2013

Large-scale cloudscapes using noise

Mario Rosa

R·I·T

Large-Scale Cloudscapes Using Noise

by

Mario Rosa

A Thesis Submitted in Partial Fulfillment

of the Requirements for the Degree of

Master of Science in Computer Science

Department of Computer Science

B. Thomas Golisano College of Computing and Information Sciences

Rochester Institute of Technology

Rochester, NY

June 12, 2013

Committee Approval:

Joe Geigel Date

Committee Chair

Reynold Bailey Date

Committee Reader

Edith Hemaspaandra Date

Committee Observer

Acknowledgments

I would like to express my deepest appreciation to all those who provided me with the support and encouragement to complete this report and my graduate degree.

Thank you to my partner, Connie, for her love, kindness, and patience.

Thank you to my parents and family, for their never-ending support, encouragement, and love.

Thank you to the Rink family, for their extraordinary kindness and generosity over the last several years.

Thank you to Dr. Hans-Peter Bischof, for the amazing opportunity and support he has given me throughout my graduate studies.

Thank you to my committee chair Professor Joe Geigel, for all of his support and guidance throughout the thesis process.

Thank you to my committee reader Professor Reynold Bailey, for his valuable time and insight.

Thank you to my friends and mentors that have supported me over the years at RIT.

Abstract

Clouds have been of particular interest in computer graphics due to the challenge they present. Clouds are considered fuzzy objects, and need specialized algorithms to model and render realistically. Many techniques exist to model and render clouds that have had much success. This research will take existing techniques in cloud modeling and rendering and create a new technique combining those with noise. The idea is that noise can be used to model large-scale repeatable 3D cloudscapes and to be able to model such cloudscapes much more quickly than current techniques. This would be beneficial to developers of virtual universes that have very many worlds numbering in the ten to hundreds to create convincing cloudscapes on each distinct world.

Contents

Committee Approval i

Acknowledgments ii

Abstract iii

Contents iv

Figures vii

Tables x

1 Introduction **1**

2 Background **3**

2.1 Cloud Formation and Classification 3

2.1.1 Cloud Formation 3

2.1.1.1 Cooling 4

2.1.1.2 Lifting Condensation Level 4

2.1.1.3 Air Parcel Lift 5

2.1.2 Cloud Classification 5

2.1.2.1 Low Level Clouds 5

2.1.2.2 Mid Level Clouds 6

2.1.2.3 High Level Clouds 6

2.2 Cloud Modeling, Morphing, and Rendering 7

2.2.1 Cloud Modeling 7

2.2.1.1 Voxel Volumes 7

2.2.1.2 Metaballs 8
2.2.1.3 Particle Systems 8
2.2.1.4 Procedural Noise 9
2.2.1.5 Textured Solid Surface Representations 9
2.2.2 Cloud Morphing 10
2.2.3 Cloud Rendering 11
2.3 Noise 12
2.3.1 Perlin Noise 13
2.3.2 Simplex Noise 14

3 Methods **16**
3.1 Overview of the Dobashi Method 16
3.2 Modifications to the Dobashi Equations 18
3.3 Generating Noise 20
3.4 Generating a Cloudscape from Noise 21

4 System **23**
4.1 Noise Maker 23
4.1.1 Making a Noise Texture 24
4.1.2 Noise Effects 25
4.2 Cloud Maker 28
4.2.1 Modeling 29
4.2.2 Rendering 29
4.2.3 Shading 30
4.2.4 Drawing 32

5 Discussion **34**
5.1 Results 35
5.1.1 Performance 35

5.1.1.1 Varying Width, Depth, and Probabilities with Identical Activation and Humidity Noise Patterns 35
5.1.1.2 Varying Probabilities with Different Activation and Humidity Noise Patterns 39
5.1.1.3 Varying Activation and Humidity Probabilities with Different Activation and Humidity Noise Patterns 41
5.1.2 Cloud Types 43
5.1.2.1 Stratus Cloud 43
5.1.2.2 Stratocumulus 46
5.1.2.3 Altocumulus 49
5.1.3 Idiosyncrasies of the Algorithm 52
5.1.3.1 Texture Size 52
5.1.3.2 Vertical Development 53
5.2 Conclusions 54
5.3 Future Work 55

6 References **57**

List of Figures

4.1 The Noise Maker. 24

4.2 Pictures of noise at different increments at a width of 480 and height of 480. (a) X-Increment:0.005 Y-Increment:0.005 (b) X-Increment:0.001 Y-Increment:0.005 (c) X-Increment:0.001 Y-Increment:0.001 26

4.3 Pictures of the effect of the move values. (a) Same as 4.2a, no move (b) 'a' with X-Move:10.0,Y-Move:-10.0 26

4.4 Pictures of the effect of the feather function. (a) Same as 4.2a, no feather (b) 'a' with X-depth:400 Y-depth:400 (c) 'b' with X-start:40 Y-start:40 (d) 'c' with X-falloff:0 Y-falloff:1 27

4.5 Pictures of the effect of the cutoff function. (a) Same as 4.2a, no cutoff (b) 'a' with Low-cutoff:0.3 High-cutoff:1.0 (c) 'a' with Low-cutoff:0.0 High-cutoff:0.7 27

4.6 The Cloud Maker. 28

4.7 A simplified overview of the Cloud Maker pipeline showing the CPU and GPU (vertex/fragment shaders) aspects. 30

4.8 The 256 different splats that can be used. Splat size increases from top to bottom, left to right. Displayed on a black background to make the splats visible. 31

4.9 Clouds using 4.2a as the activation noise, with humidity probability set at 1, activation probability set at 0.8 and extinction probability set at 0. All other values as in 4.6. 33

5.1 These images are the renderings associated with the timings from table 5.1. The non-varying probabilities were set at 0.6. The defaults for height/length is 100, with depth at 10. 37

5.2 These images are the renderings associated with the timings from table 5.2. The non-varying probabilities were set at 0.6. Height and length are 100, with depth at 10. 40

5.3 These images are the renderings associated with the timings from table 5.3. The non-varying probability is set at 0.5 with extinction set at 0.0. . 42

5.4 Pictures of the (a) humidity (with X-increment and Y-increment of 0.1 and moved 0.5 increments both the X and Y direction) and (b) activation (with the same X and Y increment as humidity) used to make stratus clouds in figure 5.5. 43

5.5 Picture of stratus clouds made from noise in figure 5.4. 500 by 500 by 5 with humidity and activation probability of 0.6. 44

5.6 Picture of a stratus cloud that is large made from noise in figure 5.10. 500 by 500 by 4 with humidity probability at 0.5 and activation probability of 0.6. There is also an accompanying video named “stratusThree.mov” on the included CD. 45

5.7 Pictures of the (a) humidity (with X-increment and Y-increment of 0.005 and moved 1.0 increments both the X and Y direction) and (b) activation (with the same X and Y increment as humidity) used to make stratocumulus clouds in figure 5.8. 46

5.8 Picture of stratocumulus clouds that are large but show lots of sky in between them made from noise in figure 5.7. 500 by 500 by 10 with humidity probability at 0.3 and activation probability of 0.6. 47

5.9 Picture of stratocumulus clouds that are large, dark and show little to no sky in between them made from noise in figure 5.4(b). 500 by 500 by 50 with humidity probability at 1.0 and activation probability of 0.8. These clouds would look strange from the side due to the shape problem discussed in the introduction to the results section 48

5.10 Picture of the (a) humidity (with x and y increments of 1.0) and (b) activation (with x and y increments of 0.05 and x and y feather depths of 400) used to make altocumulus in figure 5.11 49

5.11 Picture of altocumulus with humidity probability of 0.5 and activation probability of 0.6 with dimensions of 1000 x 1000 x 2 and a position at a height of 5000. Made with noises in figure 5.10 50

5.12 Picture of altocumulus with humidity probability of 0.5 and activation probability of 0.6 with dimensions of 2000 x 2000 x 2 and a position at a height of 5000. Made with noises in figure 5.10 51

5.13 Pictures of clouds with activation noise increment at 0.05 and activation probability at 0.8. (a) is 100 by 100 and (b) is 500 by 500. 52

5.14 The top and the bottom of the clouds have the same shape. 53

List of Tables

5.1 Timings associated with the humidity and activation noise set as the noise in figure 4.1. TTR stands for "Time To Render." FPS stands for "Frames Per Second." The value at which each variable is held constant while another variable is varied is marked with a "(D)." The associated renderings for these results are in figure 5.1 38

5.2 Timings associated with the humidity noise set as the noise in figure 4.3(a) and the activation noise set as the noise in figure 4.3(b). TTR stands for "Time To Render." FPS stands for "Frames Per Second." The value at which each variable is held constant while another variable is varied is marked with a "(D)." This test was run with a width and length of 100 and a depth of 10. The associated renderings for these results are in figure 5.2 39

5.3 Timings associated with the humidity noise set as the noise in figure 4.3(a) and the activation noise set as the noise in figure 4.3(b). TTR stands for "Time To Render." FPS stands for "Frames Per Second." The value at which each variable is held constant is 0.5, with extinction set at 0.0. This test was run with a width and length of 100 and a depth of 10. The '1.0' test for the activation rendered no clouds. The associated renderings for these results are in figure 5.3 41

Chapter 1

Introduction

Clouds are an expected part of any 3D game or virtual world that allows its players to venture into the outdoors. Clouds give the virtual world an extra sense of realism that is pleasing and can easily add beauty to the virtual world. Rendering clouds is a complex endeavor due to their complex and irregular shapes. Which do not lend themselves to the traditional rendering pathways using basic primitives. This research is meant to improve on the current modeling and rendering techniques, which will be detailed later in this report, to solve a problem that has not been given much attention.

In modern games the virtual worlds have been getting larger and larger. The typical solution to displaying clouds is to use a noise algorithm or an artistic approach to generate a 2D cloud texture. This texture is then layered onto a half sphere above the virtual world. While this gives the effect of clouds in the atmosphere it is not dynamic. The clouds are static and there are a limited number of textures. Another problem is that when a player choses to get into an airplane or fly through other means, they are unable to pass through the cloud layer as the clouds are always in the distance.

A category of games that has become very popular in recent years are massive multi-player online games (MMO), including role playing games (RPG) and first person shooters (FPS). In a larger world, such as those common to MMO games, the creation of unique cloud patterns for the entire world would be time consuming. MMOs commonly have be-

tween tens to hundreds of playable worlds. Additionally, real-time rendering on the client side is not always necessary in these worlds, especially in ones that limit free motion to specific areas of the world. For these MMOs some of the work of cloud rendering can be moved server-side, as one can assume that many users will be in one location at a time. On a large world there can be several closed areas being used for gameplay. A storm system should be able to move across each area and from one closed area to the next. If a player finishes game objectives in one area in which a storm was moving through and to the west; that player, if deployed to an area just west of their previous game, should see the storm moving in from the previous area.

The goal of this research is to develop a flexible algorithm in which large-scale repeatable 3D cloudscapes are created easily through algorithms and can move across a multiplayer world dynamically. To create this type of effect over a large area noise can be used. This is because noise has many qualities that are desirable to produce controllable pseudo-random effects. Weather patterns are certainly random in the short-term, but they follow certain patterns over time. Being able to control the large-scale patterns, while allowing for local pseudo-randomness would be desirable. While using noise to generate clouds is not new, it has typically only been used to generate 2D textures. When used for generating 3D clouds, it is only used to generate one cloud at a time, and not an entire cloudscape. This research is aimed at investigating whether or not noise can be used in an intuitive way to generate large-scale cloud patterns that mimic typical cloud patterns.

The paper is organized as follows: In chapter 2 background on the science of clouds and cloud formation is given; then the current science of cloud rendering is reviewed as well as some background in noise generation. In chapter 3 the methods are explored. In chapter 4 the implementation of the algorithm is explored. Chapter five includes the discussion of the results, conclusions, and some future work to address the limitations of the algorithm.

Chapter 2

Background

In this chapter the background information of relevant literature and work from the scientific community is presented. First cloud formation in a natural system and classification of clouds is reviewed. Then the science of cloud modeling, morphing, and rendering in a virtual system is reviewed. Finally the science of noise generation is reviewed.

2.1 Cloud Formation and Classification

It is important to understand how clouds are formed and how they behave in the natural world if one is to simulate them successfully in a virtual world. The following sections review the science of cloud formation and classifications.

2.1.1 Cloud Formation

Cloud formation information for this section was retrieved from [37]. Cloud formation is a complex process composed of multiply processes. This section is a basic review of how clouds form.

2.1.1.1 Cooling

Weather clouds form in the troposphere, the lowest level of the atmosphere. This happens through a process in which the invisible water vapor lifts through the atmosphere and cools until the saturation vapor pressure is lower than the actual partial pressure of the water vapor. This happens due to adiabatic cooling, a process of thermodynamics in which heat is loss from the system, and isolates the system from its surroundings. In relation to cloud formation this occurs to rising air parcels. As the air parcel rises through various means of lift, it expands due to lower atmospheric pressure which lowers the internal temperature. The energy in the air parcel decreases due to the ideal gas law.

There are three other ways in which clouds can form that involve non-adiabatic cooling. Conductive cooling happens when warm air moves across a cold land mass. Radiational cooling is most common at night when there is no heat source and the air radiates the collected heat from the day. Evaporative cooling happens due to the heating of water sources to the point that the water evaporates.

2.1.1.2 Lifting Condensation Level

The lifted condensation level is the height where relative humidity in the air parcel has reached 100% through adiabatic cooling. This is the level which determines the approximate cloud base. A little ways above this level the saturation vapor pressure is much lower than the actual pressure of the water vapor causing supersaturation. Supersaturation is typically required for the water vapor to transition to water by coalescence to cloud condensation nuclei. Cloud condensation nuclei are particles in the atmosphere of very small size, typically 1/100th the size of a water droplet, that stay aloft due to normal air circulation and atmospheric drag. In cooler atmospheric regions these water droplets can transition to ice. Vapor will coalesce to the newly formed water droplets or ice crystals until a point at which it becomes too heavy to stay aloft.

2.1.1.3 Air Parcel Lift

There are three different methods in which air parcels can be lifted into the atmosphere. The first method is by means of frontal or cyclonic lift due to weather fronts and areas of low pressure. This type of lift happens in stable air and produces weather clouds that cause rain, freezing rain and snow. The second type of lift is convective lift. This type of lift is associated with unstable air and severe weather systems. This happens due to significant solar heating of the ground or because of high absolute humidity. These systems typically grow through the tropopause, the area between the troposphere and the stratosphere, and create heavy rain and thunderstorms. The third type of lift is orographic lift that happens due to geographical formations such as mountains. In this case the air can either stay stable through the process and form lenticular clouds, altocumulus or stratocumulus. If the air becomes sufficiently moist or unstable it can cause showers and thunderstorms.

2.1.2 Cloud Classification

Cloud classification information for this section was retrieved from [37]. Cloud classification has a very long history and has stayed relatively stable. There are some cloud formations that do not fit into a specific category due to rarity or because it's formed as part of a complex interaction between multiple formations. This section will review cloud classifications and the most common cloud types. In general, the higher altitude cloud are more difficult for this system to simulate.

2.1.2.1 Low Level Clouds

The low level clouds consist of clouds from ground level to about 6,500ft. These clouds are Cumulus, Stratocumulus, Stratus, and Cumulonimbus. Cumulus clouds are those that are most associated with big puffy white clouds that have a good vertical development. Cumulus clouds form from convection. They typically appear alone, in lines or in clusters

with plenty of sky to be seen. Stratocumulus are Cumulus clouds that have failed to develop vertically due to inversion, typically due to a high temperature above the cloud. Stratus clouds are typically formed from Stratocumulus clouds that faced very strong inversion and spread out to a point at which they become "hazy". Cumulonimbus clouds are those clouds that have extreme vertical development and are associated with severe weather patterns. In the system presented in this report, clouds with more horizontal development are easier to implement due to a limitation in vertical development of the clouds. Because of this Cumulonimbus cannot be simulated accurately. Additionally it is difficult to get a "hazy" look with this algorithm.

2.1.2.2 Mid Level Clouds

The mid level clouds consist of clouds from 6,500ft to 25,000ft. These clouds are Altocumulus, Altostratus, and Nimbostratus. Altocumulus clouds are very streaky clouds that form in a layer, and have a fluffy appearance. Some variations are thin enough in that you can see the position of the sun and moon. Altostratus clouds are thin uniform-gray to bluish-gray and form a large horizontal that is wavy and featureless, but can be broken due to wind. Altostratus are formed from large rising air masses, and can form in thick layers and resemble Stratus clouds. Nimbostratus clouds are darker than Altostratus and are typically formless cloud layers. Nimbostratus are often caused by thick Altostratus clouds that descends into lower altitudes. It is difficult to create the more featureless clouds in this genus with this algorithm. Although the clouds are mostly featureless, the features that are present are very small and difficult to simulate.

