University of Nevada
Reno

Sandstorm: A Dynamic Multi-contextual GPU-based Particle System using Vector Fields for Particle Propagation

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science
in Computer Science

by

Michael John Smith

Dr. Frederick C. Harris, Jr., Thesis Advisor

May, 2008

Abstract

The use of virtual reality (VR) to visualize scientific phenomena is a quite common and useful application. Virtual reality allows scientists to immerse themselves in the phenomena that they are studying. One such phenomenon currently under study is the dust kicked up by a helicopter's down-draft. This phenomenon is being studied because of the difficulty of landing a helicopter in a desert environment. Such a terrain would have large amount of loose sand that can easily be kicked up by a helicopter and interfere with the normal operation of a helicopter. In order to study such a phenomenon, an application will need to employ a particle system designed to create such visual effects. A particle system that creates such effects would need to be able to handle the updating and rendering of large amounts of particles. These particles need to follow a well defined scientific model during the course of their life time. Also such a particle system would need to be aware of multiple contexts, considering the virtual reality environment in which it would be used in. Sandstorm is a dynamic multi-contextual GPU-based particle system that uses vector fields for particle propagation, is presented in this Thesis.

Acknowledgments

This work is funded by the STTC CAVE Project (ARO# N61339-04-C-0072) at the Desert Research Institute.

First of all I would like to thank my committee for their help in getting me this far. I would like to personally thank my committee chair, Dr. Frederick C. Harris Jr., who has been a valued teacher and mentor throughout my entire college carrier. I would also like to thank my other committee members, Dr. Sergiu Dascalu and Dr. Scott Bassett, who have both been instrumental throughout my graduate school experience. I extend my thanks to all the help that I have received from my co-workers, especially Roger Hoang. Finally, many thanks go out to all of my family and friends, who's undying support has helped me through the most difficult of times.

Contents

Abstract **i**

List of Figures **iv**

List of Tables **v**

1 Introduction **1**

2 Background **4**

2.1 Helicopter and Dust Simulation 4
2.2 Virtual Reality 4
2.2.1 Depth Cues and Stereoscopic Displays 5
2.2.2 Multiple Contexts 8
2.2.3 Virtual Reality Toolkits 8
2.3 Wrath of Khan and the Genesis Effect 10
2.3.1 Particle Generation 12
2.3.2 Particle Attributes 14
2.3.3 Particle Dynamics 15
2.3.4 Particle Extinction 15
2.3.5 Particle Rendering 15
2.4 GPU offloading 16
2.4.1 Vertex and Fragment Shaders 16
2.4.2 Geometry Shaders and Shader Model 4 17
2.4.3 Scientific Computation on the GPU 18
2.5 Related Work 19
2.5.1 Uber Flow 19
2.5.2 Building a Million Particle System 20
2.5.3 Particles 23

3 Sandstorm the Concept **25**

4 Sandstorm Software Specification and Design **27**

4.1 Sandstorm Goals 27
4.2 System Requirements 28

4.3 Software Specifications 29
4.4 Use Cases 29
4.5 Modeling and Design 30
4.5.1 Main Class Hierarchy 32
4.5.2 Utility Classes 35

5 Sandstorm Prototype **37**
5.1 GPU-Based 37
5.1.1 Creating and Destroying Particles 37
5.1.2 Updating Particles 41
5.1.3 Rendering Particles 41
5.2 Vector Fields 43
5.3 Multi-contextual 44
5.4 Dynamic 45

6 Results **47**

7 Conclusions and Future Work **52**
7.1 Conclusions 52
7.2 Future Work 53

Bibiliography **55**

List of Figures

2.1 Screenshot from the Dust Framework project. 5
2.2 An example of a CAVE system. 6
2.3 A example of a Head mounted display. 7
2.4 A sample application written with VRUI. 9
2.5 Surface impact. 11
2.6 Spearding across the surface. 12
2.7 Typical spherical generation shape [24]. 14
2.8 OpenGL Programmable pipeline [26]. 17
2.9 An example of UberFlow's capability. 20
2.10 An example of how Latta stores particle information. 21

4.1 Use cases for the Sandstorm prototype. 31
4.2 The main system structure of the Sandstorm system. 33

5.1 An example how Sandstorm store particle information. 39
5.2 The main system structure of the Sandstorm system. 45

6.1 An another example of multi-contextualism. 49
6.2 An example of particles following a vector field. 50
6.3 Another example of particles following a vector field. 51

List of Tables

4.1 Functional requirements for the Sandstorm prototype application. . . 30
4.2 Non-functional requirements for the Sandstorm prototype application. 32

5.1 Table of dynamic attributes. 46

Chapter 1

Introduction

Currently around the world helicopter pilots are finding it very difficult to land in desert environments. Such environments contain vast amounts of loose sand due to the lack of vegetation. This loose sand is easily affected by winds, generated by a helicopter. Just as the desert environment is affected by a helicopter, so too is the helicopter affected by the desert terrain. These effects that helicopters experience include engine clog, very low visibility, and unstable flying conditions. Many helicopter crashes and deaths have been attributed to these effects that are caused by helicopters producing large amounts of wind and in turn creating dangerous dust levels. This thesis presents Sandstorm, a particle system that was designed to create such atmospheric phenomena as mentioned above. Sandstorm will be used in both scientific simulations and military training applications that will give both scientists and pilots a better understanding of what happens in such cases.

In the realm of virtual reality it is quite common for an application to implement a particle system to heighten the quality of immersion or to display scientific data. This is the case in fluid dynamic applications, dust simulations, military training simulations, and fire simulation applications. Nonetheless, whatever application the particle system is used in, it needs to look realistic or at least behave realistically. Vector fields can be used to 'guide' particles according to real scientific data, while also adding to realism. The idea of using vector fields to guide the motion of particles and other objects is not new, and has been discussed in [6]. In [6], the authors propose that vector fields be used to manipulate particle systems and a class of soft

objects.

Virtual reality applications and simulations require a multiple contextual environment in which to run [9], [27]. A main context, which is in charge of updating simulations and communication, controls several rendering contexts. Each of these rendering contexts are responsible for rendering to one of the screens of a virtual reality environment. The rendering contexts will run independent of each other, and communication between them is usually only done when they are ready to draw. This design requirement can cause problems when using context sensitive data and algorithms, such as particle systems. Therefore, special consideration has to be taken into account when dealing with particle systems in a multi-contextual environment. One such consideration is that most particle systems use random numbers to seed a particular particle's initial speed and direction. Since a copy of the particle system will be run per-context, we need to make sure that each context gets the same random number every time one is needed. Also, the particles need to be updated consistently across all particle systems. Both of these considerations will be discussed later in this thesis.

Virtual reality applications and simulations utilize a multiple context/thread environment to handle both complex data manipulation and intensive rendering. This complex data manipulation leaves little to no room for a particle system to run on a large scale, large enough to allow for rich data sets. GPU offloading techniques have been proven to allow applications and simulations to offload work, such as particle systems calculations, on to the graphics hardware freeing up more resources on the CPU. Textures and vertex buffer objects can be used to encode complex functions or to relay large data sets to the shaders. One has to be careful however; a GPU offloading algorithm has to be designed such that data does not need to be constantly passed back and forth between the GPU and the CPU. This can cause the bus to be overloaded and slow the rendering process down. Transform Feedback and Geometry Shaders, both of which are part of Shader Model 4, can help alleviate this data passing problem [20, 21].

It is also often the case that particle systems are written such that once they are set into motion there is very little or no changing of the variables that are part of the particle simulation. Keeping this in mind, this thesis describes Sandstorm, a particle system that has the ability to be dynamically changed while being multi-contextual and utilizing GPU offloading and vector fields for particle propagation.

In the following chapters you will find a description of Sandstorm and all the issues surrounding its development. In Chapter 2, you will find a background on the project, which includes all the information you will need to understand the issues surrounding the development of such this project. Chapter 3 will introduce what Sandstorm hopes to become. An overview of how Sandstorm was designed is presented in Chapter 4. In Chapter 5 you will find a description of how Sandstorm was engineered. Results can be found in Chapter 6. Conclusions and Future Work can be found in Chapter 7.

Chapter 2

Background

2.1 Helicopter and Dust Simulation

Before the question of 'what needs to be done' is answered, the question of 'why' needs to be addressed. The Helicopter and Dust Simulation, Heli-Dust, is a scientific simulation in which the effect of a helicopter's downdraft on the surrounding desert terrain is studied. Heli-Dust, currently being developed at the Desert Research Institute, is designed to run in a virtual reality environment and will need the use of a particle system to display the dust cloud generated by a helicopter. Heli-Dust was written using the Dust Framework, a framework which allows the developer to setup a scene using an XML file. A current illustration of Heli-Dust is shown in Figure 2.1.

Early prototypes for Heli-Dust implemented a very simple particle system in which particles where emitted and accelerated upwards. However, it was quickly determined that Heli-Dust would need a much more in-depth particle system, one in which gathered scientific data could be used to 'guide' the particles' motion. It was also determined that the particle system used would have be as efficient as possible, so GPU offloading was considered. The following sections cover different aspects and issues one would encounter when developing such particle systems.

2.2 Virtual Reality

The ability to immerse oneself in an application or simulation has interested both scientists and engineers alike. Such applications and simulations require knowledge

Figure 2.1: Screenshot from the Dust Framework project.

about how virtual reality works and what constraints you will be facing. In the following section you will find a description of the basics of a virtual reality environment like the one that will be used with Sandstorm, such as depth cues and stereoscopic displays. Also in the following subsections there is a description of the relevant constraints when developing a piece of software such as Sandstorm in a virtual reality environment.

2.2.1 Depth Cues and Stereoscopic Displays

First we must discuss visual depth cues. A depth cue is an indicator in which a human can perceive information regarding depth [28]. These depth cues come in many shapes and sizes and include monoscopic, stereoscopic, and motion depth cues. A monoscopic depth cue is one in which only information from a single eye, or information for a single image is available. This information can include: position, if an object is behind something else; size, things in the distance appear smaller; and brightness. A stereoscopic depth cue is one in which information is obtained from two eyes; this

information is derived from the parallax between the different images received by each eye. Thus, stereoscopic depth cues depend on the parallax between objects, which is the apparent displacement of objects viewed from different locations. Motion depth cues come from a phenomenon known as motion parallax, which is caused by the changing relative position between the head and the object being observed. Depth information is observed by noticing that objects in the distance move less than objects closer to the viewer.

Now that we have looked at what a depth cue is, let's look at how virtual reality uses these depth cues to fool our eyes into believing that a 2D image is a 3D image. Rendering a particular viewport of each eye on a stereoscopic display allows the user to view stereoscopic depths where otherwise they would not be able to. Stereoscopic displays come in all shapes and sizes including but not limited to: CAVE systems that allow the user to "step" in the world(Figure 2.2), head mounted displays that afford some privacy(Figure 2.3), and single walled displays.

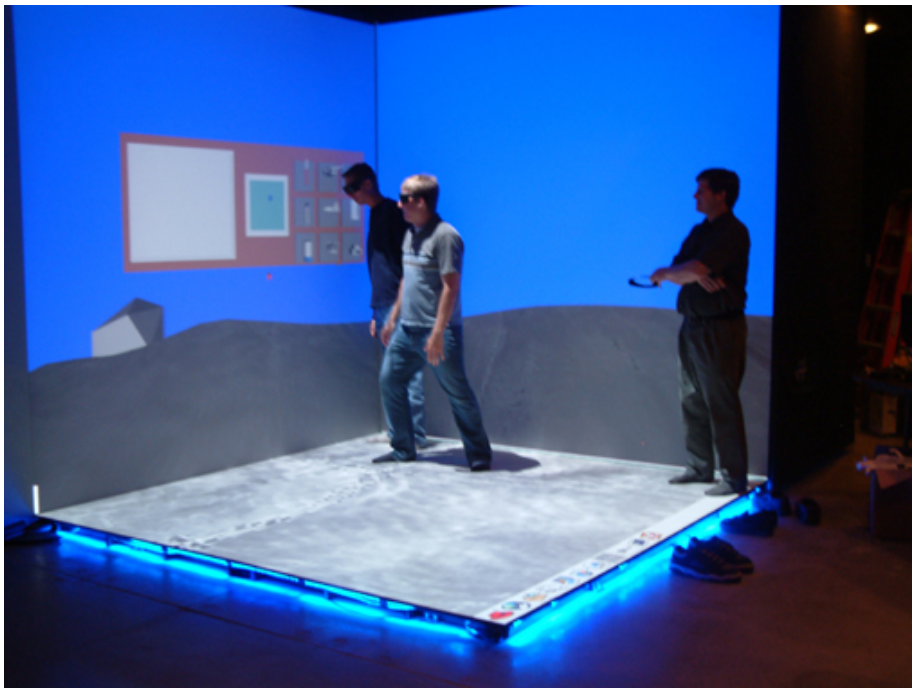

Figure 2.2: An example of a CAVE system.

