2011

Scientific visualization using Pixar's RenderMan

John Lukasiewicz

Scientific Visualization Using Pixar's RenderMan

John Lukasiewicz
Computer Science M.S. Thesis
Department of Computer Science
Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York

June 29, 2011

Approved by:

Advisor: Professor Hans-Peter Bischof, Ph.D.

Reader: Professor Reynold Bailey, Ph.D.

Observer: Professor Joe Geigel, Ph.D.

Abstract

This thesis will attempt to visualize astrophysical data that is proprocessed and formatted by the Spiegel software using Pixar's RenderMan. The output will consist of a large set of points and data associated with each point. The goal is to create images that are both informative and aesthetically pleasing to the viewer. This has been done many times before with software rendering and APIs such as OpenGL or JOGL. This thesis will use Pixar's Photorealistic RenderMan, or PRMan for short, as a renderer. PRMan is an industry proven standard renderer that is based on the RenderMan Interface Specification which has been in development since 1989. The original version was released in September of 1989 and the latest specification, version 3.2 was published in 2005.

Since aesthetics is a subjective quality based on the viewers' preference, the only way to determine if an image is aesthetically pleasing is to survey a general population. The thesis includes an experiment to assess the quality of the new renders.

Acknowledgements

Thanks goes out to my Advisor, Dr. Hans-Peter Bischof for allowing me to work on the Spiegel project and pushing me to do my best while working on the project. I would also like to thank all the Professors who taught computer graphics, including Professor Joe Geigel, Professor Reynold Bailey and Professor Warren Carithers for giving me the basis and fundamental knowledge which enabled me to write this thesis.

I would also like to acknowledge my friends who I was able to run ideas back and fourth and get feedback for the render images and progress I was making. I would also like to acknowledge the students in the REU program who wrote the initial RenderMan framework in Spiegel. Much of their code was used in the plugins created in this thesis.

I would also like to thank my parents and family for their prayers and support for me through all these years. They always did everything they could to help me get to where I am today.

List of Figures

Contents

1 Introduction

Scientific visualization is a very important field which enables researchers to gain a more broad understanding and idea of what large sets of n-dimensional data look like. Scientific visualization allows one to take a large amount of information and see it all at once. The data represented by visualization could be graphs, abstract quantities or changes over time. It could also be actual physical and engineering data gathered through various means or theoretical concepts. An example would be volume rendering an MRI or in the case of this thesis; gravitational waves.

Scientific visualization is a very broad field that is usually applied to an existing project or research data. Most visualization systems and techniques have been created specifically for one type of visualization. For example, constrained inverse volume rendering[18], or MacVis[13]. Spiegel, on the other hand, is a general purpose visualization system which is designed to meet future visualization needs that are currently not supported.[4]

A similar implementation to the RenderMan Shading Language was used to render astrophysical data before. In their paper, Corrie et al. write about Data Shaders that were designed for visualizing volumetric data.[8] They describe an extension to the existing types of shaders in PRMan called a Data Shader. Data Shaders are designed to perform the shading and classification of the sub-volume around a sample point in a volume data set. The shader does not actually work on RenderMan, instead it is implemented on a programmable volume rendering system based on cap_vol[8], a volume renderer that has been developed at the Australian National University.

This thesis modifies a RenderMan pipeline in a visualization software, Spiegel, that was developed at RIT and implements Spiegel plugins to render three different astrophysical objects using the RenderMan pipeline. The objects are a nebula, gravitational wave and a point cloud in order to evaluate and analyze the results compared to previous results that were rendered in OpenGL. Several plugins were developed to visualize the objects. The objects themselves are contained in text files that contain point data which

is read in by Spiegel, which has support for the different file formats. Spiegel already had an existing RenderMan pipeline, but support for polygons was added as well as RenderMan specific optimizations to improve rendering times.

1.1 Problem Statement

This thesis explores several methods of visualizing four dimensional astrophysical datasets consisting of four dimensional point-density data sets. The data is written in a text file so it is impossible to discern any meaningful information from reading the numbers alone. By rendering this data into a visible image, it is able to take all the individual values and display them at once in a meaningful image that is able to aid in the comprehension of the data. Spiegel has a Render system capable of rendering such data. This thesis will look into the advantages and disadvantages of using a full fetured, industry standard Renderer.

Three different objects will be rendered: a nebula, gravitational wave and the surface of a non-uniform point cloud. These will be used to judge the feasibility and evaluate the use of RenderMan in the context for scientific visualization. The methods and results will be compared to previous techniques that were implemented in the pipeline. The rendering times as well as the advantages and disadvantages of the design will be discussed.

1.1.1 Applications Used

Two applications are used in this thesis, Spiegel, a visualization framework for large sets of data [4] and Pixar's Photorealistic RenderMan, or PRMan, which is based on the RenderMan interface. The RenderMan Interface is a standard communications protocol between modeling programs and rendering programs capable of producing photorealistic-quality images.[2]

1.2 Thesis Outline

This thesis will begin by describing the JOGL Pipeline as well as the existing RenderMan pipeline. It will discuss the shortcomings of the OpenGL approach and the motivation for adding RenderMan support. It will also delve into the technical aspects of RenderMan, and what makes it an effective choice for as a tool for scientific visualization.

Algorithms that were used in the plugins for rendering the astrophysical objects will then be described. Some of these were implemented as RenderMan shaders, while others were implemented as Spiegel plugins. The algorithms that this thesis will cover in depth are Perlin Noise, volume rendering techniques, Marching Cubes, color spaces and algorithms for polygonizing a point cloud.

The next part will discuss the technical aspects of the implementation and how exactly the above algorithms were used in the pipeline when rendering the objects. The first part will be the RenderMan pipeline and the support for RenderMan objects including lights and shaders. It will also discuss each shader that was written for displaying the objects and the motivation behind the choice of shader used. The next part will describe exactly how the RenderMan pipeline works with Spiegel.

The thesis will describe the technical aspects of rendering the nebula, including fractal noise and fractional brownian motion. It will describe how color spaces were used to illustrate the density in certain areas as well as what noise parameters work and parameters that produce unsatisfactory results.

The gravitational wave will discuss the marching cubes algorithm and how it was implemented using the uniform grid data provided. Various methods of shading in order to get the clearest image will be discussed as well as rendering multiple layers of densities. This will also discuss the optimizations that were done to increase rendering times and decrease memory usage to minimize swapping. Specifically the way RenderMan handles rendering in buckets and delayed read archives will be discussed.

Rendering of a non-convex surface of a non-uniform point cloud will also be discussed and will describe the surface algorithms for polygonizing a point cloud that were used.

The next section of the thesis will describe the resulting renders and how they compare to existing work in terms of rendering time, file size and complexity of implementation.

Finally the conclusion will describe possible further work that can be done to the pipeline to add support for various RenderMan features that have not yet been implemented into Spiegel.

2 Previous Work in Spiegel

Previous work in astrophysical visualization using Spiegel included OpenGL and RenderMan. The OpenGL implementation used was JOGL, a Java implementation of OpenGL. [19] The RenderMan Pipeline only had support for spheres and many options were hard-coded into specific plugins. The pipeline was completely redone in this thesis into a more modular and simple design that allows for more flexibility and software reuse.

2.1 OpenGL

OpenGL was used to render the data before any RenderMan support was written. This section will discuss the way OpenGl was used to render objects in Spiegel.

2.1.1 Design

The OpenGL pipeline was built into the plugin classes. When using OpenGL it is necessary to pass data back and fourth from the api to the application. This becomes complicated when using many shaders because it is necessary to set each shader's uniform variables from within the application. Each plugin had to specify the uniform variables for shaders that were used. This made it difficult to separate the OpenGL aspects from the algorithms that generated the geometry.

2.1.2 OpenGL Pipeline

The OpenGL Rendering Pipeline is a series of processing stages. [5] The rendering pipeline is like an assembly line where primitives in display lists are assembled into triangles, shaded and drawn on the screen. The pipeline starts with putting all data in a display list.

The first stage is per-vertex operations which converts the vertices in the display list into primitives. A shader program is run on every vertex, which allows the position of the vertex to be transformed, rotated and scaled. Spacial coordinates are projected from

a position in the 3D world to a position on the screen. Texture coordinates are also generated and transformed in this stage. Lighting, if enabled, can also be performed in this stage. Clipping is also a major function of this stage, if geometry goes beyond the edge of the screen it is culled and the rest of the geometry is eliminated. The result of this stage are complete geometric primitives, which are transformed and clipped with related color, depth and texture-coordinate values and other data that is used in the rasterization stage.

Next the geometry and pixel data is converted into fragments. Each fragment is a square that corresponds to a pixel in the framebuffer. [5] This process is called rasterization. Color and depth values are assigned for every fragment square. If a custom shader is written, then the shader program is run on every fragment. In this stage, texture mapping is also done. A texel, or texture element, is generated from texture memory for each fragment. Fog calculations, scissor test, alpha test, stencil test as well as the depth buffer test are all performed in this stage of the pipeline.

Blending, dithering, logical operation and masking by a bitmask is performed afterwards. Finally the fragment is drawn to the frame buffer.

2.1.3 RenderMan REYES Pipeline

Pixar's Photorealistic RenderMan is based on the Reyes Image Rendering Architecture. The rendering system was developed at Lucasfilm Ltd. and its goal was to be an architecture optimized for fast high-quality renders of complex scenes.[7] [21] RenderMan was designed to model scenes with hundreds of thousands of geometric primitives. In order to shade each primitive, a shading language was developed. Ray tracing is also supported but is optional since not every object requires raytracing for optimal image quality. The original goal was to have a rendering speed of about 3 minutes per frame, which, assuming 24 frames per second, would allow a 2 hour movie to be rendered within a year.

The Reyes pipeline has for main stages, dice/split, shade, sample and visibility/fil-

ter. The Reyes architecture divides the primitives into a grid of micropolygons using a technique call Dicing. Micropolygons are the common basic geometric unit of the algorithm.[7] If the primitive cannot be diced, it is split into two primitives which are then diced into micropolygons. Micropolygons are advantageous because every micropolygon can be considered independently of what kind of primitive it is. Since the rendering algorithm requires a large amount of z-buffer memory, the screen is divided into rectangular buckets.[2]

After all of the primitives in a bucket have been split or diced, their micropolygons are put into every bucket they overlap. The number of micropolygons in memory can be managed by setting the grid size. The rendering time is proportional to the number of micropolygons. This means that the larger the resolution size, and the more objects there are, the slower it runs.

After each primitive is diced, shading is done in eye space through procedural shaders. Each micropolygon is flat shaded. [21] The final stage is the visibility/filter stage. This is where the visible pixel is chosen based on the depth buffer. Then the pixel values are filtered by a user defined filter to render the final image.

2.1.4 Differences Between OpenGL and Reyes

Both pipelines are similar in function. In both the application produces geometry in object coordinates, which are then shaded, projected into screen space, rasterized into fragments, and composited using a depth buffer. The pipelines differ in three ways: shading space, texture access characteristics and the method of rasterization. [21]

The OpenGL pipeline shades in two stages, on vertices and fragments with respective shaders for each stage. The Reyes pipeline supports a single shading stage on micropolygons. In OpenGL texturing is a per-fragment screen-space operation and textures are filtered by using mipmapping. In Reyes, texturing is a per-vertex eye-space operation. Reyes uses "coherent access textures" [21] which do not require any filtering and can be accessed sequentially. Coherent access textures, or CATs are textures for which texture

coordinates s and t are $s = au + b$ and $t = cv + d$.[7] This means that s and t are linear functions of u and v texture coordinates. OpenGL's basic primitive is a triangle while the fundamental primitive of the Reyes pipeline is the micropolygon.[7]

2.2 Motivations for using RenderMan

RenderMan has many benefits over OpenGL. It is designed as a tool for rendering geometry generated in other software such as commercial modeling applications like Maya or custom in-house tools. The Reyes pipeline that is used by RenderMan was designed to be able to compute a feature-length film that is virtually indistinguishable from live action motion picture photography and to be able to create visually rich as real scenes. [7] In this case, the Spiegel Visualization Framework[4] is being used to format data into files useable by RenderMan. The framework is designed in such a way that allows for maximum code reuse through modular plugins. This allows one to have a general plugin for outputting geometry into a RenderMan scene description file called a RIB file and a separate plugin to create the geometry. This allows one to separate the renderer from the data processing and geometry generation code.

2.2.1 OpenGL Hardware Compatiblity

Not every OpenGL feature is compatible with existing graphics hardware today. Many newer features are implemented through extensions which are not always supported by older hardware. This creates compatibility issues that RenderMan does not have because it is a software based renderer that runs on the CPU.

2.2.2 RenderMan Filters

Another benefit are the filters that are built into RenderMan. Unlike OpenGL the images that RenderMan outputs are filtered through custom pixel filters such as the gaussian filter or a catmull-rom filter. [2] This allows for a higher quality image, and allows one to control the aliasing of the image by choosing a suitable filter for the image being rendered.

