dispersion model has been incorporated into the snow rendering algorithm to provide an animation of falling or blowing snow, effectively increasing the realism of moving particles. Our particle dispersion system performs advection of snow particles according to mean and fluctuating wind quantities using an unsteady, random-walk turbulence model [10,11]. Particles are reflected off of the ground and simple, geometric structures. The resulting motion is characteristic of real fluid flows in empty space and also around buildings.

The overall goal of this research tact is to create safer winter driving conditions by applying information about human perception to the design and configuration of snowplow lighting and paint color. To accomplish this, we are building a virtual driving simulation environment in which falling or blowing snow is rendered and animated. Using the simulator system, we can experiment with different snowplow lighting configurations and snowplow paint colors. Through this virtual prototyping system's results, we expect to be able to provide reasonable input for real-world tests and experiments validating any alternative configurations.

The key components of the research in this project were to (1) develop an effective visual simulation of snow for use in a virtual environment, and (2) begin experimental analysis of how human perception under these circumstances is affected.

The remainder of the report will highlight the information from the first year of this project, funded by the NATRSL FY 2008 program. Chapter 2 provides background information on the state of snow rendering and also discusses the mechanism by which snow is moved within our simulation framework. Chapter 3 provides the snow rendering implementation details, including the wind turbulence and particle advection, scattering, snow particle sorting, and display. Chapter 4 will present the results from the first year, and provide guidance on how the project should proceed in future years to provide the most benefit to other researchers in this area.

Chapter 2. Background and Related Work

Very little work has been published on the subject of modeling snow in real-time situations, despite the fact that snow is such a common material when rendering natural scenes. Stationary, fallen snow has been modeled previously by Chrisman [12] using the optical characteristics of ice and snow. Real-time Cloud rendering by Harris and Lastra [13] presents methods for simulating realistic clouds using multiple scattering in the light direction. Clouds are illuminated by multiple directional light sources with scattering from each one. Our method uses imposters to accelerate cloud rendering by exploiting frame to frame coherence. Using imposters is an effective way to render clouds that may contain other objects, such as airplanes or birds. This is important for snow rendering since snow clouds and snowing situations often have objects within them, such as cars or snowplows. Wang and Wade [14] have modeled a snow domain with static textures without using a particle system.

Our system uses a dynamic particle system to simulate particles in our virtual environment. The particles in the system use a wind simulation model to simulate the particle motion within the confines of the simulated environment. The particles move about in the environment similar to how wind would cause them to disperse. We also have a collision detection mechanism in place where in the particles can collide against simple structures such as buildings, the ground, or a rough approximation to a snowplow, thus changing particle motion.

We use the QUIC-Plume dispersion model that was reworked to run on the GPU [10, 11, 15]. The movement of the particles is decided using the wind field generated by QUIC-URB along with turbulent fluctuating winds. Particles are released into a domain with an initial position given by a source. The source can either be a point source, a line source, a sphere source or a plane source. Particles once released from the source will travel until they are outside the boundary of the domain. Figure 2.1 shows screen captures from this system.

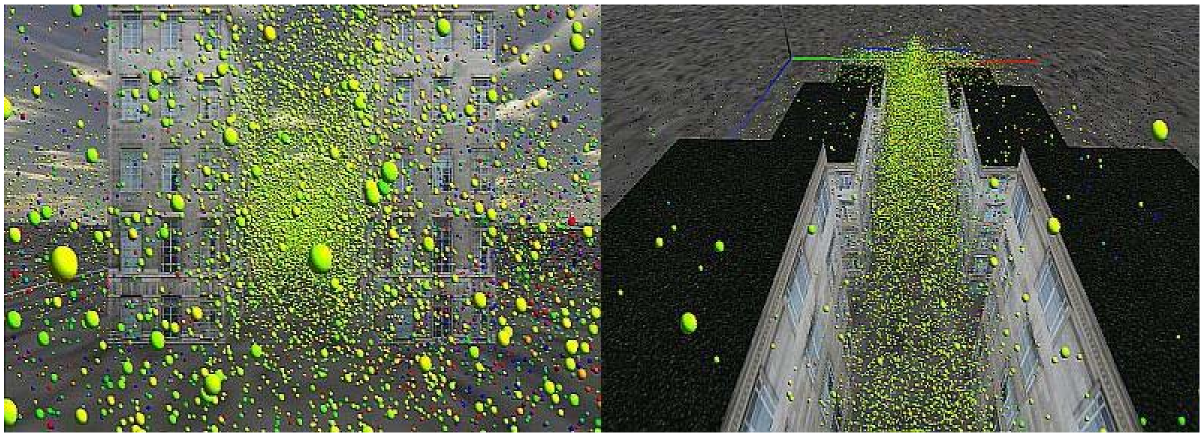


Figure 2.1: A virtual environment of buildings with the colored particles. The particles collide against the buildings and change their direction.

The particles in our system are modeled using point sprites. Point Sprites enable us to model 3D objects as single points on the screen using 2D texture images for mapping them. This reduces the load of drawing many 3D objects such as spheres in case of particles. In short, we can draw many points with 2D textures that will look like 3D spheres. Point Sprites make it easier to render millions of particles in real time, which we require to make our system interactive. It is necessary that we render at fast rates for the interactive environment to work well. For example, if a user was driving behind a car, he should immediately be able to brake if the car in front of him stops. This will be possible to render if we have enough time between a user braking and rendering it on screen. Generally we want to render a frame at a rate which feels real to the user. Typically the refresh rate for a computer screen is 1/60 of a second. So ideally we should render 60 or more frames per second for the effects to be real.

To speed up the process of rendering a million particles per frame, we can parallelize the computations using the graphics processor. A graphics processor is a highly parallel processing unit dedicated for rendering graphics onto the screen, which is also known as a GPU (Graphics Processing Unit). A GPU is a vector processor that contains highly parallel stream processors used to display real-time 3D graphics. Previously, the CPU used to do all the calculations necessary for rendering an image onto the screen. In particular, the CPU had to calculate the color of each pixel per frame. As the scenes in graphics became more and more complex over time, we needed a dedicated processor for graphics calculations. A GPU is meant for rendering high end graphics, which works on the Single Instruction Multiple Data (SIMD) architecture and it is capable of doing many floating point calculations simultaneously.

Before we go any further with the graphics processor, we need to understand the basic idea of graphics and the graphics processing pipeline. We can draw complex objects on the screen by using 3D geometry. We can either apply color to these objects or apply textures to them. For example, we can apply a tile texture to a floor to enhance its appearance. These 3D objects are projected onto the 2D screen after being processed through the graphics pipeline. An object undergoes various transformations before it gets rendered on the screen. A 3D object is first converted from local coordinates to viewing/eye coordinates which are the coordinates of the object relative to the camera. This is the Viewing or Modeling transformation. Then we define the viewable volume in the Projection Transformation which will clip the parts of objects which are not visible to the camera, and then the Viewport Transformation converts the 3D coordinates to 2D coordinates on the screen. As we render objects to the screen, every object is broken down into individual fragments or pixels and sent to Fragment Processing. Here we compute the color of every pixel before rendering it to the frame buffer. These per vertex and per fragment calculations can be parallelized using the stream processors on the GPU for fast rendering.

The fixed functionality of a graphics pipeline can be overwritten with the use of vertex and fragment processing units called shaders. Shaders are low level programs written using specialized C-like languages such as OpenGL Shading Language and Cg. By using shaders, programmers have more control over their applications to create better graphics and increase performance. The vertex shader overwrites the per vertex stage in the pipeline while the fragment shader overwrites the per fragment stage of the pipeline, giving us control on every fragment or pixel on the screen. By overwriting parts of the fixed functionality of graphics pipeline we get a low level access to the graphics card.

Even though we have moved a large chunk of the data and operations to the GPU, there still remains some necessary communication between the CPU and the GPU. The data representation of graphics is done on the CPU and then sent to the GPU for processing. Also, many times the

CPU needs to send small bits of information over to the GPU for use in the shaders. So the challenge is to minimize the use of CPU memory and reduce the CPU-GPU communication.

To sum up, the GPU’s have enabled programmers to come up with complex and realistic looking scenes in 3D games and high end 3D rendering. GPU’s are very flexible for a wide variety of computations, and in many circumstances execute the operations faster than on a CPU.

The use of textures forms the main technique to manipulate GPU memory through programs. Textures can be considered as a form of the main memory in the GPU. A texture is an array of vectors where each vector, called a texel, is a color defined by red, green, blue and alpha values. Normally these values are read in the fragment shader and the color retrieved from the texture is applied to the corresponding pixel/pixels on the screen depending on the texel-pixel mapping. Generally, textures can be one, two or three dimensional. We can also use these textures to store other information, such as position of particles, normals of the particles, or any other information that we want to manipulate using the shaders.

We use textures extensively to store and manipulate the particle data. For example, in the following figure we can see a 2D texture of size 4X4 which will hold positions for 16 particles. Each particle position is a vector of 4 floating point values that store the x, y, z, w values. Each cell in the texture holds a position vector for a particular particle.

