8-2013

3D Fractal Flame Wisps

Yujie Shu
Clemson University, yujies@clemson.edu

3D Fractal Flame Wisps

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Fine Arts
Digital Production Arts

---

by
Yujie Shu
August 2013

Accepted by:
Dr. Jerry Tessendorf, Committee Chair
Dr. Robert Geist
Dr. Donald House

Abstract

This thesis presents a method for integrating two algorithms, fractal flames and wisps, to create visually rich and interesting patterns with 3D volumetric structure. Twenty-one single 3D flame variations are described and specified. These patterns were used to produce an aesthetically designed animation, inspired by both Hubble Telescope photographs and data from a simulation of a predicted collision between the Milky Way and Sagittarius galaxies. The thesis also describes Python tools and a Houdini pre-visualization pipeline that were developed to facilitate the animation design and production.

Acknowledgements

I would like to express my sincere grattitude to my advisor, Dr. Jerry Tessendorf, for providing guidance, support, and inspiration throughout my research. His insight into visual effects industry and his mentorship are invaluable to this work.

I would also like to offer special thanks to committee member, Dr. Robert Geist, for introducing me to the Digital Production Arts Program and the field of Computer Graphics. My sincere thanks also go to committee member, Dr. Donald House, for introducing me to Physically Based Animation.

In addition, I would like to thank Dr. Jeanette Myers and Dr. Lih-Sin The in Physics and Astronomy Department for providing galactic simulation data. Also, special thanks go to a DPA alumnus Robert Helms who gives an important feedback to the galaxy animation.

Finally, I would like to thank my mother, Xianping Shu, and my hunsband, Weixin Wu, who have always supported me.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this thesis, the word "flame" does not refer to combustion products but to the artistic visual style pioneered in fractal flames. In 1992, Scott Draves developed the fractal flame algorithm which uses a 2D Iterated Function System (IFS) of non-linear functions to create enchanting images. The three main differences between fractal flame algorithm and standard IFS functions are: (1) affine transformations are replaced by non-linear functions, (2) tone mapping is used in fractal flames to generate log-density displays, and (3) color is assigned to the flame according to the density map [1].

Richard "Doc" Baily was one of the pioneers who brought fractal flames into film production. His feature film credits include *Superman Returns* (2006), *Stay* (2005), *The Core* (2003), and *Solaris* (2002) [2]. He developed a proprietary high-speed particle renderer, SPORE, to generate fascinating 3D fractal flames. The SPORE system uses a proprietary method to build a collection of iterated functions that tranform a 3D seed point into another 3D output point [3].

The images in Figure 1.1 [4] are examples of his fractal flames with artistic style. The image on the left appears as a black hole and indicates the swirl motion with stylized strokes. The one on the right, named "Angel", illustrates a comet with a nice tail in front of the aurora. In both images, those fractal flames are created by SPORE using millions of particles. In this thesis, I hope to achieve this type of artistic feel from wisps and volume rendering.

Widely adapted in medical and data visualization, volume rendering has become an essential tool in visual FX production. It deals with true 3D volumes, including volume modeling, volume animation, and photorealistic rendering [5]. One of the primary applications of volume rendering in film production is to generate FX elements such as clouds, smoke, fire, wisps, water splashes, etc [6].

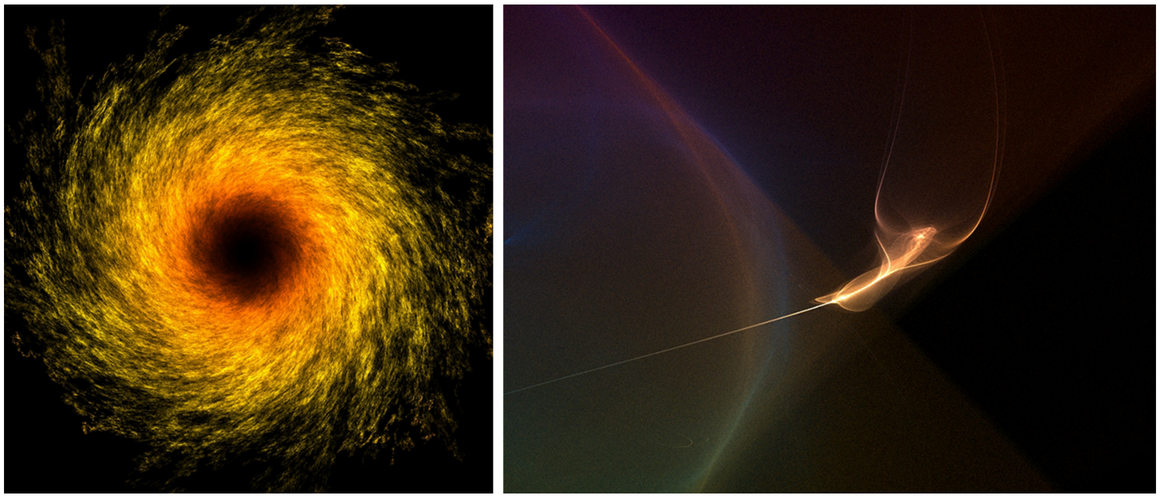


Figure 1.1: Flame Artwork By Richard “Doc” Baily [4]

Fields are mathematical functions that return a value at any point in 3D space. Volume modeling is a process to construct fields. There are five types of fields: (1) scalar field, (2) vector field, (3) matrix field, (4) color field, and (5) signed distance field. A scalar field $f(x)$ returns a scalar value which can be a floating-point value at a point x; a vector field $f(x)$ returns a vector value with 3 scalar components; a matrix field $f(x)$ returns a 3x3 matrix value; a color field F returns a rgba-value at the point x; and a signed distance f returns a floating-point value to represent the distance from a point to a geometry surface [6]. Many math operations can be used on the fields to construct new fields, such as addition, subtraction, multiplication, division, sine, cosine, tangent, etc. Transformations, such as scaling and rotation, can be applied to fields, too.