2.1.2.3 High Level Clouds

The high level clouds consist of clouds from 10,000ft to 60,000ft. These clouds are Cirrus, Cirrocumulus, and Cirrostratus. Cirrus clouds are very wispy clouds that can be seen on clear days, and reveal the direction of the wind in the high atmosphere. These form at the very top and outer edges of weather systems and can stretch across continents when

pulled by the jet-stream. Cirrocumulus are clouds that are thin but look puffy from the ground, they are typically in large groups. These clouds, like the other cumulus variants, are formed by convection. Cirrostratus clouds are very thin, large layers of clouds. These clouds form from Cirrocumulus that become heavy due to supercooled water droplets freezing, causing inversion. Due to artifacts and other limitations in the modeling and rendering process these are the most difficult to simulate.

2.2 Cloud Modeling, Morphing, and Rendering

Much research and work has been done in the scientific community to simulate clouds in the virtual world. The following sections review the science of cloud modeling, morphing, and rendering in a virtual world.

2.2.1 Cloud Modeling

Clouds are irregular and complex objects that do not lend themselves to the traditional modeling techniques using only primitives. In order to model clouds one must take into consideration the way in which clouds behave. Clouds are fuzzy on the surface, but generally have a solid density filling their volume. The following will review the most common methods of modeling clouds.

2.2.1.1 Voxel Volumes

A voxel or a volumetric pixel is a volume cell in a regular three-dimensional grid. Voxels are a commonly used tool when rendering implicit 3D objects because they provide an uniform sampling through the object space. Voxel grids are easy to create using 3D textures, lend well to physically-based simulations, and there are a large variety of methods to render voxel grids. Kajiya and Herzen [15] used voxels to store the results of a physical cloud simulation, the voxels were then rendered using ray tracing. Dobashi et al. [9] used a cellular automata method based on a voxel grid to simulate clouds similar to

earlier work by Nagel and Raschke [34], which approximates some of the physical cloud dynamics in a less accurate way. Miyazaki et al. [33] used the same rendering method as [9] but used a Coupled Map Lattice method to perform the cloud simulation on a grid. This model was able to simulate multiple different cloud types. Miyazaki et al. [32] extended this work in 2002 to produce more accurate cumulus clouds. The results of partial differential equations can be stored on a voxel grid for cloud rendering such as in [12,16,24,25], these methods are more accurate to the physical dynamics of clouds. Harris [24] is perhaps the most realistic of the physical based methods. As in other interests in computer graphics, this area is no stranger to the tradeoffs between realism and real-time rendering. Koehler [25] tries to take a faster approach to [24] using a less accurate physics model, consequently losing out on some of the realism.

2.2.1.2 Metaballs

Metaballs are mathematically described shapes defined by several sources of field potential similar to isosurfaces. Each source is defined by a center, radius, and strength of the local field potential [35]. Nishita et al. [10] first used metaballs for modeling clouds by sparsely hand placing metaballs, and using a fractal method on top of the placed metaballs to fill in detail. Dobashi et al. has used metaballs in two different applications. In [31] metaballs were used to fill in cloud volumes created from satellite imagery. In [9] metaballs are used to collect density from clouds simulated using a voxel grid, the metaballs were then splatted for rendering. Miyazaki et al. also used the Dobashi metaball splat method in [32,33]. Splatting has become one of the main rendering techniques used in cloud rendering since it lends itself to many modeling techniques and graphics tricks such as billboards [11].

2.2.1.3 Particle Systems

Particles are small geometric primitives which are described by a position and other properties [1]. They are easy to render and inexpensive to manage which make them

attractive for real-time applications. They are often used in gaseous modeling as a volume filler as particles can be used in physic engines. But even though particles are inexpensive, it takes a very large number of particles to represent small details. Harris and Lastra used particles in their rendering engine [11] and achieved high frame rates by using a splatting method and billboard imposters. This method was static, however, and was not based on a physical model. Particles have become particularly popular for use in this area and have been used in many other methods. Harris continued the use of particles in [12] and [24]. C.M. Yu and C.M. Wang [14] introduced methods to morph particles into various shapes. In [20-21] N. Wang used artistic methods to model clouds and used textured particles to create a dozen different cloud types.

2.2.1.4 Procedural Noise

Continuous density data is a technique using noise to fill fuzzy volumes [2,3,26]. Ebert extends the use of noise and turbulence to fill volumes by using a two-level approach. First by allowing a user to place implicit geometric primitives, which are then unionized, to create the cloud macro structure. The primitives are then used as implicit function inputs into a procedural solid noise perturbation method to create the cloud micro structure and rendered using a scan line technique [28,27,7]. Schpok et al. [13] expanded on this work by moving parts of the algorithm to the graphics processors. In Bouthors et al. [18] hypertextures were used to create the surface characteristics of clouds. They found they were able to control the creation of the hypertexture to create small details of many different forms and shapes.

2.2.1.5 Textured Solid Surface Representations

Textured solids are a mix of different planar and curved surfaces that are shaded by a mathematical formula. In this approach only the surface of the clouds is computed and not the volume. In Gardner [17] a fractal texturing technique was used on multiple textured and shaded ellipsoids. Westover [30] proposed a method to interactively render

clouds using a variant of [17]. Most recently Bouthors et al. [18] uses a surface mesh combined with a hypertexture that allows them to render a large variety of cloud shapes.

2.2.2 Cloud Morphing

Many morphing techniques involve simulating the physical dynamics of cloud formation and morphing [12,15,16,24,25]. The first to demonstrate a cloud simulation in computer graphics is Kajiya and Herzen [15]. They generated cloud data sets using a set of partial differential equations (PDE) known as the Navier-Stokes equations of incompressible fluid flow. A more detailed physical model also based on PDEs was demonstrated in Overby et al. [16] using stable fluid simulation algorithms to solve the Navier-Stokes equations. A faster and more realistic simulation is introduced in Harris et al. [12] (and continued in Harris [24]) by solving more realistic PDEs and by moving the simulation to graphics hardware.

Others such as [9,34,32,33] have simulated cloud morphing in a less physically accurate way but with good results. Nagel and Raschke [34] introduced the use of a rule-based approach using cellular automata that is able to spread humidity around a voxel boolean grid and form clouds in the humid cells. Dobashi et al. [9] extended this research by adding a cloud extinction mechanism. Extending the cellular automata Miyazaki et al. [32] used a coupled map lattice that used continuous values in the voxel grid; allowing them to approximate many of the physical dynamics of clouds. In C.M. Yu and C.M. Wang [14] they introduced a method of morphing particles into a tetrahedron model extracted by skeletonization from the chosen models. This approach to cloud morphing is more artistic and allows the user to select a source and target model. The models can be of any shape, such as the word 'CLOUD' or the shape of a bunny. The morphing in the system presented in this report is not done in real-time, instead the morphing is done in a pre-render step and then kept static through-out the rendering.

2.2.3 Cloud Rendering

Rendering clouds involves displaying the illuminated and shaded volume or surface. The most obvious approach to rendering volume densities, but most expensive is ray tracing [15]. The most popular algorithm used to render clouds is the splatting algorithm first used in this application in Nishita et al. [10] and used in [9-12, 14, 19-22] as a part of their rendering. This algorithm is popular due to its flexibility in that the modeling of the clouds can be done with different techniques as long as a density can be made. C.M. Yu and C.M. Wang [14] proposed an improvement on the splatting technique by using a two line integral convolution (LIC) in a hybrid method to avoid artificial Gaussian blobs. This improves the look of the clouds near the edges, where the difference would be most noticed by the user. The basic idea is that the 3D density of a cloud is splatted onto many 2D billboards. How many billboards needed depends on the use. Harris and Lastra [11] found that using multiple billboards improved the look of fly-throughs by placing one in front of, and one behind the object from the viewers perspective. While clouds farther away only required one billboard. N. Wang [20-21] extended the use of billboards by surrounding the current position with many large billboards that display the far away clouds, while nearby clouds are rendered in volume using smaller billboards.

Illuminating the clouds involves shading the clouds, as well as shading the ground and in some cases using volumetric rendering to show sun rays. Shading of clouds typically involves calculating the scattering of light though the clouds. Light scattering was first explored in Blinn [36] to render the rings of Saturn. Dobashi et al [9,31] and Miyazaki et al. [32] uses single scattering of light using line integral methods that is more appropriate when a good approximation is needed for real-time applications. The method in [9] involves a OpenGL blending technique that is very intuitive. This method is extended in Harris and Lastra [11] to account for multiple forward scattering using particles and was extended again in Harris et al. [12] to voxel based clouds.

The more complex and accurate approach is simulating multiple forward scattering and sky light [10,11,12,18,24,25,38]. Kajiya and Herzen [15] used a single scattering

method in their work, but also described a multiple scattering method using spherical harmonics. In Nishita et al. [10] a finite element method was used by dividing the cloud volume into voxels and computing radiative transfer between the voxels. The method in [10] had a low cost due to two simplifying observations; water droplets are favorable to forward scattering due to their highly anisotropic phase function; and the contribution of scattered light decreases quickly, allowing for scattering computations to be limited to an order of three. A technique capable of illuminating many different volumes with good results using a line integral approach is presented in Kniss et al. [38]. This method is able to simulate absorption and multiple forward scattering using a "half angle slice" technique. This technique allows for the illumination and the display of the volume to be done together in alternate passes as they traverse through the volume.

For this algorithm rendering the clouds using the above techniques would not be necessary to demonstrate the hypothesis, as the idea would be that this research could be extended to any rendering model. For completeness, a splatting technique will be used with single scattering of light. Since this algorithm relies on multiple clouds being created, an impostor for each cloud would not be viable. Instead billboards will be placed at the center of metaballs placed periodically throughout the clouds as in Dobashi et al. [9].

2.3 Noise

Noise is a repeatable pseudorandom function that has coordinate inputs in n dimensions and an output of one dimension [7]. A truly random function would have no inputs, as repeated inputs would lead to different outputs. Noise is important in computer graphics as it allows for textures over 3D objects to be consistent from frame to frame within animation. In general the input coordinates are transformed into a hash value that can be used as a pseudo-random number or used as an index to fetch a pseudo-random number. Noise has a known range from -1 to 1, this allows for scaling of the noise function in

predictable ways. Noise is statistically invariant under rotation (isotropic) and translation (stationary), and has a narrow bandpass limit in frequency with a maximum of about 1.

The statistically invariant features allows the noise to appear pseudo-random, while allowing for the ability to create repeatable textures without the need for storing the texture in memory. This means that an input to the noise function will give back the same results. This allows for the creation of complex and natural looking textures that can be rotated and translated in space. Because humans are sensitive to continuity this has allowed for the fast and easy creation of realistic texturing of 3D objects. Popular examples from literature include the simulation of marble, water, fire, bumpy surfaces and fractal noise. The next property, a narrow bandpass, allows for all of the features of the noise to appear the same size. In other words, no one feature overwhelms another. This allows the noise to be easily controlled and inserted into more complex noise functions. Although noise functions are pseudo-random and will express patterns over a period, the periods can be made to be very large to hide any obvious periodicities.

Two of the most popular noise functions, Perlin and Simplex noise, are described below. Simplex noise is used in the algorithm described in this report.

2.3.1 Perlin Noise

Perlin noise was introduced by Ken Perlin as a way to create procedural texture primitives [2]. Perlin noise has been used extensively in the film industry, for which Perlin was awarded an Academy Award. Perlin noise has even been used to create hypertextures, 3d representations of the noise modeled by density functions [3].

Perlin noise is implemented by using a pseudo-random spline on a regular cubic grid lattice. This has problems when moving to higher dimensions as it is inherently a $O(2^N)$ problem. Given the input point a hash value is computed. This hash value is used as an index to get a pseudo-random gradient vector for each of the closest grid points. These gradients are combined with a vector from the grid point using a scalar product. Then a bilinear function is used to blend the results together.

Perlin noise has received some improvements. In Perlin [4], the noise algorithm was corrected to remove two defects in the original algorithm. One of the defects was the use of the Hermite blending function which has a second derivative that created discontinuities due to none-zero values at the endpoints. Changing this function to a fifth degree polynomial solved that issue. The second defect was the use of a pseudo-random gradient distribution that was needlessly expensive and resulted in clumping. This was fixed by pre-specifying unit vectors in different directions from the center of a unit square. This also improved performance by removing 25 multiplies from the scalar product calculations. More improvements Kensler et al. [6] are made to introduce a better hash function, as well as better band-limiting.

2.3.2 Simplex Noise

Simplex noise is an improvement on the classic Perlin noise developed by Perlin [5] and is most clearly detailed in Gustavson [8]. Simplex noise has many benefits over the classic noise function, though it is not a direct replacement of the classic algorithm. The most prominent benefit is the reduction in computational complexity and the number of multiplications. For example, simplex noise can be calculated in $O(N^2)$ time for N dimensions.

The reason for this speed up is due to the use of a simplex grid instead of a cubic grid. Only $N+1$ corners are needed to make a simplex grid of N dimensions in contrast to the 2^N corners needed for the classic algorithm. The number of multiplies are reduced because interpolation is no longer required. Instead the summation of contributions from the corners of the simplex that contains the point are used to make the noise value. In order to find the simplex cell that contains the input, the input point must first be skewed into the simplex space. Then the distance from the cell origin must be deskewed to input space and the magnitudes of the distance between itself to the input point must be calculated for each dimension. These magnitudes are used to determine which simplex the point is in. Once this is found the corners of the simplex can be traversed

to find the contribution of each using a gradient approach similar to Perlin noise. These contributions are then summed and scaled to its final value. [6] improvements can also be applied to simplex noise.

Chapter 3

Methods

The developed algorithm is a combination of two loosely coupled algorithms. One is the noise maker algorithm that uses at its core a simplex noise algorithm ported from the Gustavson implementation [8]. The second is the cloud maker algorithm based on the work of Dobashi et al. [9]. The two algorithms work together with the results of the noise maker algorithm acting as one of the inputs to the cloud maker algorithm.

3.1 Overview of the Dobashi Method

In this research the methods used are most similar to Dobashi et al. [9] where a voxel gird of three boolean values is used to simulate the cloud dynamics. This is less accurate because it does not use physics simulations and uses boolean values which can only represent 0 and 1. The three values that are used in the simulation are humidity (hum), activation (act) and cloud (cld). The humidity and activation values interact with each other in order to create clouds through transition functions and probability calculations. A humidity value of one in a cell means that cell has the required humidity to activate. An activation value of one means that cell is able to make a phase transition from vapor to water. The cloud values keep track of whether there are clouds (cld = 1) or no clouds (cld = 0) in a given cell.

A set of transition rules use boolean math and are applied at each cell at each time step. These rules are:

$$hum(<i,j,k>,t_{i+1}) = hum(<i,j,k>,t_i) \wedge \neg act(<i,j,k>,t_i) \vee IS(rnd < p_{hum}(<i,j,k>,t_i)) \tag{3.1}$$

$$cld(<i,j,k>,t_{i+1}) = cld(<i,j,k>,t_i) \vee act(<i,j,k>,t_i) \wedge IS(rnd > p_{ext}(<i,j,k>,t_i)) \tag{3.2}$$

$$act(<i,j,k>,t_{i+1}) = hum(<i,j,k>,t_i) \wedge \neg act(<i,j,k>,t_i) \wedge f_{act}(<i,j,k>,t_i) \vee IS(rnd < p_{act}(<i,j,k>,t_i)) \tag{3.3}$$

with f_{act} defined as follows:

$$\begin{aligned} f_{act}(<i,j,k,>,t_i)) = {} & act(<i+1,j,k>,t_i) \vee act(<i,j+1,k>,t_i) \\ & \vee act(<i,j,k+1>,t_i) \vee act(<i,j-1,k-1>,t_i) \\ & \vee act(<i-1,j,k>,t_i) \vee act(<i,j-1,k>,t_i) \\ & \vee act(<i-2,j,k>,t_i) \vee act(<i+2,j,k>,t_i) \\ & \vee act(<i,j-2,k>,t_i) \vee act(<i,j+2,k>,t_i) \\ & \vee act(<i,j,k-2>,t_i) \end{aligned} \tag{3.4}$$

A cell can become humid or activated randomly based on user set probability values (p_{hum} or p_{act} respectively). A cell cannot become unhumid or unactivated by random means, but a cell with a cloud can suddenly lose a cloud through random means based on a user set probability of cloud extinction (p_{ext}). A cell cannot be humid and activated at the same time. This is because a cell that is activated "uses up" its humidity and humidity is set to 0. Therefore a humid cell can transfer to an activated cell, but an activated cell cannot transfer to a humid cell. A humid cell will only transfer to an activated cell if cells nearby are also activated, more specifically if the cells two levels below or two levels to the side or one level above are activated. This builds on the dynamics of clouds, in that clouds form where clouds already exist, and generally build upwards. An activated

cell will quickly become a cloud in the next time step. This activation process also means that cells that become extinct randomly will reactivate and become a cloud quickly if they are in the middle of a cloud formation, while cells at the edge are more likely to stay extinct.