2.2.3 RenderMan Shaders and Shading Language

The RenderMan Shading Language is a programming language that is designed to describe the interactions of lights and surfaces.[2] It provides the means to extend the shading and lighting formulae used by a rendering system.[14] For a given primitive, a shader is a program that is executed on every micropolygon that a primitive is divided into. PRMan supports four different kinds of shaders: Surface, Displacement, Light and Volume.[2] Each shader has an intrinsic set of variables that is inherited based on the geometry that is being shaded. For example, all shaders inherit the opacity and color of the the primitive and the local illumination environment described as a set of light rays.[14] This allows for the programmer to manipulate given variables to achieve the desired effect.

RenderMan supports complex shaders that perform operations that would be difficult to do with OpenGL shaders. The REYES architecture divides the image into micropolygons which allow complex operations on individual micropolygons that would not be possible in the OpenGL pipeline such as displacement shaders. [7] OpenGL uses the traditional triangle rasterization pipeline that is hardware based and designed for speed. RenderMan is designed as an offline renderer, which means that it is not used for realtime. This allows for more detailed and accurate operations to be performed on the image that one is rendering.

The REYES pipeline also allows for simpler shaders to be written. OpenGL uses a multi stage pipeline where certain stages of the pipeline are programmable. The modern pipeline allows for vertex, geometry and fragment shaders. RenderMan has different types of shaders that can be written, but each shader only requires one file to be written, instead of two or three separate shaders which handle different stages of the pipeline. Because geometry in the REYES architecture, which RenderMan uses, is divided into "micropolygons" which are all processed by the shader the same way regardless of the geometry. [7]

2.2.4 Aliasing

Aliasing is an important issue when working with a fixed resolution. Since every pixel on screen is really just sampling the geometry and objects underneath it, problems are encountered when the sampling rate goes below the Nyquist limit. The Nyquist limit is the highest frequency that can be adequately sampled, which is half the frequency of the samples themselves. In other words, the samples must be at least twice the highest frequency present in the image or it will not recreate the image.[2] PRMan, the renderer used in this thesis automatically antialiases texture lookups and allows the user to set various filters and to adjust the sample rate on the image.

2.3 Previous Spiegel RenderMan Pipeline

RenderMan was perviously used in rendering a galaxy merger.

2.3.1 Design

The design was similar to the current one developed in this thesis but the previous design had a lot of hardcoded functionality. Multiple shaders were written from within the RIB generation file and transformations were hard coded into the file for the one specific render of the galaxy merger.

2.4 Modifications to Pipeline

A few modifications were made in order to handle the new renders. The gravitational wave involved rendering many transparent layers, which ended up using al physical ram and ended up swapping causing the renders to take a very long time. In order to mitigate this, RIB Archives were used. The support for polygons was also added.

The RIB file generator plugin was completely revamped. The code originally had many shaders hardcoded for a specific render. It was re-written so any object could be passed in and rendered. Nothing is hardcoded, everything can be modified. This allows

for one plugin to be used to render all three different renders that were created in this thesis.

2.4.1 New Plugins Written

New plugins written were a nebula cloud plugin, a marching cubes implementation and the pivoting circles implementation for rendering non-uniform point cloud surfaces. The plugin for outputting shaders was also written, as well as changes made to the way directories were read so problems would not be encountered when switching from Windows to Mac OS X. Support for RenderMan polygons was also added.

2.4.2 Modular Design

Everything that had to do with generating graphics and hardcoded was taken out and replaced with a modular design. Individual shaders were taken out and can now be added graphically or through the sprache script files as needed per plugin.

2.4.3 RIB Archives to Speed Up Renders

The Reyes pipeline dices polygons into micropolygons on demand depending on the bucket size and the area it is shading. In some cases it is possible for the renderer to exhaust a large amount of memory such as when rendering motion blur or in the case of this Thesis, rendering many transparent layers. Dan Maas calls this a "micropolygon explosion." [17] The solution that is used in this thesis is to divide the scene into RIB Archive files, which are files that contain only the geometry descriptions. The renderer loads and unloads the files as needed. This greately decreased the memory usage when rendering the gravitational wave.

2.4.4 Usage of Pipeline in Spiegel

The pipeline in Spiegel was designed with simplicity and maximum reuse in mind. It was also designed to take advantage of RenderMan's RIB archive functionality. An example

of the usage is illustrated in the image above. The first step is to extract the data into a useable form. The data is usually point data with information about the points such as density. Then it is passed into the module that processes the data and outputs visual elements such as spheres or polygons. This is where the scene is basically constructed. For example, if a Marching Cubes algorithm was being used to polygonize a grid, it would be used in this plugin. The plugin creates arraylists which represent the objects in the RIB Archive file. Each object has it's own color, opacity, shader, and position. The objects are read by a separate module which creates the main RIB scene discription file with the render quality and camera settings as well as individual RIB archive files.

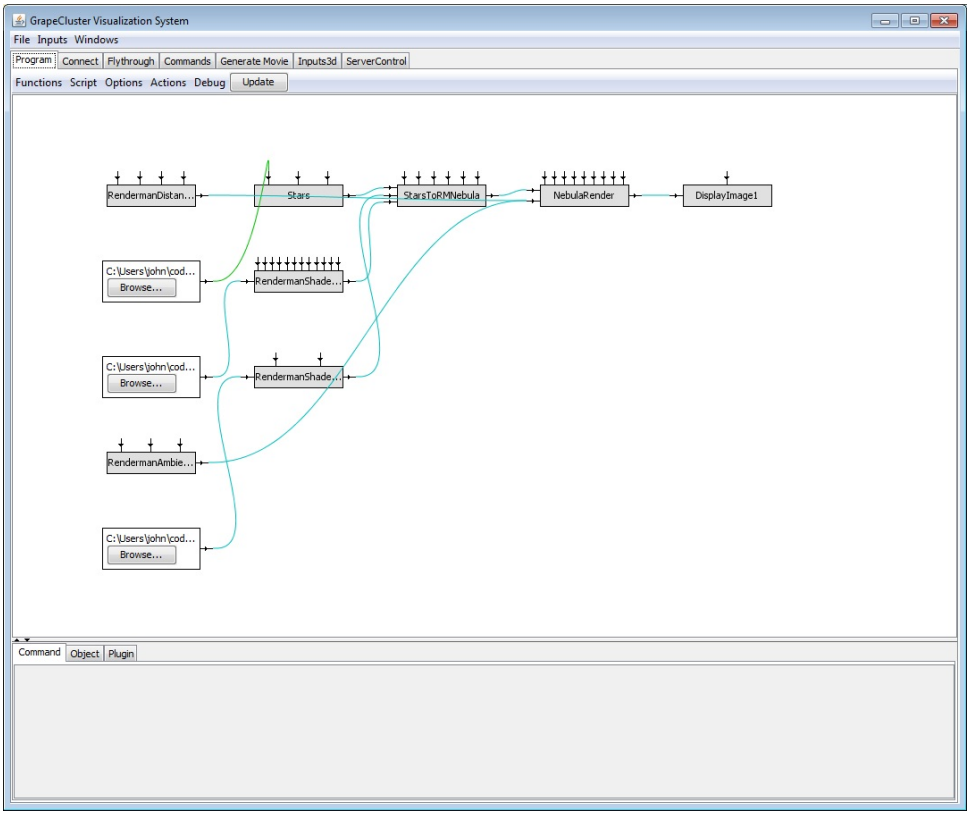

Figure 1: Example of a RenderMan program in Spiegel

The pipeline has four stages: the first is the data extraction, the second is the processing of the data into RenderMan objects, the third is setting up the camera and the

final stage is displaying the image. In Figure 1 the program is being used to display a nebula. The main inputs on the left side from top to bottom are a distant light, the star dataset, the nebula shader, an ambient light, and a star shader. The objects are then connected to the proper plugins. The stars are extracted using the star extractor plugin and the shaders are compiled with the shader plugin. All the data goes into the nebula creation plugin. The data is processed into RenderMan objects and is sent into the camera plugin which also sets up the scene with lights. Finally the rendered image is displayed using the display image plugin. This kind of design allows one to swap out components for different ones. For example if a different shader is used for the stars, one can just change the input and modify the shader parameters graphically. If one wants to extract the stars in a different way, for example, taking only half the stars from the dataset, one can write a custom plugin and replace the current star extractor with it.

3 Algorithms

The algorithms that were used for astrophysical visualization had to do with volume and surface rendering. The data is usually made up of points with certain densities and rendering the data in an understandable and aesthetically pleasing way was achieved through three different techniques. The first render was a rendering of a nebula cloud from point data. The density was determined by a number of points in an area. From the densities, spheres were created which were later shaded using Perlin Noise. This was a similar technique to the gardner cloud shader in Advanced Renderman by Apodaca and Gritz. [2] In the shader they use transparent spheres which are shaded with a turbulence shader to give a cloudy appearance. The shader was modified to give a wispier appearance rather than a cottony cloud appearance.

The next object was a gravitational wave which was rendered by using a Marching Cubes algorithm. [16] Since the data was organized in an evenly spaced grid, the Marching Cubes algorithm was well suited for polygonizing the point field. The algorithm also allowed for density data to be used at various levels and to see what the wave looks like as one increased the density threshold with time.

The last object was a surface render which used a modified Ball-Pivoting Algorithm. [3] The algorithm was simplified where instead of using a circle-sphere collision, an angle and circle radius was used. This simplified the calculations and produced a correct polygonization of a surface point cloud. The seed triangle was also found by using a technique used by the IPD algorithm. [15]

3.1 Color Models

A color model is a specification of a 3D color coordinate system and a visible subset in the coordinate system within which all colors in a particular color gamut lie. [11] The gamut of a device refers to the range of colors that it can reproduce. [2] The purpose of a color model is to allow convenient access to colors within a certain range.

3.1.1 RGB Color Model

The color model used with computer monitors is RGB. The RGB color space can be visualized as a cube in Figure 2. The main diagonal of the cube represents gray levels because that is where equal amount of red, green and blue color are. This can be represented from $(0, 0, 0)$ to $(1, 1, 1)$.

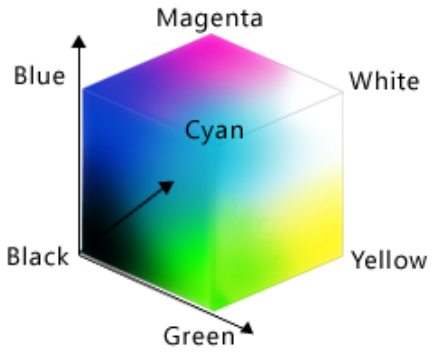

Figure 2: Visualization of the RGB color space. [1]

The RGB color space isn't intuitive to the way humans perceive color because it is an additive color space. This means that if red is added to cyan, it becomes a lighter shader of cyan instead of more purple as one would expect. This can be seen in Figure 3.

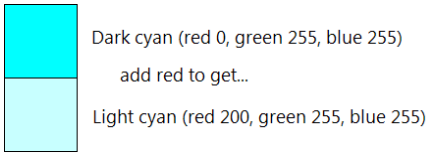

Figure 3: Adding red to cyan. [1]

3.1.2 HSV Color Model

An alternative color space is Smith's HSV model. [27] It is user oriented being based on the intuitive appeal of the artist's tint, shade and tone. [11] The three components of the color are hue, saturation and value. The hue is essentially the RGB color cube viewed along the principal diagonal. It can also be visualized as a wheel in Figure 4. Red is at

zero degrees, yellow is at 60, green is at 120, cyan is at 180 degrees, blue is at 240 degrees and magenta is at 300 degrees.

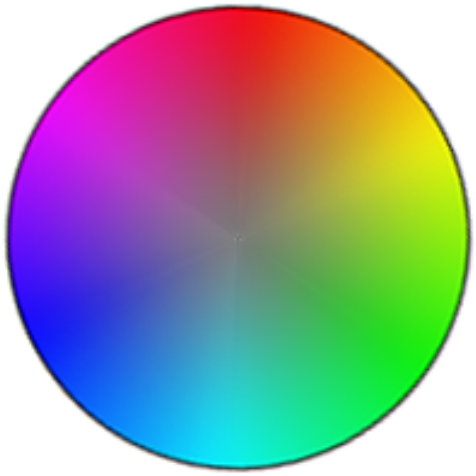

Figure 4: HUE color wheel. [1]

If we extend this wheel along the center into a cone, we end up with a visual representation of the HSV color model as shown in Figure 5.

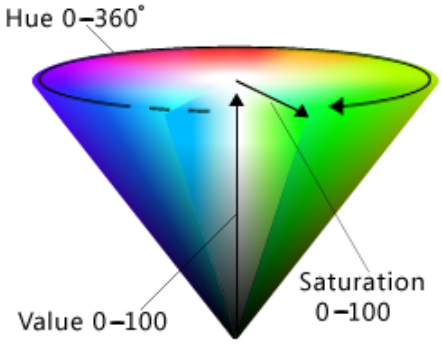

Figure 5: The HSV Cone. [1]

The cone has a height of 1. The point at the apex is black and has a V coordinate of 0. At this point, H and S values are irrelevant. If the Saturation, or S, value is 0 as the V values go up, we end up with grayscale colors and the hue or H value is irrelevant.

