UNIVERSITY OF MINNESOTA

This is to certify that I have examined this copy of a master's thesis by

Siddharth Sham Deokar

and have found that it is complete and satisfactory in all respects,

and that any and all revisions required by the final

examining committee have been made.

Peter Willemsen

Name of Faculty Adviser

Signature of Faculty Adviser

August 21, 2009

Date

GRADUATE SCHOOL

Real-Time Snow Rendering

A THESIS

SUBMITTED TO THE FACULTY OF THE GRADUTE SCHOOL

OF THE UNIVERSITY OF MINNESOTA

BY

Siddharth Sham Deokar

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

Pete Willemsen

August 2009

Acknowledgements

I would like to take this opportunity to thank my advisor, Dr. Pete Willemsen for his guidance throughout this thesis and my entire graduate career. I have gained a lot of valuable knowledge in three dimensional graphics and virtual reality with his assistance.

I also owe my heartfelt appreciation and thanks to Dr. Joe Gallian and Dr. Doug Dunham for being a part of my thesis committee.

I would also like to acknowledge Michele Olsen and Joshua Clark for their help with the snowplow along the way.

Finally, I would like to thank all my friends for their support and encouragement throughout the thesis.

Table of Contents

List of Tables

List of Figures

1. Introduction

Driving vehicles on roads in winter conditions can be extremely dangerous at times. Winter snow and fog causes problems while driving. When snow is blowing around, it is hard to see other vehicles or judge their speed. Detecting the changes in speed of the vehicle or snowplow in front of you is important for safe driving. Blowing snow and fog complicates visibility while driving. Due to the extreme weather conditions and poor visibility, it becomes difficult to detect speed changes of other vehicles. The poor visibility affects our view, resulting in the potential for rear end collisions.

We can create a 3-D simulation of the extreme winter conditions where a person can drive around or follow other vehicles or snowplows in blowing snow or foggy conditions. The interactive virtual environment can help us in studying user reaction times to the changes in speed/motion of other vehicles or snowplows. We could test the user reaction times for detecting motion of the snowplow for a variety of color and lighting configurations. Determining the optimal color and lighting configurations for snowplows could help in making driving safe in extreme winter conditions. For this virtual environment to be effective, the rendering has to be in real time. Simulating effective snow in real time is a challenging task as there are no good solutions available for real time rendering of snow.

The aim of my research is to improve the visual rendering of snow taking into account the physical interaction between light and the snow particles. Ray tracing methodology has been used until now to produce realistic effects of light-object interaction. Ray tracing has been used for rendering images ahead of time by tracing the path of light through every pixel in the image. Due to the high computational costs involved, ray tracing is not suitable for rendering interactive environments. We need a method that can approximate the ray tracing effects and reduce the computational costs while giving us acceptable and realistic results.

Our methodology for simulating snow in a real time environment is based on rendering large number of dynamic snow particles. We explicitly model the blowing snow and its effect on the environment. The lighting calculations are applied to each snow particle to characterize the absorption, emission and scattering of light. The presence of dynamic lights in the scene affect the lighting calculations for the snow particles, giving us more realistic scenes encompassing various optical effects such as scattering, high dynamic range rendering, and blurring.

We use imposters for our particle system that were developed previously. I have modeled the snow particles using snowflake shapes and a Gaussian function for transparency. The lighting calculations for every snow particle are done using the Henyey-Greenstein scattering distribution model. We have also implemented High Dynamic Range rendering to give more realistic effects to the scene. We use additive alpha blending for the transparent snow particles, which might have to be changed in the future for effects such as shadowing. We

have also incorporated aggregate snow effects, which makes the objects at distance blurry depending on the blowing snow density. Our 3-D environment for snow is very fast and works in real time, giving us realistic results of snowing and foggy conditions.

The outline of the rest of the thesis is as follows. We cover the background and related work on snow particle rendering in Chapter 2. Chapter 3 covers the implementation of the snow model, while Chapter 4 describes the results obtained and finally Chapter 5 concludes the thesis and indicates directions for future work.

2. Background and Related Work

There has not been much published work 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 or fallen snow had been modeled previously by Chrisman [Rendering Realistic Snow] using the optical characteristics of ice and snow. Real Time Cloud rendering by [Harris and Lastra, November 2001] 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 [Rendering Falling Rain and Snow, 2004] have modeled a snow domain with static textures without using a particle system.

We are using a particle system to simulate particles in our virtual environment. The particles in the system use a wind simulation model to simulate the particle motion in our system. The particles move about in the environment similar to how wind would cause them to flow. We also have a collision detection mechanism in place where in the particles can collide against simple urban structures such as buildings and off the ground, changing their direction.

We use the QUIC-Plume dispersion model that was reworked to run on the GPU [Pete Willemsen, Andrew Norgren, 2007]. The movement of the particles is decided using the wind field generated by QUIC-URB along with turbulent fluctuating winds [Andrew Norgren, 2008]. 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.

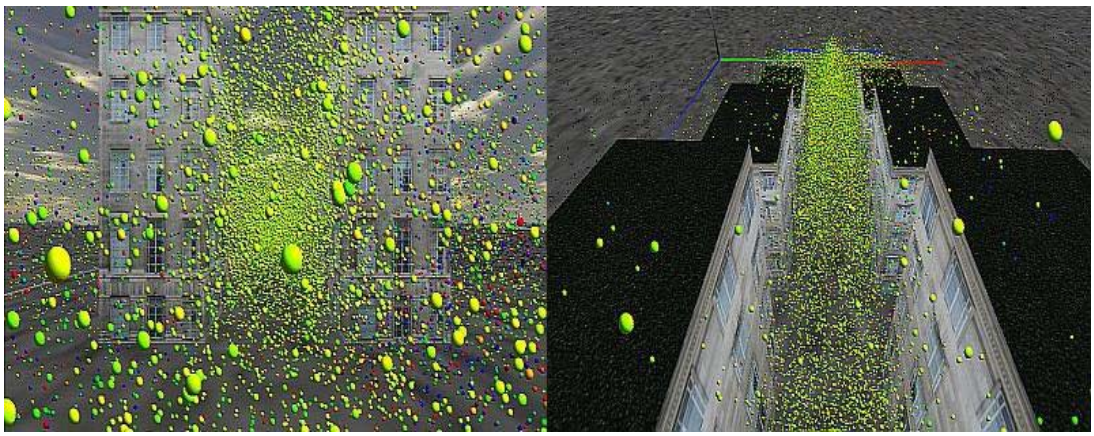

Figure 2.1:A virtual environment of buildings with the colored particles depicting motion of wind. The particles collide against the buildings and change their direction.

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.

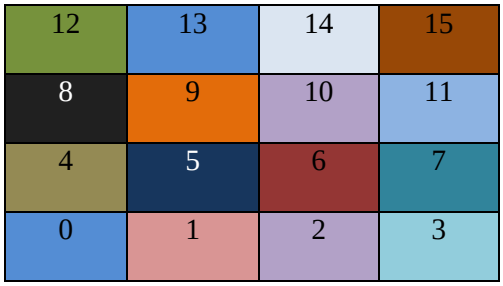

Figure 2.2 2-D Texture of size 4X4 holding 16 particle positions with each position being a vector (x, y, z, w)

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.

3. Implementation

Our snow particles are implemented on the GPU at real time rates making it possible to render effects such as High Dynamic Range Rendering (HDR), scattering, blending, and aggregate snow. 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. The shapes of the snow particles are modeled after actual snowflakes using a Gaussian transformation for translucency.

We use point sprites as particles [Andrew Norgren, 2008] for modeling our snow. 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 many dynamic lights in the scene. We then apply aggregate snow effect outside the snow domain to show the effect of snow on objects at a distance. For example, when it is snowing and you look around, after a certain distance

you cannot recognize individual snow particles, and the effect is that of a fog with objects blending with the background. We then render the scene using high dynamic range and get the blur and bloom effects before we tone the scene down to low dynamic range for rendering it to the display.

The following is a generic display function, which shows how our system code is organized to get the final image with our snow model.

Display:

Sort the particles on distance from eye [Section 3.3] OR use additive blend [Section 3.4]

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]

Apply aggregate snow equation to other objects in scene [Section 3.5]

Render scene to a frame buffer using HDR for lighting calculations [Section 3.6]

Render HDR image

Down sample and apply Gaussian blur

Apply resulting bloom mask to original image

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. For imposters to work, we store the normal at each point and calculate the color at that point using its orientation to the light source so that they look like spheres.

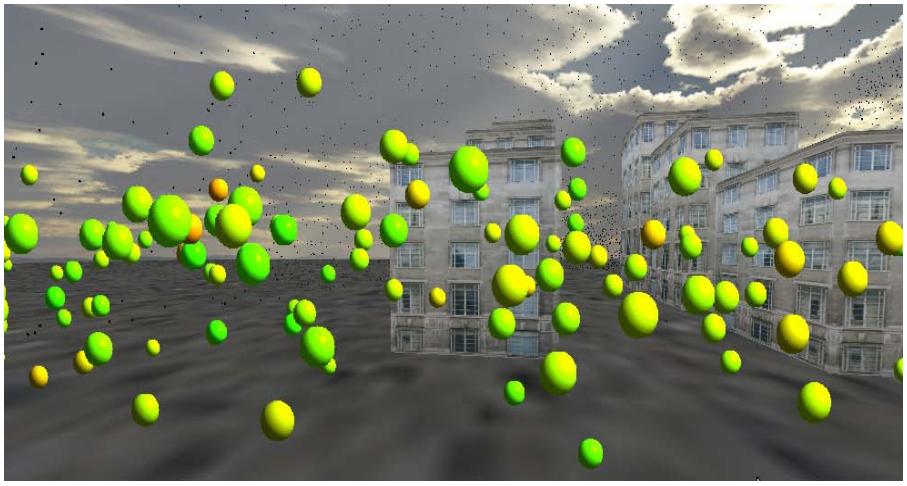

Figure 3.1 Particles rendered as a single points with a 2D texture on it.

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. They are transparent particles which interact with light and give a scattering effect. We need to 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 which peaks at the center and gradually falls off as we move away from the center. This functionality gives the snow opacity at the center and becomes transparent as we move away from the center.