Generally, there are four ways to animate volumetric data. First, traditional keyframe animation of shapes can be used to drive volumes. Second, Physically Based Simulations can be used to create motion drived by forces. Third are advections on the volume. Fourth, animation on the noise parameters will greatly change the looks of the effects. Some examples are pyroclasts, wisps, auroras and clouds.

To build a volumetrics system, three elements are needed. There are voxel fields, camera information, and lighting information. Volume rendering, the ray marcher system, makes use of this data and generates a final image [5]. To present the realistic look of the translucence within the volumetric data, opacity has to be correctly interpreted in the rendering. The ray marching algorithm accumulates opacity and opacity-weighted color along the viewing direction vector that

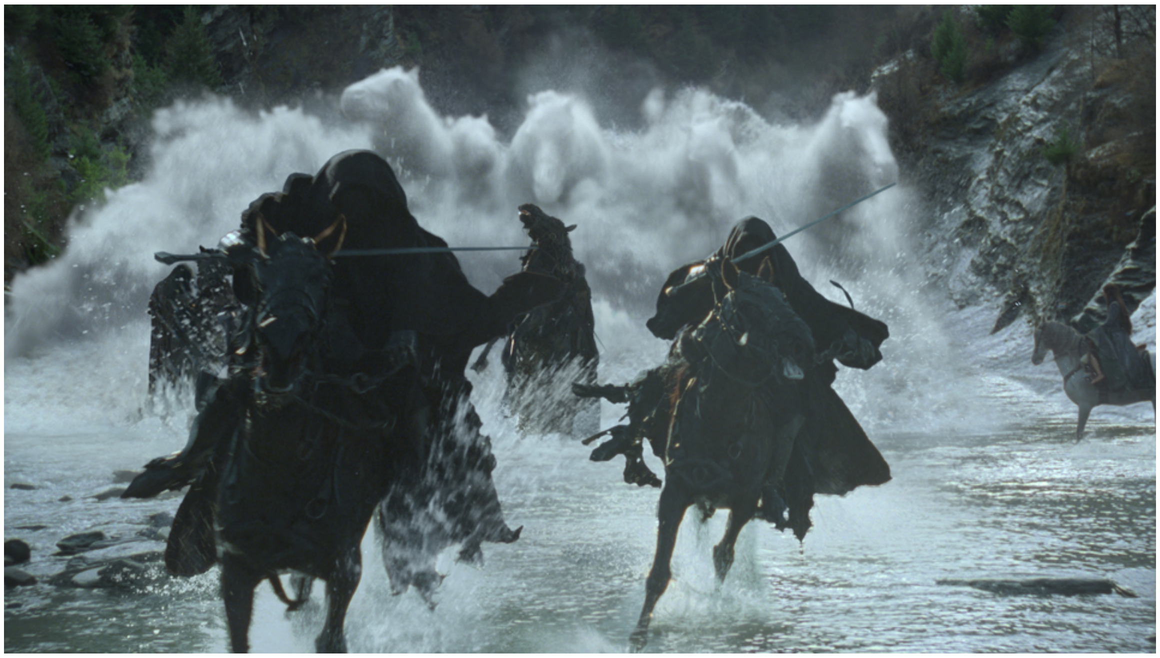


Figure 1.2: The wisp effect from The Lord of the Rings: Fellowship of the Ring [7]
Copyright 2001 New Line Cinema. All rights reserved.

traverses the viewing space [6].

Wisps are a type of volumetric FX generated from many dots sampled in a grid. The algorithm creates wisps using a uniform random number generator and two separate copies of fractal summed Perlin Noise. The random number generator is controled by an input seed parameter. Each wisp has a guide particle with attributes that drive all of the parameters in the algorithm.

Wisps have been used in many feature films, for example, *The Lord of the Rings: Fellowship of the Ring* (2001), *X2: X-Men United* (2003), *Stealth* (2005), *Superman Returns* (2006), *Happy Feet* (2006), *Pirates of the Caribbean: Dead Man's Chest* (2006), *Night at the Museum* (2006), *Pirates of the Caribbean: At World's End* (2007), *Night at the Museum: Battle of the Smithsonian* (2009), and *The A-Team* (2010). Figure 1.2 [7] shows an example of wisps in *The Lord of The Rings: Fellowship of The Ring.* This is the debut of wisps by Alan Kaplan at Digital Domain. Wisps create photo-realistic white water for this river scene with artistically controlled motion.

The Fractal Flame and Wisps generate different visually captivating and enchanting computer graphics, while the first one creates countless patterns and the second one presents elegant 3D structure. This thesis will present a method and implementation to integrate the Fractal Flame and Wisps together to create 3D Fractal Flame Wisps. An effort has been put into the exploration of the artistic look and how to control the look. In order to make an animation with an artistic design, a customized production pipeline has been developed. The rendering system is implemented in C++ and Python and pre-visualization of galaxy animation is implemented in Houdini.