The density created by this voxel grid is then transferred to a lattice of metaballs. Each metaball collects the density from the voxels within its radius using a field function. The metaball is then splatted onto a billboard. This billboard is then placed at the center of the metaball and is used to create a ground shadow texture and is used to display the clouds to the user. When creating the ground texture, the clouds are sorted based on the distance to the sun and the camera is placed at the sun location. The billboards are then blended from front to back and the ground shadow texture is displayed on the ground. When the billboards are displayed to the user they are sorted in the same way except based on the camera location and are blended from back to front. They also render shafts of light through the clouds using multiple textured spherical shells around the camera. This whole process is not done in real-time, instead a voxel grid is stored for each time step, then all the time steps are rendered into a movie.

3.2 Modifications to the Dobashi Equations

Modifications had to be made to the Dobashi equations from the originals presented in 3.1. This is due to several differences in the algorithms. This algorithm uses two noise textures as the randomness and therefore the probability values are really just thresholds against the noise textures. The equation in [9] are 3D equations, in this algorithm 3D lookup of values is not necessary since each $< i, j >$ value has the same values depth-wise due to the 2D noise textures. Many different equations were tried but through trial and error the ones presented below gave the only results that created the desired output. The results are written as boolean values which is then used in the cloud rendering step to decide where clouds are positioned.

Two noise textures are used to represent the humidity and activation. The activation texture is what the clouds look like 'now' while the humidity texture will show what clouds might appear after the rendering. One important property of the humidity noise, however, is that clouds will only appear in a spot if a cloud was already introduced nearby by the activation noise. In effect the final resulting cloudscape is a combination of the activation and humidity textures. How much of the activation and humidity texture is used is chosen by the user in the cloud maker by setting the probabilities. There are three probability inputs: humidity, activation, and extinction. The equations expressing this process are as follows.

$$tex_{cld}(< i, j >) = \frac{((tex_{hum}(< i, j >) + tex_{act}(< i, j >))}{2} \tag{3.5}$$

$$\begin{aligned} hum(< i, j >, t_{i+1}) = {} & hum(< i, j >, t_i) \wedge \neg act(< i, j >, t_i) \\ & \vee (tex_{hum}(< i, j >) > p_{hum}) \end{aligned} \tag{3.6}$$

$$\begin{aligned} cld(< i, j >, t_{i+1}) = {} & cld(< i, j >, t_i) \vee act(< i, j >, t_i) \\ & \wedge (tex_{cld}(< i, j >) > p_{ext}) \end{aligned} \tag{3.7}$$

$$\begin{aligned} act(< i, j >, t_{i+1}) = {} & hum(< i, j >, t_i) \wedge \neg act(< i, j >, t_i) \\ & \wedge f_{act}(< i, j >, t_i) \vee (tex_{act}(< i, j >) > p_{act}) \end{aligned} \tag{3.8}$$

with f_{act} defined as follows with one condition:

any test found to be 'out of range' results in a false value.

$$\begin{aligned} f_{act}(< i, j, >, t_i)) = {} & act(< i+1, j, >, t_i) \vee act(< i, j+1, >, t_i) \\ & \vee act(< i-1, j, >, t_i) \vee act(< i, j-1, >, t_i) \\ & \vee act(< i-2, j, >, t_i) \vee act(< i+2, j, >, t_i) \\ & \vee act(< i, j-2, >, t_i) \vee act(< i, j+2, >, t_i) \end{aligned} \tag{3.9}$$

These equations use boolean math to combine the humidity, activation and cloud exist values. Any values surrounded by carrots such as $< i, j >$ represents a coordinate and when followed by a t_i is a coordinate at a certain time. The values tex_{act} and

tex_{hum} represents the current activation and humidity texture values from the user set noise textures. The value tex_{cld} is derived from the current activation and humidity, and represents the presence of a cloud at that texture coordinate. The values p_{hum}, p_{act}, and p_{ext} are the user set humidity, activation, and cloud extinction probability values, respectively.

3.3 Generating Noise

Many different noise patterns can be created and, in combination with the cloud maker, can be used to create cloud patterns found in nature through trial and error. The noise maker algorithm is composed of two 2D simplex noise texture creators, each creator at its core is a wrapper around a 2D simplex noise algorithm. One creator is for the activation noise texture, the other is for the humidity noise texture. The creators are not simply a simplex noise algorithm though, as the creators run the simplex noise algorithm on many different two dimensional coordinate inputs in order to create a rectangular texture while also applying various manipulations.

The noise maker when first run sends a set of default inputs to both of the creators to generate default textures. The algorithm then sets one of the creators as the current creator. The current creator receives any inputs that have changed from the default by the user, and recreates its texture. At any time a user may change the current creator to the inactive one. Simply put the noise maker acts as a switch to determine which creator gets the user input.

The noise maker receives 15 inputs, 14 of which are passed onto the current creator. The inputs are:

- The switch that determines the current creator
- Four inputs to determine the increment pattern used to generate the two dimensional coordinate inputs to the simplex noise algorithm

- Two inputs to determine the starting 2D coordinate
- Two inputs to determine the allowable range of outputs from the simplex noise algorithm
- Six inputs to determine how a feather effect is applied to the outputs from the simplex noise algorithm

The output of the noise maker are the humidity and activation textures.

3.4 Generating a Cloudscape from Noise

The cloud maker algorithm takes in 15 inputs, 2 of which are the noise textures created by the noise maker. The inputs are:

- The dimension
- Position or center of the clouds
- The wind direction
- The orientation
- The probabilities

The dimensions of the cloudscape is controlled by setting the length, width, and depth parameters. The position of the clouds is the center of the clouds and is set with X, Y, and Z parameters. The wind direction is a vector set using X and Y parameters and moves the clouds in the vector direction and the speed of the vector magnitude. The orientation is the rotation of the cloudscape around the center of the cloudscape, this is set using degrees. The probabilities are used to control how the cloud maker uses the noise textures. The probabilities include humidity, activation and extinction parameters. The dimension of the clouds and the probabilities are the only inputs that directly affect

the outcome of the cloud creation and rendering. The other inputs only affect minor cosmetic properties of the cloudscape.

The first step is modeling. A density is extracted from the noise textures to create a 2D boolean voxel grid of humidity, activation, and cloud values. The modified Dobashi equation (equations 3.5-3.9) is used to extract the density from the noise textures into the 2D boolean voxel grid of humidity, activation, and cloud exist values.

The second step is the rendering, shading, and drawing step. The density from the 2D voxel grid of cloud values is extracted and smoothed so that the middle of each cloud has values close to '1' and the edges of each cloud has values close to '0'. The density is then transferred into metaballs placed periodically throughout the cloudscape by interpreting the 2D grid as a 3D grid by ignoring the 'z' position of the metaballs. This would be similar to repeating the 2D voxel gird Z-wise based on the user set depth parameter to create a 3D voxel grid. Each metaball is then assigned a splat based on the density of the metaball and shaded with an initial color. The metaballs are then assigned to their respective billboards based on their position in the cloudscape. To simulate scattering of light through the clouds a blending algorithm is used. The billboards are rotated to face the sun and the metaballs are rendered by distance to the sun, with the closest rendering first. This creates a ground shadow, which is then used to reshade the metaballs. Metaballs at the top of the clouds are brightest while those at the bottom become darker based on the depth and density of the clouds. The billboards, textured with their respective reshaded metaballs, are then rotated to face the virtual camera. Finally the billboards are rendered and blended back to front based on distance from the camera.

Chapter 4

System

The system designed around the algorithm is built using Objective-C and the Cocoa API along with OpenGL API's and techniques. The GUI of the system is loosely connected to the underlying algorithms through an MVC pattern. The user is presented with one window and a simple "world" consisting of a green ground and a blue sky. There are two different modes the user can enter from this window by accessing the view menu in the menu bar: the noise maker system and the cloud maker system. The following sections go into more detail on these two systems.

4.1 Noise Maker

The noise maker interface, displayed in figure 4.1 allows access to the parameters of the noise maker algorithm. The noise maker defaults to showing the humidity noise when first opened. A slide-out drawer provides access to the parameters described in section 3.3 and in more detail in subsections 4.1.1 and 4.1.2. Changing a value will automatically refresh the noise texture. A tab control is provided to allow the user to change between the humidity and activation noise controls.

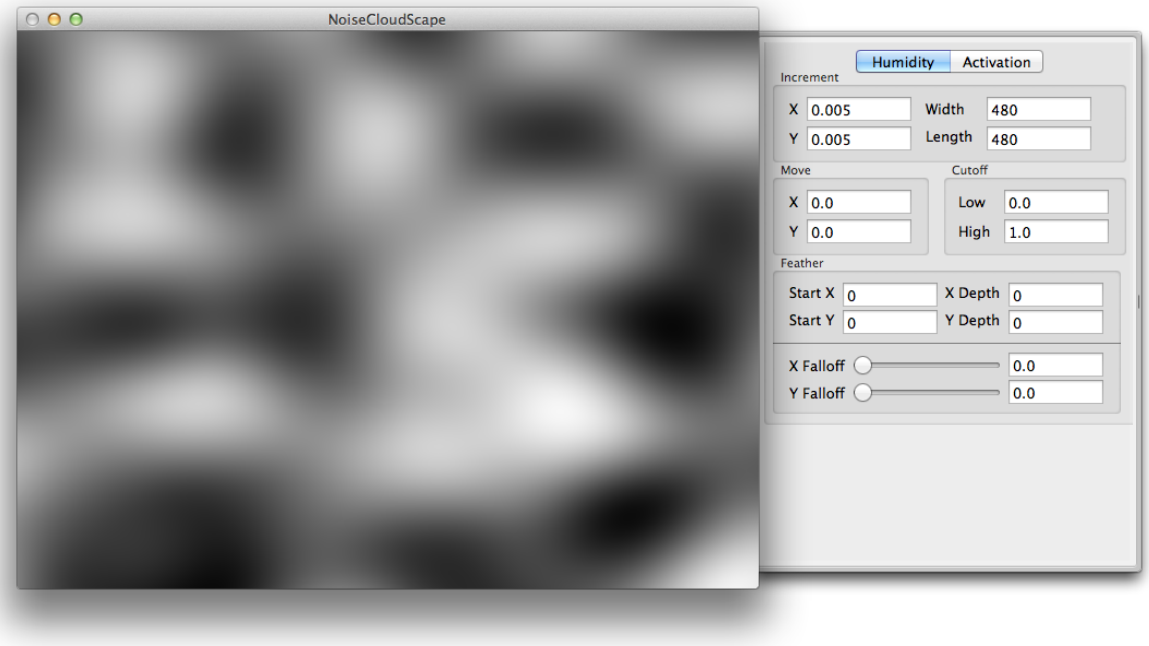


Figure 4.1: The Noise Maker.

4.1.1 Making a Noise Texture

The noise textures are generated on the CPU, and are only regenerated when the user changes a value. Each noise texture is created as in the following pseudocode:

$noiseArr \leftarrow Array(680 * 510)$
$xOffInc \leftarrow \frac{XInc * width}{680}$
$yOffInc \leftarrow \frac{YInc * height}{510}$
for $i \leftarrow 0, xOff \leftarrow xMove; i < 680; i++, xOff += xOffInc$ **do**
  for $j \leftarrow 0, yOff \leftarrow yMove; j < 510; j++, yOff += yOffInc$ **do**
    $noiseArr[i][j] \leftarrow applyEffects((simplexNoise(xOff, yOff) + 1.0)/2.0)$

The amount by which $xOff$ and $yOff$ increment is determined by four user inputs: two integers $width$ and $length$, and two floats $XInc$ and $YInc$. Examples of different noise patterns made by changing the increment can be see in figure 4.2. Two user inputs $xMove$ and $yMove$, both floats, are the initial values of $xOff$ and $yOff$ respectively; these two user inputs change the starting position of the texture pattern as can be seen in figure 4.3. The value returned by the 2D simplex noise function is a float between -1.0 and 1.0. This range of values is transformed to be between 0 and 1 and is referred to as

a brightness value.

4.1.2 Noise Effects

After the transformation the following effects are applied to the brightness value using the CPU. The first effect is a feather function. This function manipulates the edges of the current noise texture. There are six inputs, four integers and two floats. The integers are:

- $XFeatherStart$
- $YFeatherStart$
- $XFeatherDepth$
- $YFeatherDepth$

And the floats are:

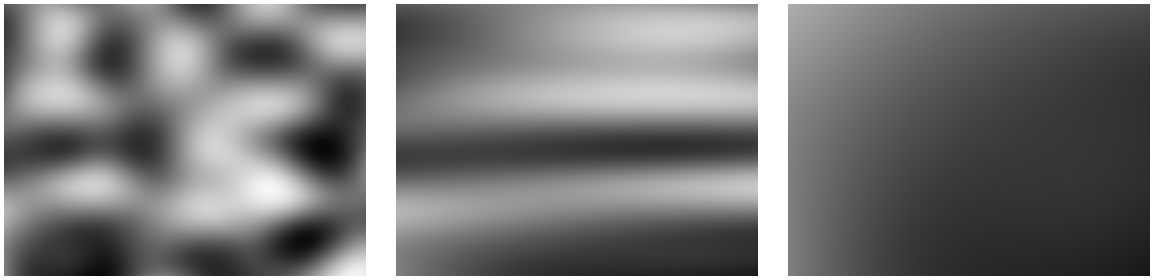
- $XFeatherFalloff$
- $YFeatherFalloff$

The feather start variables determine how far in from the edges the feather starts. The feather depth determines how far in from the edges that the feather reaches. The effect starts at the depth position and decreases brightness of noise values until it hits the start position at which point the brightness cannot be above zero. The space between the feather start and the edge defaults to a value of 0. The falloff variables, with values between 0.0 and 1.0, determines how quickly the feather fades into black. The effects of the feather function can be seen in figure 4.4.

After the feather function, the cutoff function is applied. The cutoff function has two inputs:

- $CutoffLow$
- $CutoffHigh$

This function decreases the range of values by creating a floor and ceiling and transforming intermediate values to fit the new range. A lower ceiling makes the brights brighter while a higher floor makes the darks darker. This effect can be seen in 4.5.