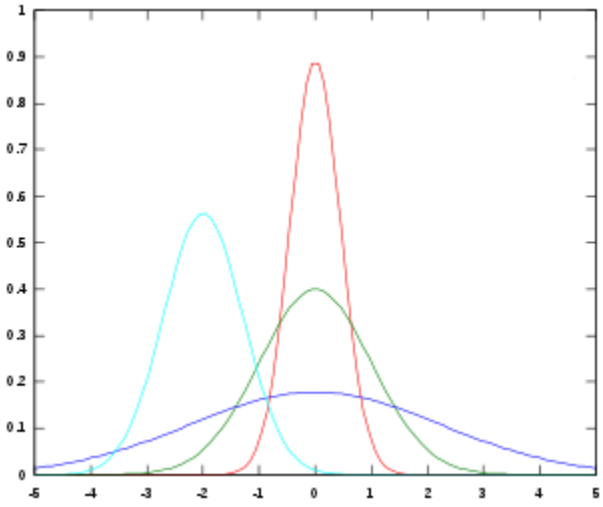

Figure 3.2 Gaussian Function Curves different values of a, b and c.

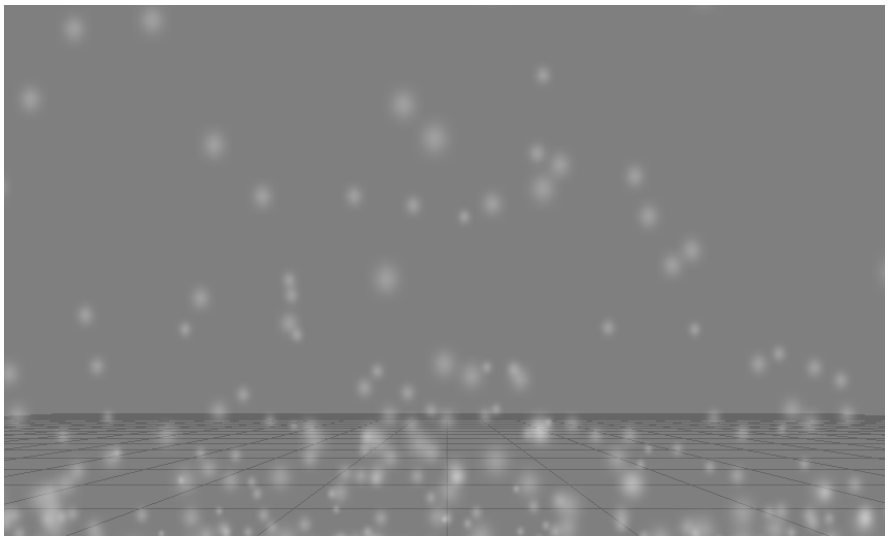

Figure 3.3 Transparency using the Gaussian Function

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,

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

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 Shapes for Snow Particles

In reality, the snow particles are not perfectly shaped as the Gaussian surface indicates. To give the snow particle a random shape, we have used actual snowflake shapes to change the uniform look of the Gaussian, so that the physical shape of the snow particles would be similar to actual snow. The shapes are assigned to individual particles in a random manner.

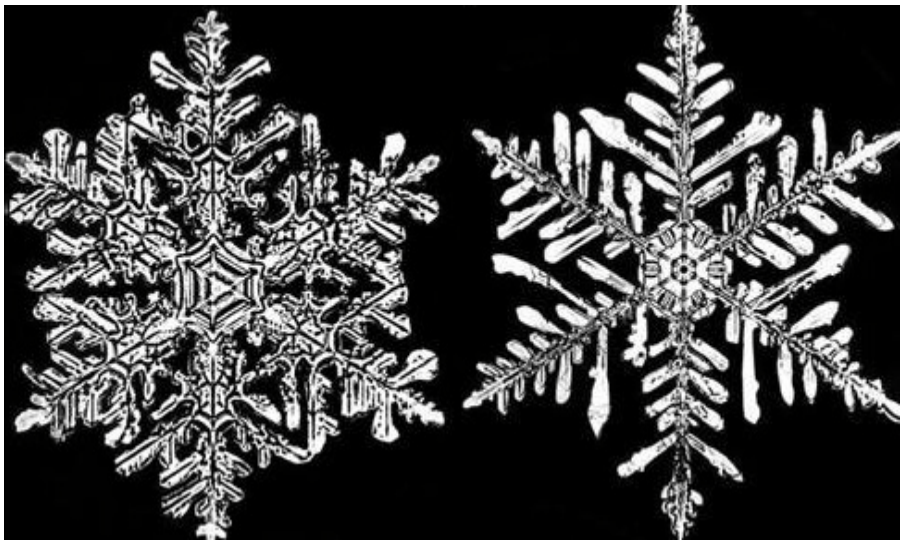

Figure 3.4 Snowflake shapes, which along with the Gaussian determine the shape of the snow particle.

We use the snowflakes shown in the figure above to model our individual snow particles. The snowflake shapes are stored in textures which are used to give the snow particles the desired shape. Our aim in using these images is to give an outer boundary to our snow particles so that they resemble actual

snowflakes. The textures are mapped to the snow particles in the sense that the outer boundary of the snowflake is preserved, and we apply the Gaussian transparency inside this shape. The snowflake images are black and white images with the snowflake in white and the background in black. So when applying the texture to a snow particle we only process the texels which are white (corresponding to the snowflake). If the texel is black, we discard the corresponding fragment on the snow particle. If the texel is white, then we apply Gaussian transparency to the corresponding fragment of the snow particle.

In short, we map the actual snowflake shape to the snow particle as follows:

```
For every point on the texture
{
    If color is black
        Discard the point
    Else
        Calculate Alpha value of Color using the Gaussian Function
}
```

The actual snow flake shapes when applied to the snow particles will appear as follows:

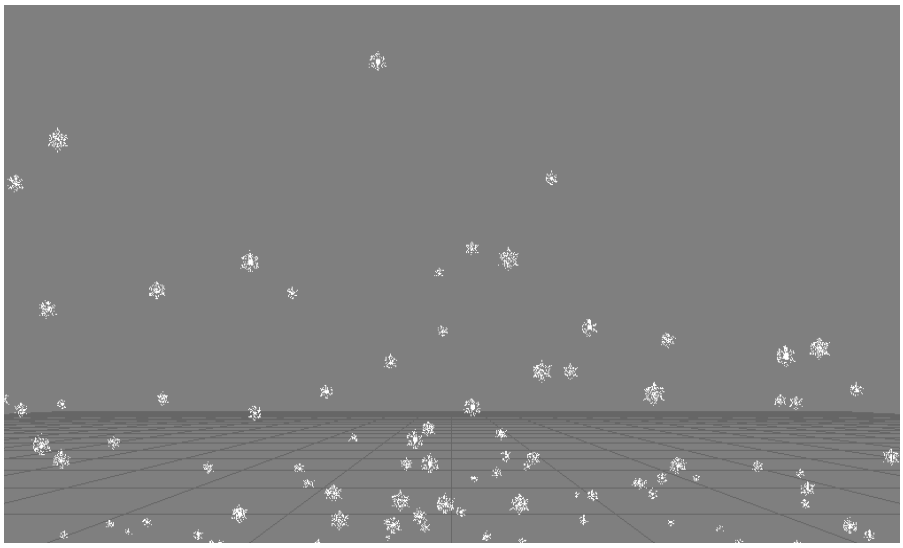

Figure 3.5 Actual snowflake shapes applied to snow particles

We then apply the Gaussian transparency to the snow particles to get the image as seen below.

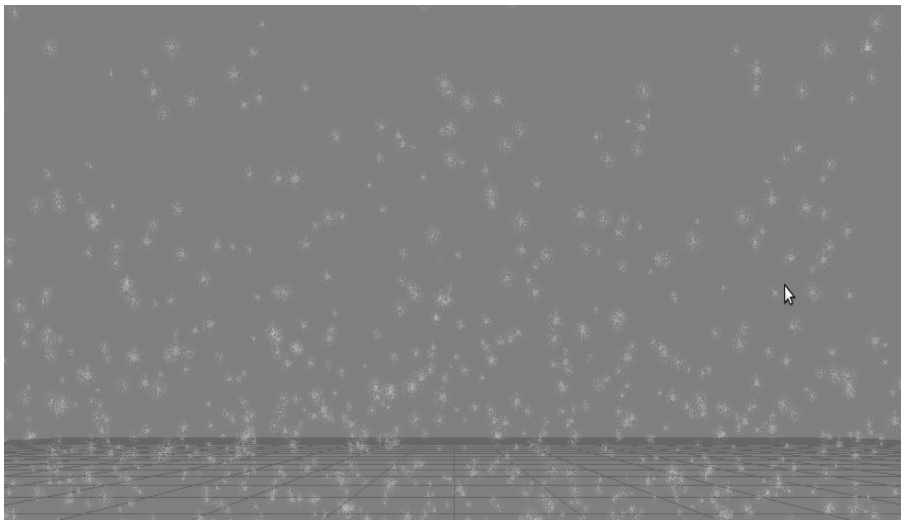

Figure 3.6 Actual snowflake shapes applied to snow particles along with Gaussian transparency

3.2.4 Dynamic Lights

We use dynamic lights in our environment; the lights are 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)

   For example: Car headlights, brake lights
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 3.7, we have a texture of size 4X5 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 3.7 A texture of size 4X5 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 the shader programs. The lights get updated every time there is a change in the 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 particle do

Read the light source texture

Get the active light sources

For every active light source

Calculate the effect of that light source on the particle in terms of intensity

Calculate scattering by Henyey-Greenstein phase function

color = color + (phase * intensity + ambient color)

Calculate the alpha value of color by the Gaussian function

3.2.5 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.