Figure 2.3: A example of a Head mounted display.

There are two ways of accomplishing stereoscopic vision, and they are active and passive stereo. An active stereo system uses a combination of glasses with built-in shutters, and a projector or projectors that rapidly switch back and forth between the left and right eye image. The shutters in the glasses allow the user to only see one image at a time, thus creating the illusion of depth. Active stereo has the benefit of allowing for a very deep sense of depth [28]. Passive stereo uses two projectors for a single screen, one view for each eye. A polarized filter is used on each projector, giving the light coming out of the projector a predetermined polarization. The user then wears a set of polarized glasses, each eye has the same polarization as one of the filters used on one of the projectors. Each eye only sees one of the images creating the the illusion of depth [28]. Most CAVE virtual reality systems will use active stereo because it gives a very deep sense of depth, giving the viewers the sense that

they are walking into another world.

2.2.2 Multiple Contexts

A typical virtual reality application is split into multiple contexts: a main context and multiple rendering contexts. The main context usually handles things like input abstraction and simulation gathering, while the multiple rendering contexts handle the rendering of the scene according to the view that its screen is suppose to have. Because of these multiple contexts, a virtual reality application developer needs to make sure that all context sensitive information and algorithms are multiple context safe. Things like random number generation need to be generated in the main context and passed down to the render contexts. Also a major consideration, especially for particle systems, is that if billboards are drawn each rendering context needs to get the correct coordinates of the camera.

2.2.3 Virtual Reality Toolkits

There are many virtual reality toolkits and libraries that are currently available for use in a virtual reality applications. Such toolkits and libraries usually handle things like creating stereosopic images, setting up the virtual reality environment, and handling distribution methods. This section describes two such toolkits/libraries: VRUI and FreeVR.

Virtual Reality User Interface, or VRUI, is a virtual reality development toolkit that was developed by Oliver Kreylos at UC Davis [9]. Figure 2.4 is an example of an application written with VRUI. VRUI's main mission is to shield the developer from a particular configuration of a virtual reality environment. This will allow developers to develop quickly and in a portable and scalable fashion. VRUI accomplishes this mission by the abstraction of three main areas that are common to all virtual reality environments and systems: display abstraction, distribution abstraction, and input abstraction.

The display abstraction provides OpenGL rendering contexts that are set up in

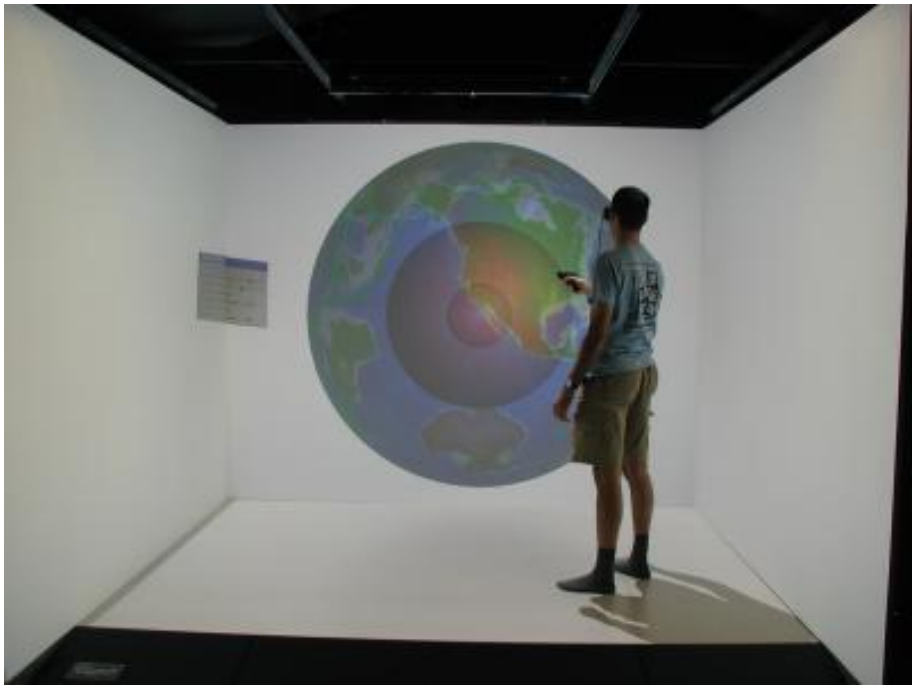

Figure 2.4: A sample application written with VRUI.

such a fashion that rendering a scene will display that scene on all rendering surfaces. These rendering surfaces run the gambit from monitors to screens to head-mounted displays. This display abstraction is also in charge of rendering the scene in the correct stereographic mode.

The distribution abstraction provides the developer with a transparent environment in which the distribution method is hidden from the developer. There are three types of distribution as defined by VRUI: split first, split middle, and split last. Split first is where an application is replicated on all the computers in the environment, and data is synchronized by hardware. Split middle is a method in which a shared data structure, such as a scene graph, is used to transmit data from a main application/computer (or cluster) to all the rendering applications/computers. Finally split last, consists of an OpenGL API calls are broadcast from a main application/computer to all rendering applications/computers.

According to VRUI, a toolkit must hide the differences in input hardware and provide a uniform view of the set of connected input devices. Also, the toolkit needs

to provide common mechanisms to access these input devices and specify input requirements at a higher level.

One of the best features of VRUI is its easy to program menu system. This system allows a developer to program menus for use in a 3D virtual reality environment. When programming for a virtual reality environment, one is usually forced to map actions to buttons. This can be somewhat of a hassle because certain actions don't always map well to a single button. Some actions map better to a slider or an interactive graph where points can be dragged to modify the state of the program. Sometimes the number of actions that need to be done might be too large for the user to remember all of them and might cause the user to forget what actions they can perform. Being able to choose actions from a menu of actions can alleviate this problem.

VRUI was chosen as the development environment for Sandstorm, due to its to program menus. The menus allow Sandstorm the flexibility to change itself on the fly.

Developed and maintain by William Sherman, FreeVR is an open source virtual reality interface/integration library [27]. FreeVR was designed to work on a diverse range of input and output hardware and also to work on existing as well as newly established VR systems. However, FreeVR is not a content engine; it does not contain a scenegraph or other content organizational features. In this regard FreeVR is a lot like VRUI; however, there are some major differences between FreeVR and VRUI. The first difference is that FreeVR has no menu system; it leaves that up to the user. Also unlike VRUI, FreeVR is currently designed to work on shared memory systems, whereas VRUI focuses on clusters.

2.3 Wrath of Khan and the Genesis Effect

The first particle system ever used was implemented for use in the movie Star Trek: The Wrath of Khan and was outlined in a paper by William T. Reeves [24]. In the paper, *Particle Systems - A Technique for Modeling a Class of Fuzzy Objects*, Reeves

describes the particle system that was used to create the Genesis Effect, which can be seen in Figure 2.5 and Figure 2.6.

Figure 2.5: Surface impact.

The paper introduces particle systems as the modeling of an object as a cloud of primitive particles that define its volume [24]. During the course of the system's lifetime; particles are generated into the system, particles move and change form within the system, and then they die and are removed from the system.

The paper categorizes particle systems as a "fuzzy" object in which they do not have smooth, well-defined, and shiny surfaces; instead their surfaces are irregular, complex, and ill-defined. A method for modeling and displaying these fuzzy objects called particle systems is presented in this paper. There are three main differences between modeling regular objects and particle systems. First, the surfaces of the fuzzy object are not smooth, well defined, and made up of many primitive elements. Instead they are defined as clouds of primitive particles that represent its volume. Second, a particle system is a dynamic object, one in which its elements are born, die, and change shape. Lastly, the object that the particle system represents usually is non deterministic, since the shape of the object isn't always well known, such as

Figure 2.6: Spearding across the surface.

fire.

Reeves determined that during each frame five steps need to be performed: (1) particles are born, (2) each new particle is assigned individual attributes, (3) any particles that have existed within the system and have passed their prescribed lifetime are extinguished, (4) the remaining particles are moved and transformed according to their dynamic attributes, and (5) an image of the living particle is rendered in a framebuffer. Any number of instructions that make sense can be used in the above set. For example the updating of the particles positions can be described by a series of partial differential equations, or even a vector field. In the following sections you will find a description of each of the steps listed above.

2.3.1 Particle Generation

By means of controlled stochastic processes, particles are generated into the particle system. One such process determines the number of particles that are created during each interval of time. The number of particles created strongly influences the density of the particle system, thus it must be controlled carefully.

Two methods are proposed for controlling the number of particles being created. First the designer can control the mean number of particles created at a frame and also a variance. The actual number of particles created per frame is

$$NParts_f = MeanParts_f + Rand() * VarParts_f,$$

where *Rand* is a procedure returning a uniformly distributed random number between -1.0 and $+1.0$, $MeanParts_f$ the mean number of particles, and $VarParts_f$ the variance.

The second method, the number of particles created during a time step can be controlled by the amount of screen area the particle system takes up. The mean number of particles created per unit of screen area and its variance can be controlled by the designer. The particle system can determine the view parameters at a particular frame and calculate the approximate screen area it covers. The equation for such a control method is

$$NParts_f = (MeanParts_{saf} + Rand() * VarParts_{saf}) * ScreenArea,$$

where $MeanParts_{saf}$ is the mean per screen area, $VarParts_{saf}$ its variance, and $ScreenArea$ the particle system's screen area. Reeves described this as a level of detail method for the particle system, the further away from the view the particle system was the less particles it created.

Another stochastic process for controlling the particle system, is to vary over time the mean number of particles created per frame. This allows the particle system to increase and decrease in intensity over time, just like fire and clouds would increase and decrease in their intensity over time. To do this the particle system uses a simple linear function

$$MeanParts_f = InitialMeanParts + DeltaMeanPartsx(f - f_0)$$

where f is the current frame, f_0 the first frame during which the particle system is alive, $InitialMeanParts$ the mean number of particles at this first frame, and $DeltaMeanParts$ its rate of change.

2.3.2 Particle Attributes

Whenever a new particle is created, the particle system has to determine values for the following attributes; (1) initial position, (2) initial velocity (speed and direction), (3) initial size, (4) initial color, (5) initial transparency, (6) shape, and (7) lifetime. The initial position of each particle is determined by a *generation shape*, in which particles are emitted from a region of space around the origin of the particle system. Reeves implemented several different generation shapes, a sphere of radius r, a circle of radius r, and rectangle of length l and width w. Figure 2.7 shows a typical spherical generation shape.

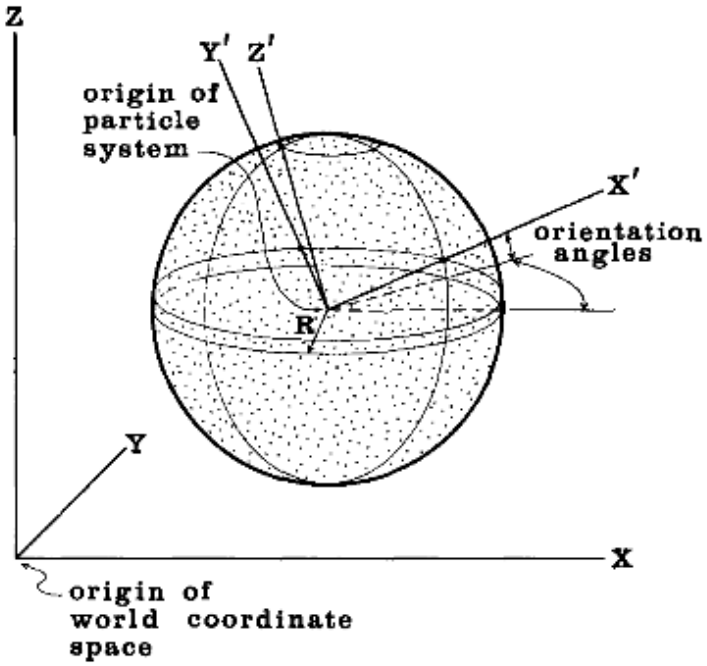

Figure 2.7: Typical spherical generation shape [24].