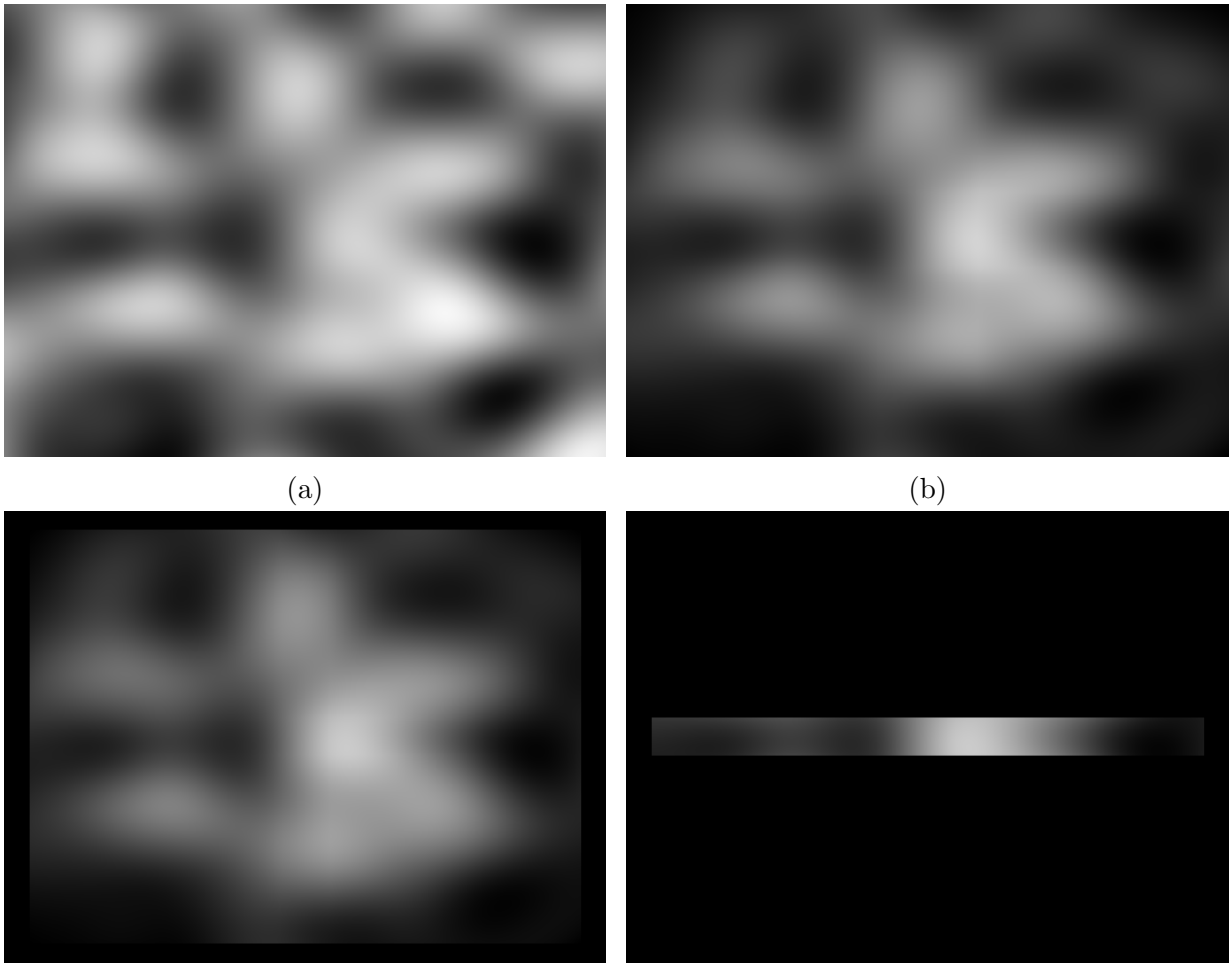


(a) (b) (c)

Figure 4.2: Pictures of noise at different increments at a width of 480 and height of 480. (a) X-Increment:0.005 Y-Increment:0.005 (b) X-Increment:0.001 Y-Increment:0.005 (c) X-Increment:0.001 Y-Increment:0.001

(a) (b)

Figure 4.3: Pictures of the effect of the move values. (a) Same as 4.2a, no move (b) 'a' with X-Move:10.0,Y-Move:-10.0



(a) (b)

(c) (d)

Figure 4.4: Pictures of the effect of the feather function. (a) Same as 4.2a, no feather (b) 'a' with X-depth:400 Y-depth:400 (c) 'b' with X-start:40 Y-start:40 (d) 'c' with X-falloff:0 Y-falloff:1

(a) (b) (c)

Figure 4.5: Pictures of the effect of the cutoff function. (a) Same as 4.2a, no cutoff (b) 'a' with Low-cutoff:0.3 High-cutoff:1.0 (c) 'a' with Low-cutoff:0.0 High-cutoff:0.7

4.2 Cloud Maker

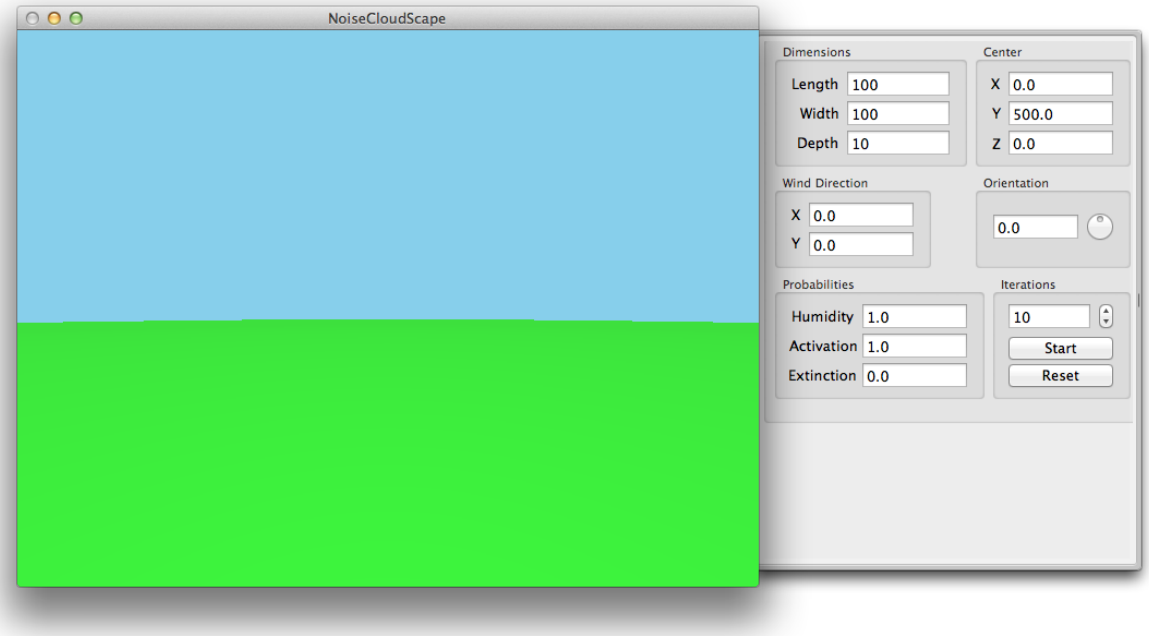


Figure 4.6: The Cloud Maker.

The cloud maker interface, displayed in figure 4.6, allows access to the cloud maker algorithm. This mode allows one to see the default world and fill that world with clouds. The parameters for the cloud maker are described in section 3.4. The user can change the parameters of the cloud maker and when the user is ready, they can run the cloud maker for a set number of iterations. After the initial run, the user can run the cloud maker for additional iterations or reset the cloud maker. The cloud maker uses various OpenGL techniques and five OpenGL vertex/fragment shaders to implement the algorithms (See figure 4.7). An example output can be seen in figure 4.9.

The dimension parameters, length, width, and depth change the size and look of the cloud space which can be see in section 5.1.3.1. The center parameters set the position of the cloud space using a 3D coordinate point. The wind direction is a 2D vector used as a velocity. The orientation rotates the whole cloud space around its own center. The probabilities affect the fullness of the cloud space; in general the lower the humidity, activation, and extinction values the more full the cloud space will be. Once the cloud

maker is started the dimension, position, orientation, and wind direction are locked out from user interaction. If the user wants to change these parameters after starting a render the cloud maker has to be reset and the rendering would have to be restarted.

4.2.1 Modeling

The cloud modeling takes the user created noise textures and extracts a density from them. Equations 3.5-3.9 are run on the GPU and are used to extract the density from the noise textures into the 2D boolean voxel grid of humidity, activation, and cloud exists values. A value of 1 in a cell of the humidity grid means that cell is humid; a 1 in a cell of the activation grid means that cell is able to become a cloud; while a 1 in a cell of the cloud grid means that cell contains a cloud. The modeling only happens a user set number of times. The first iteration starts with all zero values for the 2D boolean voxel grid. The results from each iteration is then passed as the input into the next iteration. Once all iterations have been completed, the cloud exist values from the 2D voxel grid, known as the cloud texture, is passed onto the rendering, shading and drawing steps.

4.2.2 Rendering

The cloud rendering is essentially the same as in [9], but with some differences in the treatment of the metaballs in that the metaballs are much larger in radius to accomplish real-time drawing. Although with large cloudscapes the real-time performance quickly drops off. The clouds are rendered by first finding the density of the clouds on the GPU. This step is a smoothing algorithm that takes the boolean values from the cloud texture and smooths them so that the middle of a cloud is a value of '1' and the edges of the clouds are some value close to '0' with zero being the areas of no clouds. The output is a new 2D texture known as the density texture. The step of turning the 2D density texture into a 3D density texture is not necessary as the algorithms that follow can be made to interpret the 2D texture as a 3D texture by ignoring the z positions of the metaballs.

This density texture is then used by another GPU algorithm to give density to metaballs that are positioned periodically throughout the cloud space. To reduce lattice artifacts every odd horizontal row of metaballs are offset from the even rows of metaballs. Each metaball fills itself with the density of the cells from the density texture that are within the metaball's radius. If a cell within the radius of a metaball is out of range of the density texture, a zero is returned. The returned value of each cell is multiplied by a field function as described in [9] and is summed to give the final density of that metaball which is stored in a 2D texture.

After each metaball is assigned a density value, using the CPU, each is assigned a splat. The number of splats made is 256 each with a dimension of 65x65 stored in a 2D texture (See figure 4.8). The metaballs that have a density value greater than zero are assigned splats equal to their density value. The non-zero density metaballs are then placed into an array for shading and drawing.

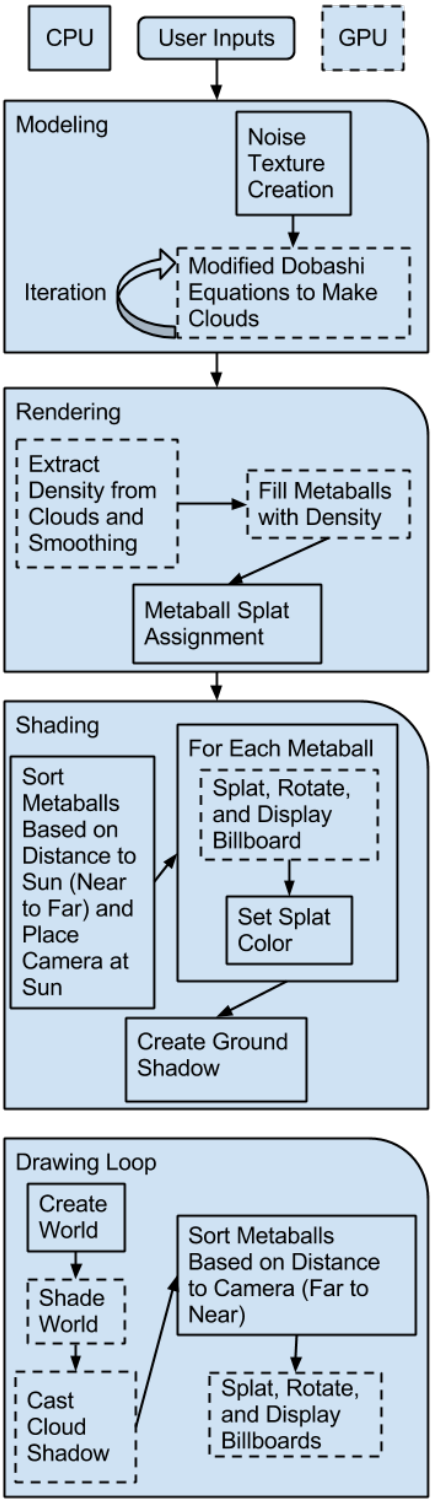


Figure 4.7: A simplified overview of the Cloud Maker pipeline showing the CPU and GPU (vertex/fragment shaders) aspects.

4.2.3 Shading

The shading algorithm only needs to be run once after a render. The shading algorithm starts by sorting the metaballs on the CPU based on the distance from the sun. An orthographic projection is created and the camera is placed at the suns posi-

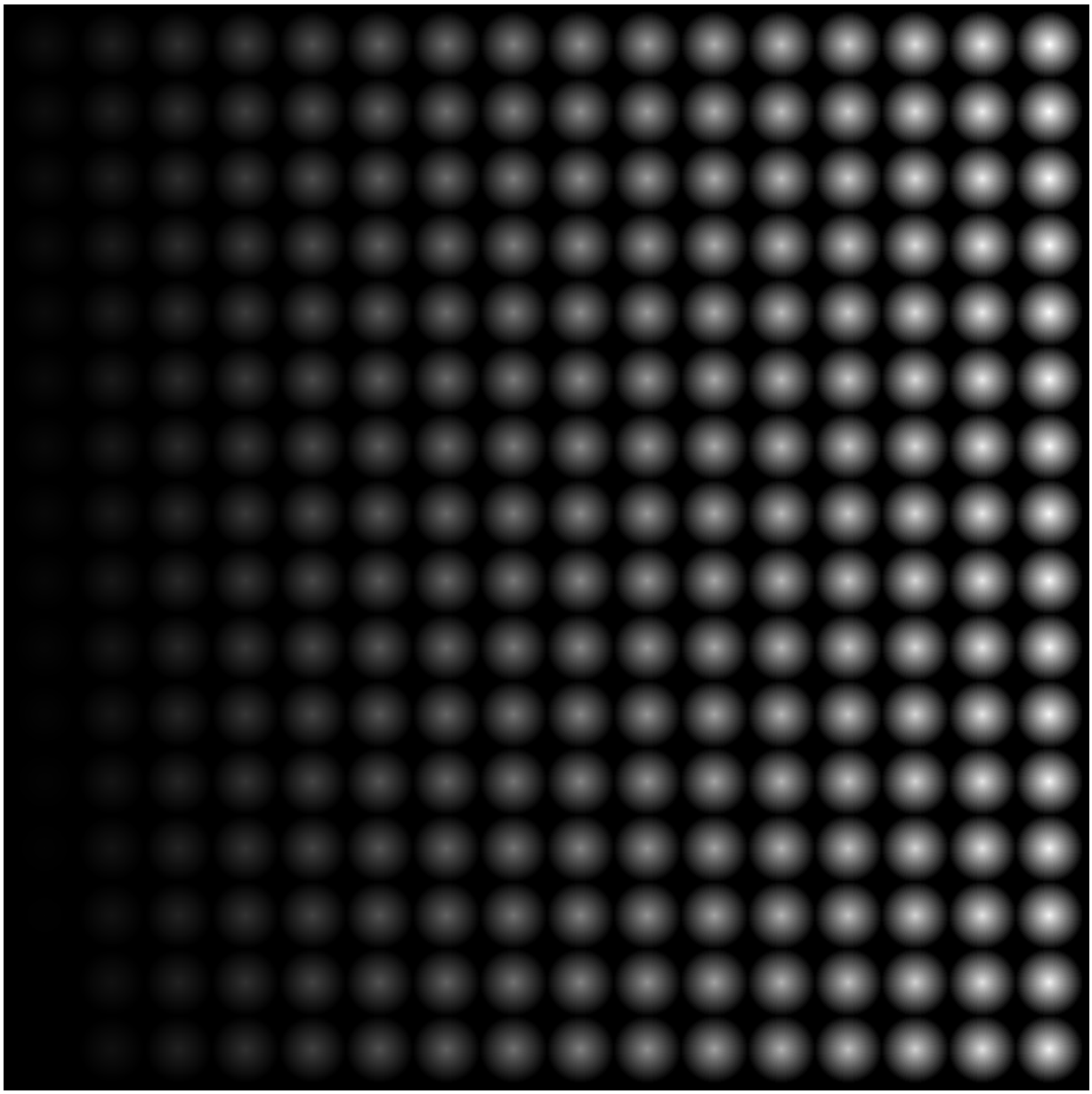


Figure 4.8: The 256 different splats that can be used. Splat size increases from top to bottom, left to right. Displayed on a black background to make the splats visible.

tion. The metaballs are then rendered front to back on the GPU by using their assigned splat to texture a billboard using an OpenGL blending function of (ONE, ONE) and a color of (0.0f,0.0f,0.0f,1.0f / 50.0f). The blend color is an arbitrary color that seems to work well. A blending function defines how an object is drawn over previously drawn objects by specifying how to modify the source and destination color before they are added together (i.e. $(source, destination)$). A blending function of (ONE, ONE) will simply add the incoming color (source) to the color already in the frame buffer (destination). After each metaball is rendered the alpha value of the backing render buffer at the projected coordinate of the metaball is read using the CPU. This value is assigned to the metaball as its color by using it as the rgb components with the alpha component set to 1. The render buffer from the shading process is then copied grid-wise into a 2D texture array using the CPU and is then used by a GPU shading algorithm to shade the ground and create the effect of shadows.

4.2.4 Drawing

Unlike the rendering and shading algorithms, the drawing step happens once every frame refresh. The metaballs are sorted on the CPU based on the distance from the camera. The metaballs are then rendered back to front using the same GPU shader used in the shading process using an OpenGL blending function of (SRC_ALPHA, $ONE\_MINUS\_SRC\_ALPHA$). This blending function will apply the following equation:

$$\begin{aligned}(R_S, G_S, B_S, A_S) &= (R_S, G_S, B_S, A_S) * (\frac{A_S}{k_A}) \\ (R_D, G_D, B_D, A_D) &= (R_D, G_D, B_D, A_D) * (1.0 - \frac{A_S}{k_A}) \\ (R, G, B, A) &= (R_S + R_D, G_S + G_D, B_S + B_D, A_S + A_D)\end{aligned} \tag{4.1}$$

Where letters with a subscript "S" is the source color or the incoming color, and letters with the subscript "D" is the destination color or the color that is in the frame buffer. The

two “k” variables are the maximum value of their subscript, in this case the maximum “A” value. The GPU shader rotates each billboard to face the camera and then renders them by using their assigned splat texture modulated with the assigned color.