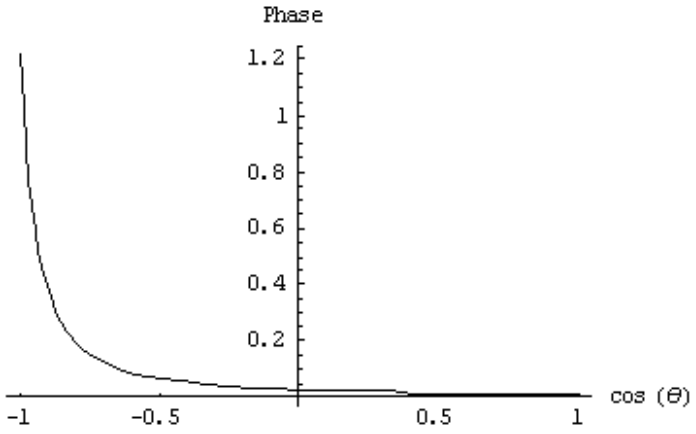

Figure 3.8 Plot of Henyey-Greenstein phase function for g =- 0.67

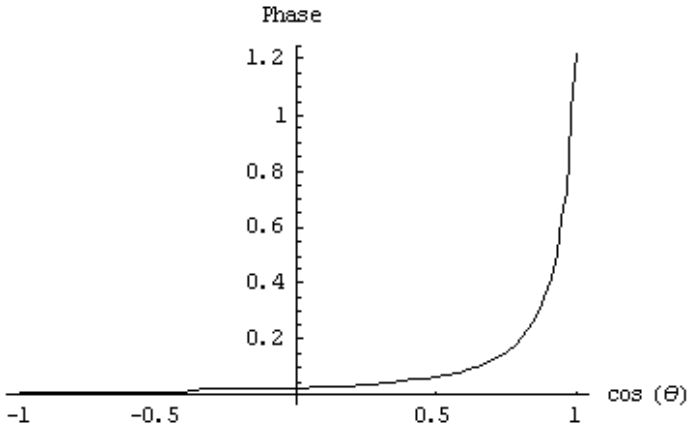

Figure 3.9 Plot of Henyey-Greenstein phase function for g = 0.67

In the plots for the Henyey-Greenstein phase function, the curve in Figure 3.8 denotes back scattering (g = -0.67) and curve in Figure 3.9 denotes forward scattering (g = 0.67).

Hence for every snow particle, we do the following:

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

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.