3.1.3 HLS Color Model

Another model for mapping the color space is the HLS color model[11], which stands for hue, lightness, saturation. HLS is essentially a deformation of the HSV color model where white is pulled upwards. This means that an L value of 1 will return white, while an L value of 0 will be black, regardless of the other values. This means that in order to get full saturation of a color, the L value has to be 0.5, which would correspond to the center of the double-hexacone as illustrated in Figure 6.

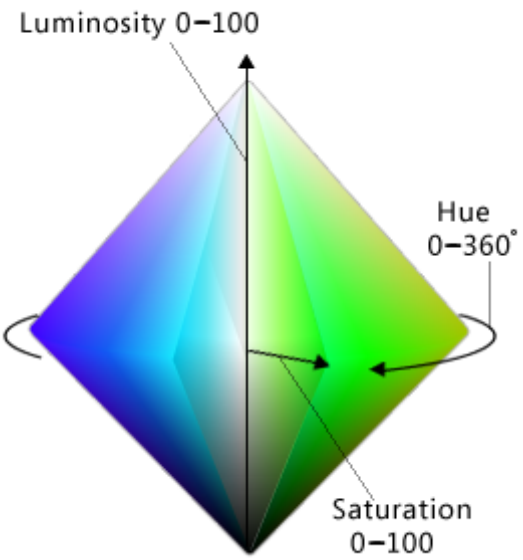

Figure 6: The HLS Double Hexacone. [1]

3.2 Perlin Noise

A noise function is designed to provide a repeatable, pseudo-random signal over $\mathbb{R}^3$ that is band limited and has statistical invariance under rigid motions.[11][10] Statistical invariance under rigid motions means that any statistical property, such as the average value of the variance over a region is about the same as the value measured over a congruent region in some other location and orientation.[11]. Band limiting means that the Fourier transform of the signal is zero outside of a narrow range of frequencies. It's essentially

the result of a band pass filter. This means that there are no sudden changes in the function. This can be done through Fourier synthesis[11] but Perlin found a simpler and faster way to create such a noise function.[23]

3.2.1 Noise in Two Dimensions

When calculating the noise in two dimensions, we have the function

$$noise2d(x, y) = z \tag{1}$$

with x, y, z as floating-point numbers. We define the noise function on a grid where gridpoints are defined for each whole number, and any number with a fractional component lies between grid points. Consider a point that lies in a grid which has a fractional component as seen in Figure 7.

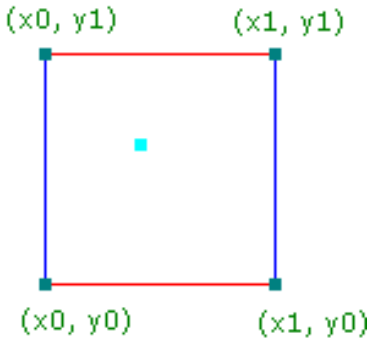

Figure 7: The four grid points around our point. [30]

We take the gradient of a length of 1 of every corner. Each gradient vector is assigned a pseudo-random direction as seen in Figure 8.
We also need the displacement of our point from each of the grid vertices as seen in Figure 9 These are calculated by subtracting our x and y coordinates from the grid corner points.

With the gradient values and corner vectors it is possible to calculate the value of the noise function. The influence of each gradient is calculated by performing a dot product of the gradient and the vector. If we consider the dot product, the result will be scaled based on the angle between the gradient vector and the corner vector. This is a scalar

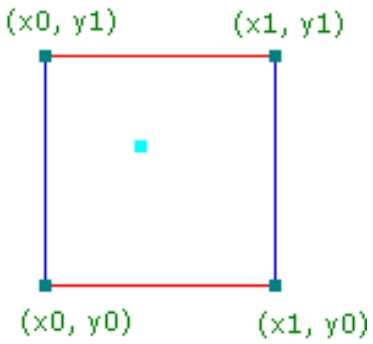

Figure 8: The four pseudo-random gradient vectors. [30]

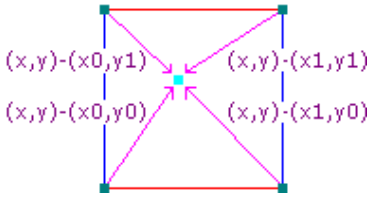

Figure 9: The four corner vectors. [30]

value that determines the percentage of each vector's influence. The four values can be calculated as follows.

$$s = g(x_0, y_0)((x, y) - (x_0, y_0)) \tag{2}$$

$$t = g(x_1, y_0)((x, y) - (x_1, y_0)) \tag{3}$$

$$u = g(x_0, y_1)((x, y) - (x_0, y_1)) \tag{4}$$

$$v = g(x_1, y_1)((x, y) - (x_1, y_1)) \tag{5}$$

The next step is to average the results together by taking a weighted average. The equation for taking the noise average was $3p^2 - 2p^3$ which creates an S curve. [23] [30] [10] [24] It turns out that this creates a curve that has nonzero values in it's second derivative. This gives us an uneven distribution and causes unwanted higher frequencies. The function was changed to the function $6t^5 - 15t^4 + 10t^3$. [10] [24] The two noise curves can be seen in Figure 10.

The averages are found by taking the value along the curve at the point $x - x_0$.

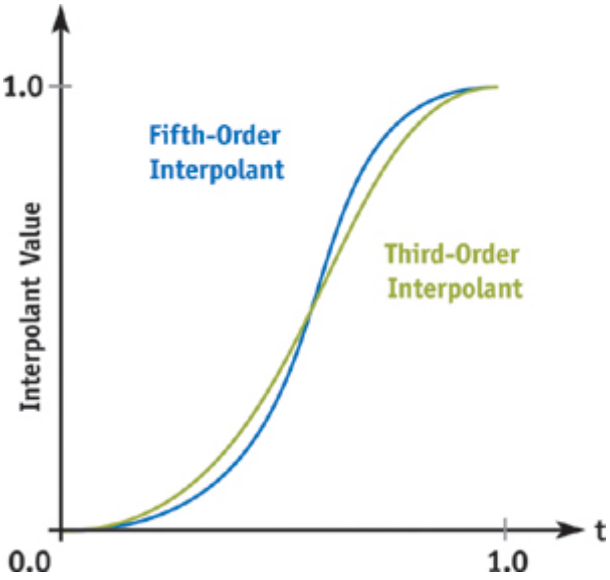

Figure 10: The two interpolation curves. [10]

$$S_x = 3(x - x_0)^2 - 2(x - x_0)^3 \tag{6}$$

Then this value is linearly interpolated from the previous values s and t by mapping the values on a range from 0 to 1. We will call this value a.

$$a = s + S_x(t - s) \tag{7}$$

The same is done for our v and u values. This results will be b.

$$b = u + S_x(v - u) \tag{8}$$

Then we find the mapping of our y value on the curve. This is the value at $y - y_0$

$$S_y = 3(y - y_0)^2 - 2(y - y_0)^3 \tag{9}$$

This value is then used to interpolate between the results for a and b resulting in our final value that is returned from the noise function.

$$z = a + S_y(b - a) \tag{10}$$

3.2.2 Noise in Three Dimensions

Noise in three dimensions is essentially calculated the same way, except instead of four corner vectors, we use eight. This can be done for any dimension. The number of grid points is calculated as 2^n where n is the dimension we are in.

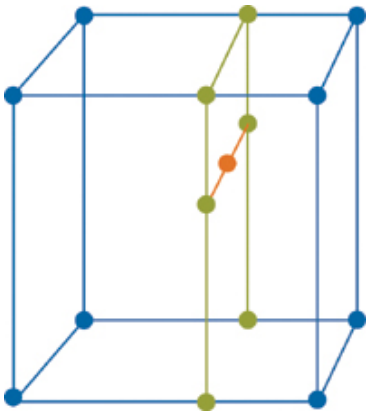

Figure 11: Noise interpolation in 3d. [10]

The interpolation in three dimensions is illustrated in Figure 11. The interpolation requires four interpolations in x, two in y and one in z. [10]

One way to speed up the noise algorithm is to have a table of gradients. This way the gradients don't have to be looked up.

3.2.3 Turbulence and Fractal Noise

Turbulence can be made by summing up and weighing several noise functions. This leads to many patterns that occur in natural objects such as marble, water, fire and clouds.[23] An example of a turbulence function using noise is in code Listing 1 [23]

```
function turbulence(p)
{
```

```
  t = 0
  scale = 1
  while(scale > pixelsize)
  {
    t += abs(Noise(p/scale) * scale)
    scale /= 2
  }
  return t
}
```

Listing 1: Example of a turbulence function

This shader can be used to make billowy looking clouds such as Figure 12.

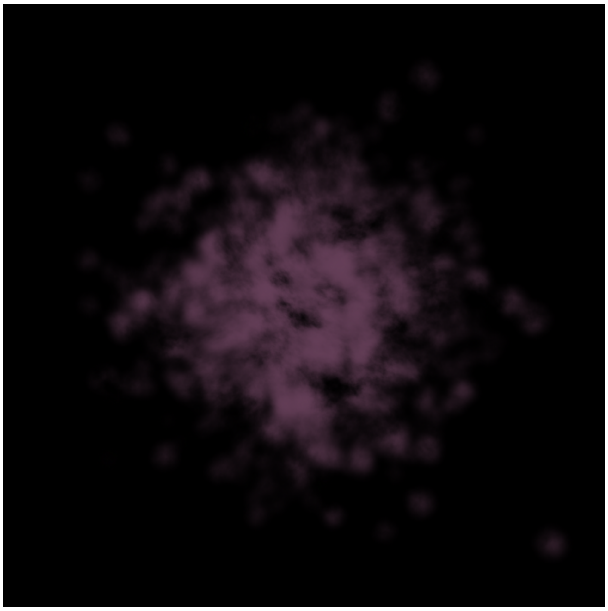

Figure 12: Turbulence used to make a wispy cloud.

3.3 Creating a Surface from Equally Distributed Points

Volumetric data sets are often rendered using indirect volume rendering or, IVR, techniques. [20] This thesis explored two such algorithms, the Marching Cubes and the

Marching Tetrahedra algorithms. Both are very similar techniques for extracting an isosurface from volume data.

3.3.1 Marching Cubes

The Marching Cubes algorithm was originally intended to create triangle models of constant density surfaces from medical data. [16] It is a sequential-traversal method. [20]

An isosurface is defined as follows. Given a scalar field F(P) with F a scalar function on $\mathbb{R}^3$, the surface that satisfied F(P)=α, where α is a constant is called the isosurface defined by α. The value α is called the isovalue. [20] This function is usually a piecewise function composed of a list of triangles. The Marching Cubes algorithm is an isosurface creation method. It takes in a density grid and outputs triangles that approximate a surface at a specific density value, or α.

The algorithm does this by "marching" a cube across the volume. The cube is positioned in such a way that the eight corners can have a density value assigned to them. If the density value is below or above our α threshold value, then the corners are marked as such. The simplest case are when all of the values are below or above the threshold. In this case, no triangles are produced and the cube is moved over to the next data points until the cube is in such a position that it is intersected by the surface. This is the case when some corners have density values above the α value and some corners are below. There are 2^8 or 256 possible way a surface can intersect a cube. All these combinations are enumerated in a list that can be looked up based on which corners are below or above the threshold. The lists hold the vertices in the correct winding order that is used upon creation.

The point where our α density lies can be estimated by linearly interpolating along the corresponding edge of the cube. If a unit-length edge E has end points V_s and V_e whose scalar values are L_s and L_e then given an isovalue α the location of intersection $I = (I_x, I_y, I_z)$ can be found with the following equation:

$$I_{x,y,z} = V_{s\{x,y,z\}} + \rho(V_{e\{x,y,z\}} - V_{s\{x,y,z\}}) \tag{11}$$

where

$$\rho = \frac{\alpha - L_s}{L_e - L_s} \tag{12}$$

The resulting interpolated values are the positions of the vertices of the triangles that make up the isosurface. After triangles are created, the normals are generated for shading. The gradient of the surface is used to generate the normals. It is used by finding the derivative of the density function:

$$\overrightarrow{g}(x, y, z) = \nabla \overrightarrow{f}(x, y, z) \tag{13}$$

The gradient is found at every corner, and it is then interpolated to the point of intersection.

There are a few optimizations that can be done to this algorithm. Since the intersections of the cube are calculated on each edge, it is only necessary to interpolate three new edges for every cube other than the first one. The other nine edges can be obtained from previous data. The list of 256 intersection topologies can also be simplified to 14 by taking into account reflection, rotation and mirroring as seen in Figure 13. [20]

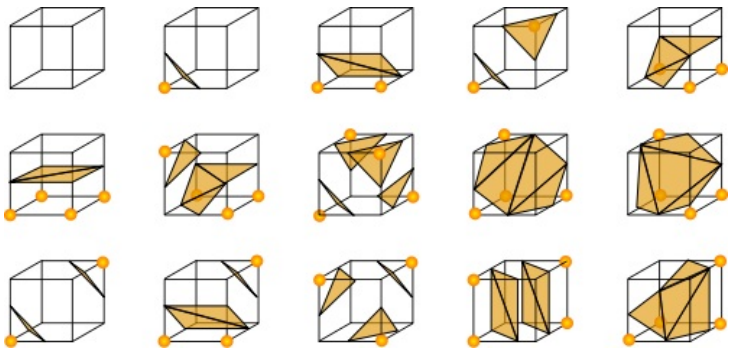

Figure 13: 14 Basic Marching Cubes Triangulations.[28]

3.3.2 Marching Tetrahedra

Marching Tetrahedra [22] is a very similar method to Marching Cubes. It was developed by Payne and Toga for surface mapping brain anatomical data from magnetic resonance imaging. Although they first created the algorithm the term "Marching Tetrahedra" was first coined by Shirley and Tuchman. [26] Instead of a cube being intersected, it is divided in to either five or six tetrahedra. Since there are for vertices per tetrahedron, there are fewer triangle subdivision combinations. The main motivation for dividing the marching cube into tetrahedra is that it allows for less ambiguities when calculating the intersections of a detailed surface.