Figure 4.9: Clouds using 4.2a as the activation noise, with humidity probability set at 1, activation probability set at 0.8 and extinction probability set at 0. All other values as in 4.6.

Chapter 5

Discussion

The goal of this research is to develop a flexible algorithm in which large-scale repeatable 3D cloudscapes are created easily through algorithms and can move across a multiplayer world dynamically. This leads to eight problems that need to be solved. The first and most basic problem was whether or not noise can be combined with cloud simulation techniques to create a cloudscape. The other problems are: flexibility of the algorithm to create many different cloud patterns, ease of creation of the desired cloud pattern, ability to repeat the cloud patterns, ability to create very large-scale cloud patterns quickly, the ability of the clouds to move across the world dynamically, the problem of realism vs. performance, and the ability of the algorithm to work across a network.

The performance aspect is discussed in the Performance section (5.1.1) of the results. The flexibility of the algorithm and ease of creation of the desired cloud pattern is discussed in the Cloud Type section (5.1.2) of the results. The rest of the problems are discussed in the Conclusion section (5.2). Possible solutions to problems that are not addressed, or that could be improved are discussed in the Future Work section (5.3).

5.1 Results

The results are presented first by showing the performance of the algorithm. Then an exploration of the flexibility of the system is discussed through examples of different cloud types. Finally some idiosyncrasies of the algorithm is discussed.

5.1.1 Performance

The test computer used for the performance is a mid-2009 Mac Book Pro. This computer has a 3.06 GHz Intel Core 2 Duo with 4GB of 1067 MHz DDR3. The GPU is a NVIDIA GeForce 9600M GT GPU with 512 MB of dedicated memory. The system was written in Objective-C using OpenGL techniques and shaders.

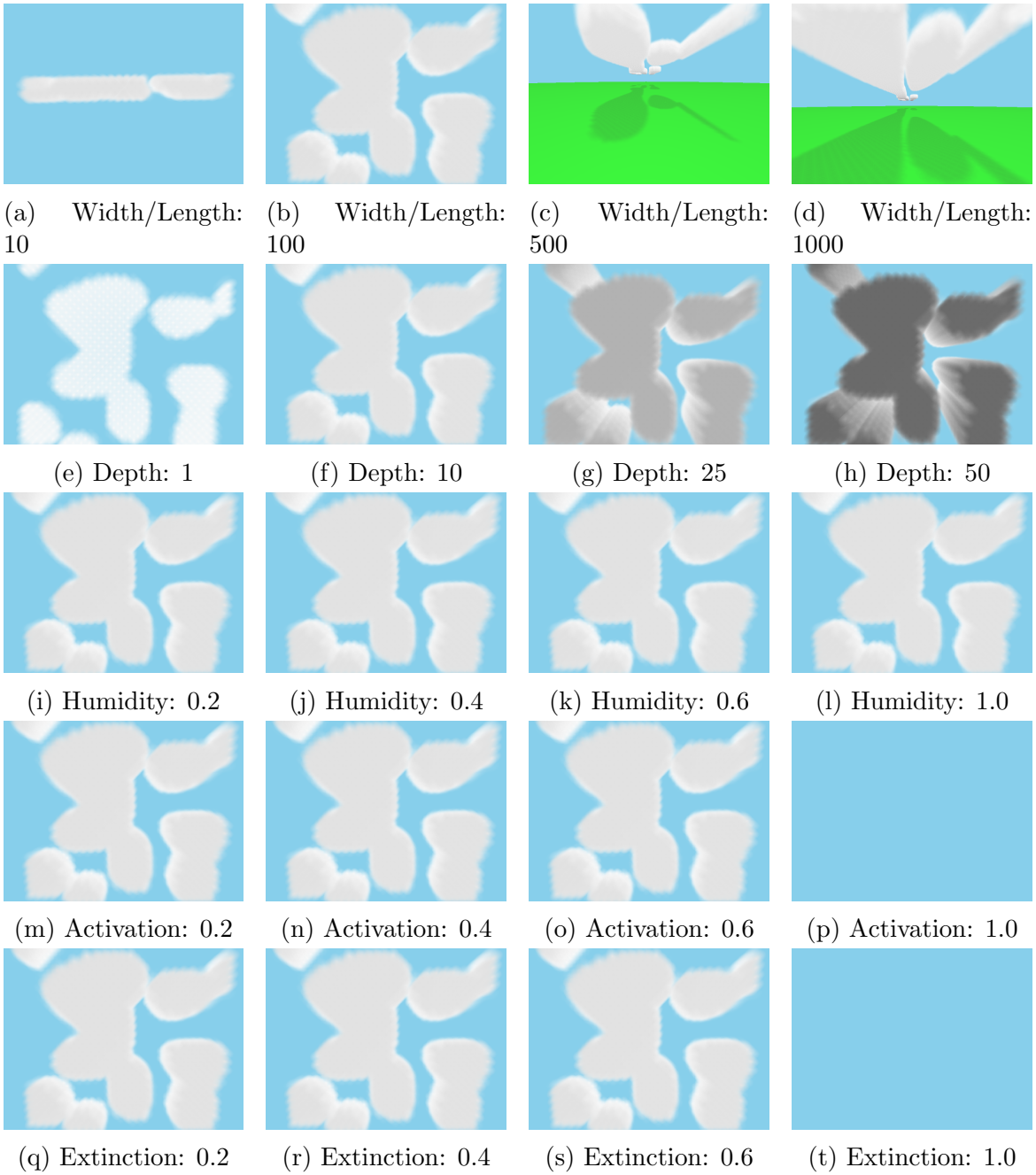
In order to support real-time rendering the metaballs were made to have a large radius of 30. But even with a large radius the real-time performance drops off quickly when the number of metaballs becomes large. All of the simulations that will be presented in the following sub-sections ran at an average of 20 - 30 fps. The other consequence of having large metaballs is a more unrealistic looking cloud: the larger the metaball, the less the collection of metaballs can effectively sample the look and the shape of the clouds.

5.1.1.1 Varying Width, Depth, and Probabilities with Identical Activation and Humidity Noise Patterns

The test in table 5.1 shows the results of having the humidity and activation noise set as the noise in figure 4.1. One variable was varied between two extremes while the other variables were held constant. The value at which each variable is held constant is marked with a "(D)." The variables chosen for this test are: width/length, depth, and the three probabilities. The width/length variable is a special case, only the length was varied while the width was held constant at 100. This is because they are identical in every way except for name. If both width and length were varied together in the width/length row one would expect the 10 column to have a smaller render time, and would expect

the 500 and 1000 column to have a much higher render time and a lower fps. The TTR values is the time to render the cloud space in seconds, this is how quickly the pre-rendering step is able to complete its work. Even though the TTR shows numbers to below microseconds, due to irregularities on the test computer the numbers are only significant to the decisecond. The FPS is the frames per seconds and accounts for the amount of work the drawing algorithm is able to achieve per second.

The biggest variations are in the width/length and depth rows. The larger the width/length or depth the slower the render time and fps become. This was the expected result because as the size of the cloudscape gets bigger so do the number of calculations and the number of on-scene objects. Most of the timings in the humidity, activation and extinction rows are equivalent due to underlying changes in conditions on the test computer. The only major changes in those rows are in the last column of the activation and extinction, where no clouds were displayed in the rendering. The humidity, activation, and extinction are not tested fairly in this test as the activation and humidity textures are set as identical noise patterns.



(a) Width/Length: 10 (b) Width/Length: 100 (c) Width/Length: 500 (d) Width/Length: 1000

(e) Depth: 1 (f) Depth: 10 (g) Depth: 25 (h) Depth: 50

(i) Humidity: 0.2 (j) Humidity: 0.4 (k) Humidity: 0.6 (l) Humidity: 1.0

(m) Activation: 0.2 (n) Activation: 0.4 (o) Activation: 0.6 (p) Activation: 1.0

(q) Extinction: 0.2 (r) Extinction: 0.4 (s) Extinction: 0.6 (t) Extinction: 1.0

Figure 5.1: These images are the renderings associated with the timings from table 5.1. The non-varying probabilities were set at 0.6. The defaults for height/length is 100, with depth at 10.

Table 5.1. Cloud Render Timings #1

Width/Length	a: 10		b: 100 (D)		c: 500		d: 1000	
Timings	TTR (s)	FPS	TTR (s)	FPS	TTR (s)	FPS	TTR (s)	FPS
	0.2786312	60	0.9309526	30	4.2753948	14	4.5196012	5
Depth	e: 1		f: 10 (D)		g: 25		h: 50	
Timings	TTR (s)	FPS	TTR (s)	FPS	TTR (s)	FPS	TTR (s)	FPS
	0.312278	60	0.9309526	30	2.3242614	20	4.3918234	10
Humidity	i: 0.2		j: 0.4		k: 0.6 (D)		l: 1.0	
Timings	TTR (s)	FPS	TTR (s)	FPS	TTR (s)	FPS	TTR (s)	FPS
	0.977354	30	0.9986212	30	0.9309526	30	0.9988256	30
Activation	m: 0.2		n: 0.4		o: 0.6 (D)		p: 1.0	
Timings	TTR (s)	FPS	TTR (s)	FPS	TTR (s)	FPS	TTR (s)	FPS
	0.996737	30	0.9549416	30	0.9309526	30	0.1178964	60
Extinction	q: 0.2		r: 0.4		s: 0.6 (D)		t: 1.0	
Timings	TTR (s)	FPS	TTR (s)	FPS	TTR (s)	FPS	TTR (s)	FPS
	0.9824722	30	0.9909266	30	0.9309526	30	0.0904568	60

Table 5.1: Timings associated with the humidity and activation noise set as the noise in figure 4.1. TTR stands for "Time To Render." FPS stands for "Frames Per Second." The value at which each variable is held constant while another variable is varied is marked with a "(D)." The associated renderings for these results are in figure 5.1

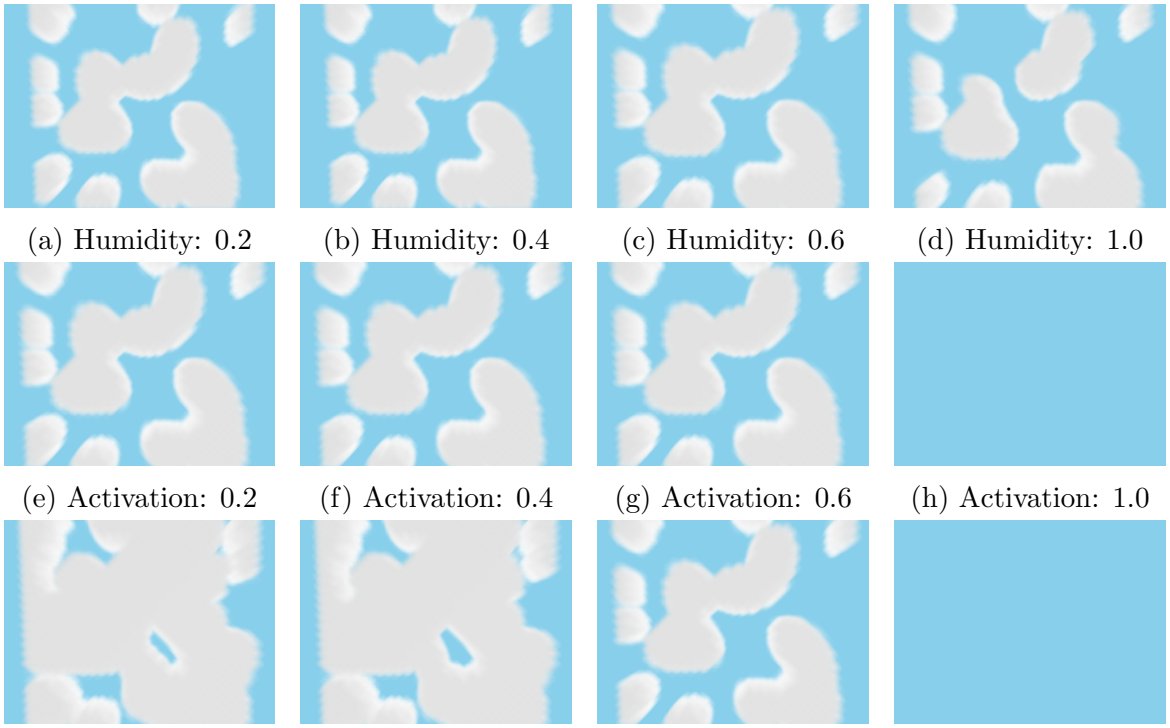
5.1.1.2 Varying Probabilities with Different Activation and Humidity Noise Patterns

The test in table 5.2 uses different noise patterns for the activation and humidity noise textures in order to test the effect of varying the humidity, activation, and extinction probabilities. The pattern used for the humidity noise is the same as in figure 4.3(a). The pattern used for the activation noise is the same as in figure 4.3(b). Width/Length and Depth variables are left out of this test as the same pattern of performance timings as in table 5.1 is expected. The humidity and activation variables don't show too much variation across the row. The 1.0 column shows the best performance for each test, granted the activation and extinction test did not produce any clouds at this level. The extinction test shows the greatest variation between the 0.4 and 0.6 column. This difference is because the extinction probability directly affects the fullness of the cloud space, but this depends on the humidity and activation probabilities. Because the humidity and activation are held at 0.6 in this test the extinction probability has its greatest effect at and over 0.6. The general trend in the timing test is that the lower of each probability, the higher the render time and the lower the fps because the fullness of the cloud space is increased.

Table 5.2. Cloud Render Timings #2

Humidity	a: 0.2		b: 0.4		c: 0.6 (D)		d: 1.0	
Timings	TTR (s)	FPS	TTR (s)	FPS	TTR (s)	FPS	TTR (s)	FPS
	0.852082	40	0.857746	40	0.873429	45	0.7718368	55
Activation	e: 0.2		f: 0.4		g: 0.6 (D)		h: 1.0	
Timings	TTR (s)	FPS	TTR (s)	FPS	TTR (s)	FPS	TTR (s)	FPS
	0.879672	45	0.860712	45	0.873429	45	0.1050106	60
Extinction	i: 0.2		j: 0.4		k: 0.6 (D)		l: 1.0	
Timings	TTR (s)	FPS	TTR (s)	FPS	TTR (s)	FPS	TTR (s)	FPS
	1.390289	30	1.3809486	30	0.873429	45	0.1023174	60

Table 5.2: Timings associated with the humidity noise set as the noise in figure 4.3(a) and the activation noise set as the noise in figure 4.3(b). TTR stands for "Time To Render." FPS stands for "Frames Per Second." The value at which each variable is held constant while another variable is varied is marked with a "(D)." This test was run with a width and length of 100 and a depth of 10. The associated renderings for these results are in figure 5.2



(a) Humidity: 0.2 (b) Humidity: 0.4 (c) Humidity: 0.6 (d) Humidity: 1.0

(e) Activation: 0.2 (f) Activation: 0.4 (g) Activation: 0.6 (h) Activation: 1.0

(i) Extinction: 0.2 (j) Extinction: 0.4 (k) Extinction: 0.6 (l) Extinction: 1.0

Figure 5.2: These images are the renderings associated with the timings from table 5.2. The non-varying probabilities were set at 0.6. Height and length are 100, with depth at 10.

5.1.1.3 Varying Activation and Humidity Probabilities with Different Activation and Humidity Noise Patterns

The last test showed the influence of extinction, but did not fully show the influence of humidity and activation. The next test will look solely at these two variables with extinction set to 0.0. As can be seen in table 5.3, the general trend of lower values leading to higher render times is still present. All of the fps cells show the same value of 30. This shows that the extinction has a large effect on the number of clouds in the cloud space, which can also be seen in the screenshots in figure 5.3 as compared to those in figure 5.2.