12	13	14	15
8	9	10	11
4	5	6	7
0	1	2	3

Figure 2.2: 2-D texture of size 4 X 4 holding 16 particle positions with each position being a vector (x, y, z, w).

Figure 2.2 shows how a 2D texture is represented on the GPU. Each colored square is an individual texel, which is a vector of four values. The size and dimension of the textures will change for different number of particles. For example, we will need a 2D texture of size 1000X1000 to store 1 million particles. The order of the numbering shown in the figure is the way in which the data is stored in the texture. For example, before texture creation data is stored on the CPU as a 1D array. For figure 2.2, the 1D array used to create a texture is of size 64, (4 X 4 = 16 texels and 16 X 4 values per texel = 64) with the values for every texel placed in the array according to the numbering shown. The first four values for texel 0 are placed one after another in the array, the next four values for texel 1 are placed one after another following the four values of texel 0, and this process continues for each texel. We can access a texture in a vertex/fragment shader using the texture coordinates. The texture coordinates give the location for a specific texel in a texture. For example, the texel 9 can be accessed using the texture coordinates s = 1 and t = 2, where s is the column and t is the row number, as the indexing of rows and columns starts at 0.

Chapter 3. Snow Rendering Implementation

Our snow particles are implemented on the GPU at real-time rates making it possible to render effects such as light scattering, particle depth sorting, and blending for more effective display. This chapter gives a detailed explanation of how our snow model works. First, an overall view of the system is outlined. Then the modeling of individual snow particles is described. Following that is the description of how we used sorting and blending on the GPU. Then, the process of creating aggregate snow is explained. Finally, there is a detailed explanation of how high dynamic range is used in rendering.

3.1 System Overview

We model the snow particles using transparency and scattering effects and do the calculations for these effects using dynamic lights on the snowplow model. The shapes of the snow particles are modeled using a Gaussian transformation for translucency.

We use point sprites as representations for particles [15] for modeling our snow system. The particles are first sorted or alternatively blended, and then modeled as snow. We model each snow particle as a transparent surface using a Gaussian transformation and then apply scattering, which is a result of potentially many dynamic lights in the scene and their interaction with each snow particle. The following is a generic display function, which shows how our system code is organized to get the final image with our snow model.

```
DisplaySnow
{
      1. Sort the particles based on distance from eye [Section 3.3]
            OR use additive blend [Section 3.4]
      2. Model snow particle
            Get the number of lights in the scene
            Calculate color for each particle [Section 3.2]
            Calculate effect of every light on the particle [Section 3.2.4]
                  Apply Gaussian Transformation [Section 3.2.2]
                  Apply Scattering Function [Section 3.2.5]
      3. Render final image
}
```

3.2 Modeling of Snow

3.2.1 Point Sprites

As we have discussed in the background section, we use point sprites for our snow particles. We draw these points in space and map them with 2D textures. These are also called *imposters* as they always face the user and they are made to look like 3D points using 2D textures witin the graphics subsystem. For imposters to work, we store the normal vector at each point and calculate the color at that point using its orientation to the light source so that they look like spheres.

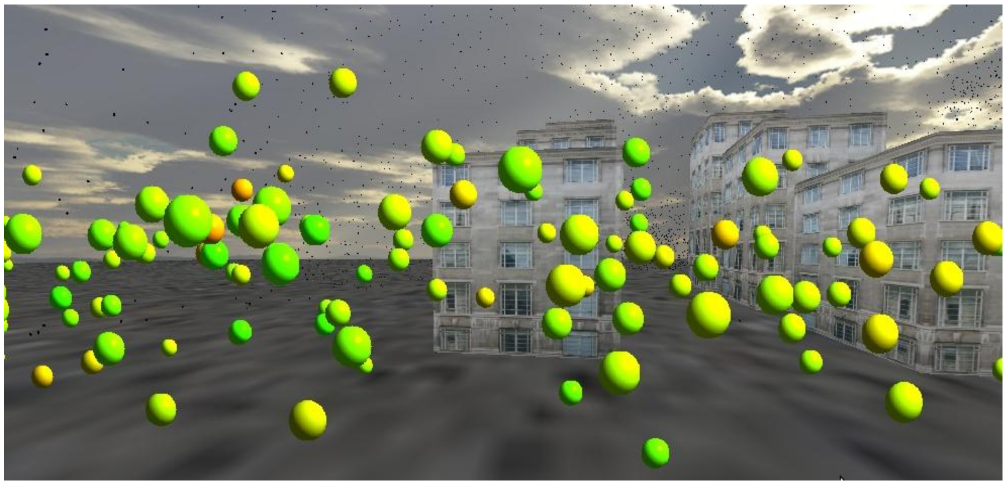


Figure 3.1: Particles rendered as a single points with a 2D texture to give the impression of a 3D sphere.

In Figure 3.1, we can see particles that look like spheres, but are actually point sprites mapped with a 2D texture to look like spheres.

3.2.2 Transparency

The snow particles that we see in reality are translucent white particles. These semi-transparent particles interact with light hitting them, often resulting in a scattering of the light at the point where it impacts the particle. We model our snow particles transparently to allow blending of different snow particles. We model the transparency of snow based on the Gaussian function which is of the form:

$$f(x) = ae^{-\frac{(x-b)^2}{2c^2}}$$

The graph of a Gaussian is a symmetric bell-curve (see Figure 3.2) which peaks at the center and gradually falls off as we move away from the center. We use this function to give the snow opacity at the center with increasing transparency as a function of distance from the center of the snow particle.

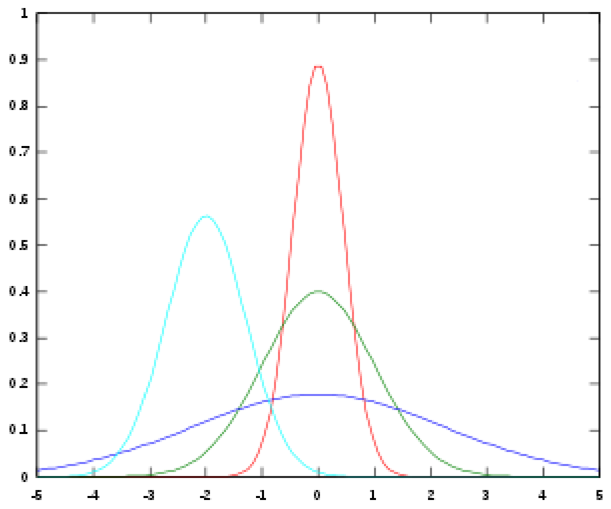


Figure 3.2: Gaussian Function Curves different values of a, b and c.

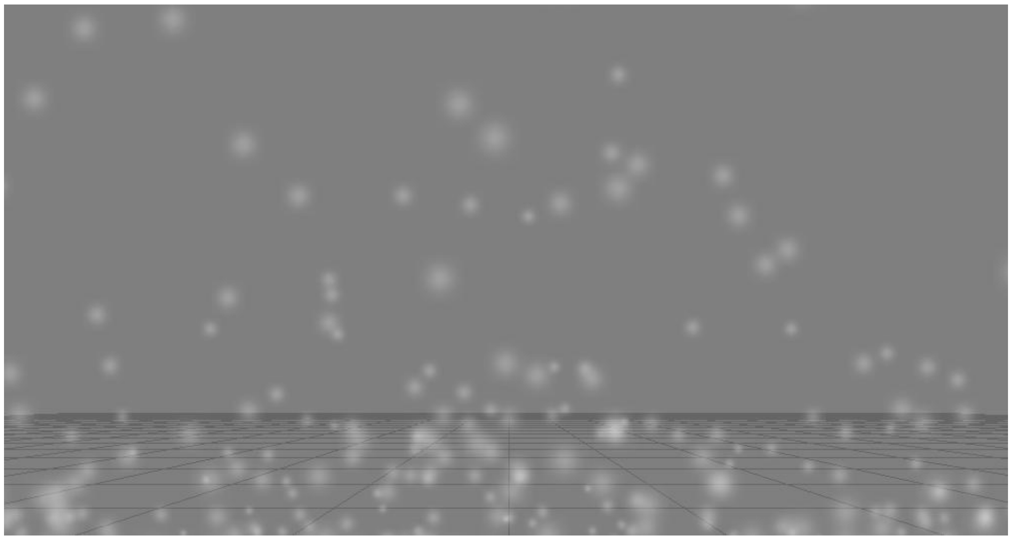


Figure 3.3: Transparency using the Gaussian Function.

The above figure shows transparent snow particles with the Gaussian function applied to them. They are opaque at the center and become transparent as you go away from the center. We use the two dimensional Gaussian function which is given by,

$$f(x,y) = Ae^{-\left(\frac{(x-x_0)^2}{2c_x^2}+\frac{(y-y_0)^2}{2c_y^2}\right)}$$

Here the coefficient A is the amplitude, x_o,y_o is the center and c_x, c_y are the *x* and *y* spreads of the Gaussian.