Inspired by Hubble Telescope's space photographs, I made a galaxy animation using those 3D Fractal Flame Wisps I created. The two images in Figure 1.3 are space photographs from Hubble Telescope. The image on the top [8] is the Andromeda Galaxy, which is the nearest spiral galaxy to the Milky Way galaxy. The Great Andromeda Nebula has an interesting wispy look in a disk shape. The one on the bottom [9] is the elliptical galaxy Hercules A. The red radio-emitting object, 3C 348, is the brightest galaxy in the constellation Hercules [10]. This galaxy has two wispy spherical shapes with tails connecting back to the galactic center. The color variance and translucency nicely demonstrate its volume.

Those images from Hubble Telescope are the inspirations for this galaxy animation project. Patterns of 3D Fractal Flame Wisps were investigated to produce a visual similarity between the galaxy nebula and 3D Fractal Flame Wisps. The galaxy could be visualized more artistically using the 3D Fractal Flame Wisp algorithm. Milky Way galaxy and Sagittarius galaxy simulation data were provided by Dr. Jeanette Myers and Dr. Lih-sin The in collaboration with the Physics and Astronomy Department of Francis Marian University and Clemson University. Based on the simulation data, I interpolated, copied, and tranformed the galaxy motion and made a galaxy animation with a center disk galaxy and three smaller galaxy groups.

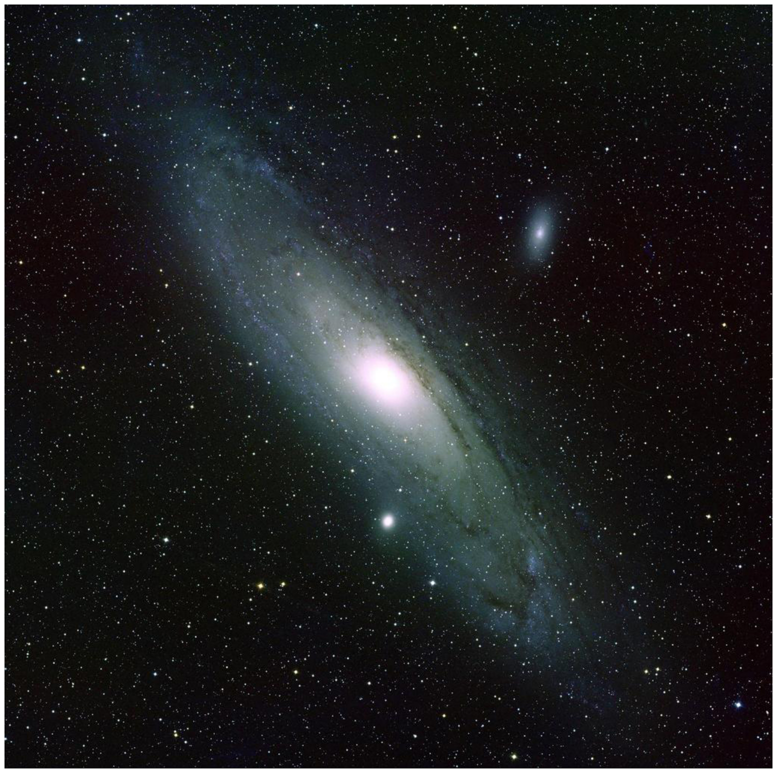
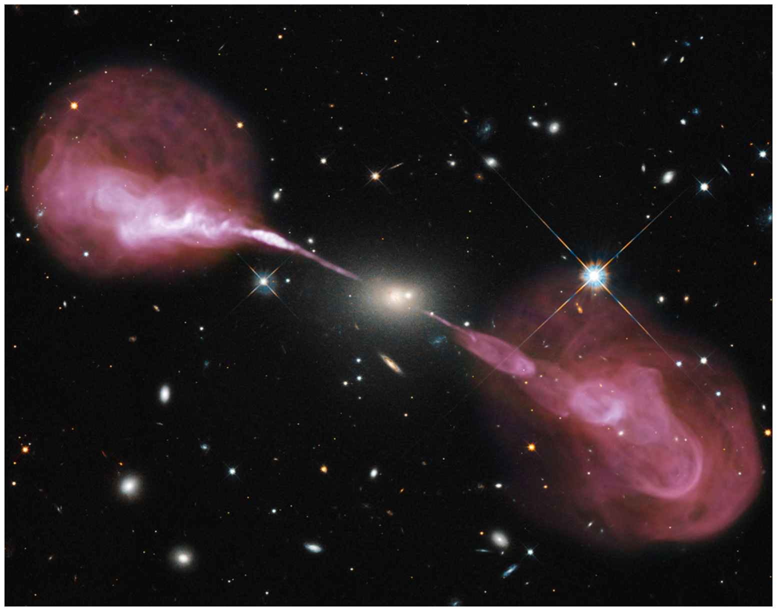



Figure 1.3: Andromeda Galaxy Photograph by Hubble Telescope [8] and Elliptical Galaxy Hercules A [9]

Chapter 2

Background

Iterated Function System (IFS) is a mathematical method for constructing self-similar forms called "fractals." The IFS is a finite collection of functions that transform a point in space $\underline{\overline{X}}$ to another point in the same space [1]:

$$\{f_i : X \to X \mid i = 1, 2, ..., N\},\ N \in \mathbb{N}$$

Fractal Flame IFS, introduced by Scott Draves, is made in 2D space. The propery of the system matches Hutchinson's set equation [11]:

$$S = \bigcup_{i=0}^{n} f_i(S)$$

where the set S is the solution of the system, n is the dimention of the space, and f_i are iterated functions. Scott Draves' Fractal Flame IFS is 2D, but according to the equation, it could be easily extended to 3D space.