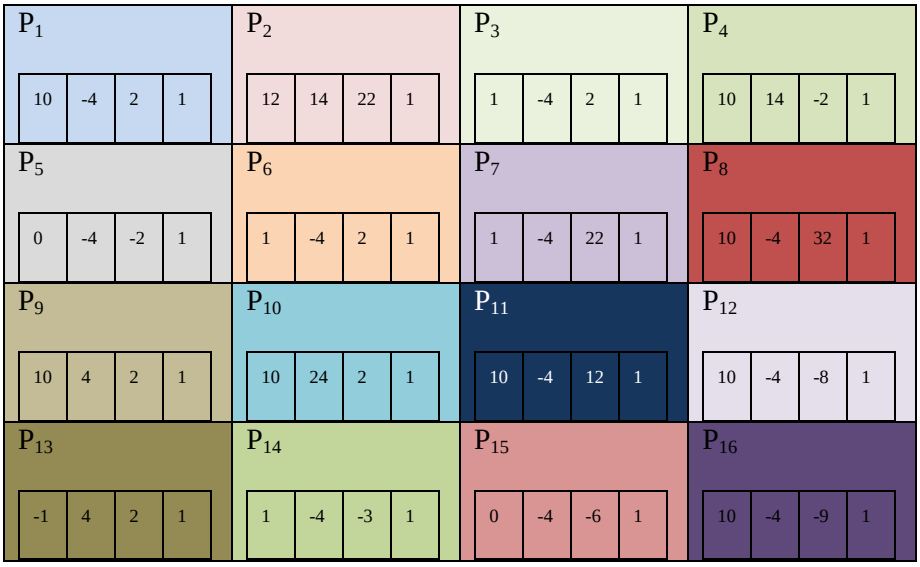

Figure 3.10 Texture of size 4 X 4 storing positions of 16 particles

In the above figure, 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 CPU to the vertex buffer
    Sort the particles based on each individual particle's distance from the
    eye
    Unmap the vertex buffer
}
```

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 [Govindaraju, 2005], 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 example, 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.

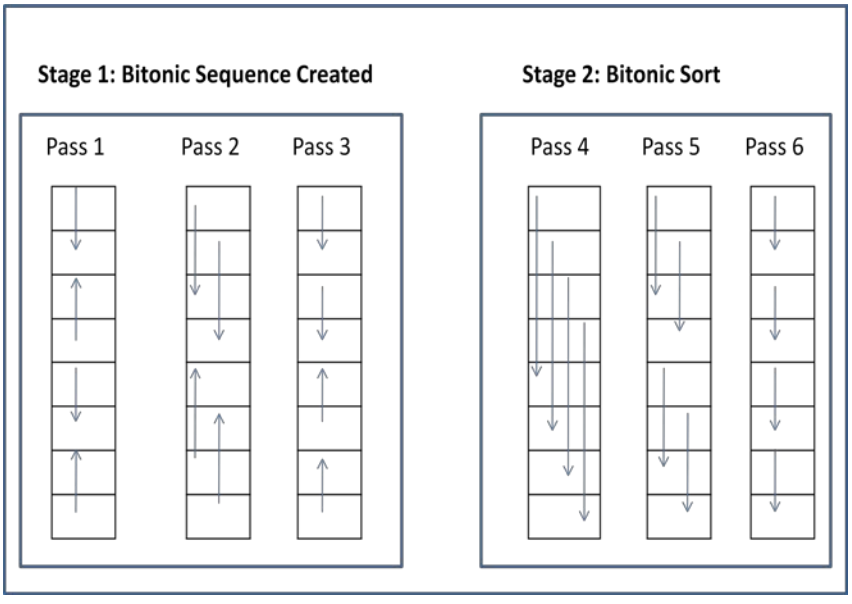

Figure 3.11 Process of bitonic sort

Figure 3.11 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 3.12. 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.

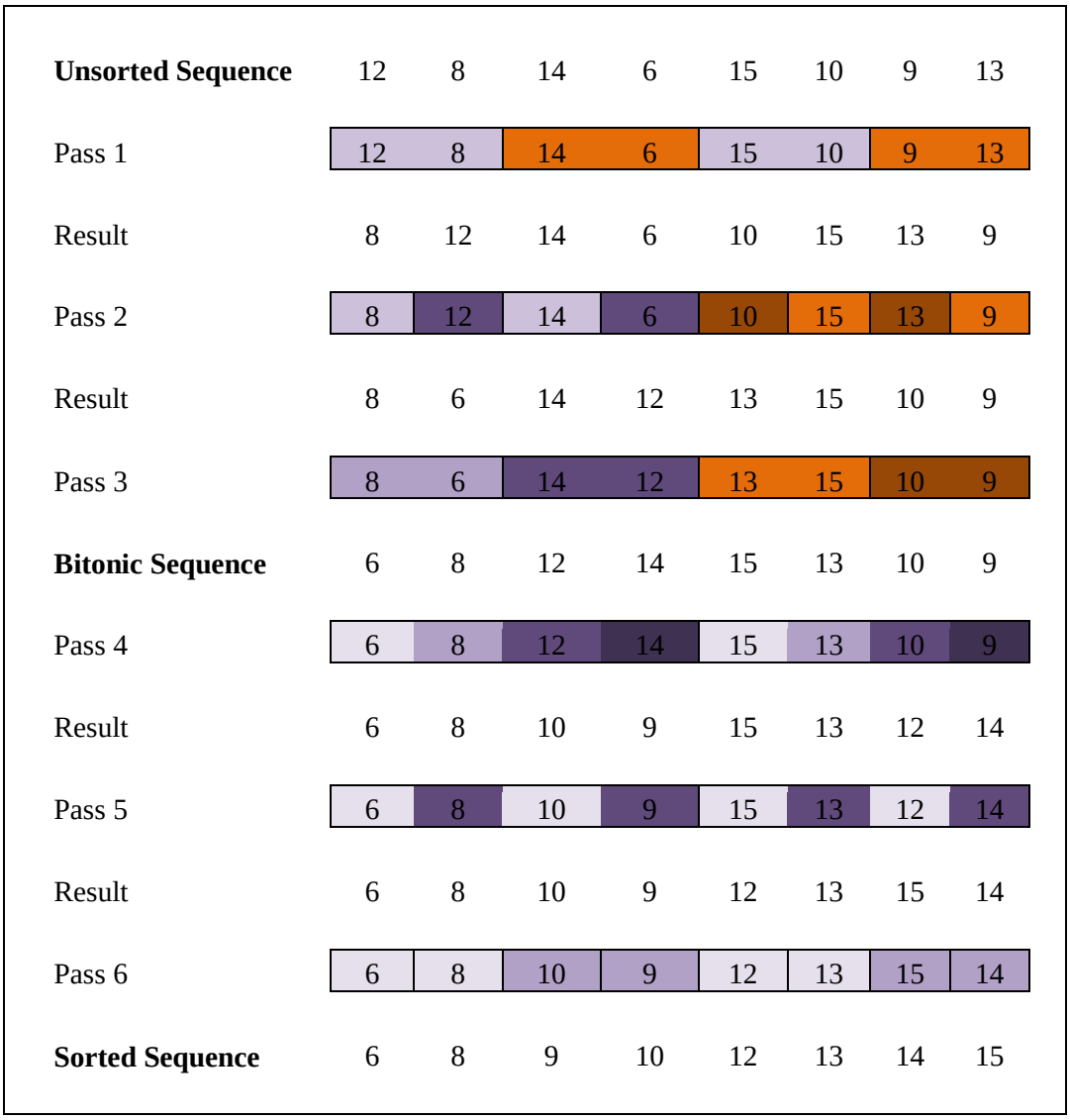

Figure 3.12 Example of bitonic sort

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,

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

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.

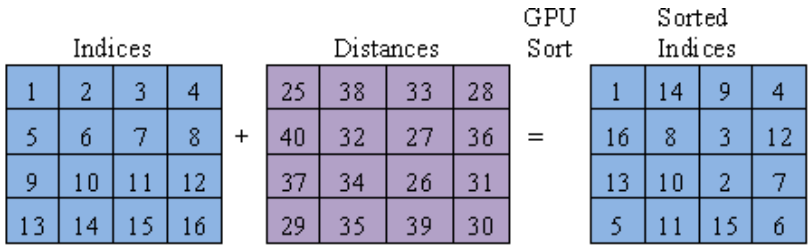

Figure 3.13 We pass the two textures, Indices and Distances for sorting and get the sorted indices of the particles.

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 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.

So we used 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. Then we came across 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.

3.3 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.

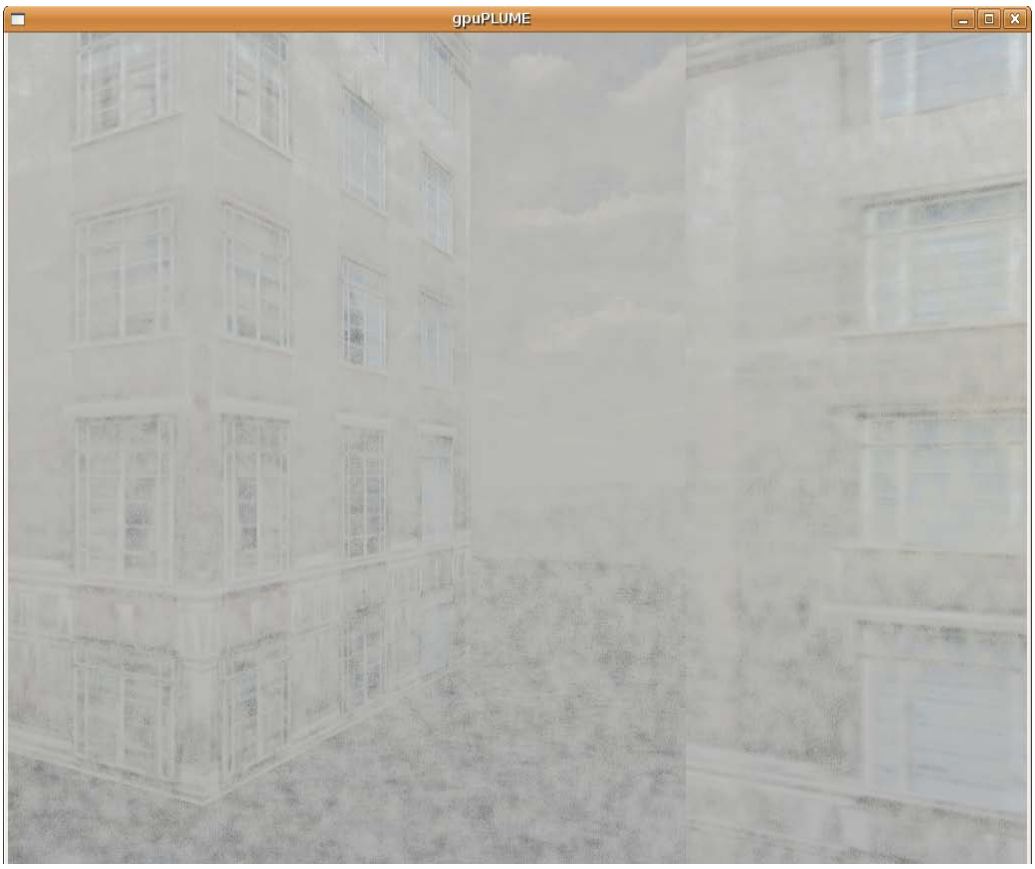

Figure 3.14 Snow particles blended using additive blending.

In the Figure 3.14, 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.

3.4 Aggregate Snow

We are rendering snow particles in a way such that they are confined in a certain domain. To show snow particles everywhere is not computationally feasible and hence we need some technique that will show the snowing effect outside our domain. In real life also you can see only the individual snow particles up to a certain distance and as the distance increases the surrounding looks more hazy or foggy. The aggregate snow effect becomes more prominent as you look toward the horizon with objects becoming less and less visible with increasing distance. We have tried to map this phenomenon with the visibility in the sense that as the snow density increases, visibility decreases.

Figure 3.15 Real world capture of snow. We cannot recognize individual snow particles beyond a certain distance.

We have used Matlab to derive a relation between the snow density in our simulation system and visibility in the simulated scene. In future we would like to tie this parameter with actual readings when it is snowing. In our system we can vary the density of snow by changing the rate of the number of particles moving per second. We have tested the situation where we have a black rectangular surface and snow blows by it. We will now present some of the images we generated for different particle rates.

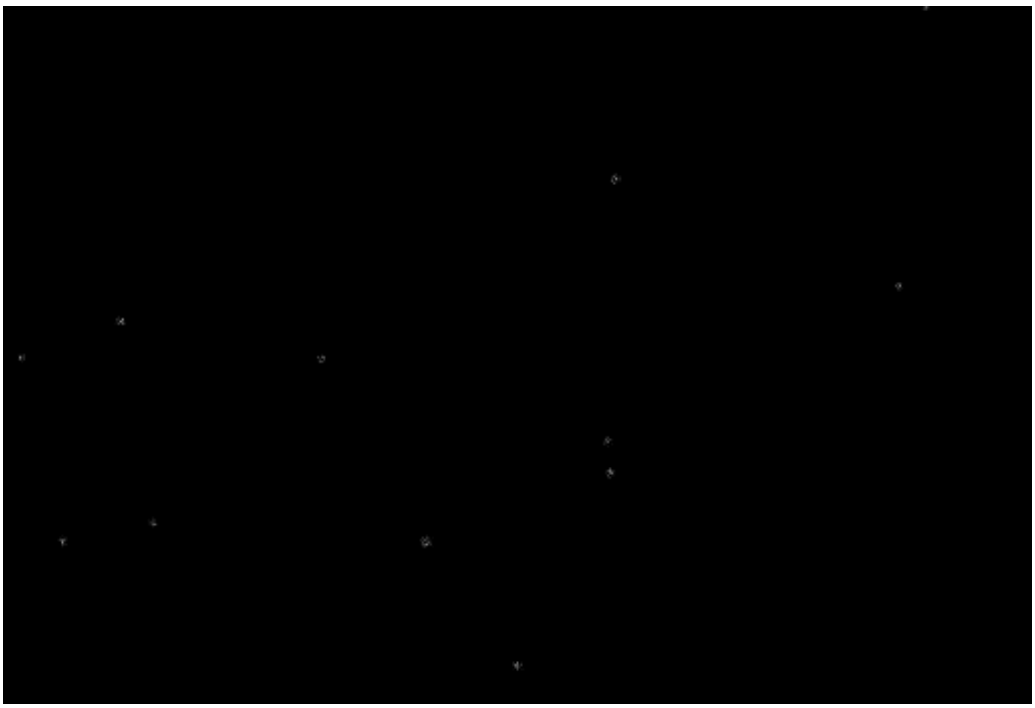

Figure 3.16 Particle rate of 100/s.

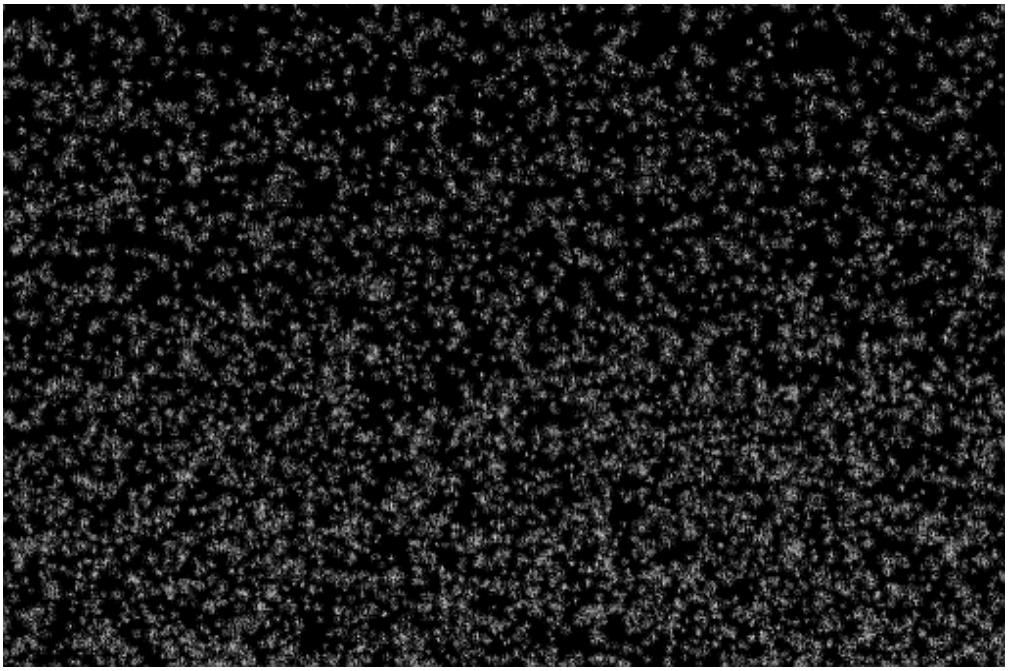

Figure 3.17 Particle rate of 50K/s.

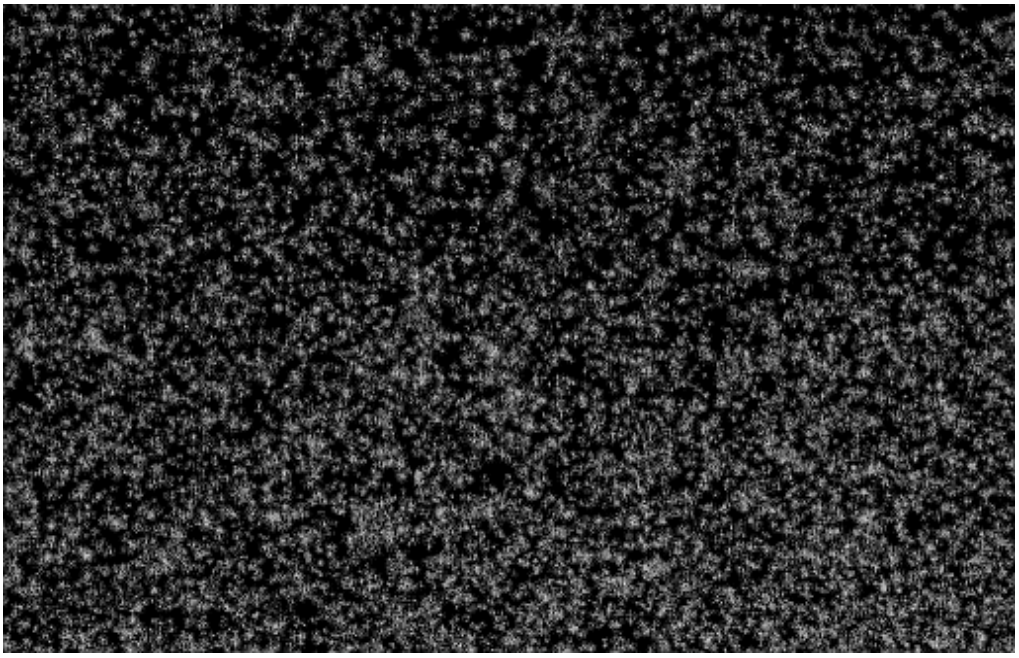

Figure 3.18 Particle rate of 100k/s.

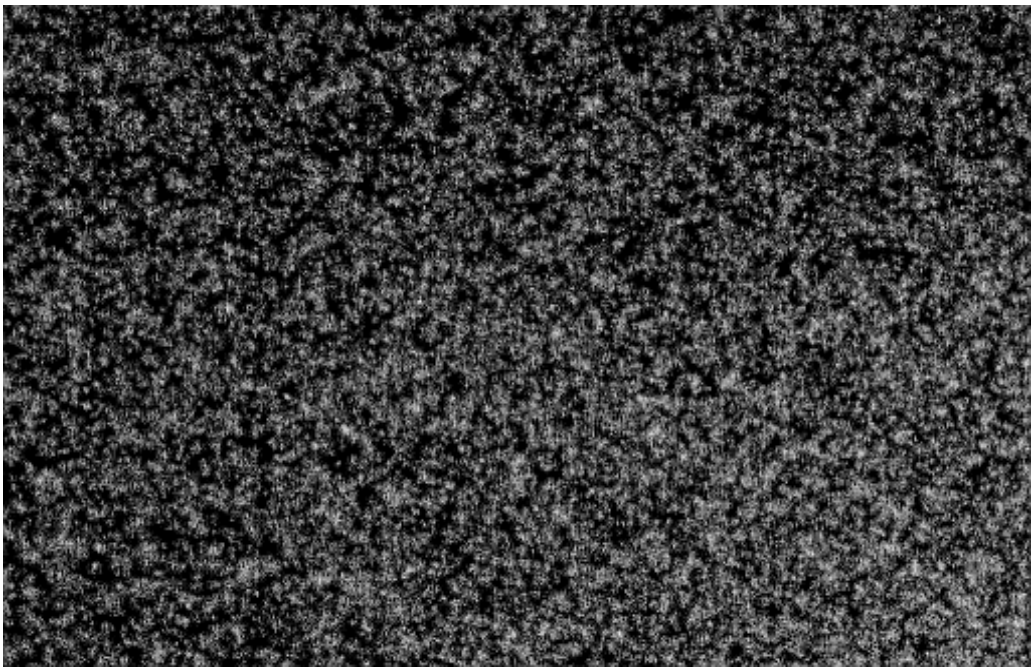

Figure 3.19 Particle rate of 250k/s.

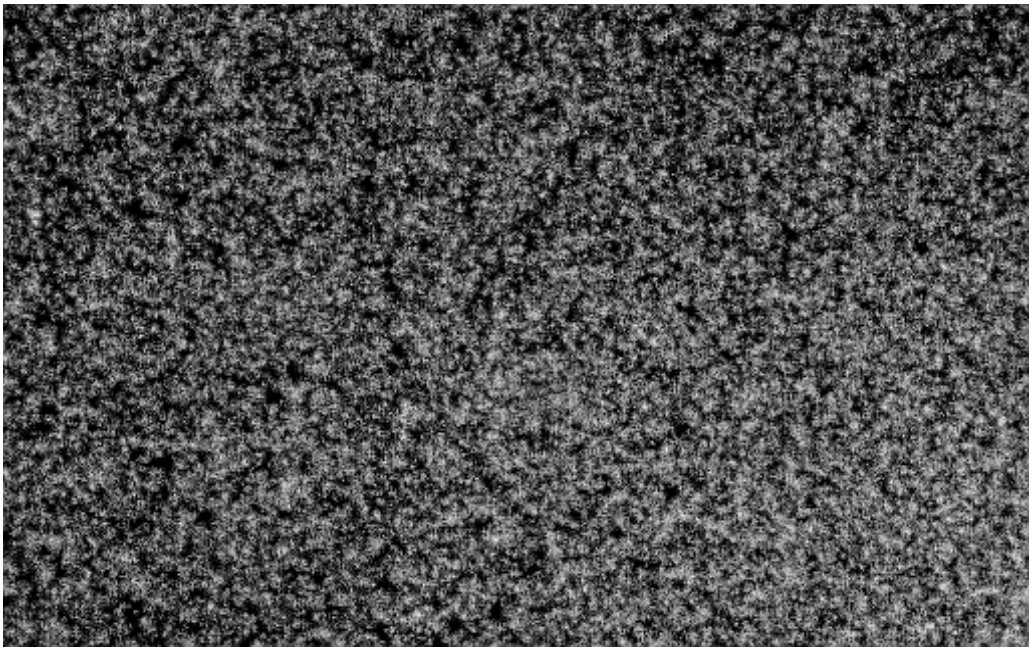

Figure 3.20 Particle rate of 1M/s.