3.4 Creating a Surface from Unequally Distributed Points

Creating a surface from unequally distributed points is a much more difficult problem than from a regular spaced grid because it is difficult to interpolate the data linearly. The data points are not guaranteed to be evenly spaced so getting the correct densities at each corner of a cube would be impossible. It is also difficult to deal with cases where the point cloud does not form a convex object, such as a torus. In order to polygonize such a surface, a few techniques were explored.

3.4.1 Ball-Pivoting Algorithm

The Ball-Pivoting Algorithm [3] is an algorithm designed to recreate a surface from a point cloud. One can imagine rolling a sphere along the surface of the sampled points and connecting each point that is touched into a triangle. This is essentially how the algorithm polygonizes the surface. Starting with a seed triangle the ball pivots around each new edge and if it touches another point, a new triangle is formed and the algorithm continues. When it cannot find another point, a new seed triangle is found and the ball continue to pivot until no more seed triangle can be found.

The Ball Pivoting Algorithm is related to Alpha Shapes.[9] Alpha Shapes are a formal

definition of a "shape" of a point set in $\mathbb{R}^3$. Let the manifold M be the surface of a three-dimensional object and S be a point-sampling of M. If we assume that S is dense enough that a ball of radius ρ, or a ρ-ball, cannot pass through the surface without touching a sample point, then the triangles created by such a ball touching every point guarantees that they have an empty smallest ball whose radius is less than ρ. This is a subset of the Delaunay triangulation of the point set (see [9], page 75).

The Ball Pivoting operation starts with finding the seed triangle. First an unused point is chosen to be used in the seed triangle. Then the closes points are considered to form a triangle. Once a triangle is formed, it is checked to make sure the normal is consistent with the normals of the points. Finally a ρ-ball is tested to make sure that the points form a triangle that can be

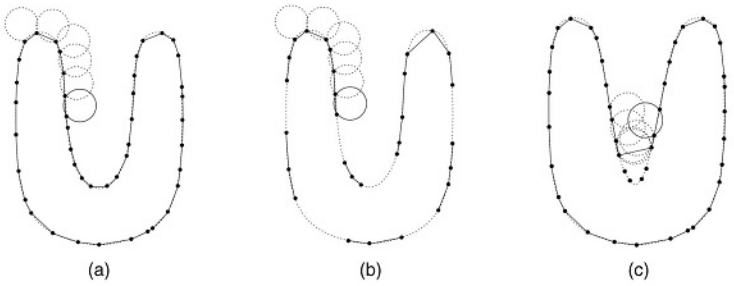

Figure 14: Illustration of the ball pivoting algorithm in 2 dimensions and it's limitations. b shows gaps caused by using a radius ρ that is too small. c shows the case when the ball radius is too large. Image taken from [3].

circumscribed within the ρ-ball.

The next step is the ball pivoting operation. We consider a ball that lies on an edge of the triangle, $e_{(i,j)}$. If we consider a ball lying on the two points of the edge $e_{(i,j)}$, let c_{ij} be it's center. The pivoting of the ball is a continuous motion where the ball stays in contact with the two endpoints of edge $e_{(i,j)}$. As the ball pivots around the edge, it's center forms a circle γ which lies on the plane perpendicular to the edge $e_{(i,j)}$. If the ball hits a new point, σ_k, a triangle is formed from the two endpoints of the edge and the new point that was hit. Each active edge is a "front" edge. A final edge is one which had a ball pivot over it without hitting any point.

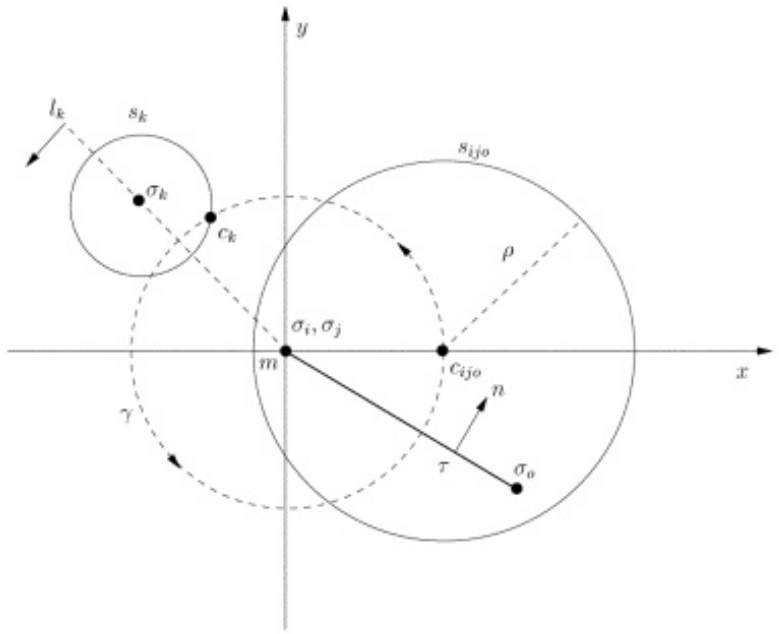

Figure 15: Illustration of the sphere-circle intersection. Image taken from [3].

In practice, point σ_k is found by computing the center of of a ball, c_x, touching the endpoints of edge $e_{(i,j)}$, σ_i and σ_k, and a potential new point in a 2ρ-neighborhood (where ρ is the radius of the ball) of the midpoint of edge $e_{(i,j)}$, which will be referred to as m. Each c_x lies on the circular trajectory γ around m and can be computed by intersecting a ρ-sphere centered at σ_x with the circle γ as illustrated in Figure 15. Of all the points that intersect with the circle γ, we select one that is first along the trajectory γ. If no intersections are found, then the edge is considered a final edge.

The pivoting continues until all edges have been marked as final edges. When a new point is added to the triangle list, it is done either through a join or a glue operation. The simpler of the two is the join operation where a not-used point is used to create a triangle. Two new front edges are added and the current edge becomes finalized. The more complicated of the two arises when the point is already part of the mesh. The first case would be if the ball pivoted all the way around and found a point already part of the mesh that is not used by any front edges. This point would not be used and the edge would be marked as final. The other case is when one of the edges that the point is part

of is a front edge. In this case, after checking the orientation of the edges, either one or two new edges are created when the triangle is added.

3.4.2 Intrinsic Property Driven Algorithm

The Intrinsic Property Driven algorithm [15] is a similar algorithm to the Ball-Pivoting algorithm [3] where edges are used to find the next point. The difference is in the way the next point is found. Instead of using a pivoting ball, an influence region is determined and based on a weighted algorithm the next point is determined within that region. The algorithm also adapts to the density of the point cloud in different regions. This only requires one pass, instead of multiple passes like the Ball-Pivoting algorithm.

The first step is finding the seed triangle. A point P is found whose z-coordinate is the largest in the point cloud. A point Q is found that is the nearest point to P and forms a line segment L_{PQ} A cylinder is constructed around the line and grows outwards until it contains one or more points. The point that is chosen, R, is the one where the sum of lengths of the edges connecting R and the points P and Q is the smallest. Such a triangle is chosen so that the normal vector can point outward. If the inner product of the triangle normal and the vector $(0, 0, 1)$ is positive, we have the desired normal. If it is negative the normal if flipped around to point the other way. This keeps the normal consistently pointing outwards when new triangles are found. For example if our seed triangle is A and we create a new triangle B, we make sure that the inner product of the normals of A and B is positive.

The second step is a loop that goes through all the active edges and creates additional triangles from them. In order to determine which point will make up the new edge, an influence region has to be defined. The size of the region is determined by the density of the points. An example of the boundary region can be found in Figure 16.

Figure 16 shows the influence region of the edge $e_{i,j}$. The dashed polygon is the projection of the influence region defined by the triangle adjacent to $e_{i,j}$. Each boundary

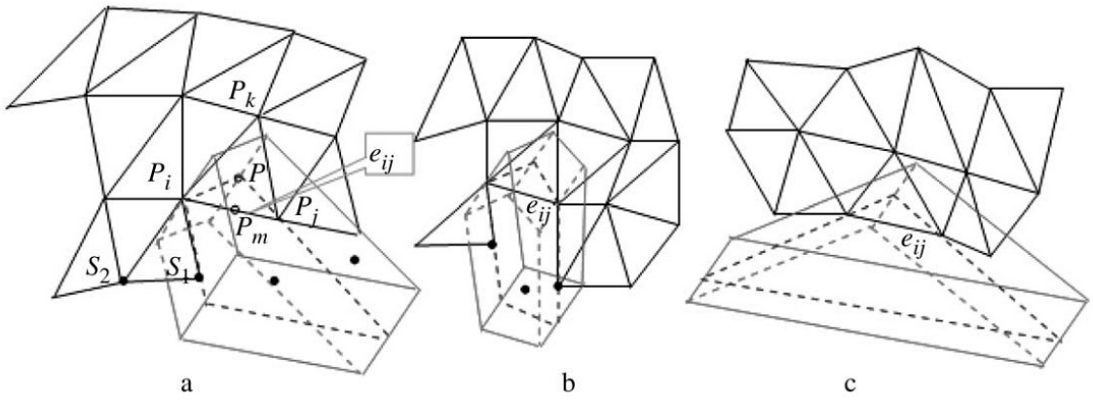

Figure 16: Examples of the influence region of edge $e_{i,j}$. The dashed polygon is the projection of the influence region unto the plain defined by the triangle adjacent to $e_{i,j}$. The dots are sample points. (image from [15]

face is calculated by using the Normal of the triangle and a point on the triangle. The point P is the barycenter of the triangle adjacent to the active edge. The normal is calculated by taking the cross product of $(P_k - P_i)$ and $(P_k - P_j)$.

If the influence region contains a points, the correct point has to be found. There are two criteria, the minimal area criterion and the minimal length criterion. Lin et al. found that the reconstructed surface using the minimal length criterion is visually better than the minimal area criterion. [15] Even so, the minimal length criterion surfaces have large difference in topology with the surfaces of the actual objects. They propose a new criterion called the weighted minimal length criterion for selecting a new point. It takes into account the aspect ratio of triangles and the length of the mesh edges. They adopt a heuristic strategy to solve the approximate minimum-weight triangulation. Suppose the triangle patch $\{i, j, k\}$ is adjacent to the active edge $e_{i,j}$, they select a new vertex P_m for $e_{i,j}$ such that that triangle patch $\{i, j, m\}$ minimizes the following sum:

$$k_{i,j}\|P_i - P_j\|^2 + k_{i,m}\|P_i - P_m\|^2 + k_{j,m}\|P_j - P_m\|^2 \tag{14}$$

The coefficient $k_{i,j}$ is calculated as follows:

$$k_{i,j} = \frac{(L_{i,k}^2 + L_{j,k}^2 - L_{i,j}^2)}{A_{i,j,k}} + \frac{(L_{i,m}^2 + L_{j,m}^2 - L_{i,j}^2)}{A_{i,j,m}} \tag{15}$$

where $L_{i,j}$ is the length of edge $e_{i,j}$ and $A_{i,j,k}$ is the area of a curved edge triangle patch $\{i, j, k\}$. $k_{i,m}$ and $k_{j,m}$ are approximated as follows:

$$k_{i,m} = k_{j,m} = 2\frac{(L_{i,m}^2 + L_{j,m}^2 - L_{i,j}^2)}{A_{i,j,m}} \tag{16}$$

3.4.3 Pivoting Circles Algorithm

The pivoting circles algorithm was a combination of elements from the Ball Pivoting Algorithm and the Intrinsic Property Driven algorithm. Instead of doing a sphere-circle collision, the radius of a circle was calculated to check if a sphere would collide with an intersecting point. The algorithm introduces a new method that is based on the gradient of the existing triangle and the current edge that is used to find the next closest unused point.

The seed triangle is found in the same way that is described in the IPD algorithm. [15] The edge is converted to a vector, and a second vector is formed from the next closest point that is found. If the dot product is negative, the point is automatically disregarded. If the dot product is positive, the point is projected unto the edge and the length of the projects is calculated. If the length is less than or equal to the edge, then the point effectively lies within the cylinder. If the length is greater, then the point is outside so it is thrown out. This continues until a point is found.