Table 5.3. Cloud Render Timings #3

Humidity	a: 0.0		b: 0.2		c: 0.4	
Timings	TTR (s)	FPS	TTR (s)	FPS	TTR (s)	FPS
	1.5607756	30	1.5715568	30	1.5457268	30
Humidity	d: 0.6		f: 0.8		g: 1.0	
Timings	TTR (s)	FPS	TTR (s)	FPS	TTR (s)	FPS
	1.4432502	30	1.2820044	30	1.1762558	30
Activation	h: 0.0		i: 0.2		j: 0.4	
Timings	TTR (s)	FPS	TTR (s)	FPS	TTR (s)	FPS
	1.5863902	30	1.583095	30	1.458362	30
Activation	k: 0.6		l: 0.8		m: 1.0	
Timings	TTR (s)	FPS	TTR (s)	FPS	TTR (s)	FPS
	1.395469	30	1.0524488	30	-	-

Table 5.3: Timings associated with the humidity noise set as the noise in figure 4.3(a) and the activation noise set as the noise in figure 4.3(b). TTR stands for "Time To Render." FPS stands for "Frames Per Second." The value at which each variable is held constant is 0.5, with extinction set at 0.0. This test was run with a width and length of 100 and a depth of 10. The '1.0' test for the activation rendered no clouds. The associated renderings for these results are in figure 5.3

(a) Humidity: 0.0 (b) Humidity: 0.2 (c) Humidity: 0.4

(d) Humidity: 0.6 (e) Humidity: 0.8 (f) Humidity: 1.0

(g) Activation: 0.0 (h) Activation: 0.2 (i) Activation: 0.4

(j) Activation: 0.6 (k) Activation: 0.8 (l) Activation: 1.0

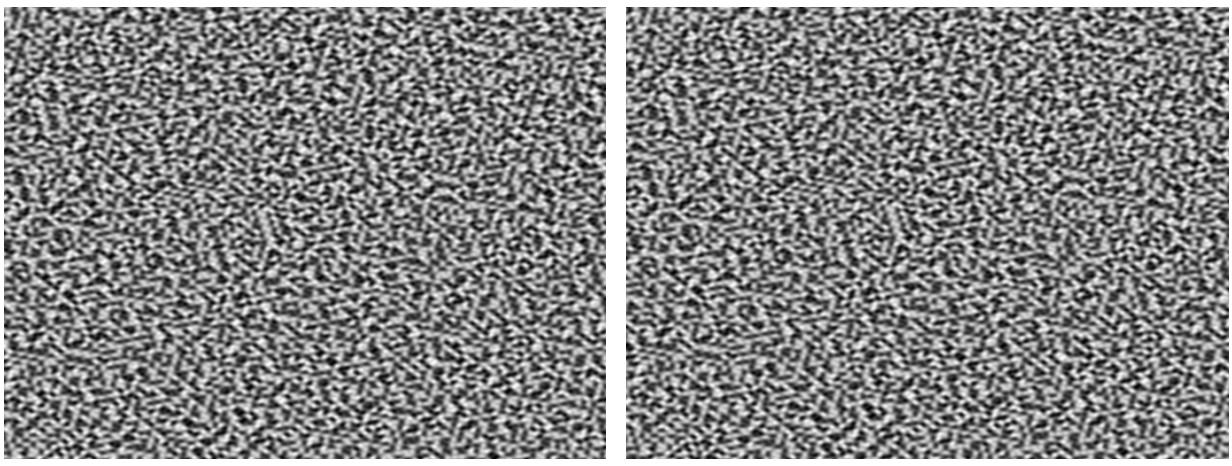
Figure 5.3: These images are the renderings associated with the timings from table 5.3. The non-varying probability is set at 0.5 with extinction set at 0.0.

5.1.2 Cloud Types

Any cloud type that has a distinctive vertical development such as the development of cumulus to cumulonimbus cannot be represented by this system. Other cloud types that have more constrained vertical development such as stratus and subtypes of stratocumulus simulated with this system. High altitude clouds are very difficult to simulate due to their extreme thinness and wispiness. The following sections will describe how to make many cloud types with this algorithm. Although a specific way to make the cloud will be described there may be many other ways to make the same cloud type.

Creating the noise for each of the simulations presented in this sub-section was made in less than ten minutes, averaging around five minutes. The altocumulus took the longest and surprised me when I decided to use a very noising pattern with the humidity: it makes the clouds seen fluffier in a much more random way.

5.1.2.1 Stratus Cloud



(a) (b)

Figure 5.4: Pictures of the (a) humidity (with X-increment and Y-increment of 0.1 and moved 0.5 increments both the X and Y direction) and (b) activation (with the same X and Y increment as humidity) used to make stratus clouds in figure 5.5.

Stratus clouds are typically what are associated with a "hazy day." They are low altitude clouds that are flat, hazy and mostly featureless with some variation in shading. These clouds were made using an activation noise of 0.1 X-increment and 0.1 Y-increment

with a humidity noise of the same increment but moved 0.5 increments in both the X and Y direction (See figure 5.4). The clouds were made with a dimension of 500 by 500 by 5 with humidity and activation probabilities of 0.6 (See figure 5.5). The only real issue with the simulation is the lack in variation of shading and the lack of a hazy appearance. To show that noise used for one cloud type can also be used for another, figure 5.6 uses the noise used to create the altocumulus simulation presented below in figure 5.10. Figure 5.6 also has an accompanying video named "stratusThree.mov" on the included CD. In this video one can see some lattice artifacts, the camera moving through the cloudscape, and the slowness of the drawing loop which ranged from 2 - 5 frames per second.

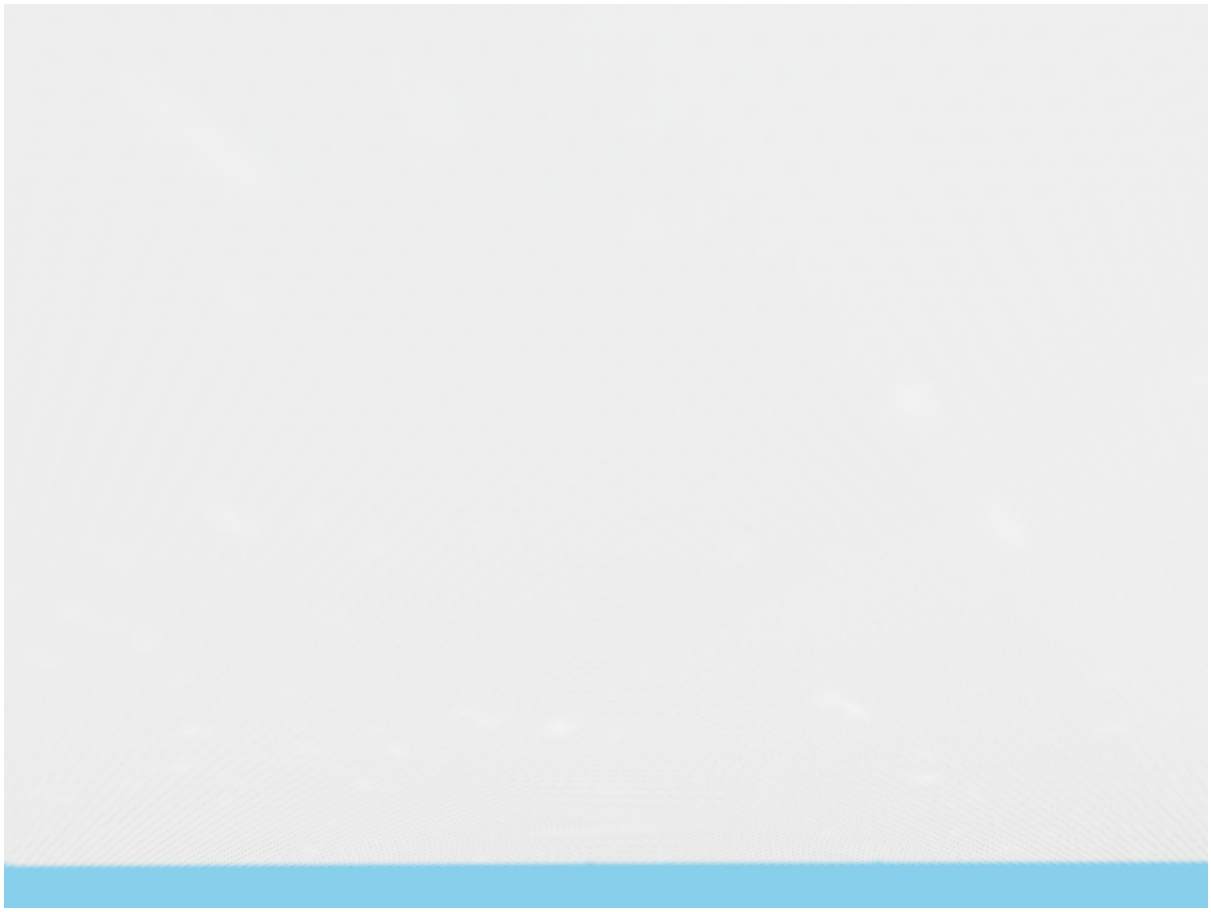


Figure 5.5: Picture of stratus clouds made from noise in figure 5.4. 500 by 500 by 5 with humidity and activation probability of 0.6.

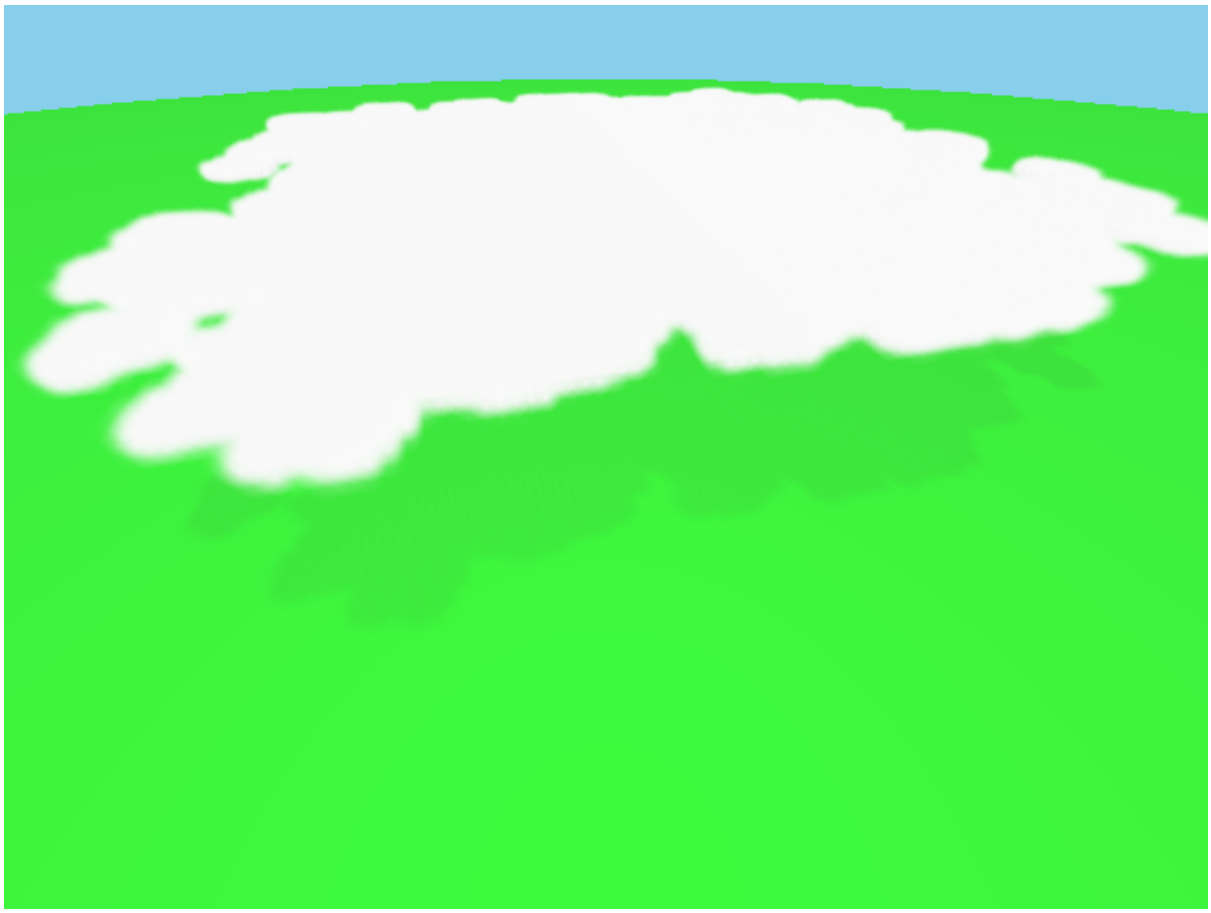
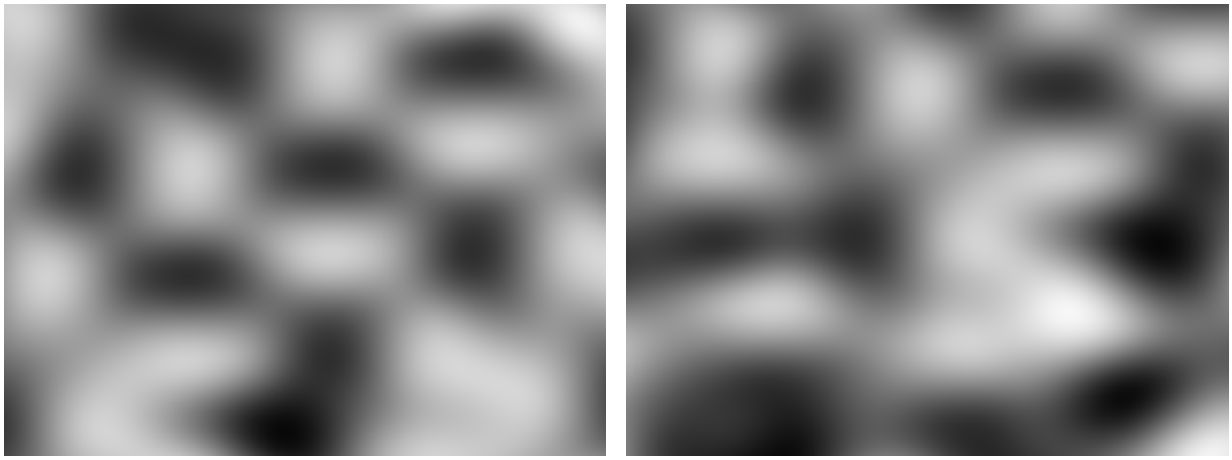


Figure 5.6: Picture of a stratus cloud that is large made from noise in figure 5.10. 500 by 500 by 4 with humidity probability at 0.5 and activation probability of 0.6. There is also an accompanying video named "stratusThree.mov" on the included CD.

5.1.2.2 Stratocumulus

There are many variations of stratocumulus clouds, many of which have vertical development and shapes that cannot be represented with this algorithm. But the horizontal variations could be simulated. In general stratocumulus are denser than stratus so that they have a much darker shading of each individual cloud. They can be either a large overcast cloud cover or partly cloudy. (See figure 5.7-5.9).



(a) (b)

Figure 5.7: Pictures of the (a) humidity (with X-increment and Y-increment of 0.005 and moved 1.0 increments both the X and Y direction) and (b) activation (with the same X and Y increment as humidity) used to make stratocumulus clouds in figure 5.8.