The color of a particle is given in the RGBA format where R specifies the red component, G decides the green component, B indicates the blue component, and A is the alpha value which defines the transparency. Hence, as explained earlier, we are calculating the alpha values for each particle using the Gaussian function.

3.2.3 Dynamic Lights

We use dynamic lights in our environment. These lights represent the lights on the back of the snowplow and can be turned on or off during the simulation. To facilitate use with our snow system, the light state is stored in a texture. We can have many lights active in the environment, each light is defined by the following parameters:

1. Data – It is a vector of 4 floating point values.
   Light Exists – Specifies whether a light source exists
   Is Light On – Specifies whether the light source is on or off
   Type – Point, Spot, or Directional Light
   Flashing – Specifies whether the light is flashing or not flashing
   For Example: Flashing lights on a snowplow
2. Position – Position of the light source (4 valued vector)
3. Intensity – Color of the light source (RGBA format)
4. Direction – Direction of light source which is a vector of 4 values is required for light calculations
5. Spot Light Parameters – Specifies the spotlight parameters if the light source is of type, spotlight. It is a vector of 4 floating point values.

An example of a texture with light sources can be seen in the following figure. In Figure 3.4, we have a texture of size 4 X 5, with each cell representing a vector of 4 floating point values. We have 4 rows representing 4 light sources, and for each row (light source) we have 5 columns specifying the light source parameters.

Data$_1$	Position$_1$	Intensity$_1$	Direction$_1$	Spotlight Parameters$_1$
Data$_2$	Position$_2$	Intensity$_2$	Direction$_2$	Spotlight Parameters$_2$
Data$_3$	Position$_3$	Intensity$_3$	Direction$_3$	Spotlight Parameters$_3$
Data$_4$	Position$_4$	Intensity$_4$	Direction$_4$	Spotlight Parameters$_4$

Figure 3.4: A texture of size 4 X 5 where each row represents a light source and individual columns represent the light source parameters.

The texture is loaded into memory and is accessible by our snow rendering shader programs. The lights get updated every time there is a change in the light state. For example, the texture gets updated for flashing lights (on/off status) and for brake lights (brake applied / not applied). Every time we want to calculate the color of a particle, we read the light source texture and get all the active light sources and then do the required calculations for determining the color.

```
For every snow particle do
{
      Read the light source texture
      Get the active light sources
      For every active light source
      {
            Compute the intensity of the light source
            Calculate scattering by Henyey-Greenstein phase function
            color = color + (phase * intensity + ambient color)
      }
      Calculate the alpha value of color by the Gaussian function
}
```

3.2.4 Scattering

For scattering of light through the snow particles we could require tracing the path of light through every snow particle using ray tracing. As explained earlier, ray tracing is computationally expensive and cannot be used in interactive environments. Hence we use an approximation to scattering given by the Henyey-Greenstein Phase function. The Henyey-Greenstein phase function is used to characterize the angular distribution of scattered light and is characterized by the average cosine of the scattering angle, g. In this phase function, a single parameter g, which is also called the asymmetry parameter, controls the distribution of scattered light.

$$p_{HG} = \frac{1}{4\pi}\frac{1-g^2}{(1+g^2-2g(\cos\theta))^{\frac{3}{2}}}$$

The values of g must be in the range (1, -1) with negative values corresponding to back scattering and positive values corresponding to forward scattering. In back scattering, light is scattered back in the direction of incident light. For example, the light scattered by the headlights of your car would be back scattering and the light scattered by the tail lights of the vehicle in front of you would be forward scattering.

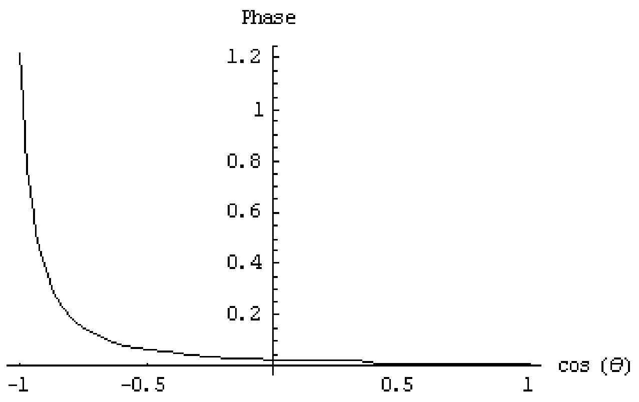


Figure 3.5: Plot of Henyey-Greenstein phase function for g =- 0.67.

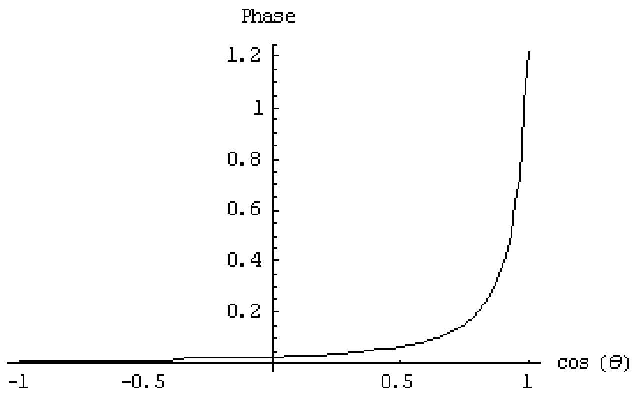


Figure 3.6: Plot of Henyey-Greenstein phase function for g = 0.67.

In the plots for the Henyey-Greenstein phase function, the curve in Figure 3.5 denotes back scattering (g = -0.67) and curve in Figure 3.6 denotes forward scattering (g = 0.67).

Hence for every snow particle, we do the following:

```
{
     Calculate the vector from the eye to the snow particle
     For every active light source do
     {
          Calculate the vector from light source to the particle
          Calculate cosine of the angle between the two vectors (dot
          Set g for back or forward scattering
          Calculate the phase by Henyey-Greenstein function
          Calculate the color as:
               color = color + (phase * intensity + ambient color)
          Sum the colors calculated for each light source
     }
}
```

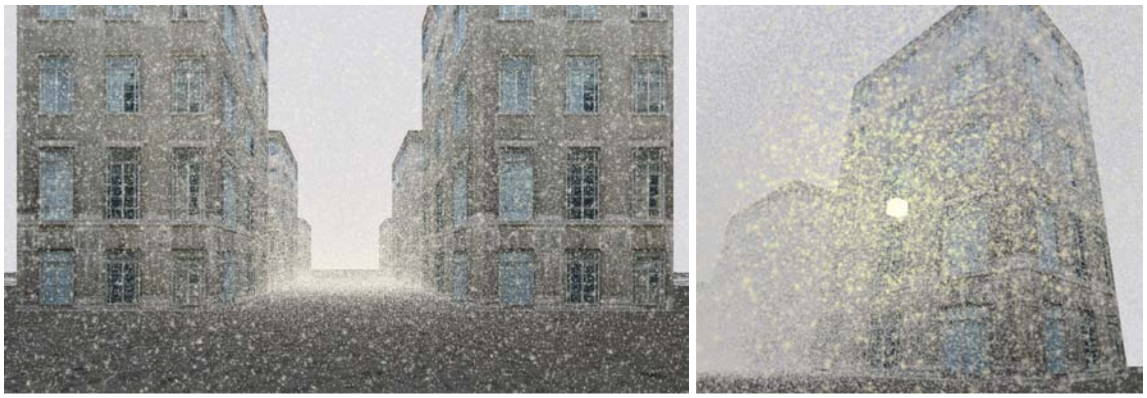


Figure 3.7: Color screen captures from the current snow rendering system. One million particles were released into each scene. The right image shows direct use of the phase function in which a streetlight illuminates particles as they pass by the viewer.

We are using the Henyey-Greenstein phase function to compute the scattering for each light source on the particles in the scene. That is, we can calculate scattering of light for the snow particles between the viewer and the snowplow. These include the lights from the following vehicle's headlights, the lights on the rear of the snowplow, and more importantly, the stroboscopic lights mounted on the top of the snowplow. Mimicking the stroboscopic lights is somewhat possible because our approach is dynamic and the lighting calculation is computed on a per-particle basis. The stroboscopic lights will be implemented as light sources of varying intensity over very small time intervals. While we may be able to roughly approximate the stroboscopic effect of a rapidly changing on/off sequence for the light, it is non-trivial to convey the brightness of the lights to a viewer on a LCD screen. Future efforts will investigate ways in which we can more effectively transmit the brightness of the snowplows lights to a viewer. We expect that techniques like High-Dynamic Rendering (HDR) may be useful.

The phase function is used in combination with an ambient light term and hardware-assisted transparency blending to simulate the accumulative effect of snow particles over distance on the rendered scene. Sorting of snow particles from the back of the scene to the viewer is required to make the transparency consistent. To achieve this, we have implemented GPU-based sorting techniques [16,17] to afford sorting of large numbers of particle positions and speed up particle rendering. This is described in detail in Chapter 3.