The next part of the algorithm is similar to the IPD and Ball-Pivoting algorithms. Instead of calculating an influence region or a collision with the sphere trajectory around

an edge, a tangent vector is used to determine the angle of the next point, to the angle of the current triangle. Points within a 2ρ region of the midpoint of the edge where ρ is the radius were found. A test was performed to find the closest point. This test was done by calculating the radius of a circle that could be drawn from the new point and the two end points of the edge. The following equation was used for calculating the radius, r, of a circle.

$$r = \frac{abc}{\sqrt{2a^2b^2 + 2b^2c^2 + 2c^2a^2 - a^4 - b^4 - c^4}} \tag{17}$$

After calculating the radius A vector was drawn from the midpoint of the edge to the new point. If the midpoint of the edge and the tangent of the edge are within a certain user set angle, then the point is accepted and edges are added. If an edge being added already exists, it is thrown out.

This greatly simplifies the calculation required in the ball-pivoting algorithm since instead of calculating the collision of a sphere and a circle, and finding the first hitpoint along such a trajectory we are taking a radius of a circle and a dot product to ensure that the point is within an influence region of the edge. This algorithm has some limitations when the angle changes drastically. Sudden changes of the tangent of the surface, or the first derivative, will cause inaccurate triangulation of the surface.

4 Implementation

This section will describe how the algorithms described in the previous section were used to render the astrophysical objects. The three objects that were rendered were a nebula, a gravitational wave and a even horizon of a black hole. In order to describe the algorithms, one must explain some preliminary ideas behind the implementation of the REYES architecture, which is the rendering pipeline that RenderMan uses, and how this pipeline was taken advantage of. Another important topic are RenderMan shaders. Various shaders were used to give a desired effect on the objects rendered.

4.1 RenderMan Pipeline

4.1.1 Taking Advantage of the Pipeline with RIB Archives

There are a few aspects of the RenderMan pipeline that are important to keep in mind when considering where optimizations can be made. RenderMan is based on the Reyes architecture [7] as describe in Section 2.1.3. The renderer dices geometry into micro polygons which are the fundamental unit used in rendering. Cook states that the extra work that the renderer does is related to the depth of the scene.[7] This is an important concept because the spec was designed originally for rendering feature films and effects. Models for films are like sets, where not all the models are built. In our case we are using geometry generated by scientific data so we don't have the freedom to designed the models in such a way that will speed up rendering.

In rendering the gravitational wave, there were several transparent layers that were being shaded. This resulted in a large amount of memory being used for rendering the image. The problem was solved by using delayed read archives in PhotoRealistic RenderMan.

4.2 RenderMan Shaders

The RenderMan shaders used were a Phong shader, a star shader, a Gooch shader and a cloud shader. This section will describe and explain the shaders used.

4.2.1 Phong Shader

Phong shading [25], also known as "normal-vector interpolation shading," [11] is an improvement of Gouraud shading. Gouraud shading is based on interpolating the shading intensity between each normal. The color value is calculated at every vertex, and is linearly interpolated over the polygon. Phong takes the idea and improves it by interpolating the normal between two vertices and shading based on the normal through the following equation:

$$N_t = tN_1 + (1-t)N_0 \qquad [25] \tag{18}$$

where $t = 0$ at N_0 and $t = 1$ at N_1.

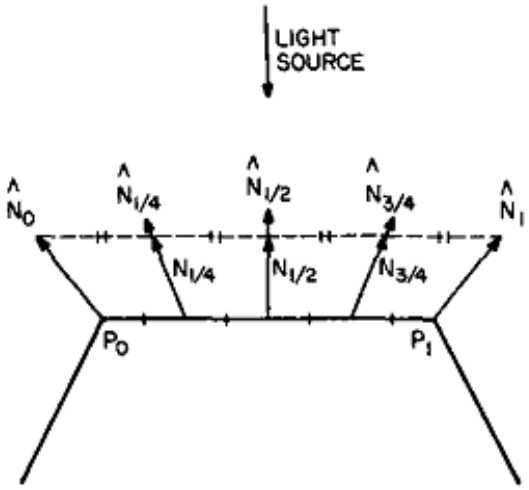

Figure 17: Normal interpolated between two points. [25]

This interpolation can be seen in Figure 17. This results in a smoother gradient and reduces the Mach Band effect. [25] The Mach Band effect occurs when there is

a discontinuity in magnitude or slope of intensity.[11] The human eye exaggerates the change in intensity of color at an edge. At the border a dark facet would look darker and a light facet would look lighter. This effected is illustrated in Figure 18. Mach banding is caused by latter inhibition of the receptors in the eye. Each receptor in the eye inhibits the receptors next to it based on how much light it receives in an inverse relation to the adjacent receptor.

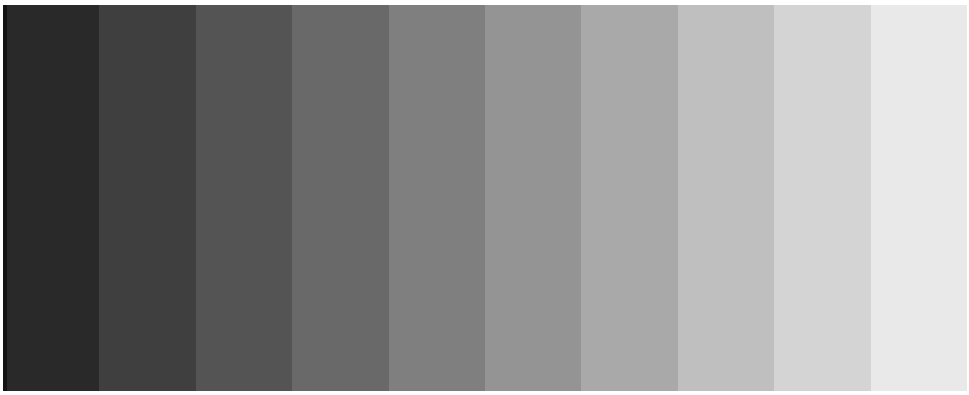

Figure 18: Example of the Mach Band effect. [6]

The Phong illumination model assumes that the maximum specular reflection occurs when the angle between the viewer and R, the reflected light vector, is 0. The Phong illumination model can be expressed as the following equation:

$$I_\lambda = I_{a_\lambda} k_a O_{d_\lambda} + f_{att} I_{p_\lambda} [k_d O_{d_\lambda} (\bar{N} \cdot \bar{L}) + k_s O_{s_\lambda} (\bar{R} \cdot \bar{V})^n] \qquad [11] \tag{19}$$

In the illumination equation above I is the resulting object color. k represents that material coefficients and O represents the object colors. k_a, k_d and k_s are the ambient, diffuse and specular coefficients respectively. $\bar{N}$ and $\bar{L}$ are the normal and the light vector respectively. The specular component is calculated via an exponential curve where the result of the dot product between the reflection vector $\bar{R}$ and the view vector $\bar{V}$ is raised to the nth power. $\bar{R}$ is calculated via the following equation:

$$\bar{R} = 2\bar{N}(\bar{N} \cdot \bar{L}) - \bar{L} \qquad [11] \tag{20}$$

This equation is illustrated graphically in Figure 19.

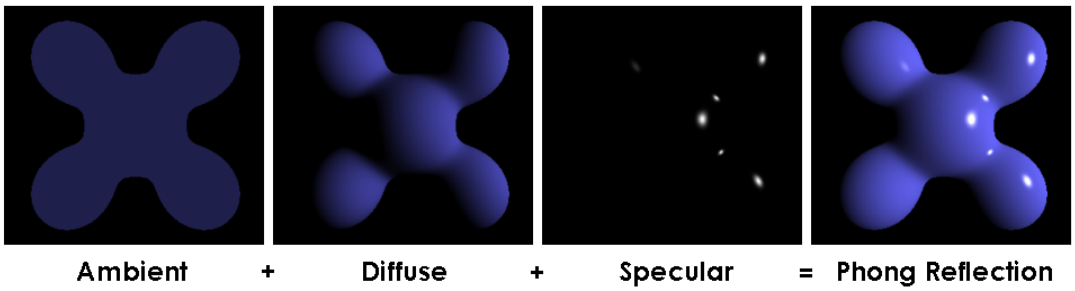

Figure 19: Illustration of Phong components. [29]

4.2.2 Gooch Shader

The Gooch Shader was used when shading the gravitational wave. The model is based on traditional technical illustration where the lighting model uses both luminance and changes in hue to indicate surface orientation. [12] Gooch et al. have observed a few characteristics that technical illustrations have in common. These are: edge lines are drawn with black curves, matte objects are shaded with intensities far from black or white with warmness or coolness of color determined by surface normal, shadowing is not shown and metal objects are shaded as if very anisotropic. In classic shading, the luminance is proportional to the cosine of the angle between the light and the surface normal of the object. This can be expressed as

$$I = k_d k_a + k_d max(0, \hat{i} \cdot \hat{n}) \qquad [12] \tag{21}$$

where I is the final color and k_d is the diffuse component and k_a is the alpha component and $\hat{I}$ is the angle of the light to the surface and $\hat{n}$ is the normal of the surface. Gooch et al. modified this equation to blend between two RGB colors based on the angle between the surface normal and the light. The equation is as follows:

$$I = (\frac{1 + \hat{i} \cdot \hat{n}}{2})k_{cool} + (1 - \frac{1 + \hat{i} \cdot \hat{n}}{2})k_{warm} \qquad [12] \tag{22}$$

Implementing this type of shader is simple in the RenderMan Shading Language. An example is seen in Listing 2.

```
surface Gooch(color specularcolor = 1, warmcolor = color(0.878, 0.996,
    0.474),
     coolcolor = color(0.580, 0.690, 0.988); float diffusewarm = .45,
     diffusecool = .45)
{

  normal Nf;
  vector V;

  Nf = faceforward( normalize(N), I );
  V = -normalize(I);

  // Set Ci and Oi
  Oi = Os * pow((1 - max(0, Nf.V)), 2);
  Ci = Oi * mix(coolcolor , warmcolor, diffuse(Nf));

}
```

Listing 2: Example of a Gooch surface shader

In Listing 2 the shader uses a mix function that is available in the RenderMan Shading Language, that combines the warm and cool shades depending on the diffuse component which is calculated through the built in diffuse() function. The opacity is also set based on the angle towards the viewer because this shader was used to shade a gravity wave. Changing the opacity based on the angle towards the user provided better results than keeping the opacity constant.

Figure 20 shows a render with the shader in Listing 2. The colors go from cool blue to the warm yellow.

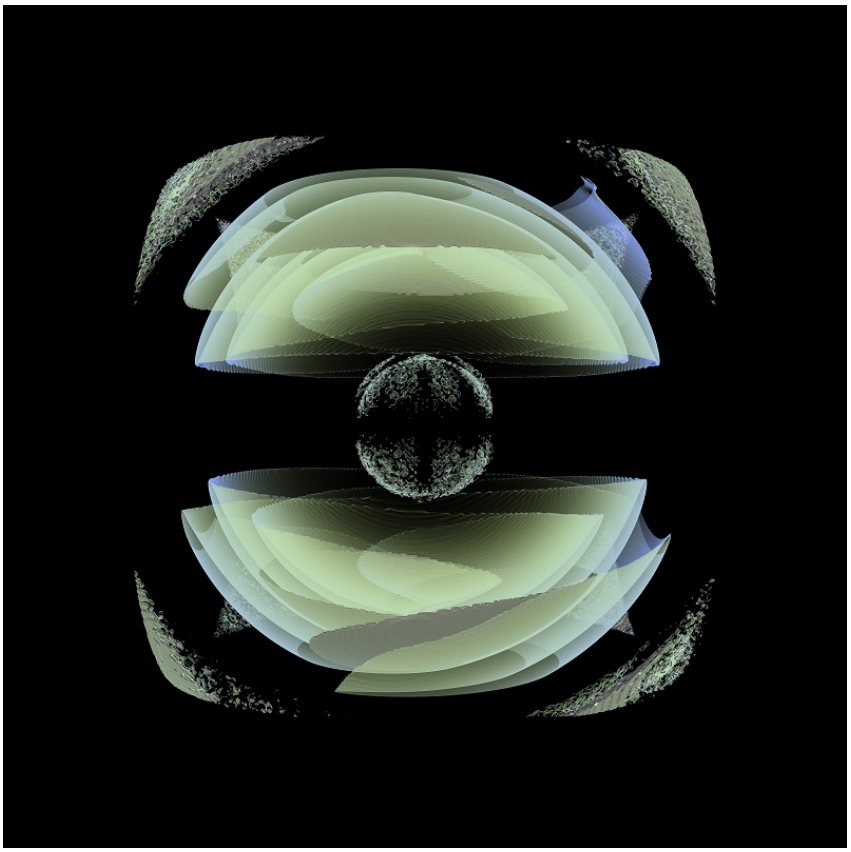

Figure 20: Gravity Wave shaded with the Gooch Shader.