As we can see in the above images, as the rate of snow increases, the white component of the image keeps increasing. We can consider the average color of the image as the snow density at a particular instant of time. In real world, as density of snow increases, visibility decreases. So we can relate the average color of our image with visibility. In short, as the average color of the image increases (snowing increases) towards white, we know that visibility has decreased. Hence, we first determine the average colors of the images using matlab that are given by Table 3.1.

Particle Rate = Particles/second	Average color value of image using Matlab
100	0.0326
500	0.2558
1k	0.5241
10k	5.2041
25k	12.0822
50k	23.3231
75k	31.9075
100k	37.7980
250k	55.6073
500k	74.5600
1M	77.2319

Table 3.1 Average color readings for different particle rates

Once we have the average color values, we need to map them to visibility. Since our system works well for one million particles, we have considered the particle rate of 1M particles/s as the rate at which the visibility is zero. In other words the image corresponding to 1M particles/s is taken as white that has average color value of 255. Thus converting the values from Table 3.1 to a range 0-255 we get the following values.

Particle Rate = Particles/second	y = (x/77.2319)*255
0	0
100	0.108
500	0.845
1k	1.730
10k	17.182
25k	39.891
50k	77.006
75k	105.350
100k	124.800
250k	183.600
500k	246.175
1M	255

Table 3.2 Mapping average color readings to the range 0-255

By using Matlab again we plot the graph for the values listed in table 3.2. We plot the average color values on the Y-axis and the particle rate on the X-axis.

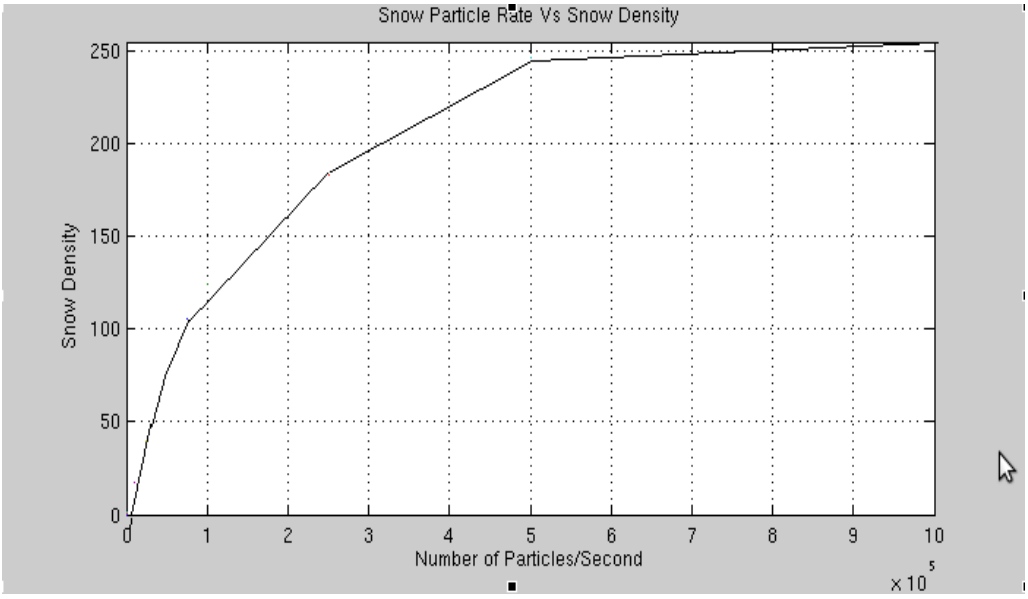

Figure 3.21 Curve for snow density Vs particle rate

From the above graph, we can find out the relation between the snow density and the particle rate, which is given by the equation:

$$f(x) = a * e^{b*x} + c * e^{d*x}$$

Where a = 325.5

b = -2.3 X 10^{-7}

c = -318.5

d = -4.2 X 10^{-6}

f(x) = density (grey scale 0 – black 1 - white)

x = number of particles flowing per second

After mapping snow density with particle rate, we bind the snow density with visibility. Currently due to lack of actual data, I have used visibility measure as a real value between 0 and 1, with 0 being no visibility (0 meters of visibility) and 1 being visibility of 600 meters. We then map this visibility with the density as follows:

$$Visibility = 1 - \frac{Density}{255}$$

Once we know the visibility factor, we can apply it get the aggregate snow effect.

Visibility Shader

{

Find the distance of the object from our snow domain

Calculate the aggregate snow fraction as

Aggregate Snow Fraction = Visibility –

((Distance of object)/Maximum Visibility Distance);

Color = (aggregate snow fraction * color of object pixel)

+ ((1.0 – aggregate snow fraction) * Background color);

}

3.5 High Dynamic Range Rendering

High Dynamic Range Rendering is the rendering of scenes by using lighting calculations done in a large dynamic range. Our visual system supports a very high contrast ratio which is generally not reflected in the graphical scenes which use Low Dynamic Range Rendering. For example, if we use 24 bits per pixel to represent color in RGB format (8 bits per color) then we get a contrast ratio of 256:1 whereas most computer monitors support contrast ratio in the range of 500:1 and 1000:1. So we use high dynamic range to render our scenes to get a better contrast ratio. HDR helps us in creating more realistic scenes as compared to the scenes using LDR for their lighting calculations. HDR can make bright things look really bright and dark things look really dark [NVIDIA].