Micharel Barnsley created the chaos game algorithm to solve the system to get the fractal image. The algorithm starts with an initial random point and then iterates [12]. Fractal Flames use this algorithm to generate enchanting images. The pseudocode is as follows [1]:

```
(x,y) = a random point in the bi-unit square
iterate
{
      i = a random integer from 0 to n-1 inclusive
      (x,y) = $F_i$(x,y)
      plot(x,y)
}
```

where x and y are both initially in [-1,1]. Because $(x, y) \in S$, $F_i(x, y) \in S$.

One of the innovations of Fractal Flames is the use of non-linear functions. These are composed by transforming the affine functions [1]:

$$F_i(x, y) = V_j(a_i x + b_i y + c_i, d_i x + e_i y + f_i)$$

To extend the Fractal Flame algortithm to 3D, a z-axis element is added in my algorithm:

$$F_i(x, y, z) = V_j(a_i x + b_i y + c_i z + d_i,\ e_i x + f_i y + g_i z + h_i,\ k_i x + l_i y + m_i z + n_i)$$

Each function V_j is a single shape variation with distinct characteristics. In Scott Draves and Erik Reckase's paper "The Fractal Flame Algorithm", they list 49 single variations in the appendix [1]. Here are some 2D example functions:

$V_0(x, y) = (x, y)$

$V_1(x, y) = (\sin(x), \sin(y))$

$V_2(x, y) = \frac{1}{r^2}(x, y)$

$V_3(x, y) = (x \sin(r^2) - y \cos(r^2), x \cos(r^2) + y \sin(r^2))$

$V_4(x, y) = \frac{1}{r}((x - y)(x + y), 2xy)$

where $r = \sqrt{x^2 + y^2}$

In my algorithm, I added the z-axis value to the equations to form 3D patterns for those basic non-linear functions. And each variation had a correspondent flame class in code. For example:

$V_0(x, y, z) = (x, y, z)$

$V_1(x, y, z) = (\sin(x), \sin(y), \sin(z))$

$V_2(x,y,z) = \frac{1}{r^2}(x,y,z)$

$V_3(x,y,z) = (x\sin(r^2) - y\cos(r^2), x\cos(r^2) + y\sin(r^2), x\cos(r^2) - z\sin(r^2))$

$V_4(x,y,z) = \frac{1}{r}((x-y)(x+y), 2xy, (x-z)(x+z))$

where $r = \sqrt{x^2+y^2+z^2}$

Based on those single variations, more variations can be generated through subsequent affine transforms which change the coordinate systems of the variations. The post transform function is generalized as follows:

$$P_i(x,y) = (\alpha_i x + \beta_i y + \gamma_i z + \delta_i,\ \epsilon_i x + \zeta_i y + \eta_i z + \theta_i,\ \kappa_i x + \lambda_i y + \mu_i z + \sigma_i)$$

then F_i is redefined as:

$$F_i(x,y,z) = P_i(V_j(a_i x + b_i y + c_i z + d_i,\ e_i x + f_i y + g_i z + h_i,\ k_i x + l_i y + m_i z + n_i))$$

Accordingly, in my volume rendering code, I built post transformation classes, such as scale, translate, rotation, and parametric transformation classes to accommodate post transformations for flame classes.

Wisps, introduced by Alan Kaplan at Digital Domain, were first used in *The Lord of the Rings: Fellowship of the Ring* (2001). The algorithm of wisps is a 3D volumetric method which generates many dots and samples in a 3D grid.

The wisp generating algorithm uses a uniform random number generator, a guide particle, and two separate copies of fractal sum Perlin noise (FSPN) [13]. In pseudo code it is:

```
for(loop over number of dots for this guide particle)
{
      1.random walk, correlated walk, or levy walk
         $\overrightarrow{X}$(x,y,z) = $R$(x,y,z)
      2.  record the $\overrightarrow{X}$ for color
         $\overrightarrow{X}_{color}$(x,y,z) = $\overrightarrow{X}$(x,y,z)
      3.  map the position to a unit sphere
```

```
        $P_0$(x,y,z) = (x,y,z)/|(x,y,z)|
    4.  displace radially from sphere using FSPN
        $P_1$(x,y,z) = $P_0$r
        r = $|FSPN_1|^{\alpha}$
    5.  map to the guide particle coordinate
        $\overrightarrow{X_0}$ = guide particle position
        $S$ = guide particle scale
        $\hat{N}$ = guide particle normal
        $\hat{r}$ = guide particle right
        $\hat{t}$ = guide particle tangent
        $\overrightarrow{X} = \overrightarrow{X_0} + [\ (\overrightarrow{P_1})_x\hat{N} + (\overrightarrow{P_1})_y\hat{r} + (\overrightarrow{P_1})_z\hat{t}\ ]S$
    6.  displace $\overrightarrow{X}$ with FSPN
        $S_2$ = guide particle scale2
        $\overrightarrow{X_{wisp}} = \overrightarrow{X} + V(FSPN_2(\overrightarrow{X})) * S_2$
        $V(FSPN_2(\overrightarrow{X})) = (FSPN_2(\overrightarrow{X}), FSPN_2(\overrightarrow{X}+offset), FSPN_2(\overrightarrow{X}+offset))$
    7.  sample the dot to the grid
        stamp(x,y,z)
    8.  sample color to grid
        color = $(FSPN_3(\overrightarrow{X}), FSPN_3(\overrightarrow{X}+offset), FSPN_3(\overrightarrow{X}+offset), 1.0)$
        stamp color
}
```