4.2.3 Star Shader

The star shader is a very simple sphere shader. As the angle goes outward, the star becomes more transparent. The color also becomes more saturated. If the color chosen is blue for example, the star will be white in the center and blue around the edges. This is done by mixing the color blue from the outside edge to the inside based on the angle between the normal and the viewer. An example of the star shader can be seen in Listing 3.

```
surface plain_star(color c = (0.3, 0.3,0.75);)
{

  normal Nf;
  vector V;

  Nf = faceforward( normalize(N), I );
  V = -normalize(I);
  float angle = 0;
  color Color = 0;
  if(N.I > 0)
  {
    Ci = 0;
    Oi = 0;
  }
  else
  {
    angle = max(0, Nf.V);
    Ci = mix(c, color(1.0, 1.0, 1.0), angle);
    Oi = angle;
  }
}
```

Listing 3: The star shader

4.2.4 Nebula Cloud Shader

The cloud shader is based on the Gardner cloud shader in Advanced RenderMan.([2], page 413) The noise function in the shader is modified to create fractal noise to produce a different effect than the standard shader in the book. The coordinates that are passed into the noise function, are themselves displaced by a noise function. This creates a more wispy looking cloud as seen in Figure 21.

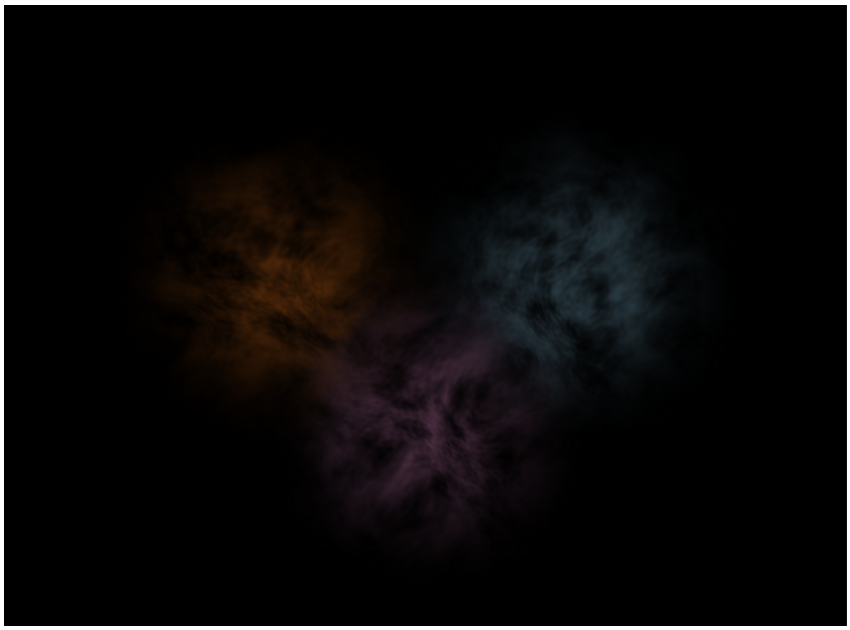

Figure 21: Three spheres shaded with the nebula cloud shader.

The shader also makes opacity fall as one travels from the center to the outside of the sphere. The opacity also falls off as one goes from the front to the back. The shader is below in Listing 4.

```
#include "noises.h"


surface nebula (float Kd = 0.5;
        string shadingspace = "world";
  /* Controls for turbulence on the sphere */
  float freq = 1, octaves = 8, lacunarity = 2, gain = 0.5, wispyness = 2;
  /* color */
  color Color = color(.8, .5, .7);
```

```
  /* Falloff control at edge of sphere */
  float edgefalloff = 8;
  /* Falloff controls for distance from camera */
  float distfalloff = 1, mindistfalloff = 1000, maxdistfalloff = 2000;
    )
{
    point Pshad = freq * transform (shadingspace, P/15);
    float dPshad = filterwidthp (Pshad);
    float f = freq;
    float weight;
    if (N.I > 0) {
  /* Back side of sphere... just make transparent */
    Ci = 0;
    Oi = 0;
    } else {  /* Front side: here's where all the action is */
    float opac = 0;

    opac += fBm(point(xcomp(Pshad) + wispyness *
          fBm(Pshad/10,dPshad/10, 3, 5, .5),
          ycomp(Pshad) + wispyness *
          fBm(Pshad/10,dPshad/10, 3, 5, .5),
          zcomp(Pshad) + wispyness *
          fBm(Pshad/10,dPshad/10, 3, 5, .5)),
      dPshad, octaves, lacunarity, gain);

    opac = smoothstep (-1, 1, opac);
    /* Falloff near edge of sphere */
    opac *= pow (abs(normalize(N).normalize(I)), edgefalloff);
    /* Falloff with distance */
    float reldist = smoothstep(mindistfalloff, maxdistfalloff,
            length(I));
    opac *= pow (1-reldist, distfalloff);
```

```
    color Clight = 0;
    illuminance (P) {
      /* We just use isotropic scattering here, but a more
       * physically realistic model could be used to favor
       * front- or back-scattering or any other BSDF.
       */
      Clight += Cl;
    }

    Oi = opac * Oi;
    Ci = Kd * Oi * Cs * Clight * Color;
    Oi -= 0.1;
    }
}
```

Listing 4: The nebula shader

4.3 Nebula Render

The nebula dataset contained thousands of stars. The objective was to make this mass of stars look like a cloudy nebula and colored based on the densities of the spheres. The stars are put into a grid based on the density of stars in the area and a cloud radius threshold. The cloud radius threshold defines how many spaces the grid is divided into. Every time a star is inserted, it is checked if another star already exists in the area. If it does, the star's number of neighbors is incremented. If not, the star is put into the list. For every point on the grid, a sphere is created with a user specified radius and inserted into the list of objects to be rendered. The user also defines whether or not the stars will be rendered. If they are, then the stars are also put into the list to be sent to be rendered. Each star is assigned a random size in a certain range upon creation. The clouds were also assigned a color based on the density. The color was translated into the HLS color

model, and the L value was modified based on the maximum density.

The spheres are rendered using the cloud and star shaders. The results looked better when the clouds were in world space, and not shader space. Since each cloud had a different position for every frame, it was necessary to keep it consistent looking. Moving the sphere and shading in shader space made a completely new cloud appearance even if the sphere was moved slightly. This was solved by shading in world space. Since we don't care about each individual cloud that is being shaded, we just want a complete looking cloud, shading in world space was optimal. Results of the render are seen in Figure 22.

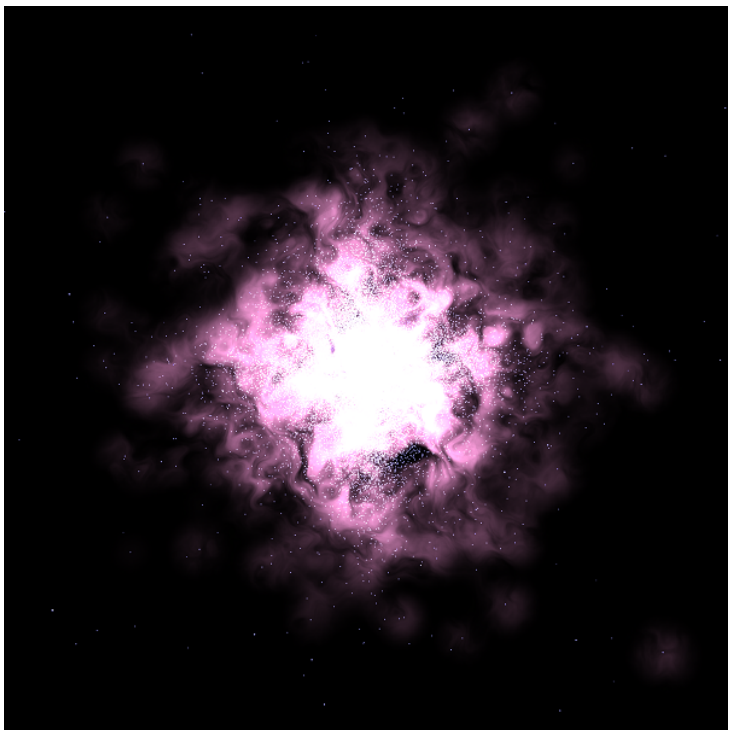

Figure 22: Result of the nebula rendered with stars.

4.4 Gravitational Wave

The gravitational wave was rendered using the Marching Cubes Algorithm. The algorithm was used to polygonize the dataset into a surface. The cloud data set is already in a 200x200x200 grid. Implementing the Marching Cubes Algorithm on such a dataset was straightforward.

4.4.1 Implementation of Marching Cubes Algorithm

The data was organized into point data within a grid. In order to get a quick idea of what the data set looked like, python scripts were written to generate the data for a RIB file. Procedural primitive support was used to run the scripts from within the RIB file. The first objective was to see what the density data looked like. When outputting the data, the HLS color space was used to help visualize the data. Each data point was rendered using the Point primitive. Figure 23 shows the result of rendering the point data. The hue value corresponds to the density.

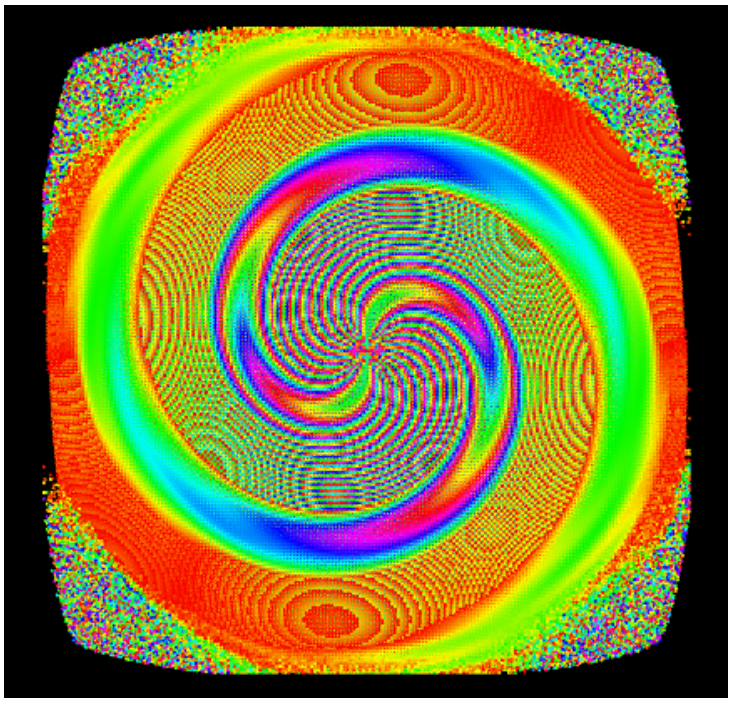

Figure 23: Density rendered as points using the H component of the HLS color model.

In order to better visualize the data, the L component of the HLS color model was used to render Figure 24.

The image still had rings that occurred. The question is whether the rings were from the data or whether it was aliasing caused by the sampling of the points in the renderer. In order to determine what was causing the rings, each datapoint in the grid was rendered with a sphere. The first 20 slices of the grid were rendered each time because of memory

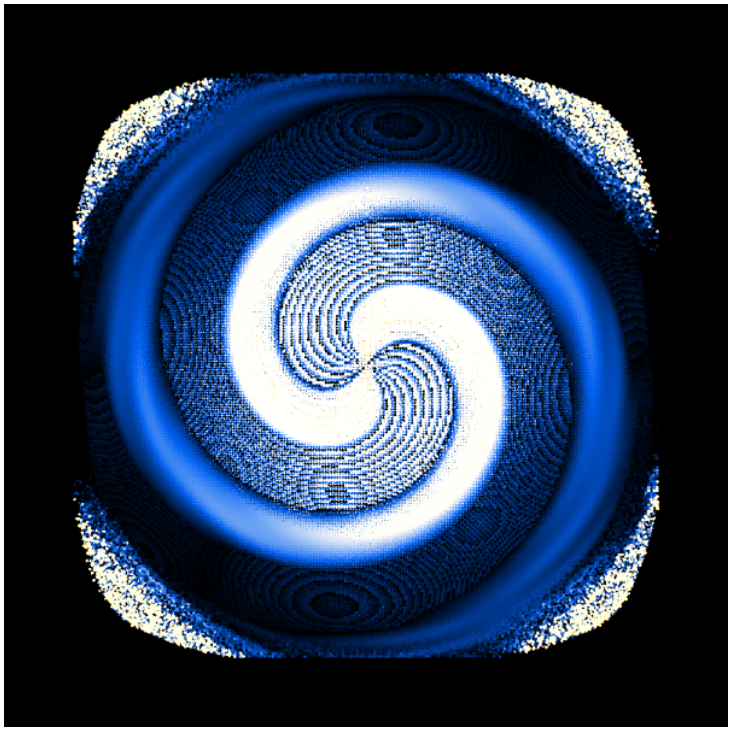

Figure 24: Density rendered as points using the L component of the HLS color model.

limitation. In Figure 25

This showed that the rings were inherently in the data and were not the result of aliasing.

The Marching Cubes Algorithm, as described in Section 3.3.1, was used to polygonize the data. The eight corners of the cube were each assigned a datapoint on the grid. This required the grid to be at least 2x2x2 in order to have eight vertices. The cube was placed in to the upper front left corner of the grid and was moved by one edge to the right. When it reached the end, it was moved back to the front and down one point. When it reached the bottom, it starts the process all over again from the top left corner, this time moving one length down in the z-direction. This process continues until the cube travels through the grid. The same data shown in Figure 25 was polygonized and shaded with a Phong shader in Figure 26.