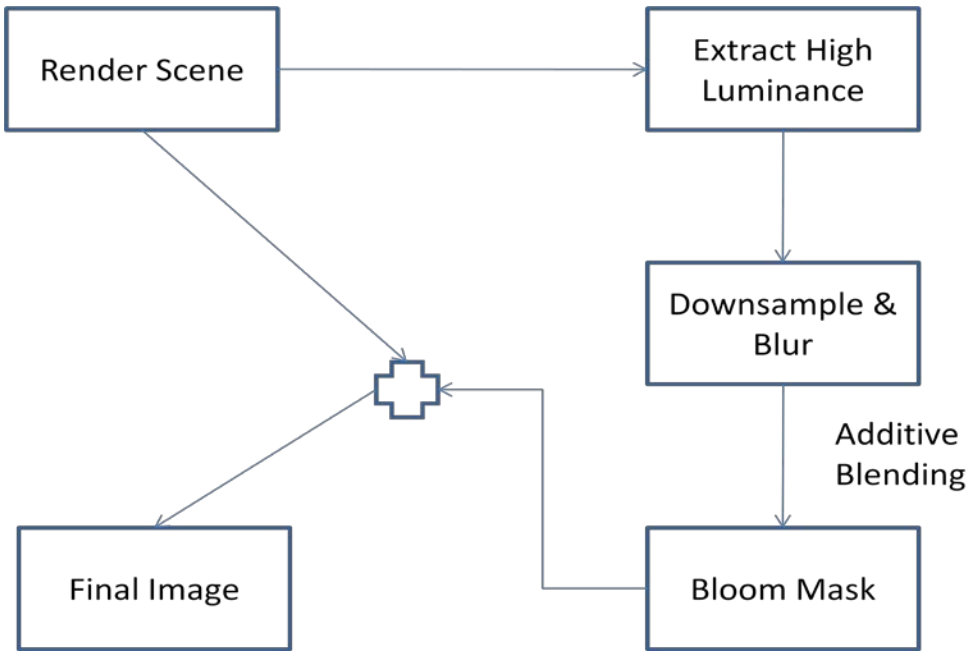

Figure 3.22 HDR Rendering Process

We render the scene to a frame buffer using bright lights in the scene. We then extract the bright areas from the scene and down sample them to create a blur effect. We down sample them multiple times which are then combined to give us a blooming effect which we can then mask with the original scene to get a final HDR image. This HDR image is then tone mapped to low dynamic range to display on the screen.

Figure 3.23 The whole scene is rendered using high dynamic range then tone mapped to convert it to low dynamic range in order to be displayed on the monitor.

We render our scene with lights having high dynamic range. For example, when we want to specify color for a yellow light in OpenGL, we would specify it as (1.0, 1.0, 0.0) in the RGB format. But in HDR, we can increase the intensity of yellow light and specify it as (10.0, 10.0, 0.0), which will be ten times brighter than the original yellow light source. We have modeled such lights on back of a snowplow with high dynamic range. We render the scene to a frame buffer object and then extract the high luminance areas from the scene. We have two bright yellow lights and two not so bright red lights on a snowplow. So when the high luminance areas are extracted we will see something like that shown in Figure 3.24.

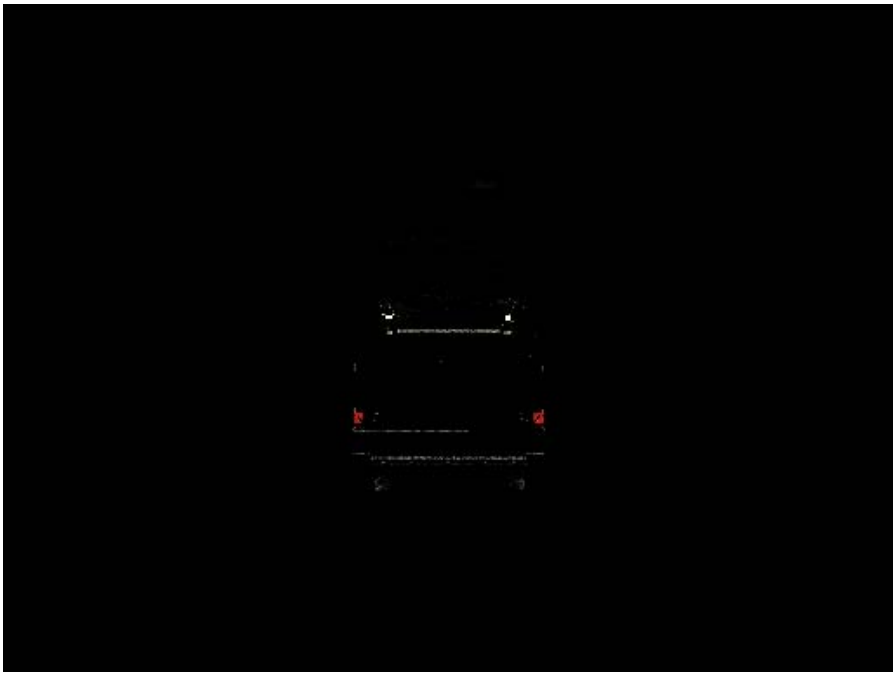

Figure 3.24 High Luminance areas extracted from the image into a texture

To extract the high luminance part we do the following:

{

For every texture coordinate do

Get the color of the texel from the texture

Set some threshold value

Calculate

Y = Max((HDR Color – threshold)/(1 - threshold))

Luminance = 0.2125 X Y.r + 0.7154 X Y.g

+ 0.0721 X Y.b

Color = Color X Luminance

}

To create a bloom effect we downsample this texture which means we reduce the texture's size a desired number of times. While downsampling we also apply a simple Gaussian blur. A Gaussian blur effect is typically generated by dividing the process into two passes. In the first pass, we blur the image in only the horizontal or vertical direction. In the second pass, we blur the image in the remaining direction.

In horizontal Gaussian blur, we add up a fixed size of neighboring texels (horizontally) to get the blur effect in the horizontal direction.

Horizontal Gaussian blur

{

For every texture coordinate do,

Get the color of a texel

Add to it the color of neighboring 256 texels (128 on the left and 128 on the right)

Return the color of the texel

}

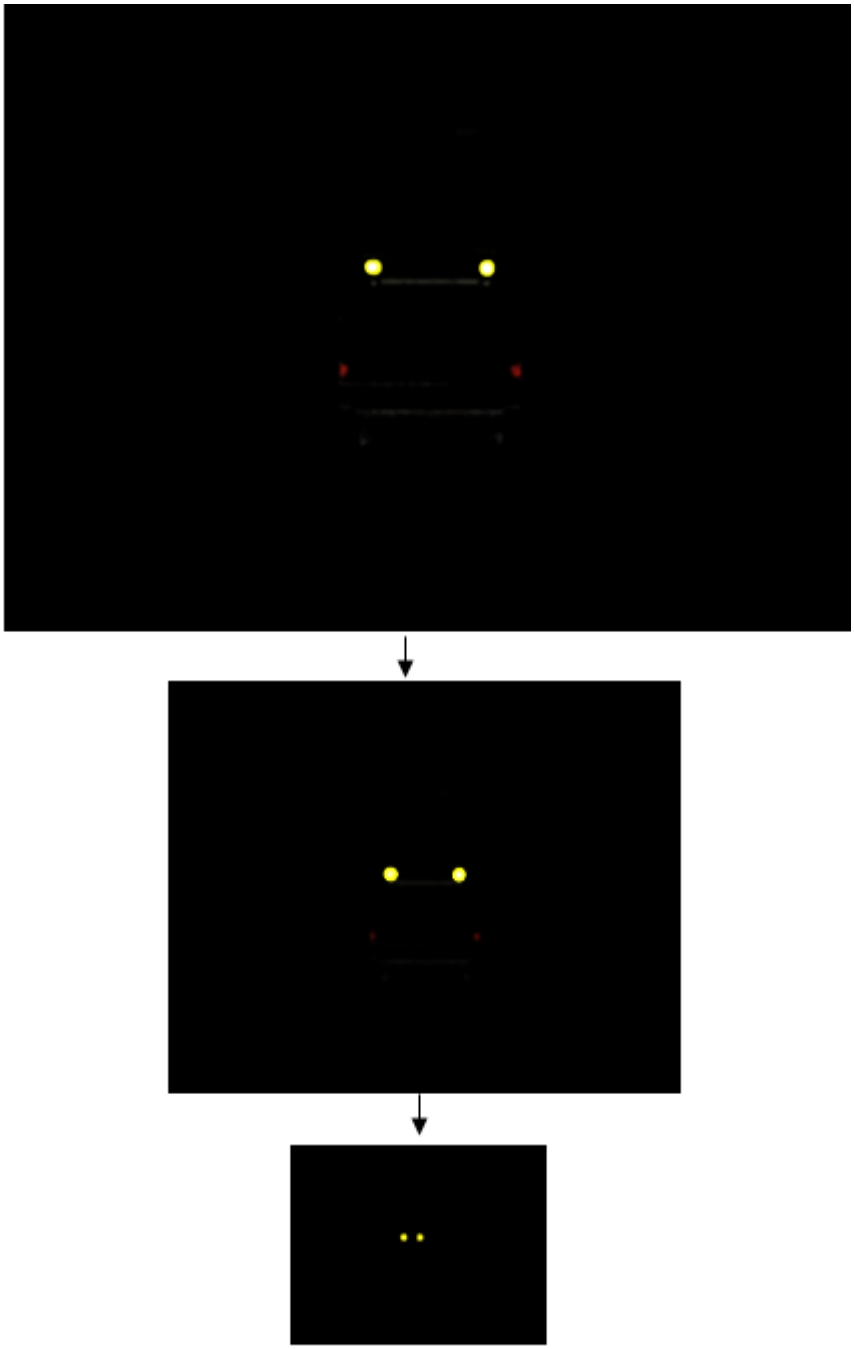

Figure 3.25 Downsampling and horizontal blur

In vertical Gaussian blur, we add up a fixed size of neighboring texels (vertically) to get the blur effect in the vertical direction.

Vertical Gaussian blur

{

For every texture coordinate do,

Get the color of a texel

Add to it the color of neighboring 256 texels (128 texels above the current texel and 128 texels below the current texel)

Return the color of the texel

}

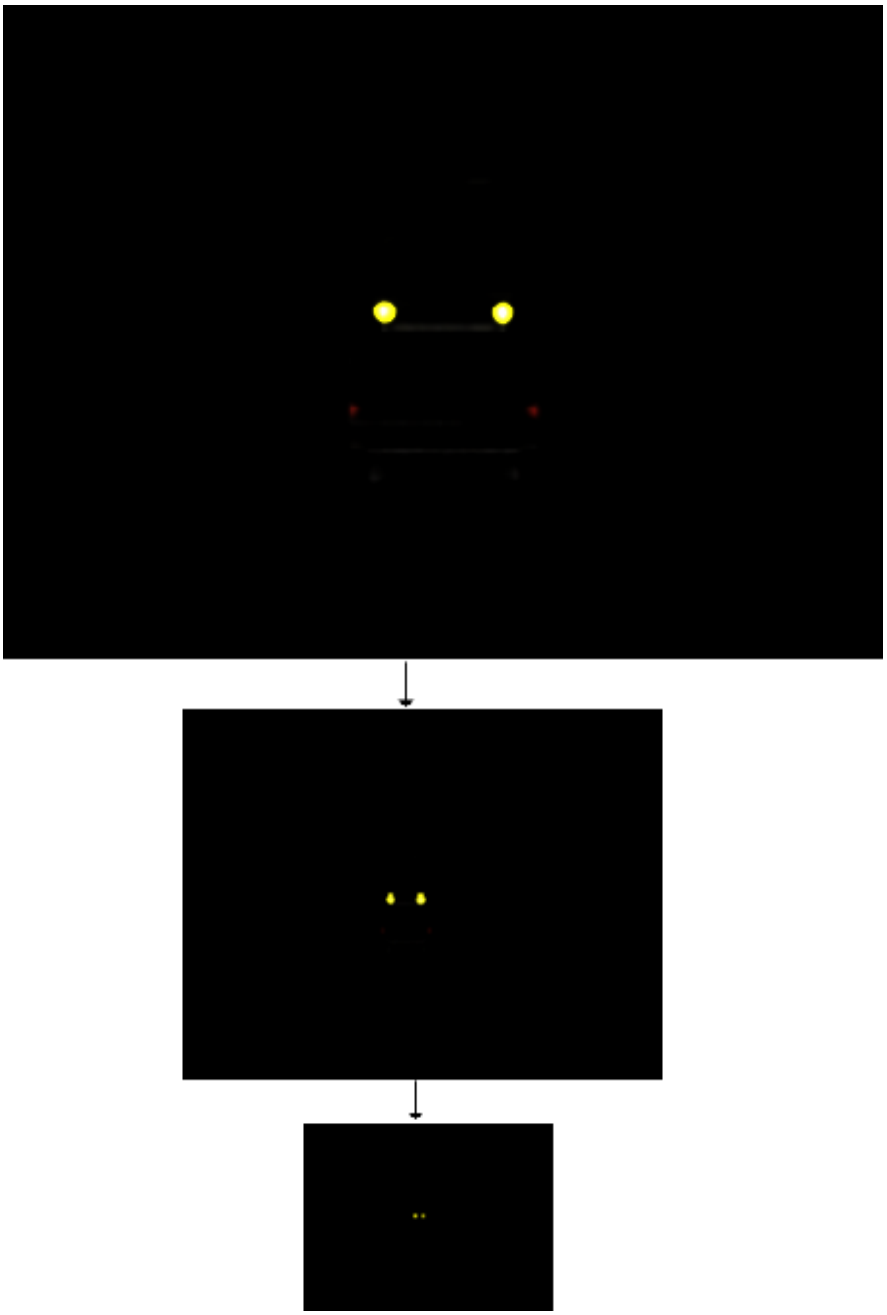

Figure 3.26 Downsampling and vertical blur

We add up all the images and then do a tone mapping to get the final image as seen in Figure 3.27. The tone mapping is done by the following formula:

$$Y = exposure\ X\ (\frac{(\frac{exposure}{maxLuminance} + 1)}{(exposure + 1)})$$