The generation shape also determined the initial direction of a particle, sphere being outward from the origin, and for the circle and rectangle the direction would be upward and away from the surface of the generation shape. Particles were allowed to deviate from the upward and away direction of the circle and rectangle by a small

amount. The remaining attributes such as shape, size, color, etc, can be determined by built in control parameters.

2.3.3 Particle Dynamics

Each particle within a particle system moves in a three-dimensional space and has other attributes changing over time. To update the particles' position the particle system simply adds the velocity vector to the position vector, obtaining the new position of the particle. An acceleration factor to modify the particles' velocity can be added to add a bit more realism and complexity to the particle system. A particular particle's color can change over time and changes according to a rate-of-color-change parameter. Other attributes such as transparency and size of particles could also be controlled with similar rate-of-change parameters.

2.3.4 Particle Extinction

When a particle is created it is given a maximum lifetime, measured in frames, in which it is allowed to exist. During each frame computation, the lifetime is decremented and once the lifetime reaches zero the particle dies.

Other methods were available to kill particles when they no longer contribute anything to the image. One method was if the intensity of a particular particle drops below a threshold the particle was killed. The other methods being that if a particle that moves more than a certain distance in a given direction from the particle system's origin of its parent the particle was also considered for termination.

2.3.5 Particle Rendering

The rendering algorithm renders the particles once their position and appearance parameters are determined. Reeves concluded that rendering fuzzy objects was just as difficult to render as regular surface based objects. Primitives of fuzzy objects could be obscured by other objects, could be transparent, and they can cast shadows on other objects, not to mention that they can coexist in a scene with regular objects.

Two assumptions where made to allow for the simplification of the rendering of the particles. The first assumption was that particles do not intersect with other surface-based objects, thus the rendering algorithm need only handle particles. The other assumption was that particles where considered to be point light sources; therefore, determining hidden surfaces was no longer a problem, each particle adds a bit of light to the pixels that it covers. With this algorithm and assumption, no visibility sorting of the particles was needed.

2.4 GPU offloading

In recent years, graphics vendors have replaced areas of fixed functionality with areas of programmability. Two such areas of programmability are the vertex and fragment processors. These two processors handle a magnitude of fixed functionality functions, but also allow for custom programs to run on them. However, if a custom program (shader) is run on one of the two processors, the fixed functionality of each processor is disabled. Several programming languages where created to aid in the development of shaders, one such language is the OpenGL Shading Language(GLSL) [5]. GLSL is a C/C++ like programming language that was developed by the OpenGL development team, and will be the only shading language that we will consider.

2.4.1 Vertex and Fragment Shaders

The vertex processor is a programmable unit that operates on incoming vertex values and their associated data. Some of the duties of the vertex processor are: vertex transformation, normal transformation and normalization, texture coordinate generation and transformation, and lighting. Data enters the vertex shader through special variables or textures and exits through special output variables.

The fragment processor is a programmable unit that operates on fragment values and their associated data. Some of the duties of the fragment processor are: operations on interpolated values, texture access, texture application, fog, and color sum. As with vertex shaders, data enters the fragment shader through special variables

and/or textures but exits as rendering results.

Together, both the vertex processor and the fragment processor are part of the OpenGL programmable pipeline. In Figure 2.8 you can find a diagram of how the OpenGL programmable pipeline is laid out.

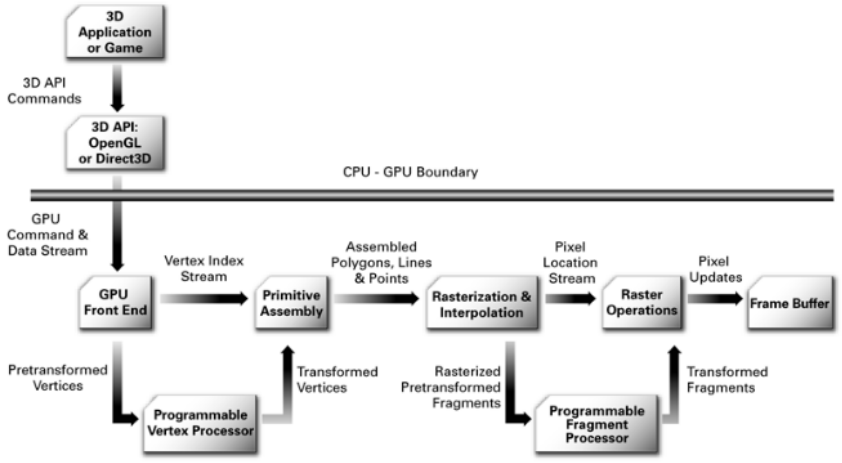

Figure 2.8: OpenGL Programmable pipeline [26].

2.4.2 Geometry Shaders and Shader Model 4

With the introduction of Shader Model 4, considering with the release of the 8800 series graphics cards from Nvidia certain enhancements were added to GLSL and OpenGL to increase the performance of GPU based program. Two such enhancements were geometry shaders and transform feedback.

Unlike its counter parts, the geometry shader has some unique features. The most important feature being the ability to create vertices on the fly. Both the vertex and fragment shader could not create vertices, all they could do is transform them. But the geometry shader has ability to create vertices, thus allowing the geometry shader to create new geometry without having to access the CPU at all.

Other enhancements to GLSL and OpenGL of interested in is transform feedback. This nice little add-on allows a shader to specify an output vertex buffer object, so

that another shader can use that same vertex buffer object as an input buffer. Thus a developer can create multi-pass shaders that do not need to relay information back to the CPU for the other passes.

2.4.3 Scientific Computation on the GPU

In the days before shaders, the GPU was used just for rendering. But with the advent programmable shaders, the GPUcan now be to aid in scientific computation. One can 'trick' the GPU into thinking that it is working on rendering information. Two methods for 'tricking' the GPU are to encode data into textures or into vertex buffer objects, or VBOs.

Texture mapping is a powerful mechanism built into OpenGL. At the time of OpenGL's birth it was nothing more than a way to paste an image on a surface; now it can be used for much more [25]. With the programmability introduced with the OpenGL Shading Language, texture mapping has become a much broader definition. Programs can now use shaders to read values from any number of textures and use them in any way that makes sense. This can run the gambit of sophisticated algorithms that use the result of one texture access, to defining parameters of another, to storing intermediate rendering results. Textures can also be used to store values for complex functions or actually encode those complex functions themselves.

Vertex buffer objects can also be used to store values for complex functions or actually encode the complex functions themselves. But unlike textures, vertex buffer objects allow data to be cached into high-performance graphics memory on the GPU [22].

Using the GPU for scientific computation can provide added realism to scientific visualization applications. This added realism can be accomplished by freeing up cycles on the CPU, thus letting more calculations to occur per time interval or increase the number of frames per second.

2.5 Related Work

In the following subsections you will find three particle systems that Sandstorm takes some inspiration from. They mostly describe different aspects of creating GPU based particle system, but also describe the general issues when developing modern particle systems.

2.5.1 Uber Flow

UberFlow is a system for real-time animation and rendering of large particle sets using GPU computation and memory objects in OpenGL [8]. The authors of Uberflow considered the use of the GPU because of the ability to offload the computation from the CPU and inherent parallelism and communication on the GPU. UberFlow's method relies on computing intermediate results on the GPU, saving these results in graphics memory, and then using them again as input into the geometry units to render images in the frame buffer. These intermediate results are stored in a texture, in which particle position, lifetime, and velocities are stored. During each time step UberFlow conducts the following events: emission, collisionless motion of particles, sorting particles, pairing of collision partners, collision response, enforcement of boundary conditions and particle rendering.

During emission, a 2D RGBA texture map containing the particles position and lifetime along with a second texture containing velocities are compared and updated. These textures are passed on to the next pass of the particle system. In this next pass, collisionless particle motion is performed with each particle's displacement updated according to the time interval from the last time. The particles are then sorted either by resolving inter-particle collisions and rendering particles as opaque primitives or it sorts particles according to their distance to the viewer and renders the particles as semi-transparent [8]. After that, the position texture is scanned for any collisions, and the output of this pass is fed into the collision response pass, where the colliding particles positions' are updated accordingly. After particle-particle collisions are

calculated, the shader collides with static parts of the scene by creating a 3D texture map of the scene and comparing it with the position texture map of the particles. Once all of this is done the particles are then rendered by looking up their position in the position texture and using those positions as input to a typical rendering process.

With these techniques UberFlow was capable of rendering a quarter of a million particles with about ten frames per second. Keep in mind that this paper was written in 2004. Since then graphics hardware have made leaps and bounds, and the techniques used in UberFlow can be adapted for use in Sandstorm. An example of UberFlow's capability is shown in Figure 2.9.

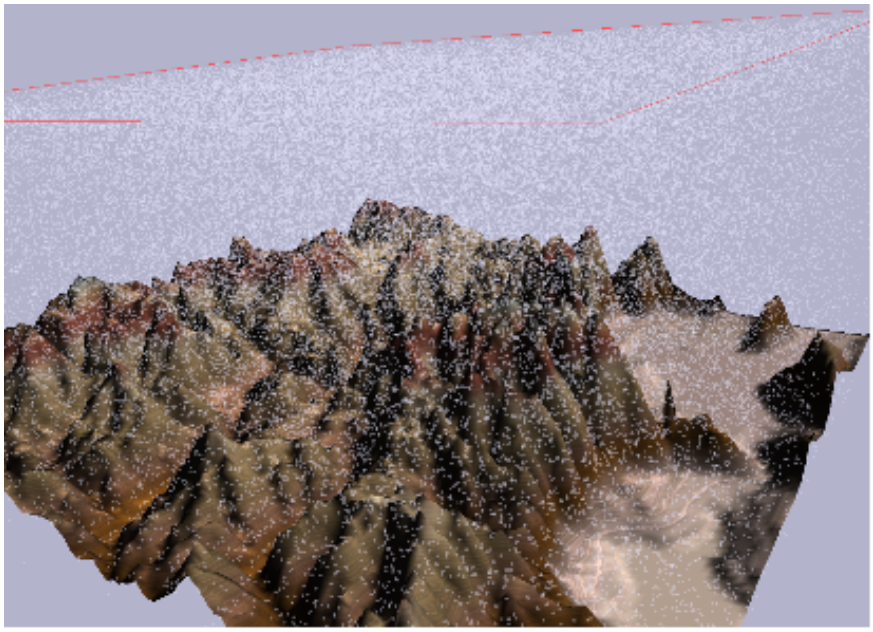

Figure 2.9: An example of UberFlow's capability.

2.5.2 Building a Million Particle System

In 2004, Lutz Latta wrote a paper entitled "Building a Million Particle System", in which Latta describes an implementation of a particle system on the GPU [10]. Latta believed that such a particle system would be capable of rendering up to 1 million particles.

Real-time particle systems are often limited by two factors, the fill rate of the particular GPU they run on, and the CPU to GPU communication. The number of pixels a GPU can draw each frame, or the fill rate, is a limiting factor when the size of the particles is large. Considering that realism of a particle system increases when small particles are used, the fill rate limitation loses importance [10]. The second limitation, CPU to GPU data transfer bandwidth, now dominates many particle systems. Because the particle system has to share the graphics bus with many other rendering tasks, most CPU based particle systems can only achieve up to 10,000 particles per frame [10]. Therefore, Latta believed that it was necessary to minimize the amount of communication of particle data across the graphics bus.

In a particle system, the most important attributes of a particle are its position and velocity. In order to create a particle system that uses the GPU, these attributes have to be stored somewhere where a shader will be able to access them. Textures were used to accomplish this goal in this particular particle system. Two pairs of textures, each pair containing one texture holding the position and the other holding the velocity of each particle, were used to allow shader access to the position and velocity information. Two pairs were used due to the fact that a texture cannot be used as both an output and input at the same time, thus creating a double buffering technique to compute new data, see Figure 2.10.