Figure 3.7 shows two images captured directly from our current software implementation. Note that these images are still considered preliminary work as we are constantly modifying and improving the snow rendering system. In these simulations, one million particles were released into the wind dispersion model. Snow particles were transported via the wind model between the buildings. The right image shows a yellowish orange streetlight (the white cube) illuminating snow particles as they pass between the light and the viewer. This is an example of the phase function described above. In general, note the low luminance contrast that develops as a result of the snow particle interaction with the background elements (e.g. buildings and ground).

3.3 Sorting

The snow particle positions are stored in 2D textures. A 2D texture is specified by its width and height. A 2D texture of size width X height can store width X height number of particle positions. Each particle position is a vector with 4 floating point values stored in a single cell of a texture. Each cell in the texture corresponds to a particle position.

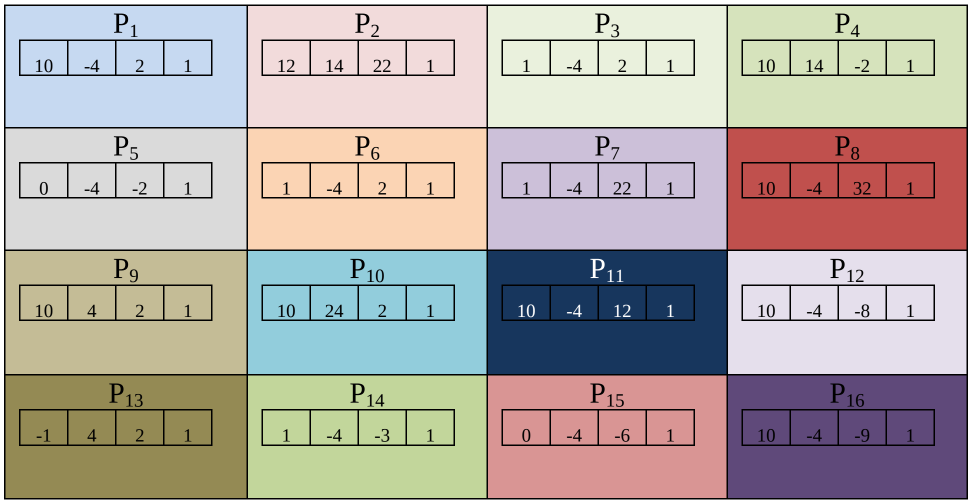


Figure 3.8: Texture of size 4 X 4 storing positions of 16 particles.

In the above figure (Figure 3.8), we can see a 4 X 4 2D texture containing 16 particles with each cell storing the position of the corresponding particle. The particles are rendered to the screen as they are read from the texture. Ideally the objects farthest from the eye should be rendered first and then the next nearer ones. This is important in our case as we are rendering translucent snow particles in our environment. For example, when you draw a transparent object, all the objects behind it should be rendered first so that you can see the objects behind through the transparent object. But in OpenGL, the objects are drawn in a sequence which is provided by the user in his code. If the user does not take care of sorting the transparent objects from the viewer position (eye) we will have artifacts in the scene in which a transparent object covers another object behind it. Hence we require a sorting mechanism which will sort the particles before rendering them to the screen. In this section, we will go through the sorting mechanisms used, quick sort and the GPU sort.

3.3.1 Quick Sort

We use the quick sort algorithm to sort particles every time, before rendering them to the screen. The sorting is done on the CPU using the Quick Sort API. When we want to draw the particles on the screen, their positions are stored in the vertex buffer on the GPU. Hence, we first map the particle positions from the vertex buffer to CPU memory, sort them and then unmap the vertex buffer for rendering. This is a very slow process, since for every frame, we have to map the GPU memory to the CPU memory for sorting. Hence, we use quick sort only for purpose of comparison with GPU sort.

```
Quick Sort
{
     Map vertex buffer of particle positions to CPU memory
     Sort the particles based on particle’s distance from the eye
     Unmap the vertex buffer from CPU memory
}
```

3.3.2 GPU Sort

The purpose of using the GPU parallelism for rendering the particles gets defeated if we use the CPU to sort the particles every frame. The GPU has to sit idle every time we send the particles to the CPU to sort. Instead, if we sort the particles on the GPU itself, we can use the inherent parallelism that comes with a GPU and also get rid of the CPU-GPU communication. Sorting on the GPU has been proven to be much faster than using the best sorting algorithms on the CPU. But most sorting algorithms such as quick sort cannot be implemented on the GPU, as it cannot write to arbitrary memory locations. The GPU does not support this functionality to avoid write after read operations by different stream processors when accessing the same memory location. A GPU sort algorithm has been implemented [17], which uses texture mapping to implement a bitonic sort algorithm. I have modified their algorithm to suit the requirements of our system.

3.3.2.1 Bitonic Sort

The bitonic sort takes a bitonic sequence as its input which is a sequence which has at most one local minimum or maximum.

Examples: 1,2,3,4,5,6,7,8   2,5,8,10,7,4,3,1   9,8,5,3,6,10,11,12

When you break a bitonic sequence at the minimum or maximum you get two sorted sequences, one ascending and the other descending. In the second set above, the maximum is 10 and splitting the sequence at 10 gives us 2,5,8,10 and 7,4,3,1 which are sequences in ascending and descending order respectively. In the third example, the minimum is 3 and splitting the sequence at 3 gives us 9, 8, 5, 3 and 6, 10, 11, 12 which are sequences in descending and ascending order respectively.

Bitonic sort operates by splitting the bitonic sequence into 2 equal halves (binary split), and then compares the two halves and switches the necessary elements. This process is repeated recursively until it will have just a single element in a sequence. In the end all the elements are combined to give a sorted sequence.

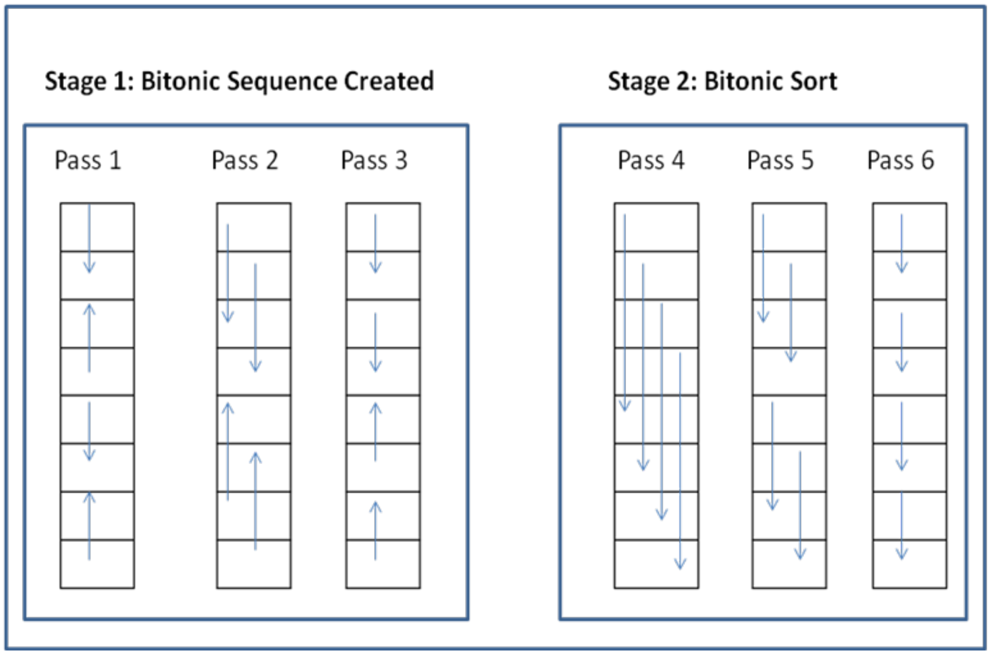
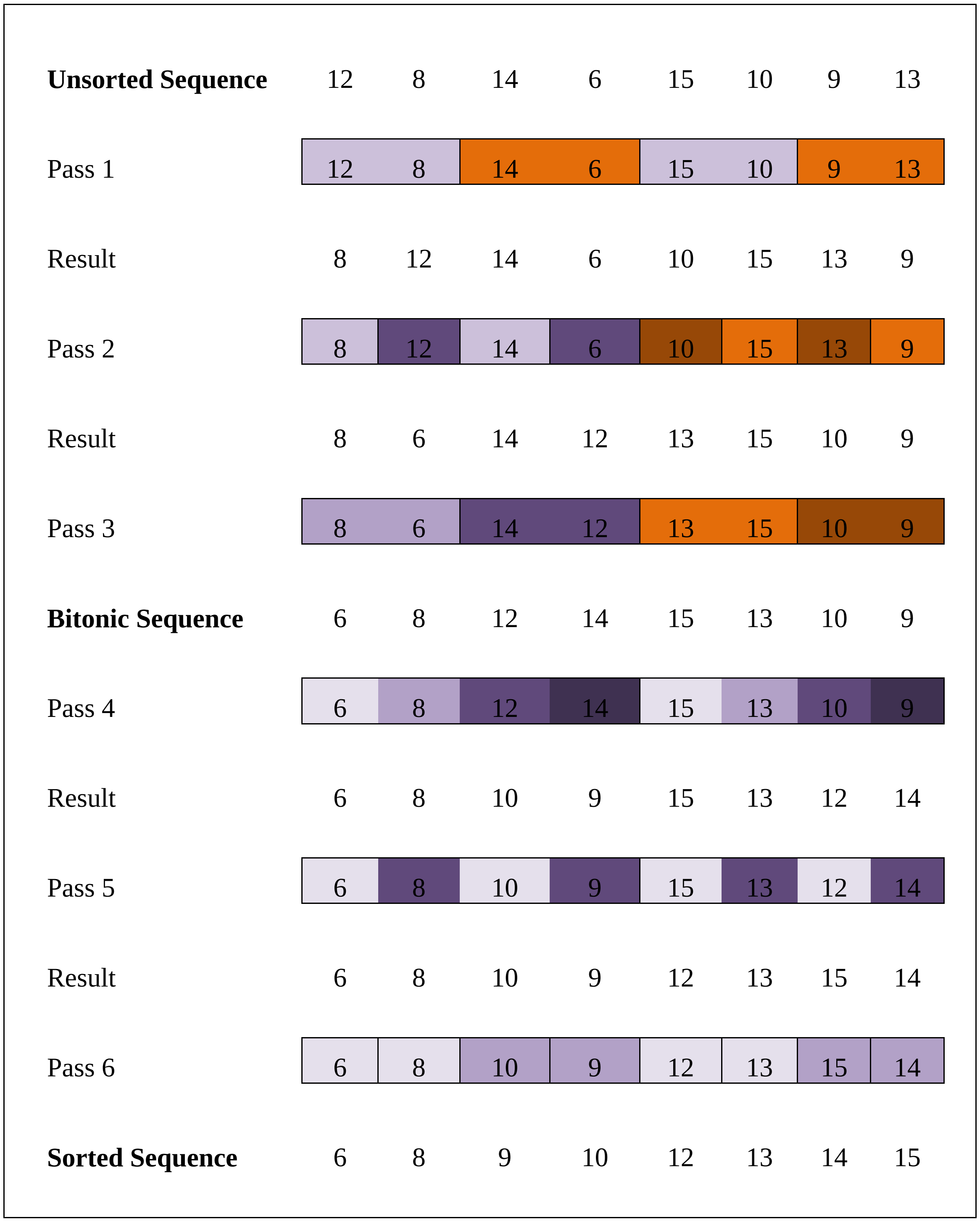