This tone mapping allows us to control the exposure which we can set through the program. With small exposure the scene is dark while it is very bright with high exposure.

Figure 3.27 Final image after bloom effect and tone mapping

After we finish with the tone mapping we are ready to render the image on the screen which looks like the image in the above figure. We can see the glow and the bloom effect created by HDR rendering around the lights on the snowplow.

4. Results

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 see how the sorting with the GPU is better than sorting on the CPU. Then, we will see actual results using LDR and HDR rendering. We will then see how aggregate snow affects our scene and finally how our scene would 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.

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

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

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.

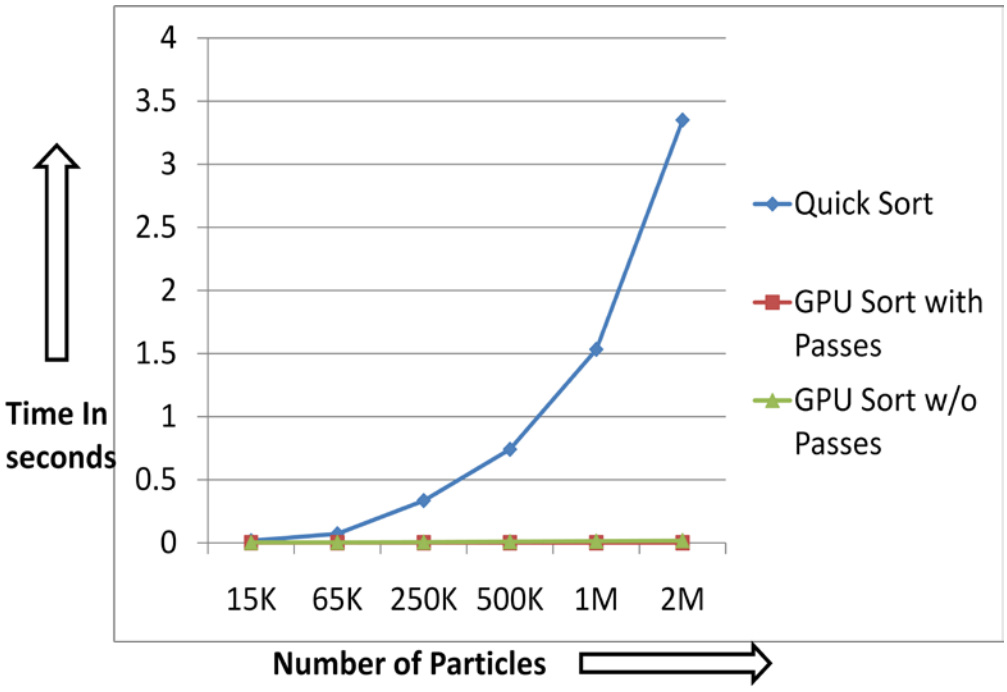

Figure 4.1 Comparison between Quick Sort (CPU) and GPU Sort (GPU)

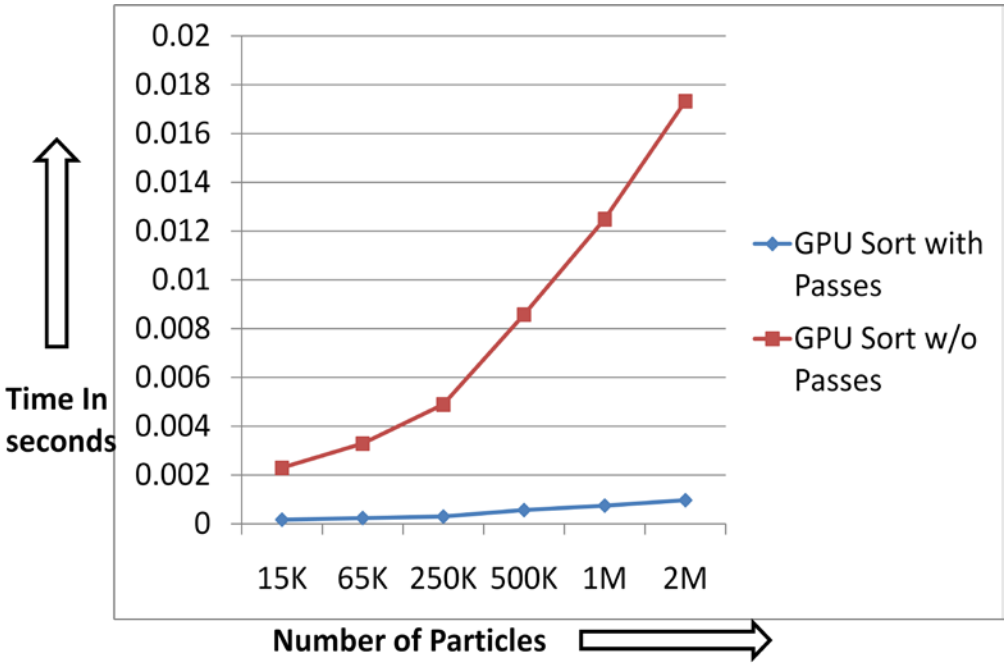

Figure 4.2 Graph showing curve for GPU Sort (with passes Vs w/o passes)

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.

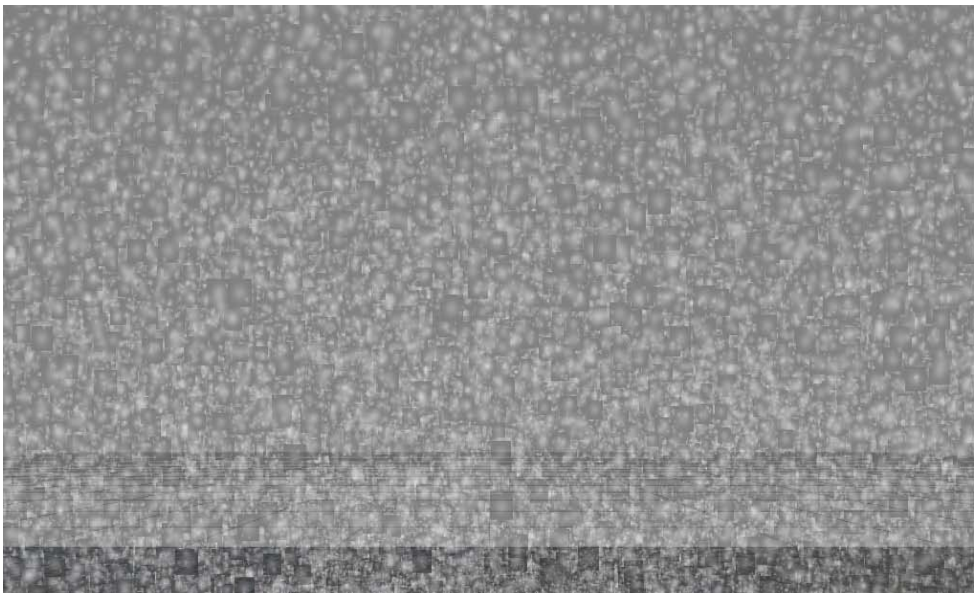

Figure 4.3 Unsorted snow particles with artifacts

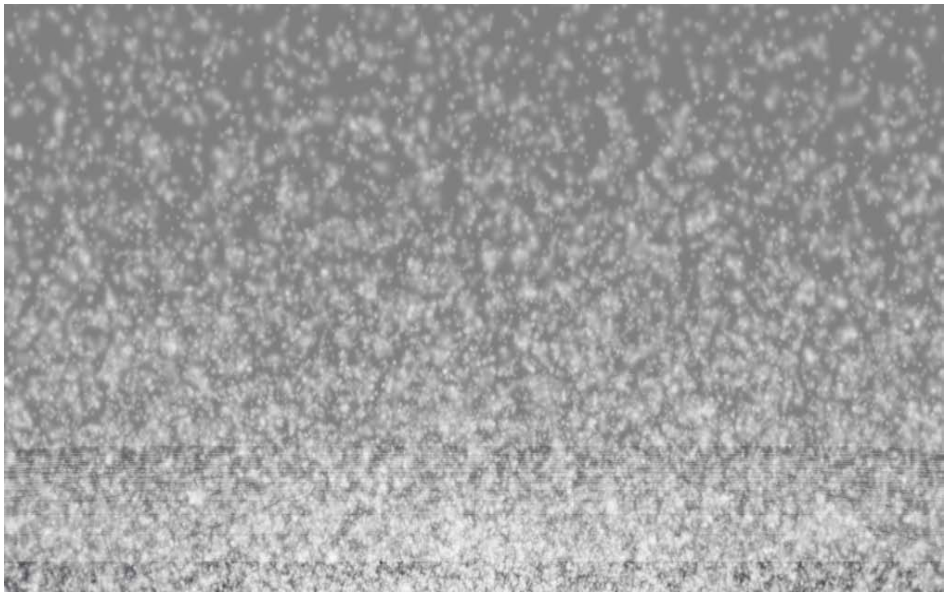

Figure 4.4 Sorted particles using GPU Sort

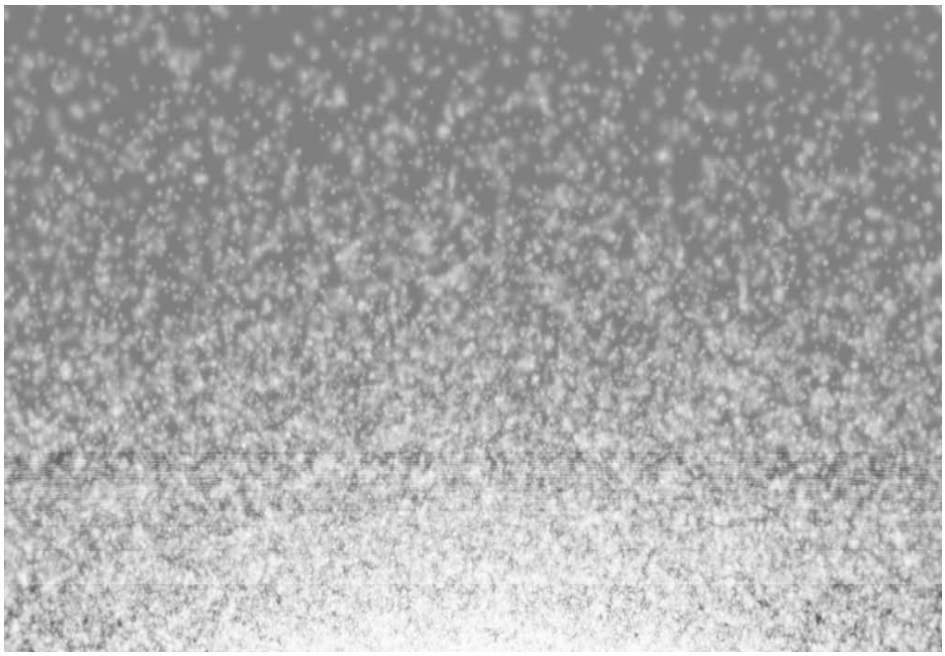

Figure 4.5 Snow particles using additive blend

As we can see from the above figures, there is not much difference between the images for sorting and additive blend. Hence we use additive blend in our snow system.

4.2 LDR Vs HDR Rendering

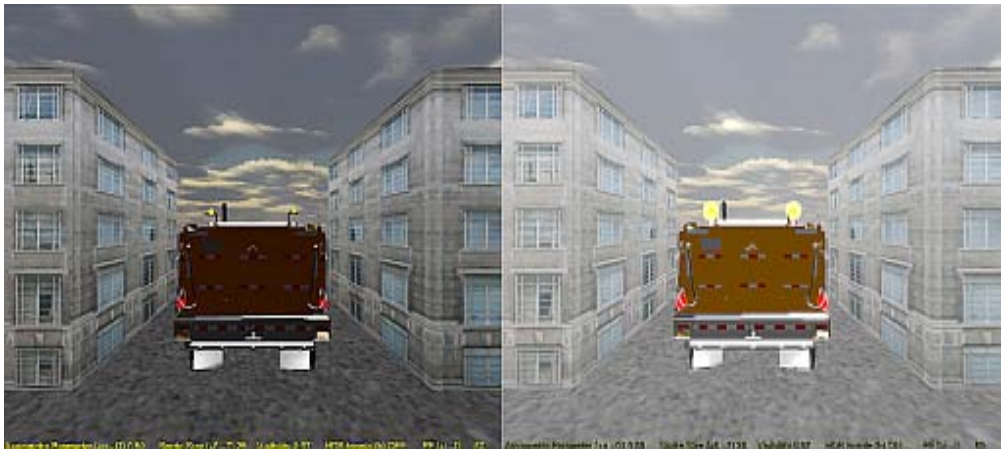

Figure 4.6 (Left) Image with LDR Rendering (Right) Image with HDR Rendering

We have modeled a snowplow [Michele Olsen, 2008] with lights on it. The lights on a snowplow truck are bright and the difference between bright and not bright lights can be highlighted by using HDR rendering. In the above figure, we can see the stark difference between LDR and HDR rendering in the two images. HDR can make bright things look much brighter like the shining flaps on the rear of the snowplow. Also, we can see the glow of the lights on the snowplow as a result of HDR rendering.

4.3 Visibility

Figure 4.7 Aggregate snow Effect

In the above images we have modeled the aggregate snow effect in snowing conditions as explained in Section 3.5. Here we have different visibilities in the three images depending how heavy the snow is. Visibility is given in the range of 0 to 1 where 0 means no visibility and 1 means totally clear/visible. The density of snow is determined by the snow particle rate. We can change the visibility in a scene by changing the particle rate.

Image 1 = > Visibility 0.76 (Less snow 6000 particles/s)

Image 2 = > Visibility 0.57 (Heavy snow 600000 particles/s)

Image 3 = > Visibility 0.16 (Very heavy snow 900000 particles/s)

4.4 Snow Model

Figure 4.8 Snow model with Gaussian transparency, HDR rendering, aggregate snow and scattering effects

We have developed our snow model which interacts with different dynamic light sources and generates effects such as scattering of light and high dynamic range rendering. In the figure above we can see the snow system with all the effects such as Gaussian transparency, scattering, HDR and aggregate snow.

5. Conclusion