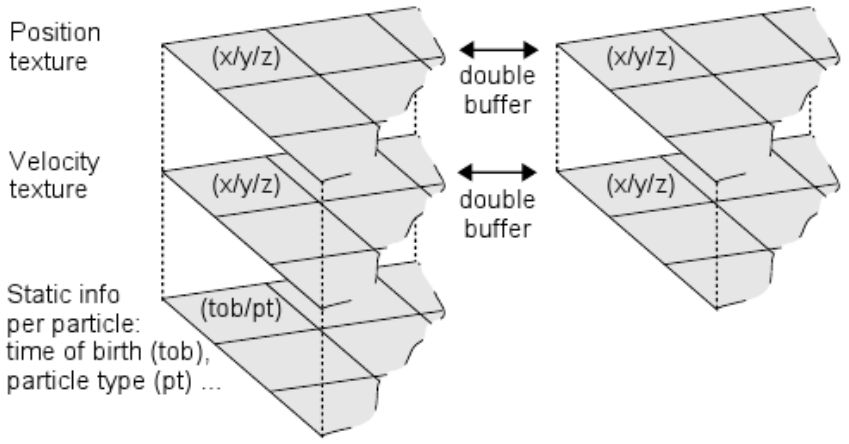

Figure 2.10: An example of how Latta stores particle information.

The simulation and rendering of the particles in the particle system were split into 6 basic steps: (1) Process birth and death, (2) Update velocities, (3) Update positions, (4) Sort for alpha blending(optional), (5) Transfer texture data to vertex data, and (6) Render particles [10].

In this particular particle system, particles could either exist forever or were given a limited time to live. The creation of new particles in the system requires the association of new data in an available index in the two information textures. Being serial by nature, allocation could not be done efficiently on a massively parallel device such as the GPU. Therefore, the allocation of a new particle was handled on the CPU, where an available index to the textures was determined and passed to the GPU. Particle death was handled both on the GPU and the CPU, when a particle dies the GPU would stop updating and rendering, and the CPU would add the index of the dead particle back to the group of indices available for allocation.

The next step was to update the velocities of the particles. A fragment shader was used for this step. The shader was given access to the velocity texture by rendering that texture to a screen-sized quad. The texture used would be one of the double-buffered velocity textures, while the other texture was set as the render target for the shader and would hold the new velocities. The shader would update the velocities by first applying any global forces, such as gravity and wind to the particles' velocities. Once that was done any local forces were applied to the velocities. Such forces include magnet attraction and repulsion. Then collision detection was perform, and the shader would determine the next position of the particle and correct its velocity accordingly.

Once the particles' velocities were updated, it was time for the particles' position. A similar method was used to bind the texture for use in the shader as was used in the updating of the velocities. A simple Euler integration was used to update the particles' positions. That Euler integration was,

$$p = p' + v * delta_t$$

where p is current position and p' is the previous position, v is the current velocity, and $delta_t$ is the time between frames.

Sorting for alpha blending (optional) was used to sort particles based on distance so that the particles in the front would be blended correctly. A simple "odd-even merge sort" algorithm was used by the GPU to allow for the exploitation of the GPU's parallelism. Also the sort had a constant number of iterations for a data set of a given size.

After the positions of the particles were determined it was time to transfer the texture data to vertex data to allow for the rendering of the particles. "Uber-buffers", also called super buffers, which basically are a data-agnostic storage of vertex or fragment data in a buffer, were used [10]. These uber-buffers allow for the accelerated asynchronous copying of data inside the GPU memory. Considering that particles can be rendered as either point sprites, triangles, or quads, the uber-buffers had to replicate the corresponding number of vertices. To avoid this overhead, the particle system used point sprites [10].

Once the particles were transformed from texture memory to vertex memory they were ready for rendering to the frame buffer. During rendering other attributes, such as color and size, were computed inside the vertex shader. As mentioned before, the particles were rendered as point sprites to reduce the workload on the vertex shader.

2.5.3 Particles

This next particle system is a bit unusual since it does not have a paper associated to it and instead may be found online [11]. This particle system, found on Microsoft's MSDN2 website, uses the latest and greatest in graphics technology to out perform its older counter parts. Also this particle system was written in DirectX10, rather than a free shading language like GLSL or Cg.

With the advent of geometry shaders, the ability to output arbitrary amounts of vertices has allowed GPU particle systems to go one step further in becoming CPU independent. Instead of visualizing particles as point sprites, the geometry shader can

now be used to create a textured billboard to be used as particles, and this particle system does just that.

Also, this sample particle system uses the stream out capabilities of the geometry shaders to store results of particle calculations into an output buffer. The output buffer is then used in a second pass and fed into the rendering shader, where the position of the particles are used to render the appropriate particle.

This particle system was studied because it described a more efficient method for the creation of particles as compared with the two previous particle systems. In this particle system, a geometry shader controls the creation and destruction of particles by either outputting new geometry, which is encoded with a particles position and velocity, or avoiding writing existing geometry. This method allows the geometry shader to allocate particle's to a vertex buffer object, which in turn can increase and decrease in size according to the amount of particles that are alive. This eliminates the need for a CPU based algorithm for the allocation of particles to a fixed size texture.

Chapter 3

Sandstorm the Concept

Sandstorm is a Dynamic Multi-contextual GPU-based Particle System that uses vector fields for particle propagation. Sandstorm is designed to use gathered scientific data, vector fields, and use it to 'guide' the motion of the particles according to this observed data. Also Sandstorm is designed to run in a 3D virtual reality environment with the presence of other applications and simulations. Therefore the performance of Sandstorm was an issue, thus a GPU-based approach was used to try and free the CPU for use elsewhere.

First off, lets take a look at what is meant by dynamic. Sandstorm is designed to have the ability to change certain attributes on the fly. Some of these attributes include, the rate at which particles were emitted, the size of the particles, the area at which the particles are emitted, the vector field that propagates the particles, and the lifetime of the particles. This ability was added to aid in the real-time scalability of Sandstorm. As less and less resources become available to the system, Sandstorm can be scaled back to allow more resources for more important tasks. This ability also brings with it an inherit level of detail, allowing Sandstorm to decrease the number of particles being emitted when the viewer is far away from the particle system.

Next, lets take a look at why Sandstorm is a multi-contextual particle system. As previously stated in Chapter 2, the typical architecture for a 3D virtual reality application/simulation is that there is a main context that handles such tasks as input gathering, updating simulations, and multiple rendering contexts that handle the rendering of a scene. Considering that there are several rendering contexts, the

generation of such information as random numbers, for use in rendering algorithms, needs to be context safe. Therefore a split first approach, described in Chapter 2, was adopted for use in Sandstorm. What is meant by this is that each context is given a copy of the entire simulation to run on it, but not everything was given to all contexts. The main context still handles context safe calculations, such as random numbers, and passes them to all the rendering contexts. This split first approach was adopted to try and take advantage of the massively parallel streaming model that are today's GPUs.

Sandstorm is designed to leverage the use of today's powerful and advanced GPUs. To take full advantage of todays GPUs, Sandstorm is designed to use Shader Model 4, in particular geometry shaders and transform feedback. By using these two aspects of Shader Model 4, Sandstorm is capable of isolating all of the particle simulation and render step to the GPU. In fact only two aspects of the particle system remain on the CPU, the passing of per frame data(such as delta time) and the updating of the vector fields. The Updating of the vector fields was left on the CPU, because the vector fields are implemented in 3D textures and to update an entire 3D texture on the GPU would put to much work on the GPU.

Finally, Sandstorm is designed to use vector fields, filled with gathered and observed scientific data to propagate the particles. This was done so that Sandstorm can mimic such scientific phenomenon as the dust cloud that is kicked up by a helicopter's downdraft.

The actual implementation for all of these attributes of Sandstorm will be described in the chapter to follow.

Chapter 4

Sandstorm Software Specification and Design

Chapter 2 presented the background and problems with creating such a particle system as Sandstorm. Also Chapter 3 presented what Sandstorm is trying to accomplish. This chapter presents a formal description of the goals, requirements, and use cases of Sandstorm. The goals will describe four major attributes that the Sandstorm prototype shall have. The requirements are presented in two categories: functional, and non-functional requirements. These describe the basics features of the Sandstorm prototype. The use cases describe the use and basic functionality of the Sandstorm. Later in the chapter, you'll find a more detailed discussion of the overall system and subsystems of Sandstorm. The software engineering practices and principles used to develop Sandstorm can be found in [1], [2], and [29]. Providing the reader with an understanding of the structure and methodology behind Sandstorm is the aim of this chapter.

4.1 Sandstorm Goals

First off, Sandstorm is meant to be a tool for the visualization of atmospheric phenomenon for scientific study, such as dust clouds and smoke. Thus, Sandstorm must allow for the input of observed scientific data to fuel the visualization of these phenomenon. For instance, the use of vector fields to guide the trajectory of the particles in an atmospheric phenomenon visualization is one example of allowing for the input

of observed scientific data. Therefore, Sandstorm uses vector fields to allow for the input of observed scientific data to fuel the visualization of atmospheric phenomenon.

According to other particle systems [8] and [10], the higher amount and smaller size of particles add to the realism of a particle system. Traditional CPU-based particle systems are stuck at around 100,000 particles [10], and are limited by the transfer of particle rendering information across the graphics bus. Newer GPU-based particle systems [8], [10], and [11], allow for a much higher amount of particles to be displayed, around 500,000. Therefore, Sandstorm uses a GPU-based approach to allow for a high amount of smaller sized particles, which adds a higher degree of realism.

Allowing the user to immerse themselves into the scientific phenomenon that they are studying may allow for a better understanding of the phenomenon. Using a 3D virtual reality environment will allow a user to immerse themselves into a particular application. Therefore, Sandstorm will utilize a 3D virtual reality environment to allow for the immersion of the user into a scientific phenomenon.

Like previously stated, Sandstorm is meant to be a tool for the visualization of atmospheric phenomenon for scientific study, such as dust clouds and smoke. Thus, Sandstorm must have the ability to allow the application programmer to customize the particle system. Such customization could include, whether or not to display the particles as points or textured billboards, the size of particles, and the maximum life time of the particles. Therefore, Sandstorm allows for a horde of attributes to be changed both programmatically and during run-time, for a complete list see Table 5.1 in Chapter 5.

4.2 System Requirements

Like all software systems, Sandstorm has minimum system requirements that must be met in order for the system to perform properly. This section will describe the minimum system requirements of Sandstorm and describe some of the specifications of the system that Sandstorm was tested and developed on.

The first requirement is the most apparent, and that is the virtual reality environment. Sandstorm was developed for use in an active stereo CAVE system. The CAVE system that Sandstorm was tested on was at the Desert Research Institute. This CAVE system was an active stereo, four-wall flex CAVE system designed by FakeSpace [3]. The system utilizes a multi-cored machine and a Nvidia quadroplex to power it [19].

Like previously stated throughout this paper, Sandstorm utilizes a GPU-based approach. However, this GPU-based approach is not your run-of-the-mill approach, it utilizes Shader Model 4, more specifically geometry shaders and transform feedback. The system that Sandstorm was developed and executed on, uses a Nvidia quadroplex with a G80 chipset in it, thus a system that is going to run Sandstorm will also have to have a G80 chipset. The video cards that have this chipset are any Nvidia 8000 series cards [18]. Currently Radeon cards don't support transform feedback. However, current Radeon cards, such as the Radeon HD 3800, support vertex stream out, which is DirectX 10's version of transform feedback [12].

4.3 Software Specifications

Before the use case and class diagrams can be presented, the requirements of Sandstorm must first be outlined (Table 4.1). These requirements were distilled through domain knowledge and the requirements of the Helicopter and Dust Simulation application described in Chapter 2.

Next the non-functional requirements (Table 4.2) show what technologies are used in Sandstorm and software features. These non-functional requirements largely control the software engineering of Sandstorm and its subsystems.