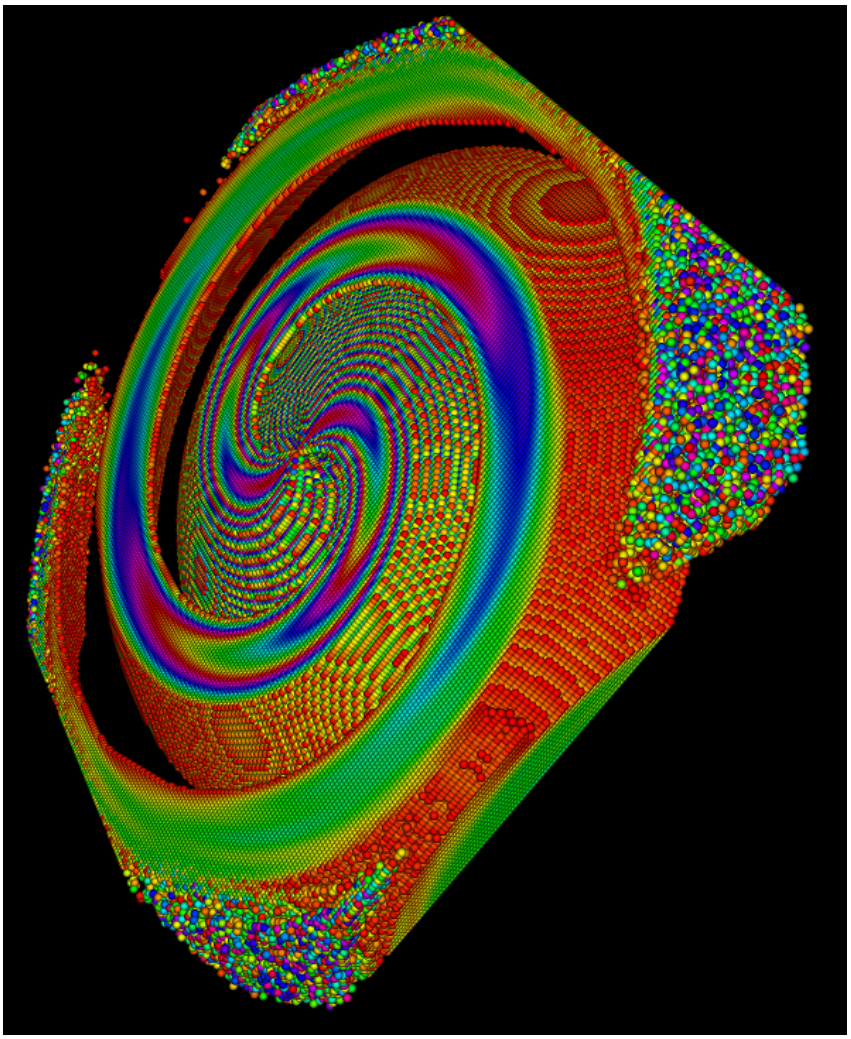

Figure 25: Density rendered as spheres using the H component of the HLS color model.

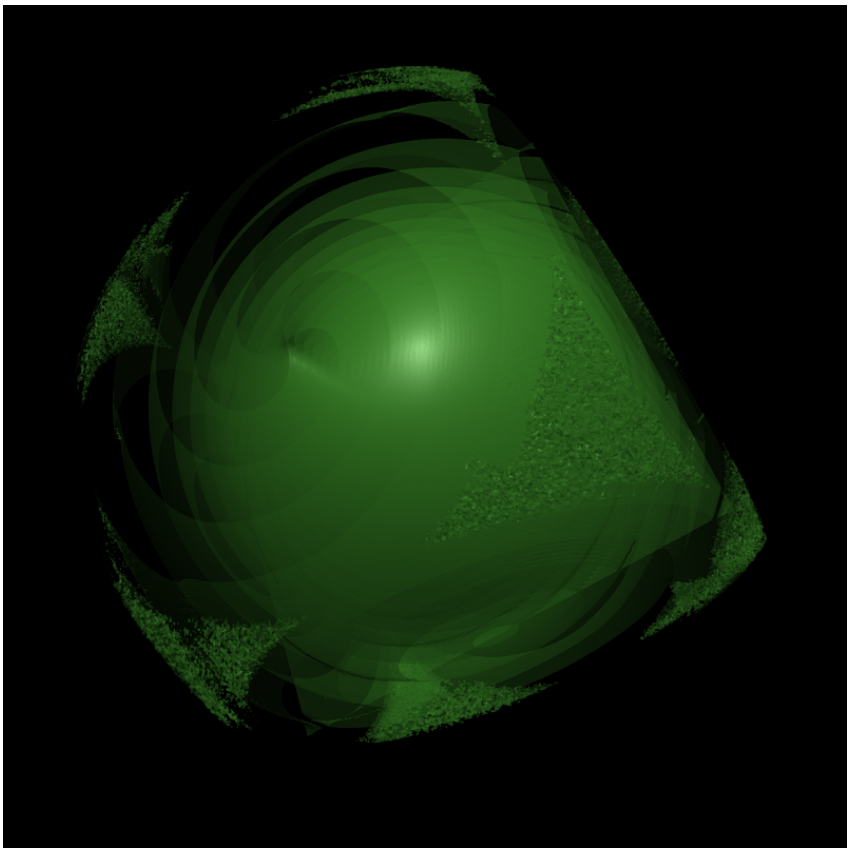

Figure 26: The result of the Marching Cubes algorithm on a density threshold of 0.

4.4.2 Optimizing RenderMan Memory Usage through Delayed Read Archives

The implementation used upwards of eight gigabytes of memory while running, which resulted in swapping and inefficient use of the CPU while rendering. This significantly increased rendering times.

In order to optimize memory usage RenderMan's Delayed Read Archives were used. Delayed Read Archives allow RenderMan to load geometry data into memory when it's bounding box is reached while rendering and later removes the data from memory when it is not needed. This saves physical ram and significantly increases rendering speed. In order to implement this, the grid was subdivided into smaller grids. This allowed one to output the data into multiple files to avoid loading all the polygons into memory at once. The subdivided grid cubes were defined by the user. For example if the user entered 40, then a 200x200x200 grid would be divided into 125 individual cubes which were then output to individual files.

4.5 Surface Reconstruction of a Point Cloud

The dataset for the point cloud was a torus with unevenly spaced sample points. An algorithm that can polygonize a non-convex and non-uniformly sampled surface had to be used for reconstruction. Several algorithms are described in Section 3.3. The final algorithm used was based on two existing algorithms, the Ball-Pivoting Algorithm and the Intrinsic Property Driven Algorithm.

4.5.1 Polygonizing a Point Cloud of a Surface

Reconstructing a point cloud was done through a modified ball-pivoting algorithm as described in Section 3.4.3. Instead of using a ball-circle collision (see Section 3.4.1) to check which point is first, an angle and radius was used. This checked if the point could be inside a sphere. In order to make sure that the point was consistent with the current surface that was generated it's angle was compared to the current edge's tangent vector.

This idea was taken from the Intrinsic Property Driven Algorithm as described in Section 3.4.2. The authors find if a point is within a geometrically defined influence region. In this case, instead of checking if a point is within a bounding volume, the angle of the next point to the current surface tangent was checked.

Rendering the torus point cloud in Figure 27 resulted in the image in Figure 28.

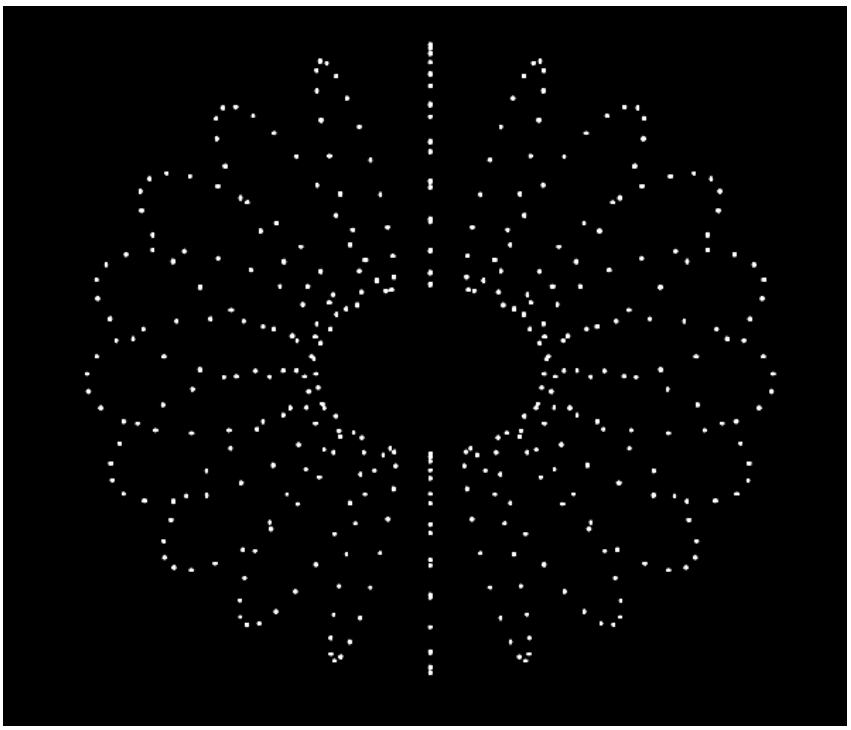

Figure 27: The torus point cloud.

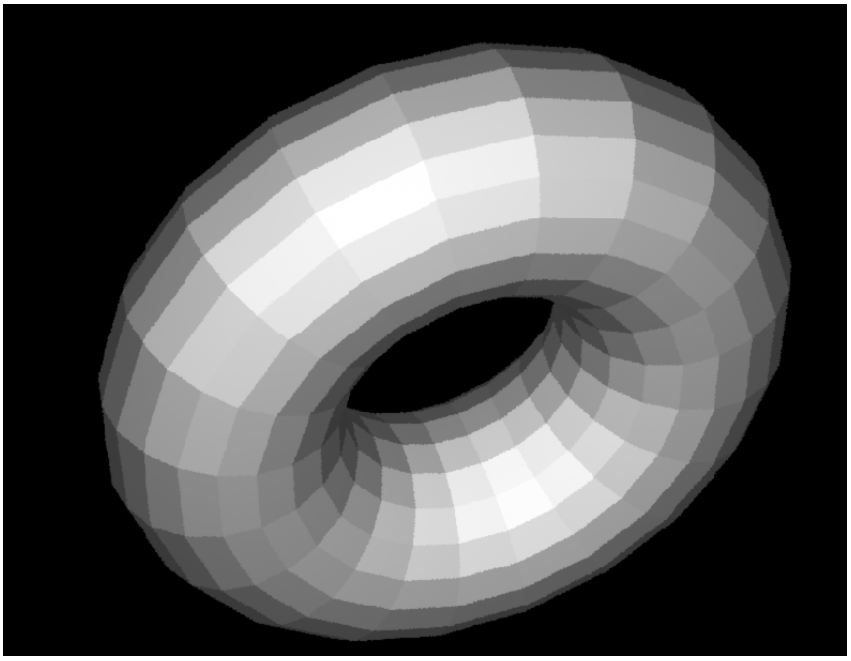

Figure 28: Poligonization of the torus point cloud.

5 Results

5.1 Data Generation and Render Benchmarks

This section summarizes the render times of the three objects with various settings specific to each object. The images were rendered on a PC running Windows 7 (64-bit) with an Intel Core i5 Lynnfield 45nm processor clocked at 2.67 ghz, and 4GB Dual-Channel DDR3 ram at 668Mhz with 7-8-7-24 timings on a ASUSTeK PCP55D-E LX motherboard.

5.1.1 Nebula Render

Of interest in the nebula render is the radius and cloud threshold value of the cloud spheres. This determines how many cloud spheres are generated and the time it takes to render. The sphere size is the radius of the sphere and the cloud threshold determines how far apart the sphere are. Both are factors that affect the length of time it takes to render an image.

Nebula Render Time Results		
Cloud Threshold	Render Time (Sphere size 14)	Render Time (Sphere size 30)
.20	6.6s	11.0s
.14	8.8s	22.6s
.10	14.5s	45.8s
.05	44.2s	100.6s

The benchmarks correspond to the rendered images in Figure 29. The top row was rendered with a sphere size of 14 with thresholds of .05, .1, .15, and .20 from left to right respectively. The bottom row used a sphere size of 30 with the same threshold values.