We have solved a computationally intensive problem on the GPU and have successfully implemented a better model of snow, which looks more realistic in an interactive environment. We have provided a snow model that renders the snow particles effectively using transparency and scattering by multiple dynamic light sources. We have provided an environment for different snowing conditions which could be used for driving in snowing conditions behind a snowplow or other vehicles. The use of High Dynamic Range rendering has contributed in making the images more realistic as we were able to support high contrast ratios.

Since rendering of snow particles is computationally intensive we cannot render snow over the entire scene. Hence we have a snow domain surrounding the user in which we render the snow particles. For the rest of the scene we have developed an aggregate snow effect which is tied in with the visibility factor during snowing. An improvement could be made to tie in the visibility with actual data from snowing conditions in the real world.

Currently we are using an additive blend for the blending of transparent snow particles. An improvement would be to use the GPU sort selectively on particles which are near to the viewer thus saving some computation time. Also, in future, it would be desirable to tie in the shadowing effect between the snow particles.

Bibliography

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

Govindaraju, N., Raghuvanshi, N., Henson, M., Tuft, D., Manocha, D. (2005). A Cache Efficient Sorting Algorithm for Database and Data Mining Computations using Graphics Processors. *Tech. Rep. TR05-016, University of North Carolina,* 2005.

Harris, M.J. (2003). Real-Time Cloud Simulation and Rendering. *PhD thesis, CS Dep., University of N. Carolina at Chapel Hill,* 2003.

Harris, M. J., Baxter, W., Scheuermann, T., & Lastra, A. (2003). Simulation of cloud dynamics on graphics hardware. *In Proceedings of the ACM Siggraph/Eurographics conference on Graphics hardware 2003,* pp.92-101, July 2003.

Harris, M. J., Lastra, A. (2001). Real Time Cloud Rendering. *In Proceedings of the ACM Siggraph/Eurographics conference on Graphics hardware 2001,* pp. 76-84.

Houlmann, F., Metz, Stephane, High Dynamic Range Rendering in OpenGL. *utbm.*

Lengyel, Eric (2007). Unified Distance Formulas for Halfspace Fog. *Journal of graphics tools, Volume 12, Number 2,* pp. 23-32.

Norgren, Andrew (2008). GPU Based Particle Dispersion Modeling with Interactive Visualization Support for Real-time Simulation. *Thesis, University of Minnesota, Duluth,* June 2008.

Olsen, Michele (2008). A Snow Plow Model. *Unpublished, University of Minnesota, Duluth.*

Preetham, A., Shirley P., Smits, B. (1999). A Practical Analytic Model for Daylight. *Siggraph 1999, Computer Graphics Proceedings, Annual Conference Series,* pp. 91–100.

Rost, R. J. (2006). *OpenGL Shading Language Second Edition.* Addison-Wesley.

Shreiner, D., Woo, M., Neider, J., & Davis, T. (2006). *OpenGL Programming Guide.* Addison-Wesley.

Wade, B.,Wang, N. (2004). Rendering Falling Rain and Snow. *In Proceedings of the ACM Siggraph/Eurographics conference on Graphics hardware 2004,* pp. 14.

Willemsen, P., Norgren, A., Singh, B., & Pardyjak, E. (2007). Development of a new methodology for improving urban fast response lagrangian dispersion simulation via parallelism on the graphics processing unit. *11th Intl Conf on Harmonisation within Atmospheric Dispersion Modeling,* pp. 363-367.