4.4 Use Cases

The use cases in Figure 4.1, depicts the functionality provided to the end user. These use cases were developed using the functional requirements presented above. There

F01	Sandstorm shall display a visual representation of phenomena such as dust clouds, i.e. particle system.
F02	Sandstorm shall use vector fields to guide particles.
F03	Sandstorm shall utilize GPU-offloading.
F04	Sandstorm shall be written to work in a 3D virtual reality environment.
F05	Sandstorm shall allow users to change the maximum lifetime of the particles.
F06	Sandstorm shall allow users to change the size of the particles.
F07	Sandstorm shall allow users to change the area in which the particles are emitted.
F08	Sandstorm shall allow users to change the rate at which particles are emitted.
F09	Sandstorm shall allow users to change the maximum number of particles that are allowed to be emitted.
F10	Sandstorm shall allow users to start/stop and pause the particle simulation.
F11	Sandstorm shall allow users to change the attributes both programmatically and during run time.
F12	Sandstorm shall allow for a large number of particles to be displayed (above 100,000).

Table 4.1: Functional requirements for the Sandstorm prototype application.

are two actors that directly influence the Sandstorm particle system. The user manipulates Sandstorm by changing the dynamic attributes of the systems. Time is also a major actor on Sandstorm, it controls the update of the simulation, and animates the particles.

4.5 Modeling and Design

This section describes the structure and design of the Sandstorm particle system. This includes a Main Class Hierarchy diagram, which includes a description of the classes including a description of the Utility Classes that were used.

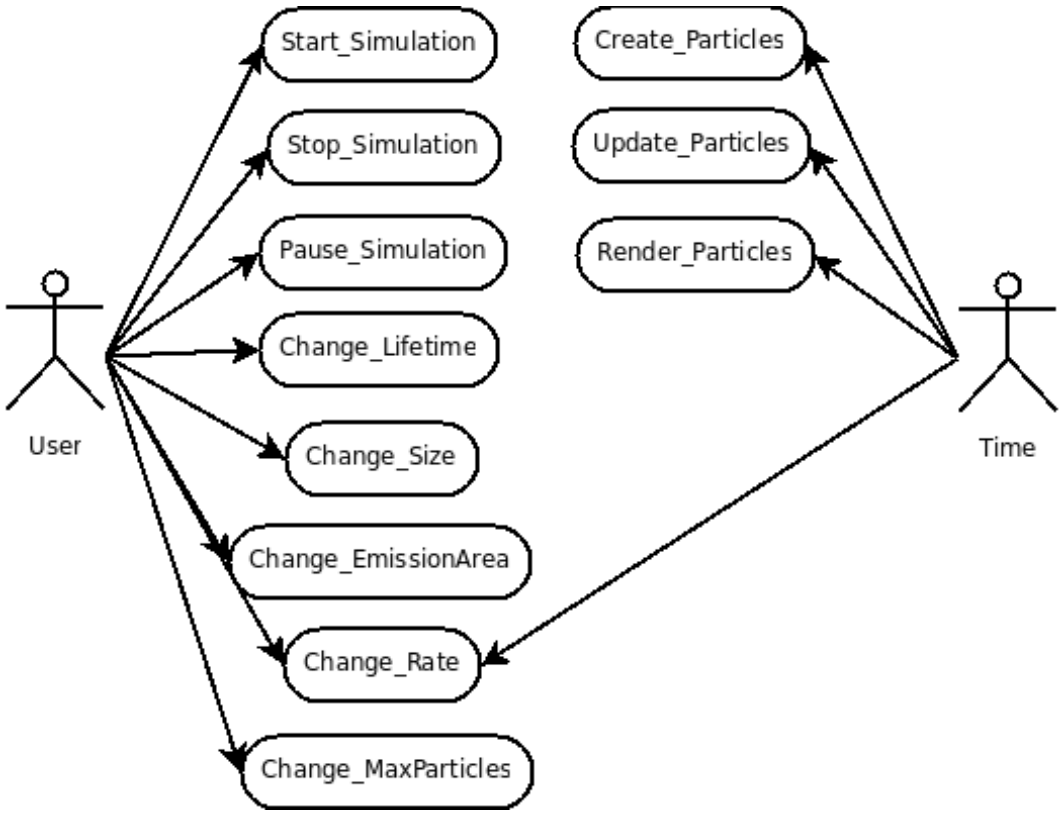

Figure 4.1: Use cases for the Sandstorm prototype.

N01	Sandstorm's CPU based portion shall be written in C++.
N02	Sandstorm's GPU based portion shall be written in GLSL.
N03	Sandstorm shall be implemented in a modular fashion allowing for the replacement of components when more efficient methods are discovered.
N04	Sandstorm shall maintain independence from windowing/VR toolkit.
N05	Sandstorm shall use the VRUI library for prototype application.
N06	Sandstorm shall maintain interactive levels of temporal fidelity, above 15 FPS.
N07	Sandstorm shall use geometry shaders for the creation and updating of particles.
N08	Sandstorm shall use geometry shaders for the rendering of particles.
N09	Sandstorm shall use transform feedback to transmit the particle data to the update/creation shader(s) to the render shader(s).

Table 4.2: Non-functional requirements for the Sandstorm prototype application.

4.5.1 Main Class Hierarchy

In Figure 4.2 you can find the main class hierarchy of the Sandstorm particle system. This is not an inheritance hierarchy, rather a flow of control between the classes. Also there are other classes that were used in the creation of Sandstorm and they will be covered in the following section.

Multicontextual ParticleSystem

This class is the main control mechanism for the entire particle systems, from controlling the flow of the simulation to the flow of data. This class contains a list of ParticleSystem class, each ParticleSystem class is responsible for a context. The list containing ParticleSystem is updated with data such as time between frames and changes in dynamic attributes.

In a multicontextual environment the generation of random numbers needs to be done on a single context to ensure that all contexts get the same random numbers. During initialization, the Multicontextual ParticleSystem is responsible for generating any random numbers that are needed for the per context simulations.

ParticleSystem

Each context contains a ParticleSystem class, which is responsible for the cre-

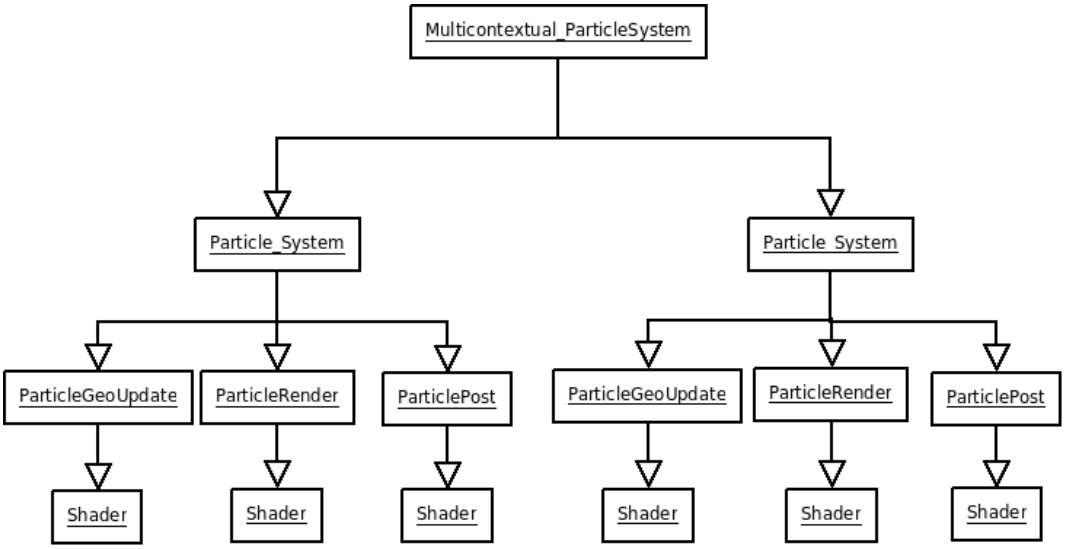

Figure 4.2: The main system structure of the Sandstorm system.

ation and coordination of the Shader objects. During creation, this class creates and initializes the shader objects with data that they need, such as pointers to render buffers and particle information buffers. First this class creates and initializes the ParticleGeoUpdate shader, which is responsible for the particle information buffers. Once the ParticleGeoUpdate shader object is initialized, the ParticleSystem initializes the ParticleRender shader object with a pointer to the particle position buffer initialized by the ParticleGeoUpdate. Finally, when the ParticleRender shader object is initialized the ParticlePost shader object is initialized with the output buffer of the ParticleRender shader object.

During the simulation the ParticleSystem class will call the ParticleGeoUpdate class to update the particle information. Once the particles have been updated, the ParticleSystem class will first call the ParticleRender shader to render the particles into an off screen buffer. After the ParticleRender shader object is called, the ParticlePost shader object is called to render the result of the deferred shading. Also while the simulation is running the ParticleSystem class directs the flow of the data to the correct shader objects, such as change in the size of the particles or the maximum life time of the particles.

ParticleGeoUpdate

In every context there is a ParticleGeoUpdate class, which is responsible for updating and creation of the particles. During initialization the ParticleGeoUpdate class creates and initializes all of the particle information buffers, such as the particle positions and particle velocity buffers. The ParticleGeoUpdate class will create two sets of these buffers so that the data is double buffered.

During the simulation, the ParticleGeoUpdate updates the particle information, which includes the particle positions, velocities, and time lived. This is done by storing the particle information into VBOs, and feeding those VBOs into a shader that will update and create the particles. Once the shader has updated and created the particles, the ParticleGeoUpdate class gets the result, using transform feedback, and swaps the two set of buffers.

ParticleRender

The ParticleRender class is responsible for the rendering of the particles into an off screen buffer for use in the ParticlePost class. During initialization the ParticleRender class receives from the ParticleSystem class, a pointer to the two particle positions buffers. Also, the ParticleRender class receives from the ParticleSystem class a pointer to the off screen buffer that it is to render to.

During the simulation the ParticleRender class uses the particle position buffer pointer to feed the render shader the particle position information. But before the ParticleRender tells the render shader to render the particles, it redirects the output of the render shader into the off screen buffer. Once the render shader has rendered the particles, the ParticleRender class swaps the pointer of the particle position buffer, so that it matches the correct one that the ParticleGeoUpdate will use next round.

ParticlePost

Once the ParticleRender class is done rendering the particles into an off screen buffer, the ParticleSystem class calls the ParticlePost class. The ParticlePost class is responsable for the deferred shading rffect described in Chapter 5. During initialization the ParticlePost class receives from the ParticleSystem class, a pointer to an off

screen buffer, which holds the results of the ParticleRender class. During the simulation, the ParticlePost class uses the off screen buffer to feed the deferred shading shader.

Shader

The Shader class is a small utility class that handles the creation, compilation, and linking of all shader code. Each of the three shader object classes ParticleGeoUpdate, ParticlRender, and ParticlePost inherit from this class. During the initialization of this class, the shader source code is located and loaded, if the shader source code cannot be located or loaded an error is returned. Once the shader source code is loaded, it is then complied and linked. If any errors occurred in the compiling or linking of the code those errors are printed. Once the shader source has been compiled and linked successfully, an identification number is stored into the Shader class. It is up to the class that inherits from the Shader class, to implement the two virtual functions that are present in this class. The first virtual function that needs to be implemented is the initialize uniforms function, this function finds and stores the location of any input variables to the shader. The second virtual function that needs to be implemented is the draw function, which is responsible for setting up the proper OpenGL state and activating the shader. Each of the Shader object classes (ParticleGeoUpdate, ParticleRender, and ParticlePost), all implement these two functions and store the location of the shader input variables.

4.5.2 Utility Classes

The class/objects/functions listed in the previous section were not the only ones used in this project, below is a brief description of some of the other pieces of Sandstorm. First the GFX Utility Library, created by Roger Hoang, is a small multi-contextual graphics library. And second, Vector Field File Loader, is a small object orientation system that loads vector data from a file for use in Sandstorm.

GFX Utility Library

GFX Utility Library, created by Roger Hoang, is a small multi-contextual graph-

ics library, that handles such things as multi-contextual textures and framebuffers and OpenGL state-saving and restoring code.