Figure 3.9: Process of bitonic sort.

Figure 3.9 illustrates a bitonic sort network on 8 data values. Each arrow between two elements indicates comparison between the two values. The maximum of the two is stored in the location pointed by the arrow head and the minimum is stored in the other location. We use textures to store the distances which need to be sorted. In every pass we apply a shader to the texture, which does the operations on the data in the texture for the given pass. We first get a bitonic sequence as shown in the above figure and then we apply bitonic sort to the texture to get the sorted values.

For example, consider the following unsorted sequence shown in Figure 3.10. In purple shaded cells, we compare whether first < second and in brown shaded cells we compare first > second. We get a bitonic sequence after the first three passes after which we are ready to apply bitonic sort to the sequence. We split the bitonic sequence into 2 halves and compare the elements in the two halves. After the bitonic sequence is generated we always compare for first < second. In every pass we divide the bitonic sequence further into two halves and compare the values and swap if necessary till we get 1 value sequences or until the bitonic sequence cannot be further subdivided.



Unsorted Sequence	12	8	14	6	15	10	9	13
Pass 1	12	8	14	6	15	10	9	13
Result	8	12	14	6	10	15	13	9
Pass 2	8	12	14	6	10	15	13	9
Result	8	6	14	12	13	15	10	9
Pass 3	8	6	14	12	13	15	10	9
Bitonic Sequence	6	8	12	14	15	13	10	9
Pass 4	6	8	12	14	15	13	10	9
Result	6	8	10	9	15	13	12	14
Pass 5	6	8	10	9	15	13	12	14
Result	6	8	10	9	12	13	15	14
Pass 6	6	8	10	9	12	13	15	14
Sorted Sequence	6	8	9	10	12	13	14	15

Figure 3.10: Example of bitonic sort.

3.3.2.2 Sorting Snow Particles

We pass a texture with the values to be sorted to the GPU sort and it returns a texture with sorted values. Internally, the sort uses two textures to perform the comparison and swapping operations. Now let us see how we need to modify this sort to suit our particle system.

The particles are stored in a 2-D texture which hold the position of every particle as a 4-D vector containing values <x,y,z,w> . We specify the number of particles in the system by the width and height of a texture. For example, a texture of width 1024 and height 1024 will contain 1024*1024 = 1 million particle positions. The actual size of the texture is width*height*4 as each particle is a vector with 4 values.

We then calculate the distance of each particle from the eye using the Euclidean distance which is given by,

$$\boldsymbol{D}(\boldsymbol{x},\boldsymbol{y}) = \sqrt[2]{\sum_{i=1}^{d}(\boldsymbol{x}_i - \boldsymbol{y}_i)^2}$$

where D(x,y) is the Euclidean distance between 2 points and d is the number of dimensions. These distance values are stored in another texture which is of the same size as the positions' texture. For example, when we calculate distance d of particle p from the eye then the value in texture for particle p would be <d,d,d,d>. This is because the fragment processor requires the memory to be of the same size to read from and to write to. Hence the position and distance textures have to be of the same size. Once we have the texture for distances ready, we must reduce the size of the texture from width*height*4 to width*height so that we have only one distance value corresponding to each particle. After reduction, we have a new texture of size width*height corresponding to the number of particles.

Now, what we have are the distances of the particles from the viewer and we need to index these distances so that we can keep track of the corresponding particles. So we create a new texture of size width*height which will hold the indices of the particles. Then we send these two textures (distance & indices) to GPU sort, which is modified to sort the distance texture and simultaneously change the values in the indices texture. After sorting, we get the sorted indices which are used to render the particles to the screen.

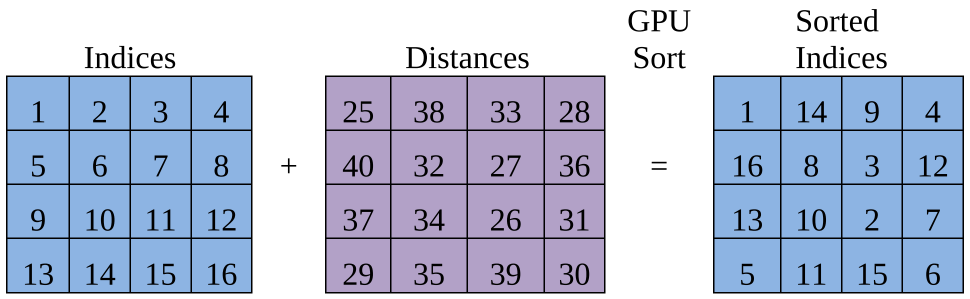


Figure 3.11: We pass the two textures, indices and distances, for sorting and get the sorted indices of the particles.

The original GPU sort code uses fragment programs written in assembly which communicate with the GPU at a very low level. We use fragment shaders written in GLSL (OpenGL Shading Language) instead, which are translations of the assembly code.

For example: Assembly to shader conversion

```
TEX R1, fragment.texcoord [0], texture [2], RECT;
->          R1          =          vec4(texture2DRect(tex2,gl_TexCoord[0]));
ADD R4, R0, -R1;
 ->               R4   =                  R0 - R1;
```

Even after sorting the particles on the GPU, the simulation is much faster than quick sort but is still slow for a real environment. The GPU sort is very fast for a small number of particles but becomes slow as the number of particles increases. For example, the time taken to render a frame with one million particles is more than the 0.01 seconds that is necessary for an interactive environment. However, as graphics card speeds increase over time, this somewhat interactively slow time will be feasible for real-time environments. Moreover, we can decrease the number of snow particles in the system and achieve real-time rates with this method.

In addition to GPUSort, we also investigated another technique in which we sort the particles over a number of passes. But this approach is restricted as the particles are moving and hence we cannot have a large number of passes to sort the particles in a particular position. On the other hand, using a few passes gives us artifacts, which are easily noticeable to the human eye. To alleviate these concerns, we have also investigated the use of additive blending which is explained in the next section. Additive blending is an alternative approach to sorting, but in future we will require sorting of particles for effects like shadowing for example.

3.4 Blending

As mentioned in the earlier section, we use additive blending as an alternative to sorting. Blending is a technique in which the alpha value is used to combine the color value of a fragment with the color of a pixel already stored in the framebuffer. Without blending, a new fragment will overwrite any existing color values in the framebuffer. This is a technique used for transparent/translucent objects which allow us to see the objects behind them. We use additive blending for our snow particles, which gives us the desired effect with the snow particles. It gives a good mixture of the translucent snow particles without any artifacts.