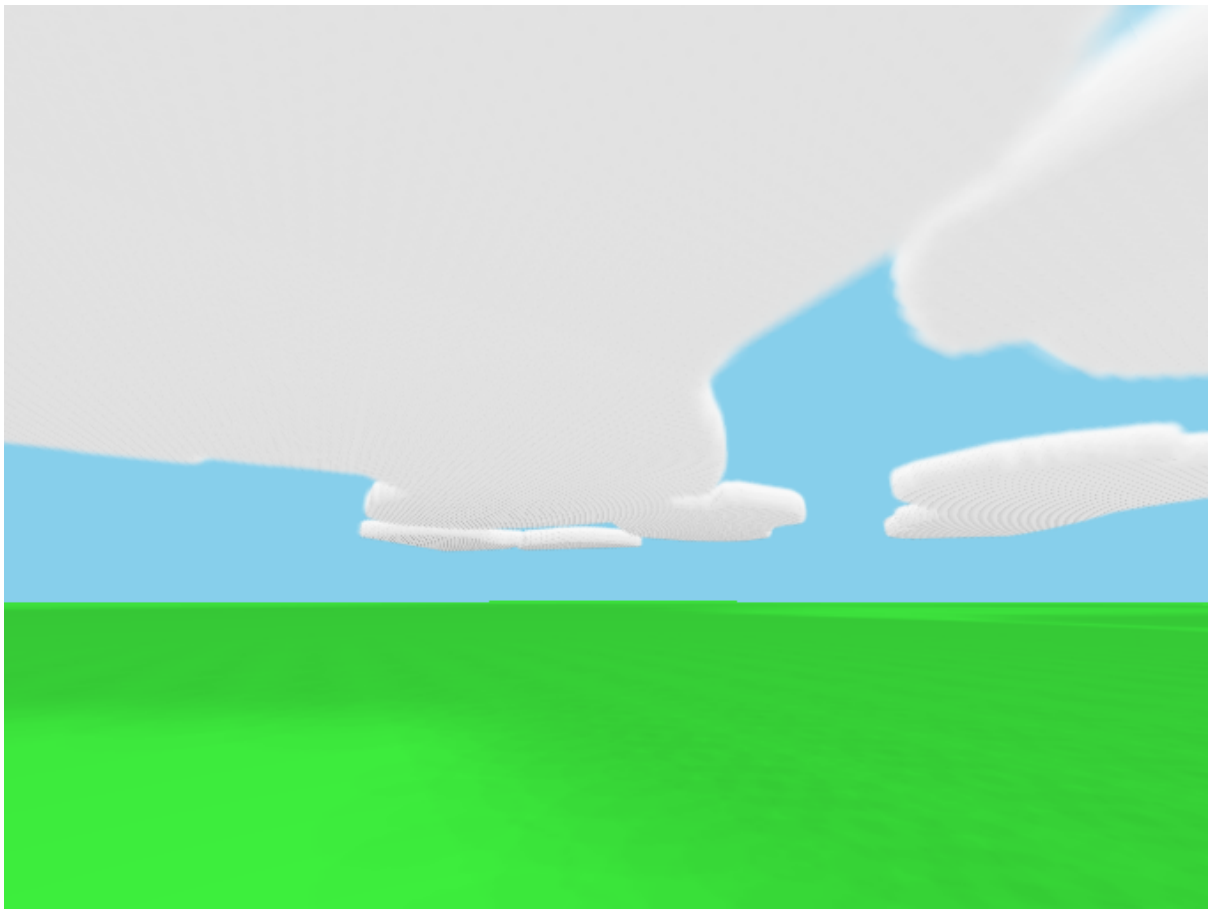


Figure 5.8: Picture of stratocumulus clouds that are large but show lots of sky in between them made from noise in figure 5.7. 500 by 500 by 10 with humidity probability at 0.3 and activation probability of 0.6.

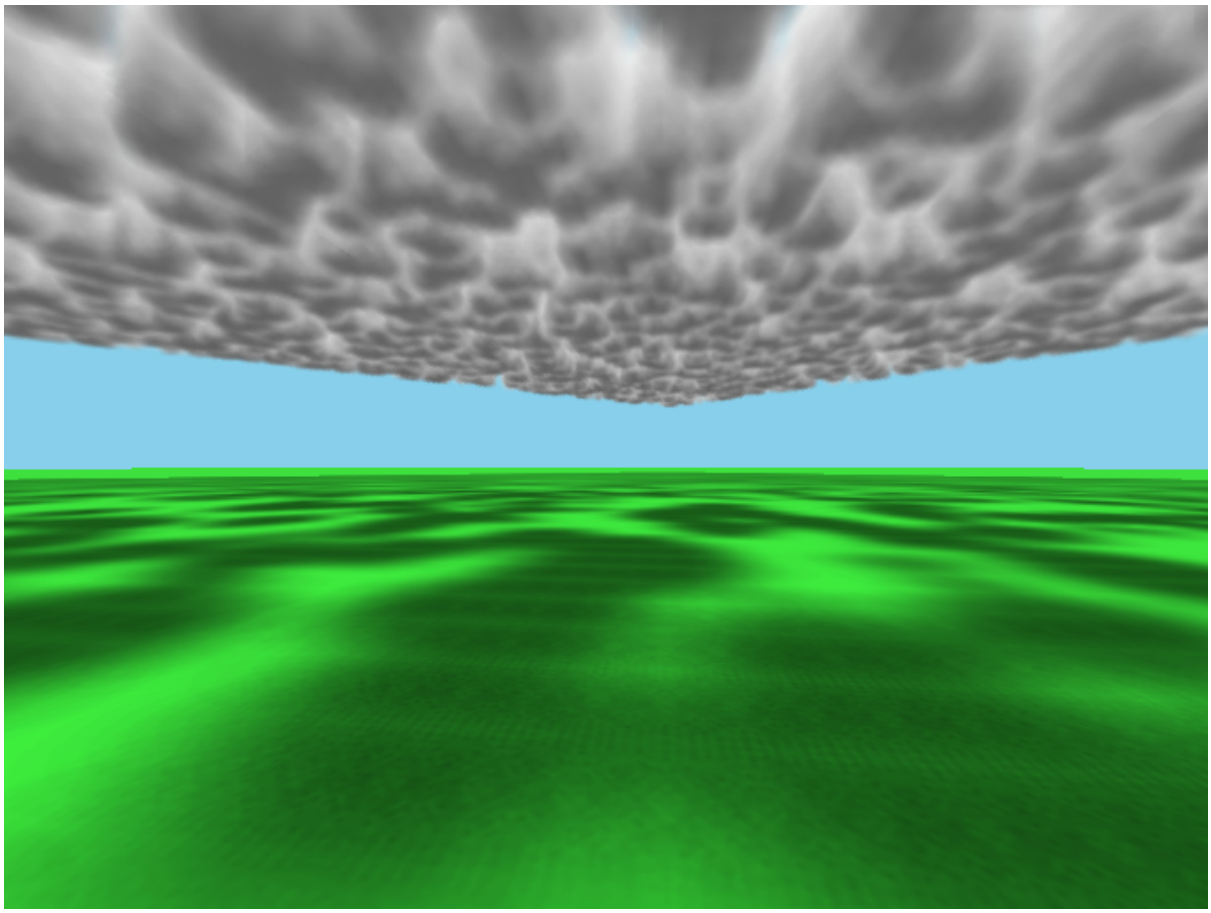
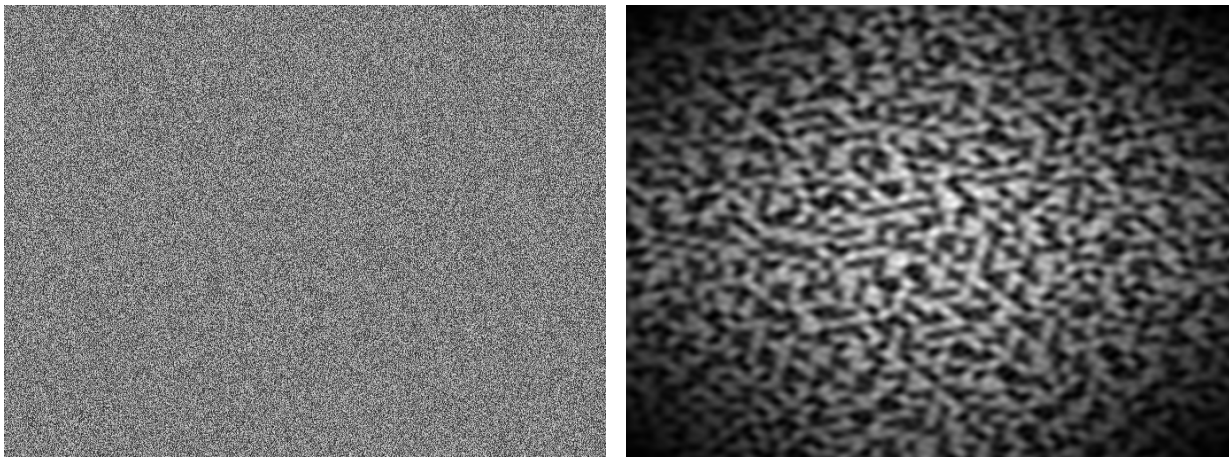


Figure 5.9: Picture of stratocumulus clouds that are large, dark and show little to no sky in between them made from noise in figure 5.4(b). 500 by 500 by 50 with humidity probability at 1.0 and activation probability of 0.8. These clouds would look strange from the side due to the shape problem discussed in the introduction to the results section

5.1.2.3 Altocumulus

Altocumulus are similar to Cumulus so many variants can be created easily. The most difficult variant is the thin-altocumulus. They look very flat and thin from the ground and so the shade of the clouds are more blue than thicker clouds (See figure 5.10-5.12). The main problem with this simulation is that you can see individual metaballs and the gridlines in between them, know as lattice artifacts, when the camera is rotated in specific ways as in figure 5.11. These artifacts can be seen in figure 5.12 but they are less obvious. These lattice artifacts will remain as long as a metaballs are used for the rendering; even in the case of the algorithm presented, where the metaballs are offset from each other every other depth level. The altocumulus images in figures 5.11-5.12 were drawn at around 3 to 5 frames per second due to the large size of the cloudscape. Ideas for increasing the FPS of the drawing loop is discussed more in section 5.3.



(a) (b)

Figure 5.10: Picture of the (a) humidity (with x and y increments of 1.0) and (b) activation (with x and y increments of 0.05 and x and y feather depths of 400) used to make altocumulus in figure 5.11

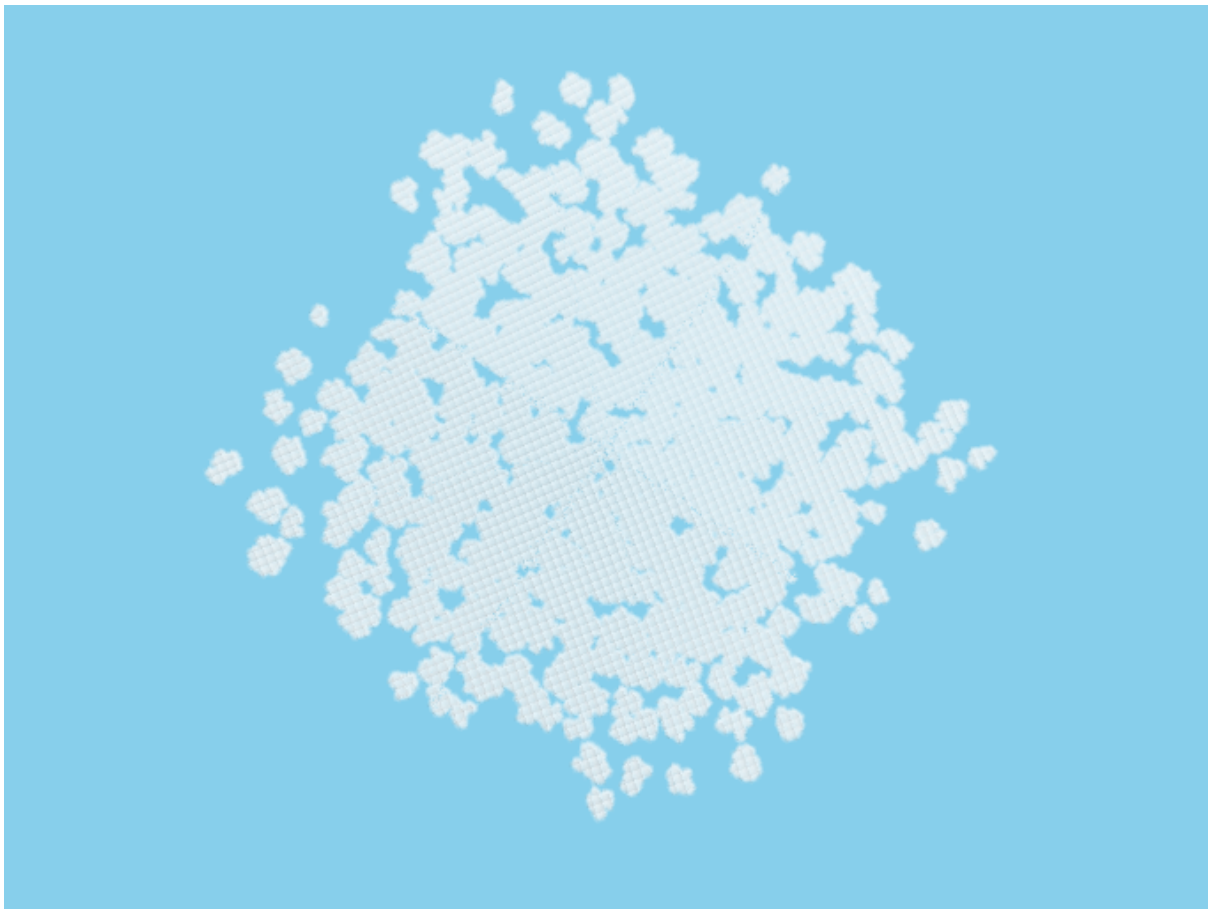


Figure 5.11: Picture of altocumulus with humidity probability of 0.5 and activation probability of 0.6 with dimensions of 1000 x 1000 x 2 and a position at a height of 5000. Made with noises in figure 5.10

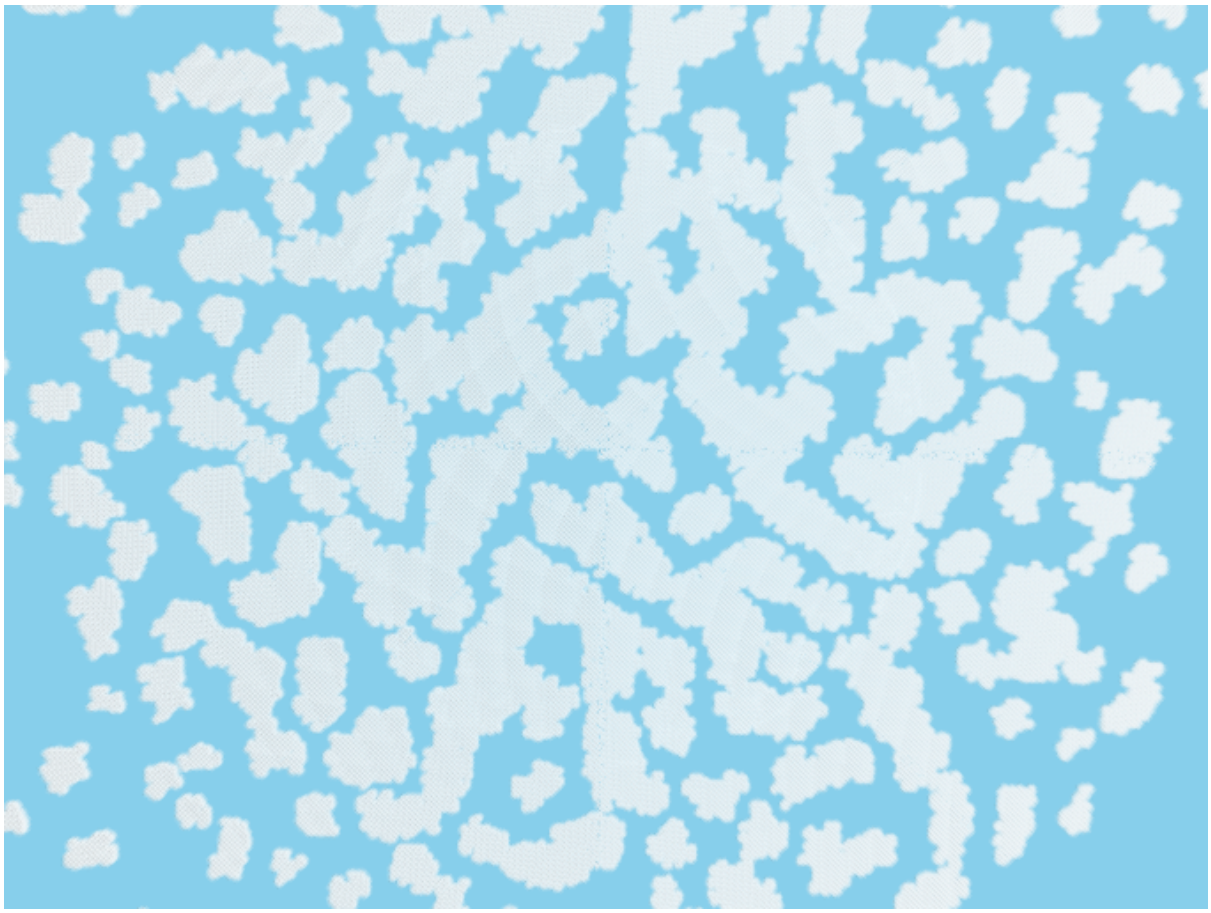
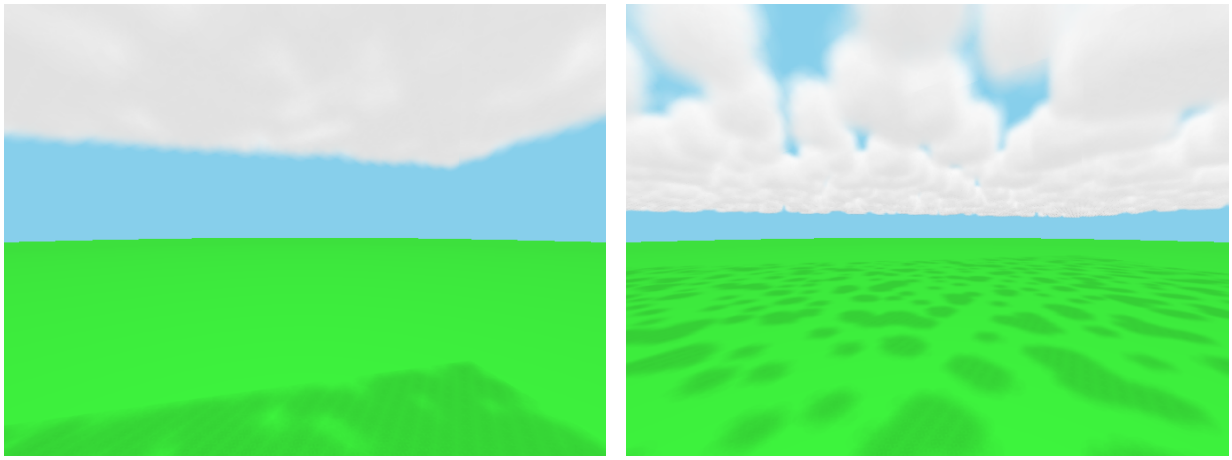


Figure 5.12: Picture of altocumulus with humidity probability of 0.5 and activation probability of 0.6 with dimensions of 2000 x 2000 x 2 and a position at a height of 5000. Made with noises in figure 5.10

5.1.3 Idiosyncrasies of the Algorithm

5.1.3.1 Texture Size

The noise textures are always of the same dimension. The user has no control over this aspect. A cloudscape larger than the texture dimension simply means that the texture is stretched to fit the larger size and each individual cloud is larger. Cloudscapes using specific noises that appear to be overcast at small scales may be scattered clouds at large scales (See figure 5.13). This is due to the sampling differences at the larger scales and the way in which the boolean logic works. At larger scales there are more metaballs between clouds. Meaning that clouds are less likely to share humidity and thus less likely to 'connect' to each other.