The GFX texture and framebuffer classes are multi-contextual texture and framebuffer classes that are used in Sandstorm for ensuring that any texture/framebuffer data is multi-context safe. This is done by taking the same approach that Sandstorm does to multi-contextualism, and that is to create a copy of the data for each context. The multi-contextual classes each contain a list that holds pre-context copy. When a new context is added, an object(texture/framebuffer) is created and initialized with the proper data, then it is added to the context list of the class.

Other functions of the GFX Library that are used in Sandstorm are the popAll and pushAll functions, which are nothing more than wrappers for OpenGL state saving and restoring code.

Vector Field File Loader

The Vector Field File Loader(VFFL), is a simple class that handles the loading of vector field data and translates the data into a form that can be used by Sandstorm. First off, the VFFL loads the file data into a three dimensional array for simplicity of loading different file formats. Sandstorm uses 3D-textures to transmit the vector field data to the GPU order to use the vector field data along with any texture the data needs to be in a single dimensional array. Once the data is loading from the file into the three dimensional array, the VFFL translates that data into a single dimensional array so that it can be used in the vector field texture.

Chapter 5

Sandstorm Prototype

In this chapter you will find a description of the inner workings of the Sandstorm Prototype. First, a description of the GPU-Based particle system will be described, then the steps to combat the multi-contextual environment. Next the use of vector fields will be discussed both on how they are used to update the particles and how the vector fields are updated themselves. Finally, a description of the dynamic nature of Sandstorm and a listing of all the attributes that are able to be dynamically changed is presented.

5.1 GPU-Based

This thesis has looked at a hand full of particle systems, some of which are GPU-based particle systems. This section describes the GPU-based particle system of Sandstorm. Like most particle systems, Sandstorm's particle system has three main phases: (1) Creation and Destruction phase, (2) Update phase, and (3) Rendering phase. The following sections describe these three main phases.

5.1.1 Creating and Destroying Particles

The most important part of a particle system is its ability to introduce new particles into the system. Without it the user would be staring at a blank screen. On the other hand, destroying particles is also an important part of a particle system. Without this ability the particle system would not be able to introduce new particles when

the system has reached its maximum. Considering that the creation and destruction of particles is handled on the GPU the problem has to be revisited. According to Reeves [24], and others [8, 10], there are three steps: (1) particles are born, (2) each new particle is assigned individual attributes, and (3) particles that are too old are destroyed.

Traditional particle systems would create particles and store them into a dynamically growing data structure, such as a linked list or an array. But with GPU-based particle systems particles can't be stored in such data structures, as is the case with Uber Flow [8] and Latta's Million Particle System [10]. In both cases textures were used to store particles and their associated data, this allowed the isolation of particles and particle data to the GPU, freeing space on the CPU. However, textures aren't the most efficient mechanism for relaying information to the GPU. Now VBOs can be used to relay information to the GPU more efficiently. Like previously stated, a VBO stores data on the high-performance graphics memory of the GPU [22]. Also VBOs can be resized to fit the exact amount of data that is used, whereas, a texture needs to describe a rectangular area that encompasses the entire area of the amount of data. For example if a data set had 19 members then a texture would have to cover an area of 20, wasting more memory than is actually needed. This is not the case with a VBO, which would just need to be of size 19. This property can allow the VBO to dynamically increase and decrease in size to accommodate large amounts of particle information. Because of VBOs' memory efficiency, Sandstorm uses them to store particle information on the GPU.

Considering that the GPU is a set streaming processors, a second look needs to be taken at how the particle information is stored. Since we never know just how many processors are reading and/or writing to a particular location in the VBO, a double buffered approach needs to be adopated. Like Latta's Million Particle System [10], Sandstorm uses a double buffered approach to store particle information on the GPU. One of the buffers is used as a read buffer and the other buffer is used as the write buffer. At the end of every cycle these buffers are swapped. Also like Latta's Million

Particle System [10] and Uber Flow [8], each of the buffers hold two VBOs. One VBO holds the particle's position in 3D space, which is stored in the first 3 members of the vertex, the fourth member of the vertex contains the lifetime of the particle. The other VBO holds the particle's velocity components, which are also stored in the first 3 members of the vertex leaving the fourth member free for use elsewhere, see Figure 5.1.

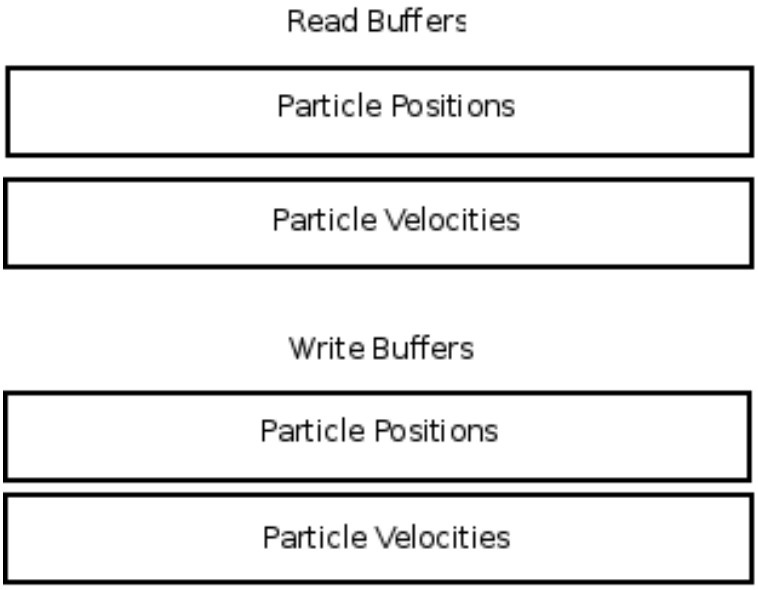

Figure 5.1: An example how Sandstorm store particle information.

Most modern particle systems have two entities associated to them, one is the particle and the other is the emitter. Considering we know what a particle is, its time to describe what a emitter is and what it is used for. An emitter is an entity that is responsible for the emission of particles, which usually has a position in space or an area for which it is responsible for emitting particles to. Emitters are entities that are not usually rendered, they are merely abstract objects responsible for the introduction of new particles into the system. On that note, in Sandstorm an emitter is stored in the same buffer as the particles. In the case of an emitter, the members of position vertex buffer holds random numbers to aid in the seeding of initial values. The case is the same for the velocity vertex buffer, but instead the fourth member

is set to one to denote that the emitter is indeed an emitter. Therefore, the fourth member of the particle's velocity vertex buffer is set to zero to denote a particle.

Now that we know how the particle information is store on the card, it's time to take a look at how the particles are created. In Shader Model 4, the geometry shader can emit one or more vertices up to a maximum amount that is hardware dependent. Keeping that in mind, geometry shaders are a logical choice for the creation and destruction of the particles.

At the beginning of the Creation and Destruction phase, the particle information is passed to the shader that is responsible for the creation and destruction of the particles, the particle shader. Once the particle shader has the particle information, it determines if the object that it receives is a particle or an emitter, by looking at the 4th value of the velocity vertex. Once the shader has determined if it is dealing with an emitter, it begins to emit particles. The first thing to determine is how many particles are to be emitted, which is done by determining how many particles have already been emitted and subtracting that from the number of particle per second. Once the difference is found, it's divided by the amount left in that cycle, each cycle is a second. This method won't always give the exact amount of particles per second, but it gives the system another measure of diversity, making the system seem more realistic.

Once the number of particles to be emitted has been determined, the particles are created but not yet emitted. Inside the aforementioned while loop, each particle is seeded with initial information that conforms to certain ranges. In order to give the particle system a more realistic feel to it this initial information is random. But creating random numbers on the GPU is currently not supported; therefore, a perturbation texture was used. This will be described in more detail later in the mutli-contextual section. Once the particle has its initial information, the emitter emits the position and velocity vertex of all the particles into the write buffer. Finally, the emitter emits itself back into the write buffer.

If the object that the particle shader is passed is a particle, then something

different happens. First, if the particle is determined to be dead the shader emits nothing freeing up its spot in the vertex buffer. If the particle is not dead, then the particle is updated which is describe in the following subsection.

5.1.2 Updating Particles

The next step in a particle system, once new particles are created and old particles are destroyed, is to update a particles' information. This section describes the Update phase.

The first thing that is done is to perform a lookup into the vector field and get the acceleration information at the position of the particle. This lookup is done by interpolating the position of the particle and dividing the members by the width, height, and depth of the vector field. Once the acceleration information is obtained the velocity of the particle is updated, then the position and the total time lived are also updated. Once the particle's information is updated it is emitted into the write buffer.

This is where Transform Feedback steps in. As previously stated, Transform Feedback allows one shader to specify an output vertex buffer object, so that another shader can use that same vertex buffer object as an input buffer. Once all of the particles have been updated and have been emitted into the write buffer, the read and write buffers are swapped and Transform Feedback feeds the read buffer to the render shader.

5.1.3 Rendering Particles

Finally, the last thing to do in the particle system is to render the particles. As previously stated, geometry shaders can emit one or more vertices to a render target. This ability can aid in the rendering of particles. When rendering particles, most particle systems render the particles as textured billboards, this gives the illusion of a volumetric cloud of dust. However, some particle systems, mostly scientific visualization systems, render particles as points in order to study the behavior of particles.

Therefore, Sandstorm has the ability to render the particles as either blended textured spherical billboards or points.

When rendering particles as textured billboards, two pieces of information must be determined by the render shader. The first and easiest is the position of the particle, which is passed to the render shader by the the particle shader through the Transform Feedback model. The second piece of information that needs to be determined is the eye coordinates, which can be obtained through the built-in variables of the shader. Once the need information is obtained, a normal centered at the particles position and points to the eye can be calculated. This normal can be found by the following equation:

$$normal = gl\_ModelViewMatrixInverse * Up\_Vector$$

where $gl\_ModelViewMatrixInverse$, is the built-in variable that describes the Inverse of the Model View Matrix, and Up_Vector is the up vector that you are using for your world, such as y-up or z-up. When that normal has been calculated the four points that make up the corners of the billboard can be calculated and the points for the billboard can be emitted to the framebuffer for rendering. However, considering that geometry shaders can not currently emit quads, two triangles that form a quad have to be emitted.

Once the billboards have been emitted they are then rendered to an off-screen buffer so that they can be run within a deferred shading shader.

Currently, Sandstorm uses a deferred shading method to blend the particles together so that the more particles there are in a scene the more dense the particle field looks. The first step is to accumulate the particles, per pixel, so that the more particles that are behind that particular pixel the denser it looks. This is done by premultiplying the particles RGB in a shader, then blend using:

$$glBlend(GL\_ONE, GL\_ONE\_MINUS\_SRC\_ALPHA)$$

which gives:

$$RGB = Src.rgb * Src.a + Dst.rgb * (1 - Src.a)$$

$$A = Src.a * ONE + Dst.rgb * (1 - Src.a)$$

Once this is done the particles are ready to be lit, this is done by taking the surface normals and doting them with the light direction. After the light is done this result can be rendered to a full screen quad with the result textured onto it [17]. One thing extra that needs to be handled because of this method is that the deferred shading shader needs to know the depth buffer of the scene you use to draw the particles onto. This would require the user of Sandstorm to render the scene to an offscreen buffer, making sure to attach a depth buffer and passing the depth buffer to the deferred shading shader.

When rendering particles as points, Sandstorm does something similar to the textured spherical billboards. Instead of rendering a particle as a small point or a small cube, a non-blended non-textured spherical billboard is rendered. This is done because it is currently computationally cheaper to calculate the normals and points and emit six vertices than it is to emit 36 vertices to make up a cube. This is done because of code reuse and code size. Simply leave out the render fragment shader and the deferred shader leads to smaller code size, rather than making a separate branch for point rendering. Therefore, the same render shader is used, except that the texturing and the blending of the billboards is replaced with a simple coloring method.

5.2 Vector Fields

The using of vector fields in Sandstorm was necessary so that the behavior particles could be controlled by gathered scientific data. This section describes how the vector fields are implemented and updated. The use of vector fields was described in the Updating Particles subsection GPU-based section of this chapter.

The vector fields that are used in Sandstorm are represented as 3D textures. Textures were used instead of VBOs, because of the existing internal methods for dealing with 3D textures, such as wrapping and indexing the texture as a 3D space.