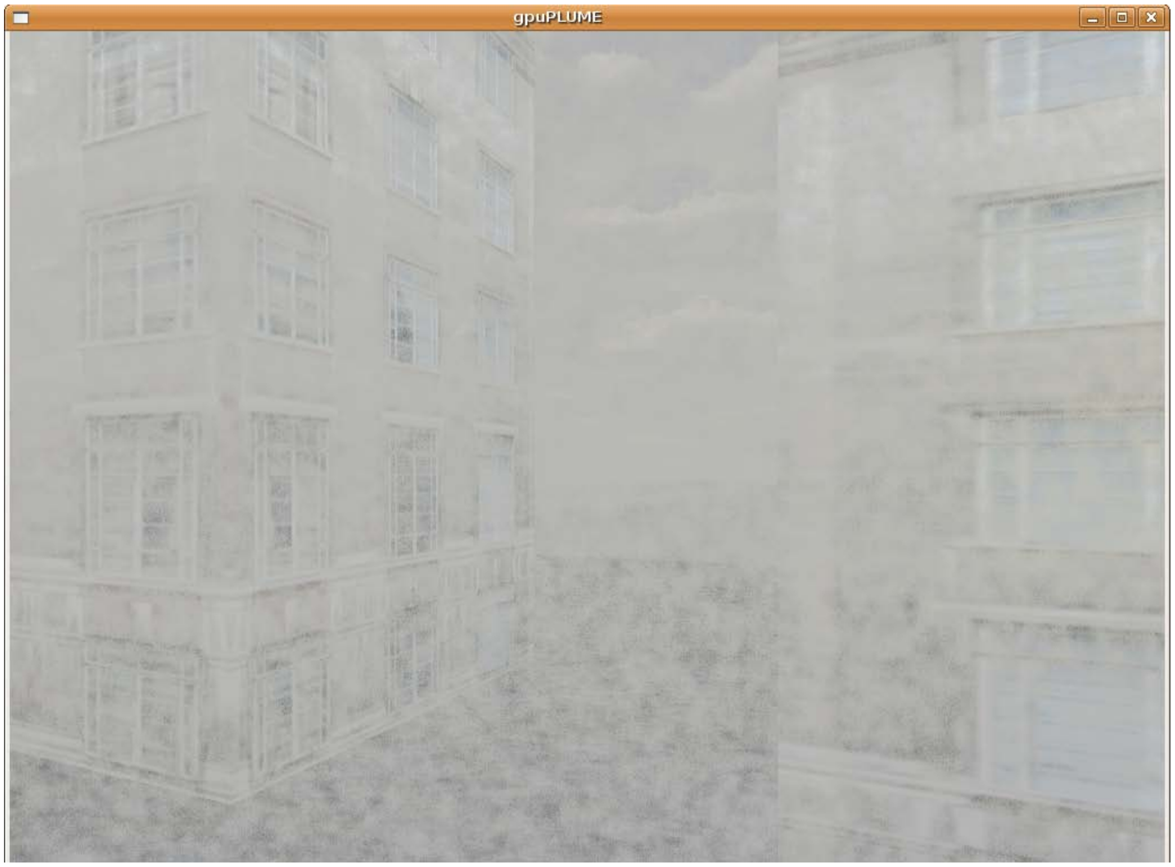


Figure 3.12: Snow particles blended using additive blending.

In the Figure 3.12, we can see a uniform cloud of snow which is developed by additive blend. The transparent particles nicely mix with the other particles giving a nice snowing effect.

3.5 Snow Plow Model

A 3-D snowplow model has been created for use in experiments to validate the snow rendering system and further explore the perceptual effects that result from driving in foggy or snowing conditions. The snowplow model has been created with Autodesk's 3DS Max modeling software. Rick Shomion supplied photographs of MN/DOT snowplows and Craig Shankwitz provided information relating to snowplow scale. We continue to improve the snowplow model by (1) optimizing the polygons from which it is constructed, (2) applying suitable material properties for proper illumination, (3) designing a modular back-end so that the rear-end lighting configuration and paint color can be swappable, and (4) outfitting the model so that tires rotate and turn based on the vehicle's motion. It may be necessary to take additional pictures of MN/DOT snowplows to refine the images used to texture the model. Two images of the 3D snowplow model are shown in Figure 3.13.

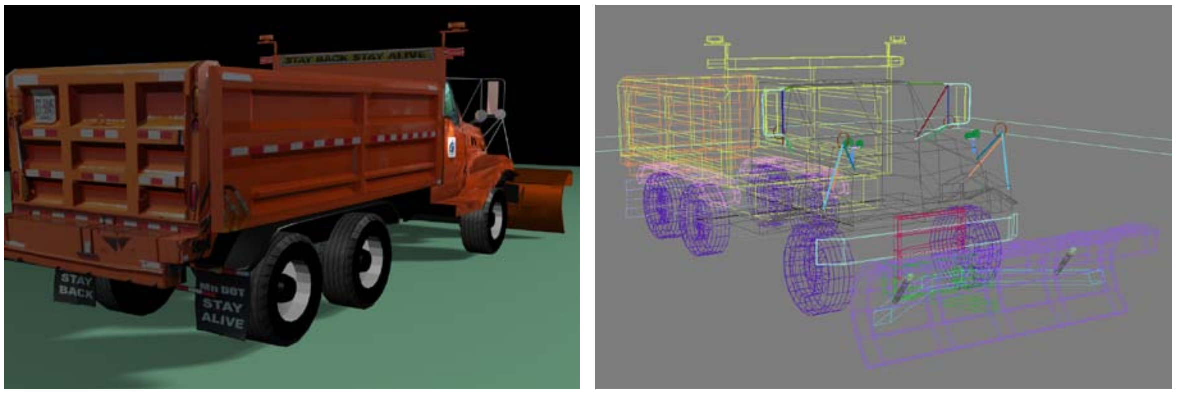


Figure 3.13: 3D Snowplow model. The left panel shows a rendered view of our 3D snowplow model. The wireframe view (right panel) shows the polygons that make up the snowplow model. The current model is a low-polygon model for use in real-time environments.

Our methodology for simulating snow in a real-time environment is based on rendering large numbers of dynamic snow particles. The motion of the snow particles is generated by utilizing a particle dispersion model to move the particles with regard to a mean and fluctuating wind component. However, in the current system, only the wind surround the plow vehicle is simulated. No other structures, such as roadside trees or other leading or following vehicles is incorporated. Those effects could be considered for future works. Alternative wind fields can be created (such as cross-winds) and loaded into this system. We also model how particles reflect off of rigid, urban structures, such as buildings or a rough approximation to the snowplow.

The lighting, and hence color, of each snow particle is calculated individually. This affords more precise lighting calculations that interact with the dynamic lights in the scene (such as from the snowplow). The result is particles that are lit by temporal and spatial components of the environment. While we have not focused on either specific daytime or nighttime conditions in FY 2008, it is likely that our system could handle both situations, at least from a particle lighting computation. For both daytime and nighttime rendering, we will need to consider the effects of snow particles at greater distances to achieve the correct dampening/reduction of light across distances.

Chapter 4. Results, Conclusions, and Recommendations

We have used a 2.4 GHz Intel Core 2 Duo Processor with an NVIDIA GeForce 8800 GTS graphics card for testing our system. In the next section we will compare how the sorting with the GPU is better than sorting on the CPU and show results of how our scene look with the new improved snow model.

4.1 Quick Sort vs. GPU Sort

The following results show the benefit of sorting the snow particles on the GPU. By sorting on the GPU we get a huge performance gain which allows us to do other complex computations such as scattering and HDR rendering. Our aim was to render a frame in 0.01 seconds which is not possible using CPU sort.

Table 4.1: Experimental values for sorting on a CPU and a GPU done on a 2.4 GHz Intel Core 2 Duo Processor with an NVIDIA GeForce 8800 GTS graphics card.

No of Particles	Quick Sort (CPU) in seconds	GPU Sort in seconds (with passes)	GPU Sort in seconds (without passes)
15K	0.017	0.000168	0.002295
65K	0.072	0.000237	0.003283
250K	0.335	0.000296	0.004891
500K	0.741	0.000562	0.008573
1M	1.533	0.000745	0.012487
2M	3.35	0.000968	0.017319

There is a huge difference in the timings for CPU sort and GPU sort as we are sorting the particles in parallel on the GPU. The following two graphs show the comparison between the two sorts.