5.1.2 Survey

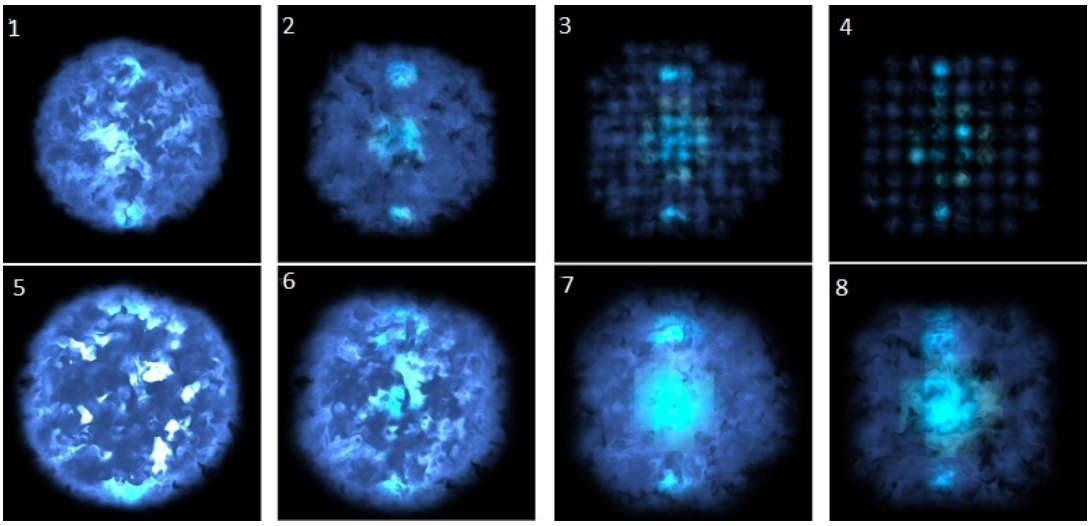

Figure 29: The image that was used to survey the accuracy and aesthetics of the images rendered above. Top row is sphere size 14 with thresholds of .05 .1 .15 and .20. The bottom row uses sphere size 30 with the same threshold values.

11 participants were asked to rate the images in Figure 29 based on how closely they convey the data in the actual raw point data rendered in Figure 30. They were asked to rate on a scale of 1 to 10 keeping in mind both how well the data is conveyed and the aesthetic qualities of the image. The table below shows the results of the survey. It

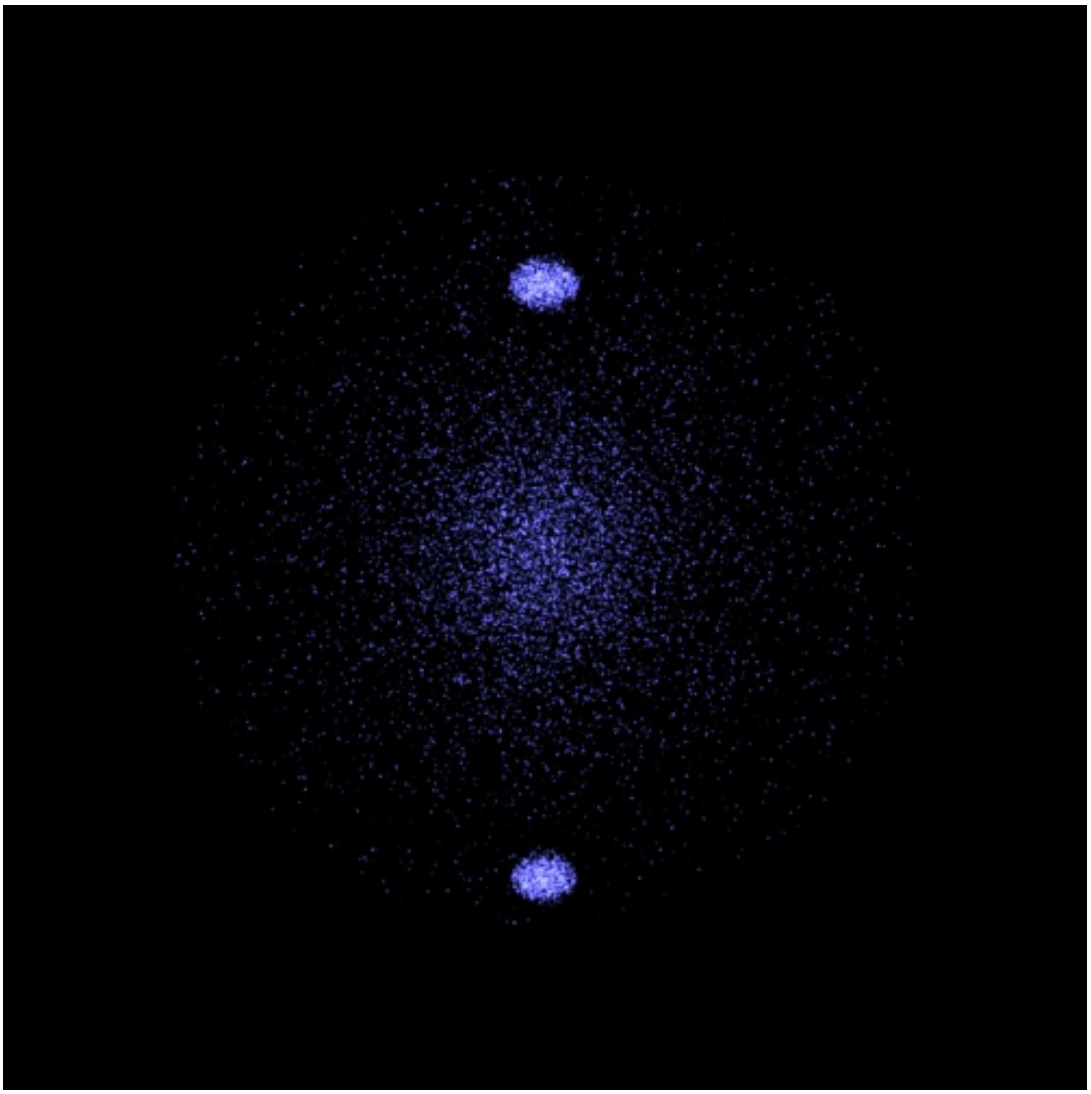

Figure 30: The image with the point data that is rendered in Figure 29.

shows that using a sphere of size 14 and threshold values of .05 and .1 provided the best results.

Nebula Image Ratings Survey								
Image Number (see Fig. 29)	1	2	3	4	5	6	7	8
Average Rating	6.6	7.5	5.1	3.2	5.6	5.4	4.8	4.1

5.1.3 Gravitational Wave with Marching Cubes

Determining how to subdivide the grid of data points is the main factor that will influence how long it takes to render. The point grid was subdivided into a cube of size 40x40x40, 50x50x50, 100x100x100, and 200x200x200. The data is run on a 200x200x200 size grid. The following is the result of the render times.

Gravitational Wave Render Time Results			
Sub-Grid Dimensions	813,290 triangles	2,465,770 triangles	2,840,238 triangles
40x40x40	224.9s	626.4s	865.4s
50x50x50	185.9s	523.2s	666.8s
100x100x100	151.8s	422.1s	481.4s
200x200x200	146.9s	465.4s	664.5s

In the table above one can see that the less subdivisions there are, the faster it writes the files to render, until we run out of memory. This happens with the higher counts of triangles. In the first column with 813,290 triangles, the renderer is capable of handling all the polygons at once in memory. The next two columns show a slow down that occurs when the renderer begins to swap memory. The most optimal subdivisions are eight 100x100x100 cubes in the last two columns.

5.1.4 Surface Polygonization

Polygonization times are determined by the number of points. The following is a table of the number of points of certain models used to benchmark the algorithm.

Point Cloud Poligonization Results			
Object	Number of Points	Triangles Generated	Render Time
Torus	400	834	4.5s
Teapot	3644	8745	11.7s
Bunny	2503	6339	20.8s

6 Conclusion

The thesis shows that RenderMan is a viable alternative to OpenGL when visualizing astrophysical data. The astrophysical data was rendered into an easy to understand and an aesthetically pleasing form and can be rendered in a timely manner on consumer hardware.

6.1 Nebula Poll Results

The poll results showed that the nebula that is rendered can indeed convey the information as well as make the image look aesthetically pleasing to the viewer. The highest rating was 7.5 which means that there are still improvements that could be made to the renders to more clearly convey the data.

6.2 Future Work

Future work in visualizing the astrophysical objects will depend on the data that is being rendered. This thesis created an easily modifiable pipeline, but there are still many aspects of RenderMan that have not been implemented. Currently there is only support for rendering spheres and polygons. RenderMan has many more built in primitives that have no been written in Spiegel. Many optimizations could be made to the algorithms to speed up the rendering time such as using optimized data structures such as kd-trees to store the data instead of grids. Many of the algorithms in Java could be optimized to run on more than one processor. The marching cubes algorithm for example could be parallelized to run on more than one processor when generating the data for individual RIB archive files. The algorithm's speed can also be improved by not writing the rib files to disk.

The way the scripts were designed to render the objects gives the user many options. The nebula has the most options which can result in many different possibilities of rendering that have not yet been explored in this thesis. A different shader could

also be written with the advent of faster machines in the future which could utilize more complicated rendering methods that are physically based. This modular design of the implementations provides a base that can currently be used to render scientific data, but can also be modified in the future based on objects that might be modeled in the future.

References

[1] *Microsoft Developer Network Online Documentation*. http://msdn.microsoft.com.

[2] Anthony A. Apodaca and Larry Gritz. *Advanced RenderMan: Creating CGI for Motion Pictures*. Academic Press, San Diego, CA, 2000.

[3] Fausto Bernardini, Joshua Mittleman, Holly Rushmeier, Claudio Silva, Gabriel Taubin, and Senior Member. The ball-pivoting algorithm for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics*, 5:349–359, 1999.

[4] Hans-Peter Bischof, Edward Dale, and Tim Peterson. Spiegel - a visualization framework for large and small scale systems. In *MSV*, pages 199–205, 2006.

[5] OpenGL Architecture Review Board, D. Shreiner, and et al. *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Version 2*. Addison Wesley, 2005.

[6] Department of Systems Carleton University and Computer Engineering. Elg 7173 - human visual system. `http://www.sce.carleton.ca/faculty/adler/elg7173/visual_system/elg7173-visual-system.html`. Accessed May 23, 2011.

[7] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The reyes image rendering architecture. *Computer Graphics*, 21(4):95–102, 1987.

[8] Brian Corrie and Paul Mackerras. Data shaders. In *In Proc. Visualization 93. IEEE CS*, pages 275–282. Press, 1993.

[9] Herbert Edelsbrunner and Ernst P. Mücke. Three-dimensional alpha shapes. In *Proceedings of the 1992 workshop on Volume visualization*, VVS '92, pages 75–82, New York, NY, USA, 1992. ACM.

[10] Randima Fernando, editor. *GPU gems: programming techniques, tips, and tricks for real-time graphics*. Addison-Wesley, pub-AW:adr, 2004.

[11] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer graphics: Principles and practice in C*. Addison-Wesley, 2nd edition, 1996.

[12] Amy Gooch, Bruce Gooch, Peter Shirley, and Elaine Cohen. A non-photorealistic lighting model for automatic technical illustration. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '98, pages 447–452, New York, NY, USA, 1998. ACM.

[13] Hans Hagen, Achim Ebert, Rolf Hendrik van Lengen, and Gerik Scheuermann. Scientific visualization - methods and applications. *Lecture Notes in Computer Science*, 2000/2001:311–327, 2001.

[14] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculations. *Computer Graphics*, 24(4):289–298, 1990.

[15] Hong-Wei Lin, Chiew-Lan Tai, and Guo-Jin Wang. A mesh reconstruction algorithm driven by an intrinsic property of a point cloud. *Computer-Aided Design*, 36(1):1 – 9, 2004.

[16] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics*, 21(4):163–169, 1987.

[17] Dan Maas. What the rispec never told you. In *ACM SIGGRAPH 2006 Courses*, SIGGRAPH '06, New York, NY, USA, 2006. ACM.

[18] Marcus Magnor, Gordon Kindlmann, Neb Duric, and Charles Hansen. Constrained inverse volume rendering for planetary nebulae. *IEEE Visualization*, pages 83–90, 2004.

[19] David McDonald and Robert F. Erbacher. Hardware accelerated graphics in java. December 30 2008.

[20] Timothy S. Newman and Hong Yi. A survey of the marching cubes algorithm. *Computers and Graphics*, 30(5):854 – 879, 2006.

[21] John D. Owens, Brucek Khailany, Brian Towles, and William J. Dally. Comparing reyes and opengl on a stream architecture. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '02, pages 47–56, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[22] B.A. Payne and A.W. Toga. Surface mapping brain function on 3d models. *Computer Graphics and Applications, IEEE*, 10(5):33 –41, sep 1990.

[23] Ken Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19:287–296, July 1985.

[24] Ken Perlin. Improving noise. *ACM Trans. Graph.*, 21:681–682, July 2002.

[25] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18:311–317, June 1975.

[26] Peter Shirley and Allan Tuchman. A polygonal approximation to direct scalar volume rendering. In *Computer Graphics*, pages 63–70, 1990.

[27] Alvy Ray Smith. Color gamut transform pairs. *SIGGRAPH Comput. Graph.*, 12:12–19, August 1978.

[28] Wikipedia. Marching cubes. `http://en.wikipedia.org/wiki/Marching_cubes`. Acessed May 19, 2010.

[29] Wikipedia. Phong shading. `http://en.wikipedia.org/wiki/Phong_shading`. Acessed May 23, 2010.

[30] Matt Zucker. The perlin noise math faq. `http://webstaff.itn.liu.se/~stegu/TNM022-2005/perlinnoiselinks/perlin-noise-math-faq.html`, February 2001. Acessed May 19, 2010.