Also considering that, currently, a vector field describes a 3D rectangular area of space, the whole texture will be used, unlike the particle information textures used in Latta's Million Particle system [10] and other particle systems. A copy of the vector field is represented as a 3D array on the CPU to conform to the split first method. Doing this allows a main object to maintain the vector field, by updating the vector field according to a simulation, such as a helicopter-desert terrain interaction model.

5.3 Multi-contextual

As previously stated, when dealing in a multi-contextual environment there are several methods that can be used, three of which are: (1) split first where the simulation is replicated on all contexts, (2) split middle where a shared data structure is transmit to all contexts, and (3) split last where the rendering calls are transmitted to all contexts. Sandstorm uses a split first method to combat the multi-contextual environmental constraints of a 3D Virtual Reality environment. This method is used for three main reasons:

- The use of Vector Fields to guide particles allows for the predictability of the behavior of particles once created.
- The use of a perturbation texture for creation of particles allows one perturbation texture to be created and passed to all simulations at initialization.
- Finally, the use of a GPU-based particle system, allowing the simulation to be distributed across multiple graphics hardware and eliminating the need for communication between GPUs.

Considering these three reasons, Sandstorm can safely use a split first method for the distribution of the simulation across multiple contexts.

Figure 5.2, shows how Sandstorm is designed to utilize the split first method. At the top of the architecture sits a main object that coordinates the initialization of all the simulation objects and controls the updating and rendering of the simulation

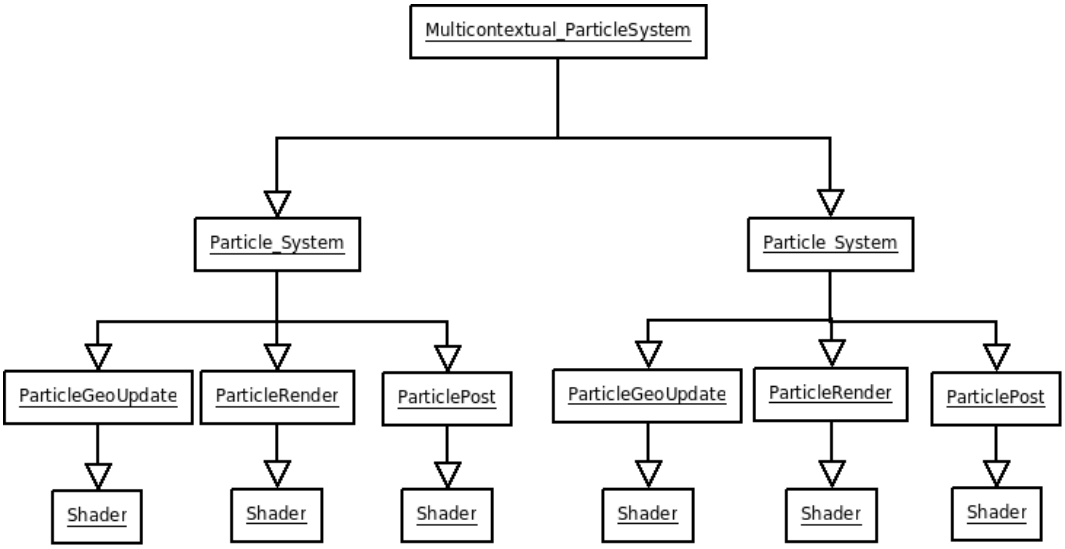

Figure 5.2: The main system structure of the Sandstorm system.

objects. This main object is in turn controlled by the main context of the 3D virtual reality environment. Each of these simulation objects contain a particle system and the same set of data, data that is given to the simulation object by the main object. The data that is given to each simulation includes the following: (1) the time lapsed between frames, (2) an updated vector field if one exists, (3) any random numbers that need to be generated, and (4) particle attributes.

5.4 Dynamic

In order to allow for a more diverse and realistic behaving particle system, certain attributes are given the ability to be changed by the user of Sandstorm. The split first method that Sandstorm's architecture is modeled after, aids in the controlling of the dynamic attributes of the particle system. The main object passes down any changes to the attributes to the simulation objects and in turn down to the GPUs that the simulation are running on. A list of Dynamic attributes can be found in Table 5.1 .

Size	This attribute determines the size of the particles when they are rendered.
Lifetime	This attribute determines how long a particle can live.
AreaWidth	This attribute determines how wide the emission area is.
AreaDepth	This attribute determines how deep the emission area is.
Maximum	This attribute determines maximum amount of particles allowed to be emitted into the system. Keep in mind that this is a soft max and that there is a predetermined maximum amount of particles.
Position	This attribute determines the positions of the emission area.

Table 5.1: Table of dynamic attributes.

Chapter 6

Results

In order to test the Sandstorm particle system, a sample application was created. The sample application was created using the Dust Framework for the creation of a scene to work with. The terrain that was used in the scene was of a desert area outside of Yuma, Arizona. The object that was used in the scene was a blackhawk helicopter, that was not accurate to real world specifications. A simple control scheme for the helicopter was used, where the wand could be tilted toward the user to increase the throttle and away from the user to decrease the throttle of the helicopter.

A basic vector field was used in this sample application. Considering that Sandstorm is not a vector field interaction model, where vector fields are calculated according to scientific models, a single vector field was used. In order to create a dynamic test of the Sandstorm particle system, a simple helicopter-vector field interaction model was created. This interaction model was nothing more than as the throttle of the helicopter was increased, the amount, life time, and rate of the particles increased. On the other side, as the throttle of the helicopter was decreased, the amount, life time, and rate of the particles was decreased. Also as the distance of the helicopter and the ground passed a predetermined point, the amount, life time, and rate of the particle, was also decreased.

The system that this test was run on has the following hardware specifications:

- A multi-cored shared memory machine, with four quad-core chips
- 48 GBs of RAM

- Running Ubuntu 7.10 Linux
- A Nvidia Quadroplex, housing a G80 chipset

This system was responsible for powering a four-walled active stereo virtual reality CAVE system.

This test was designed to test two major aspects of Sandstorm: the ability of the particles to follow a vector field, and the ability to be run in a multi-contextual environment. During the course of development of this test it was determined that VRUI could not be used as a VR library for this test on the machine that it was suppose to run on. The reason for this is that VRUI was not designed to run on a shared memory machine. Instead of creating a multi-contextual environment where general attributes can be passed to separate contexts, VRUI creates a multi-thread environment where each thread has its own set of non-shared variables. VRUI was abandoned for FreeVR, because of its ability to run on shared memory systems. Luckily, the Dust Framework had a FreeVR version.

Once Sandstorm was intergrated into the FreeVR, the sample application was run in the CAVE system described above. Figure 6.1 is a picture of Sandstorm running in a CAVE. Sandstorm was tested for three things: the ability to run in a multi-contextual environment, the ability of particles to follow a vector field, and frame rate. Sandstorm's ability to run in a multi-contextual environment and the ability for the particles to stay consistent between screens can be seen in Figure 6.1. In this figure, a yellow line separates two of the screens to which Sandstorm is displaying. As you can see, in the figures, particles are consistent between the two screens or contexts. During the test, Sandstorm was responsible for rendering and updating 300,000 particles, well above the required amount specified in Chapter 4. Sandstorm rendered and updated the particles at 15-20 FPS while standing in the middle of the particle system, and at about 65 FPS while standing a good distance back for the particle system. While the vector field that was used for this test was not one that was an actual helicopter downdraft vector field, the particles still followed the vector

field as prescribed, see Figure 6.2.

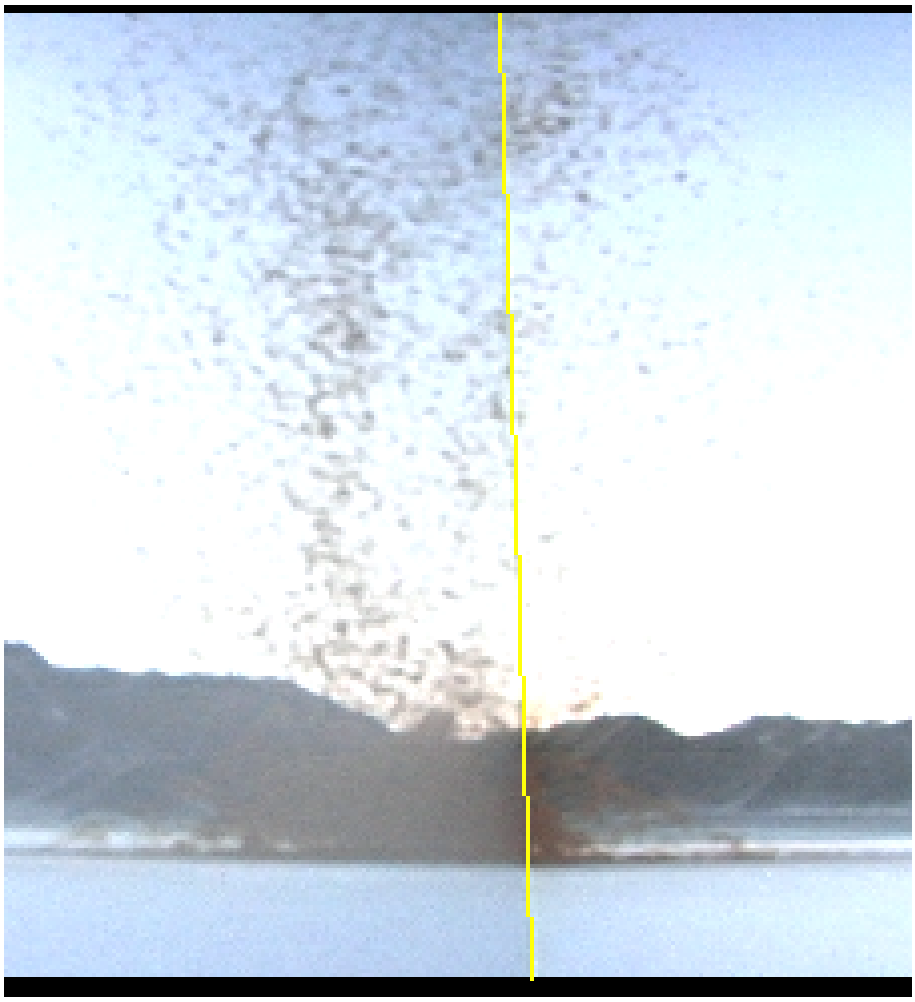

Figure 6.1: An another example of multi-contextualism.

Another example of the particles' ability to follow a vector field can be seen in Figure 6.3, which is a vector field created to simulate a tornado. At the center of the image you can use a dense funnel of particles, and as the particles increase in height they start to fall back down to the ground.

Another attribute of Sandstorm that was tested during the running of the sample application was the dynamic attributes of Sandstorm. This can be seen by the changing rate of emission and changing max life time of the particles as the throttle of a helicopter changed.

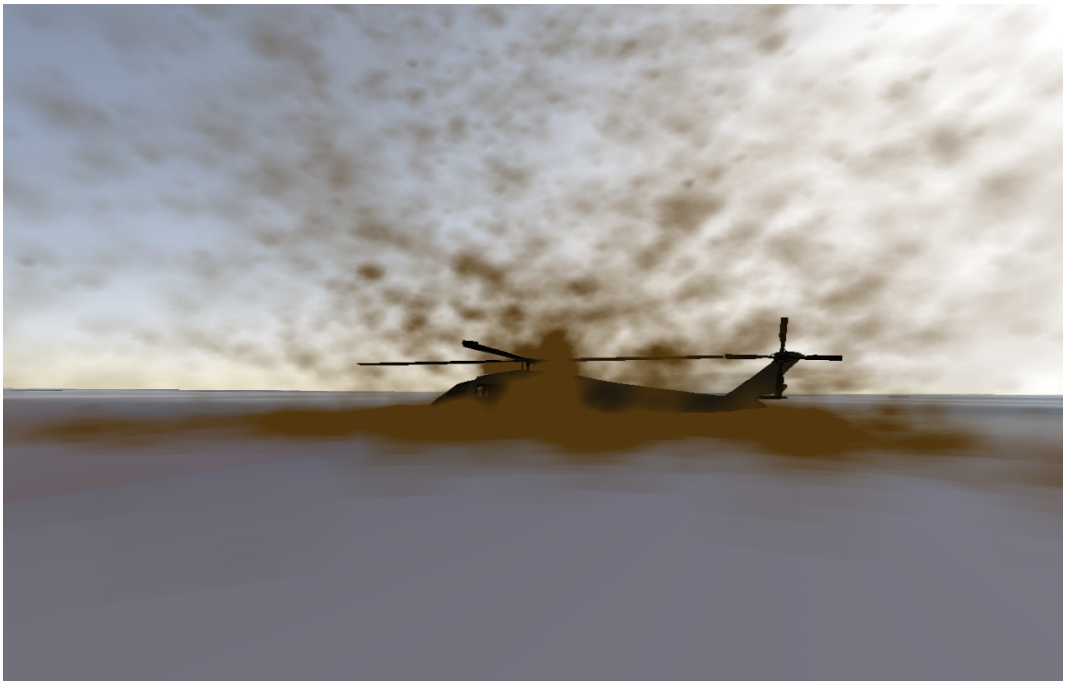

Figure 6.2: An example of particles following a vector field.