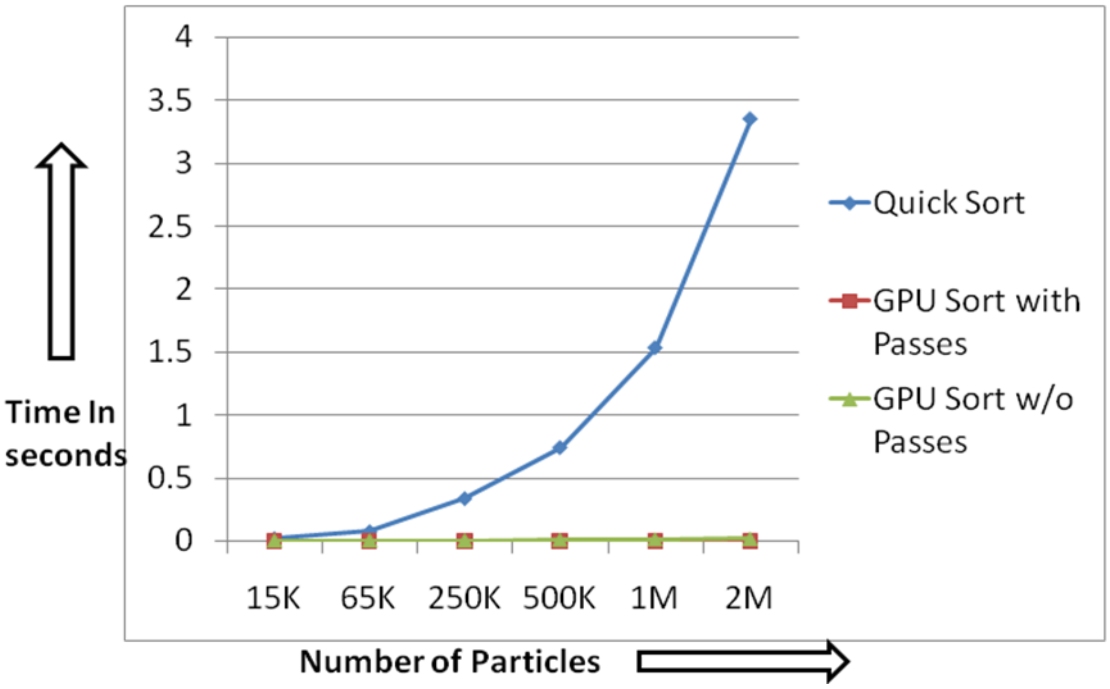


Figure 4.1: Comparison between quick sort (CPU) and GPU sort (GPU).

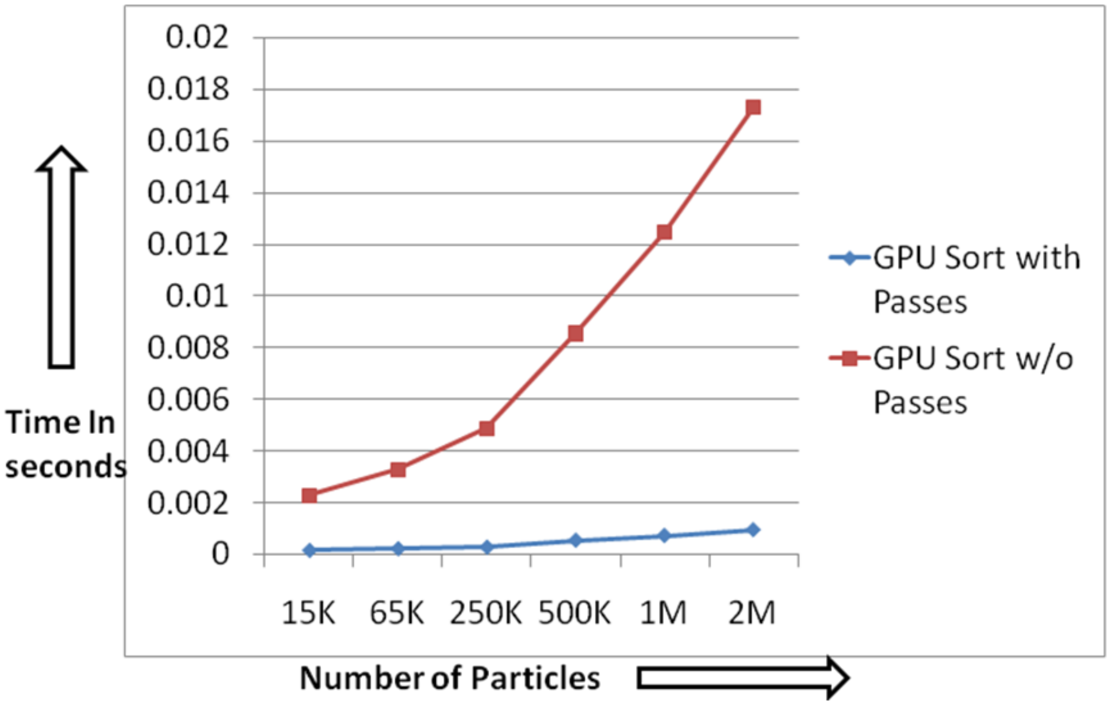


Figure 4.2: Graph showing curve for GPU sort (with passes vs w/o passes).

Figure 4.1 shows the graphical comparison between the two sorts. Figure 4.2 shows the comparison between GPU sort with passes and GPU sort without passes. As we can see, GPU sort without passes takes more than 0.01 seconds for 1 million particles which is slow. Hence we needed a different approach such as sorting over multiple passes to reduce the times.

The following are three snapshots taken from our system which show the sorted, unsorted and blended (additive blend) particles.

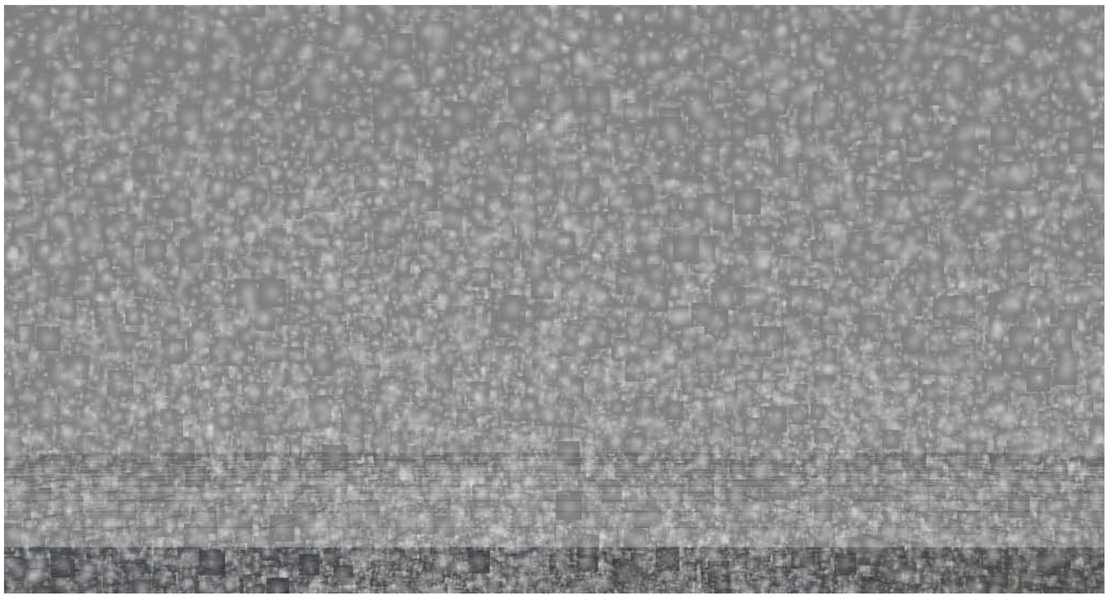


Figure 4.3: Unsorted snow particles with artifacts.

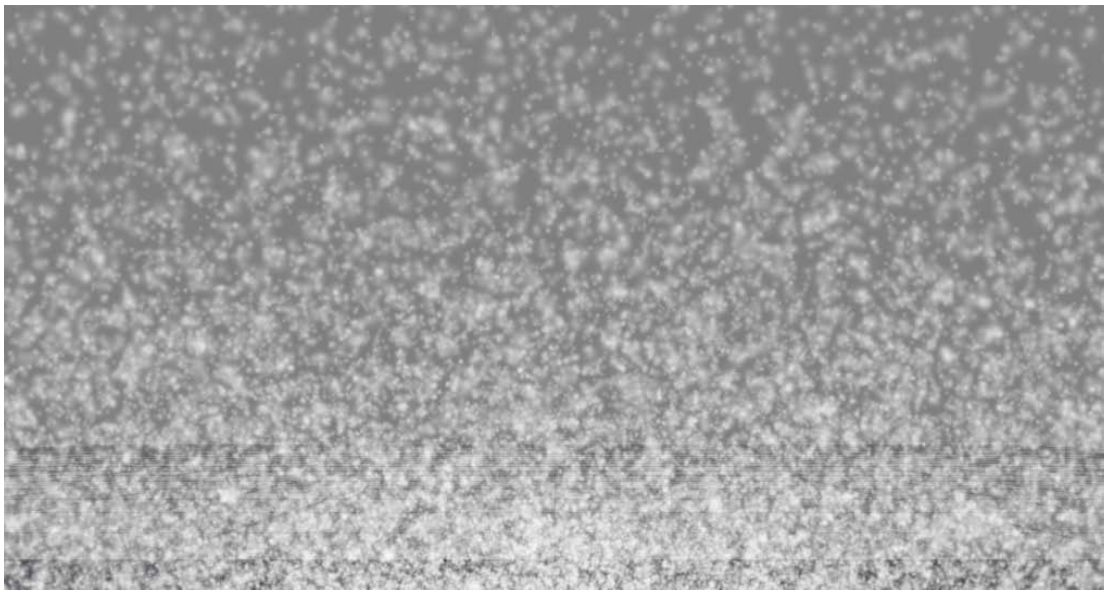


Figure 4.4: Sorted particles using GPU sort.