For step one in pseudo code, the three types of walks that can be used to generate wisps are random, correlated, and levy. Random walk is the most basic. It generates a randomized position using uniform pseudo random number generator (UniformPRN) with Mersenne Twister [14]. A random walk is generalized as follows:

$$x = 2 * prn - 1$$
$$y = 2 * prn - 1$$
$$z = 2 * prn - 1$$

where prn is a sample value from a UniformPRN generator. (x,y,z) are contained in a 2x2x2 cube with each element in [-1,1]. The left image in Figure 2.1 shows an example of random walk. Each wisp particle is totally random and has no correlation with the particle in previous step.

The correlated walk adds a procedure. A correlation parameter (ϵ) is used to make correlations with the last value, so that the wisp looks more continuous.

$$\begin{aligned}
&v_x = 2 * prn - 1\\
&v_y = 2 * prn - 1\\
&v_z = 2 * prn - 1\\
&x = x(1-\epsilon) + v_x\epsilon\\
&y = y(1-\epsilon) + v_y\epsilon\\
&z = z(1-\epsilon) + v_z\epsilon
\end{aligned}$$

If $\epsilon = 1$, then the wisp particle does not move and the positions are completely correlated. The image of wisp would be just a single dot. If $\epsilon = 0$, then the walk is totally random and uncorrelated. The image of wisp would be the same as the random walk image. For $0 < \epsilon < 1$, the collection of values is a random walk with positions that are partially correlated. The middle image in Figure 2.1 shows an example of correlated walk with a correlation coefficient $\epsilon = 0.7$. Each wisp particle is 30% random and has 70% correlation with the particle in previous step. So the wisp shape is much more coherent.

The levy walk is a correlated walk with an additional scaling to make the correlation uneven.

$$\begin{aligned}
&v_x = 2 * prn - 1\\
&v_y = 2 * prn - 1\\
&v_z = 2 * prn - 1\\
&x = x(1-\epsilon) + v_x\epsilon\\
&y = y(1-\epsilon) + v_y\epsilon\\
&z = z(1-\epsilon) + v_z\epsilon\\
&mag = x^2 + y^2 + z^2\\
&x = x/mag^q\\
&y = y/mag^q\\
&z = z/mag^q
\end{aligned}$$

where user-defined parameter q is a scalar value to control the divisor. The right image in Figure 2.1 shows an example of levy walk. Each wisp particle has different randomness and correlation with the particle in previous step. So the wisp shape is more coherent than random walk but more spread-out than correlated walk.

There are several important parameters inside noise to control the shape of wisps, including frequency, octaves, roughness, fjump, and translate. The total collection of parameters are encap-

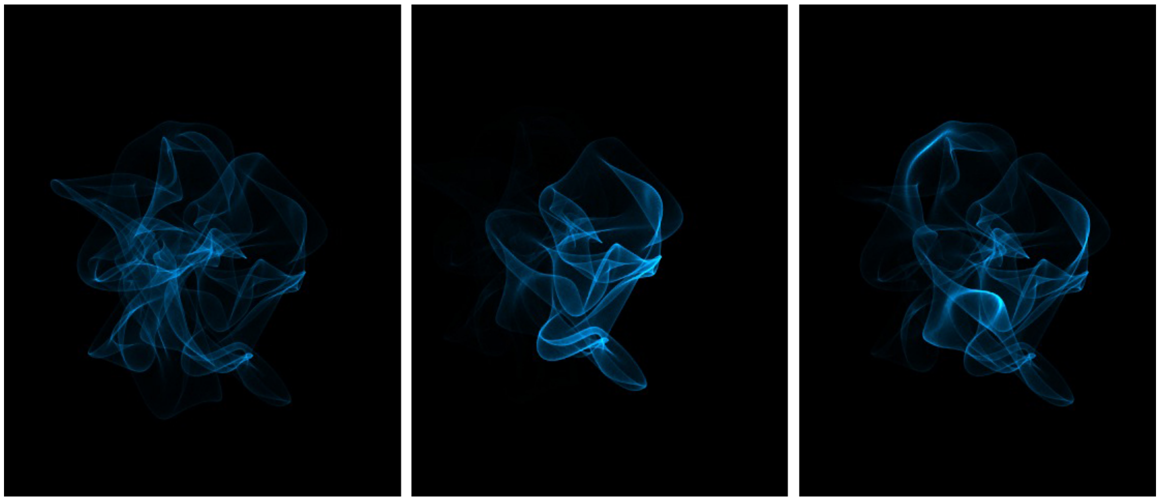


Figure 2.1: Wisp Generated Using Random Walk (Left), Correlated Walk (Middle), and Levy Walk (Right) Algorithms

sulated in the guide particle class. Since there are two copies of FSPN, two sets of noise parameters are defined in the particle class. Five noise parameters are used very often, shown as follows:

```
float frequency
float octaves
float fjump
float roughness
Vector translate
```

The FSPN equation is generalized as:

$$FSPN = \sum_{i=0}^{n-1} r^i PN((\vec{x} - \vec{x}_t))f(f^i_{jump})$$

where n is the number of octaves, f is the frequency, r is the roughness, f_{jump} is fjump, and $\vec{x}_t$ is the translate vector.