(a) (b)

Figure 5.13: Pictures of clouds with activation noise increment at 0.05 and activation probability at 0.8. (a) is 100 by 100 and (b) is 500 by 500.

5.1.3.2 Vertical Development

While the realistic look of the shading is effected by the size of the metaballs, the shape of the clouds is effected by the use of noise textures. Currently the same humidity and activation noise texture is used for every height level of the cloud. This means that the bottom of a cloud and the top of the cloud have the same shape, which makes clouds that look strange (See figure 5.14). Clouds typically have smaller tops then they do bottoms, similar to mountains.

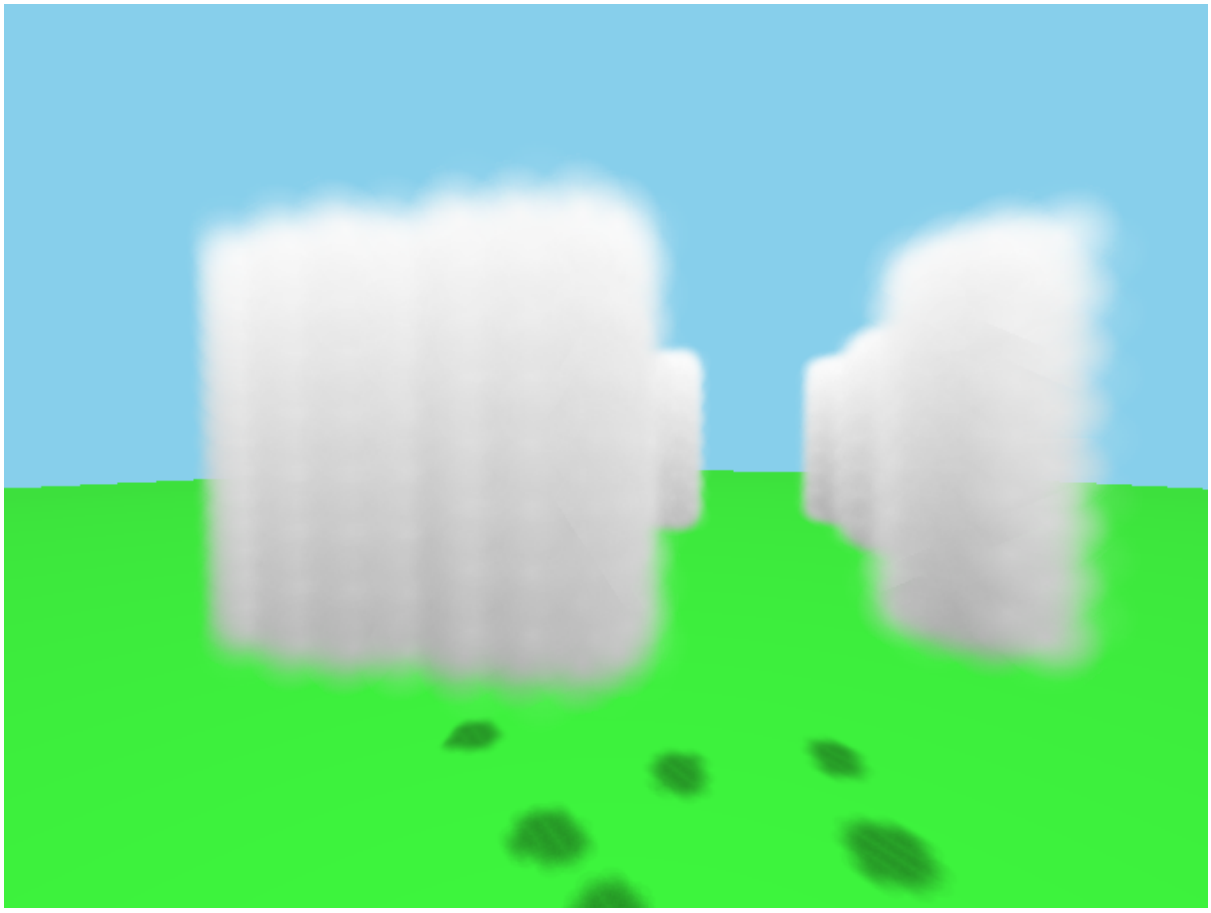


Figure 5.14: The top and the bottom of the clouds have the same shape.

5.2 Conclusions

Creating clouds using noise textures in combination with the techniques from [9] has proven to be quite flexible at simulating certain cloud patterns as can be seen in section 5.1.2. But it is unable to simulate many more cloud patterns. There are many different combinations of options to select from in this algorithm to create many different patterns; many of which produce great results.

The ease of use of the algorithm could be improved. It can be confusing, at times, trying to work with the many options if one does not understand how the humidity, activation and extinction work together.

The ability to repeat the cloud patterns is solved by using noise as the underlying randomness. Using noise also partially solves the problem of creating large-scale cloudscapes quickly; unfortunately the drawing method is too slow to create large-scale cloudscapes at real-time performance (5.1.1), and many improvements can be made to add to the realism. Moving the clouds across the world was only addressed with a simple wind vector. More work will need to be done to create a more complete algorithm to move the clouds around the world in a much more natural way.

On the problem of realism vs. performance, the algorithm is much too slow to be used in any system that requires the simulation of more than just clouds. This slowness is due to the large number of metaballs that need to be sorted based on the camera position, rotated to face the camera, and then drawn one by one. The other problem is that all metaballs, except those with no density, are drawn, regardless of whether they are visible or have been completely obscured through the blending algorithm.

One problem that was not address is the server/client aspect. Though I see this as more of an implementation detail, it does have some implications for the above problems. Such as solving aspects of the problem of performance, since the entire modeling and rendering pipeline would be moved to the server. A server would also allow for the clouds to be more dynamic, as described in the Introduction, as the server can continuously run

the modeling and rendering of the clouds and send updates to the clients. A server does add the normal problems of network based systems that would have to be addressed.

From the results presented it can be seen that the cloudscapes look very convincing for very little effort from the user. And even though there is much to be improved with several aspects of the clouds, the basic problem of using noise to define the cloudscape through combination with a cloud simulation algorithm has been shown to work quite well.

5.3 Future Work

The cloud rendering and drawing mechanism need to be improved by intelligently deciding which metaballs are displayed to the user, and which can be safely culled. An algorithm for picking out individual clouds close to the camera and splatting them to small billboards would also be helpful. Farther away clouds can be splatted onto large billboards placed around the viewers area. This would reduce the number of on-scene objects, but whether or not it will be faster will have to be tested.

The shading of the clouds is dependent on an arbitrary blending color of (0.0f,0.0f,0.0f,1.0f / 50.0f). This should be changed to allow the user to set it. But should it just be one value for the whole cloudscape? Should it also be a noise texture? This would require some testing. The idea of using a noise texture to affect shading would be interesting. The size of real clouds doesn't necessarily effect the shading. In the density algorithm in this thesis it is assumed that if all the voxels around you are activated, then you are dense. If a texture defined the denseness than the shading of a cloud would not be so uniform as it is now, though whether or not this would effect the realism would have to be seen.

The shape of the clouds can be improved by allowing the user to specify different noise textures to be used at different heights. The system would then interpolate between the noise textures to fill in the intermediate heights allowing for a smooth transition. This

would give the clouds a more natural and organic look.

Another improvement would be to allow for the creation of oddly shaped noise textures. Currently since the textures are square, the cloudscapes are rendered as large squares. This looks very strange when the cloudscape is small or the cloudscape is high in the atmosphere as can be see in figure 5.11. This improvement could be implemented by allowing one to draw an arbitrary feather edge in the noise maker.

Users should be able to select from several predetermined settings that make clouds of the most common types and then allow the user to customize those settings. Allowing the user to create multiple separate cloud layers would also be a desired improvement. Unlike the improvement mentions above using multiple textures to define a cloud single cloudscape, this improvement would produce a whole new cloudscape that is separate and will have its own unique look and feel. This would be very desired because when one looks up at the sky many different cloud layers and cloud types can be present at the same time. This improvement would give a big boost to the realism of a virtual world.

One desired addition to the system would be to add the ability to render the cloudscape on a server, and then move the rendered information to the client computer. This way, the client computer can work on just drawing the cloudscape, while the server can focus on rendering and morphing the clouds. This would work well for multi-player games that have multiple users on the same server, within the same area of a world.

Chapter 6

References

1. W. Reeves. “Particle Systems - A Technique for Modeling a Class of Fuzzy Objects”. ACM Transactions on Graphics, vol. 2, no. 2, April 1983.

2. Ken Perlin. “An Image Synthesizer”. SIGGRAPH 1985, pp. 287-296.

3. Ken Perlin and Eric M. Hoffert. “Hypertexture”. SIGGRAPH 1989, pp. 253-262.

4. Ken Perlin. “Improving Noise”. SIGGRAPH 2002, pp. 681-682.

5. Ken Perlin. “Noise hardware”. In Real-Time Shading SIGGRAPH Course Notes. M.Olano, Ed. ch. 9. 2001.

6. A. Kensler, A. Knoll, and P. Shirley. “Better gradient noise”. Tech. Rep. UUSCI-2008-001, SCI Institute, University of Utah. 2008.

7. D. S. Ebert, F. K. Musgrave, D. Peachey, K. Perlin, and S. Worley. Texturing and Modeling; A Procedural Approach, Third Edition. Morgan Kaufmann, San Francisco.

8. Stefan Gustavson. “Simplex noise demystified”. Linköping University, Sweden. March 2005.

9. Y. Dobashi, K. Kaneda, H. Yamashita, T. Okita, and T. Nishita. "A Simple, Efficient Method for Realistic Animation of Clouds". SIGGRAPH 2000, pp. 19-28.

10. T. Nishita, Y. Dobashi, and E. Nakamae. "Display of clouds taking into account multiple anisotropic scattering and sky light". International Conference on Computer Graphics and Interactive Techniques, 1996. pp. 379-386.

11. Mark J. Harris and Anselmo Lastra. "Real-Time Cloud Rendering". EUROGRAPHICS 2001, vol. 20, no. 3.

12. M. J. Harris, W. V. Baxter, T. Scheuerman, and A. Lastra, "Simulation of cloud dynamics on graphics hardware". Graphics Hardware, 2003. pp. 92-101.

13. Joshua Schpok, Joseph Simons, David S. Ebert, and Charles Hansen. "A Real-Time Cloud Modeling, Rendering, and Animation System". EUROGRAPHICS 2003. pp. 160–166.

14. Chung-Min Yu and Chung-Ming Wang. "An Effective Framework for Cloud Modeling, Rendering, and Morphing". Journal of Information Science and Engineering. vol 27. pp. 891-913. 2001.

15. J. Kajiya and B. V. Herzen, "Ray tracing volume densities". International Conference on Computer Graphics and Interactive Techniques, 1984. pp. 165-174.

16. D. Overby, Z. Melek, and J. Keyser, "Interactive physically-based cloud simulation". Pacific Conference on Computer Graphics Applications, 2002. pp. 469-470.

17. G. Y. Gardner. "Visual simulation of clouds". Conference on Computer Graphics and Interactive Techniques. vol. 19. pp. 297-304. 1985.

18. A. Bouthors, F. Neyret, N. Max, E. Brunet, and C. Crassin. "Interactive multiple anisotropic scattering in clouds". ACM Symposium on Interactive 3D Graphics and Games, 2008. pp. 173-182.

19. A. Bouthors and A. Neyret, “Modeling clouds shape”. Computer Graphics Forum 2004. pp. 1-4.

20. N. Wang. “Realistic and fast cloud rendering in computer games”. ACM SIGGRAPH Conference on Sketch and Applications, 2003. pp. 27-30.

21. N. Wang. “Realistic and fast cloud rendering”. Journal of Graphics Tools. vol. 9. pp. 21-40. 2004.

22. H. S. Liao, T. C. Ho, J. H. Chung, and C. C. Lin. “Fast rendering of dynamic clouds”. Computer and Graphics. vol. 29. pp. 29-40. 2005.

23. Roland Hufnagel, Martin Held, and Florian Schröder. “Large-scale, realistic cloud visualization based on weather forecast data”. CGIM 2007. pp. 54-59.

24. M. J. Harris. “Real-Time Cloud Simulation and Rendering”. Dissertation, UNC Chapel Hill, 2003

25. Oliver Koehler. “Interactive simulation of clouds based on fluid dynamics”. Thesis, University of Koblenz-Landau, 2009

26. J. P. Lewis “Algorithms for solid noise synthesis”. In Proceedings of SIGGRAPH 1989, pages 263–270.

27. D. S. Ebert “Volumetric modeling with implicit functions: a cloud is born.” In ACM SIGGRAPH 97 Visual Proceedings: The art and interdisciplinary programs of SIGGRAPH ’97, page 245.

28. D. S. Ebert and R. E. Parent. “Rendering and animation of gaseous phenomena by combining fast volume and scanline A-buffer techniques.” In Proceedings of SIGGRAPH 1990, pages 357–366.

29. P. Elinas and W. Stürzlinger “Real-time rendering of 3D clouds”. The Journal of Graphics Tools, 5(4):33–45.

30. L. Westover "Footprint evaluation for volume rendering". In Proceedings of SIGGRAPH 1990, pages 367–376.

31. Y. Dobashi, T. Nishita, H. Yamashita, and T. Okita. "Using metaballs to modeling and animate clouds from satellite images". The Visual Computer, 15:471–482.

32. R. Miyazaki, Y. Dobashi, and T. Nishita. "Simulation of cumuliform clouds based on computational fluid dynamics". In Proceedings EURO- GRAPHICS 2002 Short Presentations, 2002.

33. R. Miyazaki, S. Yoshida, Y. Dobashi, and T. Nishita. "A method for modeling clouds based on atmospheric fluid dynamics". In Proceedings Pacific Graphics, pages 363-372, 2001.

34. K. Nagel and E. Raschke. "Self-organizing criticality in cloud formation?" Physica A, 182:519–531.

35. J. F. Blinn. "A generalization of algebraic surface drawing". In Proceedings of SIGGRAPH 1982, pages 273–274.

36. J. F. Blinn. "Light reflection functions for simulation of clouds and dusty surfaces". In Proceedings of SIGGRAPH 1982, Computer Graphics, pages 21–29.

37. William R. Cotton, George H. Bryan, Susan C. van den Heever. Storm and Cloud Dynamics. Elsevier Science & Technology, Saint Louis, MO, USA.

38. J. Kniss, S. Premože, C. Hansen, and D. S. Ebert. "Interactive translucent volume rendering and procedural modeling". In Proceedings of IEEE Visualization 2002, pages 109–116.