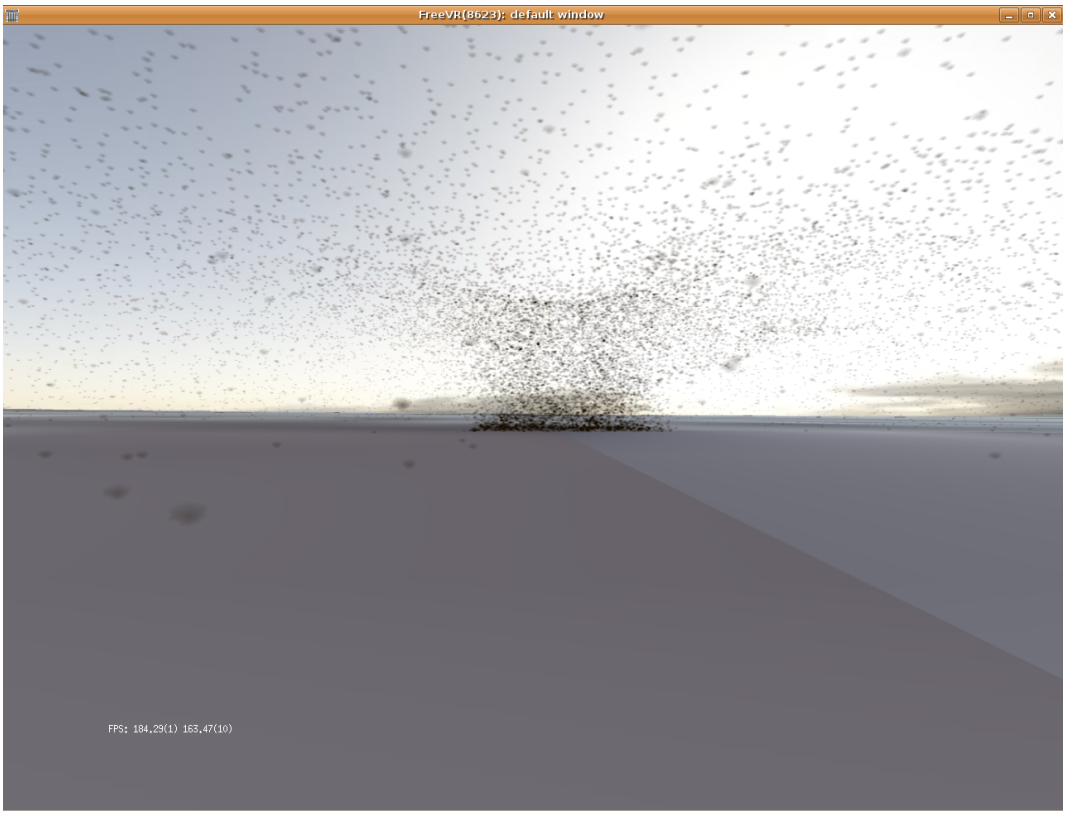

Figure 6.3: Another example of particles following a vector field.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

We have seen that Sandstorm has met all of its design goals listed in Chapter 4, the ability to use a vector field to 'guide' particles along predetermined paths. We can use this ability not only to guide particles according to a simulated helicopter downdraft model, but it can also be used to visualize many other atmospheric or visual phenomena. Such phenomenon could be, but are not limited to tornados, hurricanes, wind fields, and even fires.

We have also seen that Sandstorm has the ability to run in a multi-contextual environment. While running in the multi-contextual environment described in Chapter 6, the performance was not as great as it was on a single context. This can be attributed to the fact that the shared memory machine that Sandstorm was run on only had two graphics cards, forcing two screens to share one graphics card. The Future Work section below describes some possible areas of optimization for Sandstorm.

Other design goals that were listed in Chapter 4 and that Sandstorm has met are as follows. Sandstorm is a particle system that has the ability to create a visual representation of phenomenon such as dust clouds. Sandstorm has the ability to change attributes such as emission area, rate, lifetime, and maximum number of particles. Sandstorm utilizes GPU-offloading to update and render particles. Sandstorm can update and render above 100,000 particles in such an environment, currently above 300,00 particles.

So we have seen that Sandstorm is capable of creating such atmospheric phenomena as dust clouds, and that Sandstorm can use observed and simulated scientific data to guide the particles according to the data. We can now consider Sandstorm for use in both scientific and training applications that focus on helicopter-desert environment interaction. Such applications would help the user to understand the interaction between helicopter and desert environments. Hopefully, such understanding will decrease the number of helicopter crashes and deaths cause by the aforementioned interaction.

7.2 Future Work

Considering that Sandstorm is currently just a prototype there are many things that can be done to increase the quality of the system.

The first item to address is that the current Sandstorm has areas that can use a bit of optimization. Currently, both emitters and particles reside in the same buffer objects, causing the shaders that use these buffers to contain branching. Branching in a shader can be computationally expensive, therefore it might be better to separate emitters and particles into different buffers. Another thing about the current Sandstorm is that the size of the buffers are currently of static size and do not dynamically increase or decrease in size. Instead of using a soft maximum amount of particles to create a pseudo-dynamic size, it might be beneficial to create a dynamic sized buffer. These dynamic sized buffers will definitely increase speed when the amount of particle is small, but they might not help at all as the size of the buffers reach large numbers.

Other features that can be added to Sandstorm to increase the quality of the system are as follows:

- Collisions between particles and static objects can also be implemented. Such a static object would be the terrain in the sample application, a method for this can be found in [8].

- Collisions between dynamic objects, such as the helicopter and the particles,

could be added to Sandstorm. This would require a voxelization of dynamic objects described in [13].

- Soft particles could be added to Sandstorm to allow for more realistic looking particles [31].
- Motion Blur is another effect that could be added to make the particles look more verisimilar [14].
- Light Scattering could be used to give the illusion that particles are blocking out light [15].
- Finally, a shader based physics model could be implemented to allow users of the system to change the behavior of the particles without having to recompile.

Other improvements that can supplement Sandstorm are the creation of a vector field simulator and a vector field creator/editor. A vector field simulator could be created to feed Sandstorm dynamically changing vector fields so that the particle motion acts more natural. A vector field creator/editor can be created to help scientists visualize vector fields before they are used in Sandstorm. This application could also be developed for a 3D virtual reality environment and is also currently designed and being implemented.

Bibliography

[1] Jim Arlow and Ila Neustadt. *UML and the Unified Process: Practical Object-Oriented Analysis and Design*. Addison-Wesley, 2002.

[2] Albert Endres and Dieter Rombach. *A Handbook of Software and Systems Engineering: Empirical Observations, Laws, and Theories*. Pearson Addison-Wesley, 2003.

[3] Fakespace Systems Inc. Fakespace Systems Inc. CAVE®. `http://www.fakespace.com/flexReflex.htm` (Accessed April 23, 2008).

[4] Gold Standard Group. OpenGL - The Industry Standard for High Performance Graphics. `http://www.opengl.org/`. Accessed March 10th, 2008.

[5] Gold Standard Group. OpenGL Shading Language. `http://www.opengl.org/documentation/glsl/`. Accessed April 23rd, 2008.

[6] Travis L. Hilton and Parris K. Egbert. Vector fields: an interactive tool for animation, modeling and simulation with physically based 3d particle systems and soft objects. In *Computer Graphics Forum*, pages 329–338, Aire-la-Ville, Switzerland, 1994. Eurographics.

[7] Intersense. IS-900 Wireless Tracking Modules. `http://www.intersense.com/products/prec/is900/wireless.htm` (Accessed April 23, 2008).

[8] Peter Kipfer, Mark Segal, and Rüdiger Westermann. Uberflow: a gpu-based particle engine. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 115–122, New York, NY, USA, 2004. ACM.

[9] Oliver Kreylos. VRUI. `http://graphics.cs.ucdavis.edu/~{}okreylos/ResDev/Vrui/index.html` (Accessed April 23rd, 2008).

[10] Lutz Lata. Building a million particle system. In *Proceedings of the Game Developers Conference 2004*, 2004.

[11] Microsoft. ParticleGS Sample. `http://msdn2.microsoft.com/en-us/library/bb205329(VS.85).aspx` (Accessed April 23rd, 2008).

[12] Microsoft. Stream-Output Stage (Direct3D 10). `http://msdn2.microsoft.com/en-us/library/bb205121(VS.85).aspx` (Accessed April 23rd, 2008).

[13] H. Nguyen. *GPU Gems*, pages 633–676. Addison-Wesley, 3rd edition, 2008.

[14] H. Nguyen. *GPU Gems*, pages 575–632. Addison-Wesley, 3rd edition, 2008.

[15] H. Nguyen. *GPU Gems*, pages 275–348. Addison-Wesley, 3rd edition, 2008.

[16] H. Nguyen. *GPU Gems*, pages 513–528. Addison-Wesley, 3rd edition, 2008.

[17] Nicholas Francis, Over The Edge Entertainment. Deferred Particle Shading, Cooler Looking Smoke For Games. http://unity3d.com/blogs/nf/ (Accessed April 23, 2008).

[18] Nvidia. GeForce 8 Series. http://www.nvidia.com/page/geforce8.html (Accessed April 23rd, 2008).

[19] Nvidia. Nvidia Quadro Plex VCS. http://www.nvidia.com/page/quadroplex.html (Accessed Apirl 23rd, 2008).

[20] Nvidia Corporation. OpenGL Geometry Shader 4 Extension. http://developer.download.nvidia.com/opengl/specs/GL_EXT_geometry_shader4.txt(AccessedApril23,2008).

[21] Nvidia Corporation. OpenGL Transform Feedback Extension. http://developer.download.nvidia.com/opengl/specs/GL_NV_transform_feedback.txt(AccessedApril23rd,2008).

[22] Nvidia Corporation. Using Vertex Buffer Objects. http://developer.nvidia.com/object/using_VBOs.html (Accessed April 23, 2008).

[23] Open Source Community. OpenSceneGraph. http://www.openscenegraph.org/ (Accessed April 23, 2008).

[24] William T. Reeves. Particle systems–a technique for modeling a class of fuzzy objects. In *SIGGRAPH '83: Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, pages 359–375, New York, NY, USA, 1983. ACM Press.

[25] Randi J. Rost. *OpenGL Shading Language*. Pearson Education Inc., 2 edition, 2006.

[26] Ren Shao. Ren shao-work. http:/blogs.vislab.usyd.edu.au/index.php/RenShao?cat=136 (Accessed April 29, 2008).

[27] William R. Sherman. Freevr. http://www.freevr.org/(AccessedApril23,2008).

[28] William R. Sherman and Alan B. Craig. *Understanding Virtual Reality*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 2003.

[29] Iam Sommerville. *Software Engineering*. Addison-Wesley, 7 edition, 2004.

[30] Standard Performance Evaluation Corporation. Vertex Buffer Objects (VBOs). http://www.spec.org/gpc/opc.static/vbo_whitepaper.html (Accessed April 23, 2008).

[31] Tristan Lorach. Soft Particles. `http://developer.download.nvidia.com/whitepapers/2007/SDK10/SoftParticles_hi.pdf` (Accessed April 23rd, 2008).

[32] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide Sixth edition The Official Guide to Learning OpenGL Version 2.1*. Addison Wesley, 5th edition, 2007.