Animating the parameters above creates an animated wisp. The `frequency` can change the frequency of the noise. The `octaves` numbers add more bumps to the wisp. The `roughness` and `fjump` can make the noise shape expand or shrink to look more rough or plain. The `translate` vector changes the position of the noise. I animated those 5 parameters in my galaxy animation for the 3D Fractal Flame Wisps.

To add colors to Wisps, another FSPN can be used: $FSPN_3(p_0)$. To make sure colors are

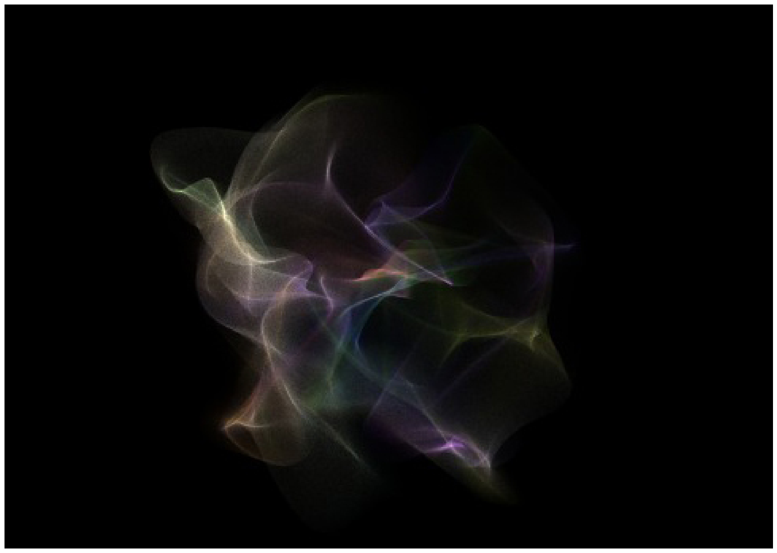


Figure 2.2: Wisp With Random Color

continuous throughout the wisp, p_0 should be the position before displacement between step 2 and 3 in pseudo code. Figure 2.2 shows an example of Wisp with random color.

To explore new looks of Wisps, the step one of random walk, correlated walk, or levy walk has been replaced by the fractal flame algorithm. Some of the test results are shown in Figure 2.3. Those patterns are distinguished from the usual wisp patterns. Later, a more complete algorithm of 3D Fractal Flame Wisps has been generalized. Comparing the previous wisp pseudo code, step one has been replaced by the flame functions and step 3 to 5 are deleted.

To make a single 3D Fractal Flame Wisp, wisp and IFS algorithms can be combined together as follows in pseudo code:

```
for(loop over number of dots for this guide particle)
{
    $P_0$(x,y,z) = a random point in the 2x2x2 cube with axes between [-1,1]
    for(loop over IFS variation numbers)
    {
        1.  fractal flame functions
            $\overrightarrow{X}$(x,y,z) = $f_i$($P_0$(x,y,z))
        2.  record the $\overrightarrow{X}$ for color
            $\overrightarrow{X}_{color}$(x,y,z) = $\overrightarrow{X}$(x,y,z)
```