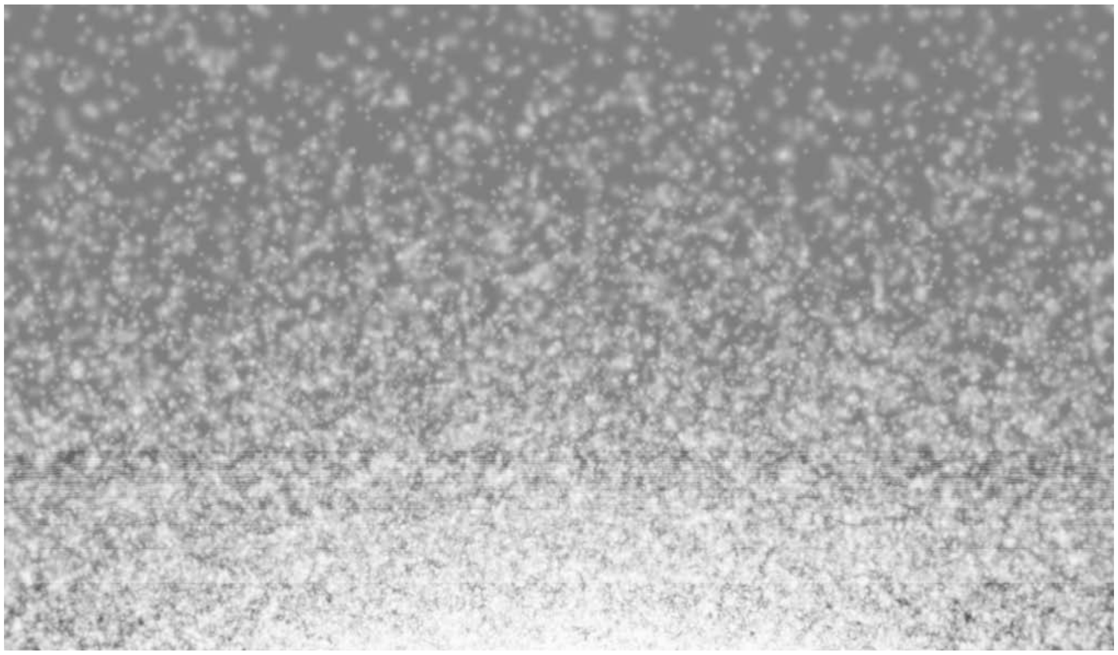


Figure 4.5: Snow particles using additive blend.

As we can see from the above figures, there is not much difference between the images for sorting and additive blend. Hence we are currently using additive blend in our snow system for development. However, we can switch on the GPU sort for complete sorting of particles in the environment as needed.

4.2 Conclusions

The bulk of our efforts over the last year have been to develop a software architecture for rendering blowing snow in a real-time virtual environment. Our system utilizes commodity graphics hardware (video game cards) and performs a simplified wind simulation on the graphics hardware as well. Snow particles move with respect to a mean wind field and interact with buildings and other structures (like snowplows). Snow particles are illuminated using a per-snow-particle rendering so that the effect of the snowplow truck's rear lighting configuration can influence each particle. Our system runs at real-time, interactive rates (60Hz screen refreshes) with around one million snow particles. We expect to increase this number and be able to consistently render our snow field in a driving simulation framework using 60Hz screen refresh rates.

We are currently setting up pilot experiments that use the snowplow model and a simple 3D driving simulation environment [18,19] to replicate studies by Yonas et al. [1,2] as well as Snowden et al. [3]. The results from these studies will be used to design our experiments planned for future work on this project. We will investigate how lighting configuration and snowplow color affect perception when following a snowplow. Work-study students in Dr. Yonas' lab are being hired to acquire Institutional Review Board (IRB) approval for the experiments, prototype experimental design, and help with running the upcoming experiments. We are currently beginning to conduct the first round of experiments.

As such, we are unable to make any recommendations for alternative snowplow lighting configurations. However, the snow rendering system is showing great potential and we fully expect to integrate it with our driving simulation environment. Our efforts over the next year will improve the snow rendering system and begin to provide information on alternative lighting configurations that may help reduce collisions with the back ends of snowplows.

References

1. A. Yonas and L. Zimmerman. *Improving the ability of drivers to avoid collisions with snowplows in fog and snow*. Minnesota Department of Transportation, Report No. 2006-02, 2006.

2. A. Yonas, L. Zimmerman, H. Seo, A.J. Alexander, A.S. Olinick, and S.Z. Polley. “The effect of luminance contrast and stroboscopic presentation on the threshold for the discrimination of approach from withdrawal.” [abstract]. *Journal of Vision*, 5(8), 2005.

3. R.J. Snowden, N. Stimpson, and R.A. Ruddle. “Speed perception fogs up as visibility drops.” *Nature*, 392, pp. 450, 1998.

4. M.S. Langer, L. Zhang, A.W. Klein, A. Bhatia, J. Pereira, and D. Rekhi. “A spectral-particle hybrid method for rendering falling snow.” *Rendering Techniques, 2004 Eurographics Symposium on Rendering*. Norrkoping, Sweden. A. Keller and H. W. Jensen (eds.). pp. 217-226, 2004.

5. A. Preetham, P. Shirley, and B. Smits. “A Practical Analytic Model for Daylight.” *Proceedings of the 26th Annual Conf. on Computer Graphics and Interactive Tech. (SIGGRAPH 99)*. Los Angeles, CA. pp. 91-100, 1999.

6. E. Lengyel. “Unified Distance Formulas for Halfspace Fog.” *Journal of Graphics Tools*, 12(2), pp. 23-32, 2007.

7. W.J. Wiscombe and S.G. Warren. “A Model for the Spectral Albedo of Snow i: Pure Snow.” *Journal of the Atmospheric Sciences*, 37, pp. 2712-2733, 1980.

8. S.G. Warren. “Optical Properties of Snow.” *Reviews of Geophysics and Space Physics*, 20(1), pp. 67-82, 1982.

9. R. Lawson and Q. Mo. “Observations of ice crystals at the South Pole using a CPI and polar nephelometer.” *Proceedings of the 8th Conference on Polar Meteorology and Oceanography*. San Diego, CA. 2005.

10. P. Willemsen, A. Norgren, B. Singh, and E.R. Pardyjak. “Development of a New Methodology for Improving Urban Fast Response Lagrangian Dispersion Simulation via Parallelism on the Graphics Processing Unit.” *11th International Conference on Harmonisation within Atmospheric Dispersion Modeling for Regulatory Purposes*. Cambridge, UK. July 2-5, 2007.

11. E.R. Pardyjak, B. Singh, A. Norgren, and P. Willemsen. “Using Video Gaming Technology to Achieve Low-cost Speed up of Emergency Response Urban Dispersion Simulations.” *American Meteorological Society Seventh Conference on Coastal Atmospheric and Oceanic Prediction and Processes*. San Diego, CA. September 2007.

12. C. Chrisman. *Rendering Realistic Snow*. Unpublished Abstract. University of California, San Diego.

13. M.J. Harris and A. Lastra. “Real Time Cloud Rendering”. *Computer Graphics Forum* (*Eurographics 2001 Proceedings*), Manchester, UK. 20(3), pp. 76-84, 2001.

14. B. Wade and N. Wang. “Rendering Falling Rain and Snow.” *Proceedings of the ACM Siggraph Sketches 2004*, Los Angeles, CA. pp. 14, 2004.

15. A. Norgren. “GPU Based Particle Dispersion Modeling with Interactive Visualization Support for Real-time Simulation.” Thesis, University of Minnesota, Duluth, June 2008.

16. P. Kipfer and R. Westermann. “Improved GPU sorting”, *GPU Gems 2*, M. Pharr, Ed. pp. 733–746, 2005.

17. N.K. Govindaraju, M. Henson, M. Lin, and D. Manocha. “Interactive Visibility Ordering and Transparency Computations among Geometric Primitives in Complex Environments”. *ACM Symposium on Interactive 3D Graphics and Games*, Washington, D.C. pp. 49-56, 2005.

18. P. Willemsen, J.K. Kearney, and H. Wang. “Ribbon Networks for Modeling Navigable Paths of Autonomous Agents in Virtual Environments.” *IEEE Transactions on Visualization and Computer Graphics*, 12(3), pp. 331-342, 2006.

19. H. Wang, J.K. Kearney, J. Cremer, and P. Willemsen. “Steering Behaviors for Autonomous Vehicles in Virtual Environments.” *Proceedings of IEEE Virtual Reality Conference*, Bonn, Germany. pp. 155-162, March 2005.