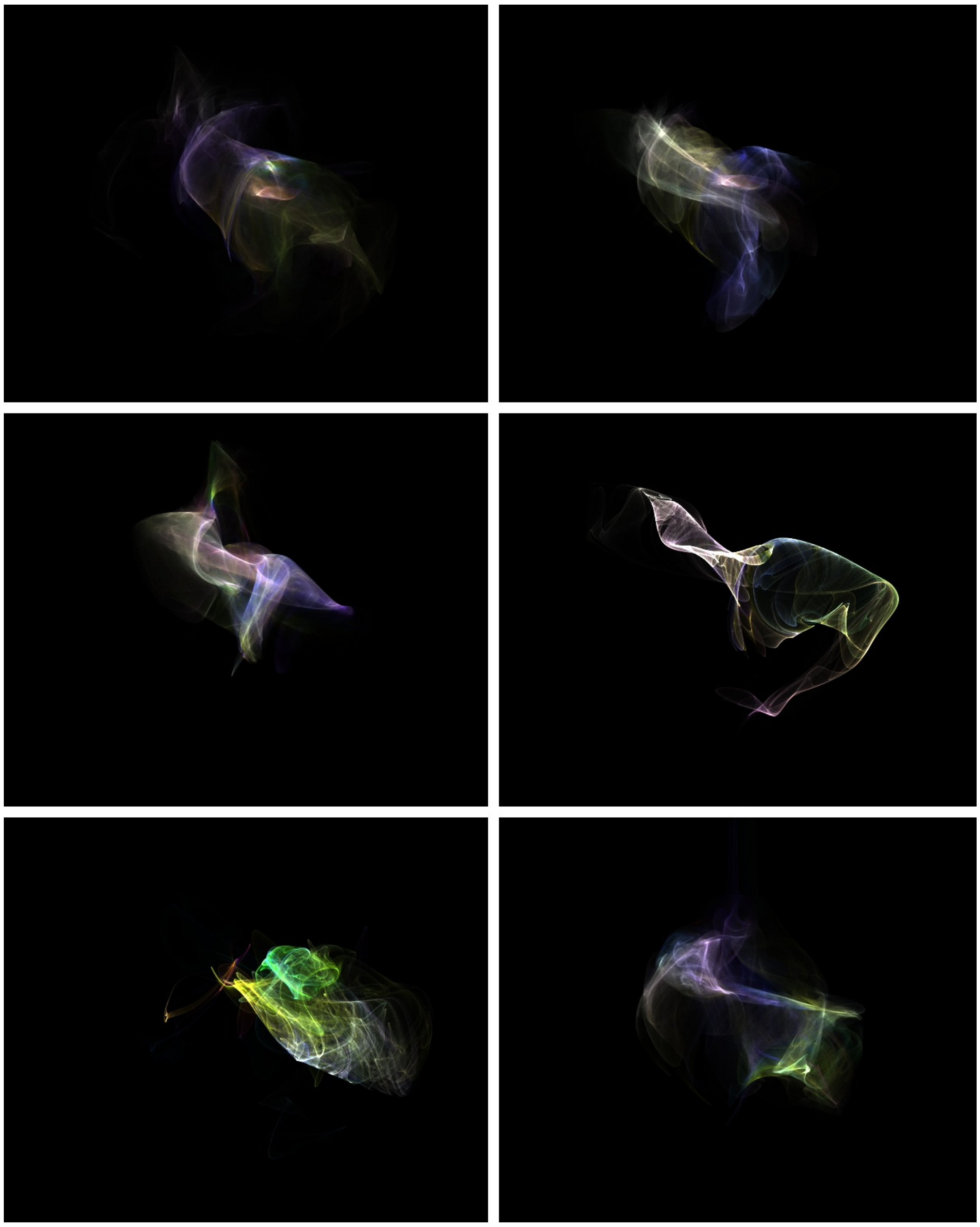


Figure 2.3: 3D Fractal Flame Wisps Pattern Example Images.
These images were selected for aesthetic properties

```
                3.  displace $\overrightarrow{X}$ with FSPN
                    $S =$ user-defined displacement coefficient
                    $P_0 = \overrightarrow{X} + V(FSPN(\overrightarrow{X})) * S$
                    $V(FSPN(\overrightarrow{X})) = (FSPN(\overrightarrow{X}), FSPN(\overrightarrow{X}+(0.1,0,0)), FSPN(\overrightarrow{X}+(0,0.1,0)))$
                4.  sample dot to grid
                    pscale = guide particle scale
                    stamp $P_0$*pscale
                    $P_0$ is the input of the next point in the IFS loop
                5.  sample color to grid
                    color = $(FSPN_2(\overrightarrow{X}), FSPN_2(\overrightarrow{X} + (0.1,0,0)), FSPN_2(\overrightarrow{X} + (0,0.1,0)), 1.0)$
                    stamp color
          }
    }
```

This thesis uses this new algorithm to explore new looks and patterns of 3D Fractal Flame Wisps. Also, a great effort has been put into experimenting with artistic controls of the shapes, colors, and the animations. More details will be discussed in the next chapter.

Chapter 3

Implementation

The research implementation consists of three main components: the Volume Render, Python interface, and 3D Fractal Flame Wisp patterns. The Volume Render, which is the main renderer for this thesis, utilizes the ray marching algorithm to image the volumetric data in voxel buffers. Written in C++, the Volume Render is not sufficiently user-friendly for testing the impact of a large number of parameters. Therefore, a Python wrapper has been added to the volume rendering code to improve usability. The workflow for testing and rendering is embedded in one or more Python scripts. The Fractal Flame class includes all 3D Fractal Flame Wisp base patterns used to experiment within this thesis. Also, IFS and noise displacement has been added to the flame to improve the visual quality of the shapes. Last but not least, color design is critical to the looks of the 3D Fractal Flame Wisps.

The essential algorithm for Volume Rendering is the ray marching algorithm, which is the numerical approximation of the rendering equation [6]:

$$L(\mathbf{x}_C, \mathbf{n}_P) = \int_{s_0}^{s_{max}} ds\ C^T(\mathbf{x}(s))\ \kappa\ \rho(\mathbf{x}(s))\ e^{-\int_0^s ds'\ \kappa\ \rho(\mathbf{x}(s'))}$$

$L(\mathbf{x}_C, \mathbf{n}_P)$ is the light received by a camera at the position $\mathbf{x}_C$ with a pointing direction $\mathbf{n}_P$ for pixel P. Since the volumetric material only takes a finite volume of space, setting up starting point $\mathbf{x}_{near}$ and finishing point $\mathbf{x}_{far}$ can reduce the time of integration. The ray marcher traverses along the line:

$$\mathbf{x}(s) = \mathbf{x}_{near} + s\mathbf{n}_P$$

with a step size Δs and a unit direction vector $\mathbf{n}_P$, $s \in [\,0, |\mathbf{x}_{far} - \mathbf{x}_{near}|\,]$. C^T is the color representing the amount of light emittable at the point $\mathbf{x}(s)$. ρ is density and κ is the extinction coefficient [6].

Wisps are represented by scalar fields and color fields that are discretized consisting of many small voxel samples. Wisp particles' density and color are rasterized into a voxel grid before rendering. Four types of grids have been implemented in this thesis to accelerate the rendering process.

A rectangular grid is a typical method for a voxel buffer. It is a 3D orthogonal array, which stores scalarfield values (such as wisp density) or colorfield values (such as wisp color) in each voxel. The memory is allocated all at once when initializing the grid. The normal range of the resolution for rectangular grids is up to roughly 1000^3.

In implementation, a rectangular grid can store its data as a contiguous one dimentioanl array. For example,

```
float * grid = new float[sizeX * sizeY * sizeZ];
```

And the mapping from a 3D coordinate to 1D array can be done through an index function:

```
int RectangularGrid::index(int i, int j, int k)
{
      return i + sizeX * (j + sizeY * k);
}
```

A sparse grid has two layers of information: data pointers and actual data. The data pointers are allocated dynamically where wisps dots are located. They are not assigned ahead of time. A sparse grid is shown in Figure 3.1 [5]. The outline of the larger cube is the dimention of the 3D grid. The smaller cubes represents the valid voxels with data. All the empty space in the larger cube shows those voxels do not have any memory allocated. This method is more memory efficient comparing to rectangular grid. Therefore, when initialing the sparse grid only the memory for data pointers is allocated. An extra parameter, partition size, is needed.

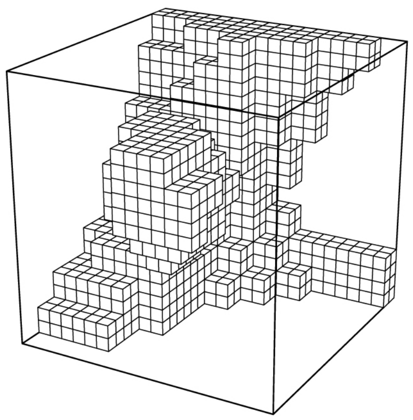


Figure 3.1: Sparse Grid Visualization from SIGGRAPH course notes 2010 [5]

```
int blockSizeX = sizeX / partitionX;
int blockSizeY = sizeY / partitionY;
int blockSizeZ = sizeZ / partitionZ;
float *grid = new float*[blockSizeX * blockSizeY * blockSizeZ];
```

Later, when sampling the particles, if there are values needing to be assigned to the voxels then the memory will be allocated through the data pointers.

```
if( grid[bi] == NULL )
{
      grid[bi] = new float[partitionX * partitionY * partitionZ];
      for(int i=0; i<partitionX*partitionY*partitionZ; i++)
            grid[bi][i] = dvalue;
}
```

To render 3D Fractal Flame Wisps, a sparse grid would be a better choice than a rectangular grid in memory usage. It saves memory because wisps have a large amount of empty space where sparse grid will not need allocated memory.

The double sparse grid adds another partition layer to the sparse grid in order to accommodate larger dimensions of grids. The grid dimension is about 3000^3 for sparse grid while double sparse grid can go up to 5000^3 for my animation project. It helped me to get finer details of the 3D

Fractal Flame Wisps.

```
int blockSizeX = sizeX / (partitionX * partitionX);
int blockSizeY = sizeY / (partitionY * partitionY);
int blockSizeZ = sizeZ / (partitionZ * partitionZ);
float *grid = new float**[blockSizeX * blockSizeY * blockSizeZ];
```

When sampling the particles later, if values need to be assigned to voxels, the memory will be allocated through the two layers of data pointers.

```
if( grid[bi] == NULL )
{
      grid[bi] = new float* [partitionX * partitionY * partitionZ];
      for(int i=0; i<partitionX*partitionY*partitionZ; i++)
            grid[bi][i] = NULL;
}
int dbi = dindex(i,j,k);
if( grid[bi][dbi] == NULL )
{
      grid[bi][dbi] = new float[dparsizeX * dparsizeY * dparsizeZ];
      for (int i=0; i<dparsizeX*dparsizeY*dparsizeZ; i++)
            grid[bi][dbi][i] = dvalue;
}
```

To visualize the volumetric data, camera information decides the viewing frustum. Since information outside the viewing frustum is not visible, it does not need to be sampled onto the grid. Furthermore, voxels closer to the camera requires more details than the voxels further away. To better accommodate situation like this, a frustum-shaped grid is a more suitable choice.

The frustum grid uses information from a camera to shape the grid in the space projection as the camera. Each voxel has a different size.

The transformation of a point in space $\overrightarrow{P}$ to a location in the frustum volume projects the point to the camera plane location $\overrightarrow{X}$:

$$\overrightarrow{X} = \frac{\overrightarrow{P} - \overrightarrow{P_c}}{\hat{n}_c * (\overrightarrow{P} - \overrightarrow{P_c})} - \hat{n}_c$$

where $\overrightarrow{P_c}$ is the camera position, and $\hat{n}_c$ is the unit vector of camera viewing direction.

To further opitimize the volume render, the frustum grid can be combined with the sparse grid, as shown in Figure 3.2 [5]. The sparse grid changes from cube shape to frustum shape. Also, the frustum grid uses the camera information to cut out the data outside viewing frustum. It is very

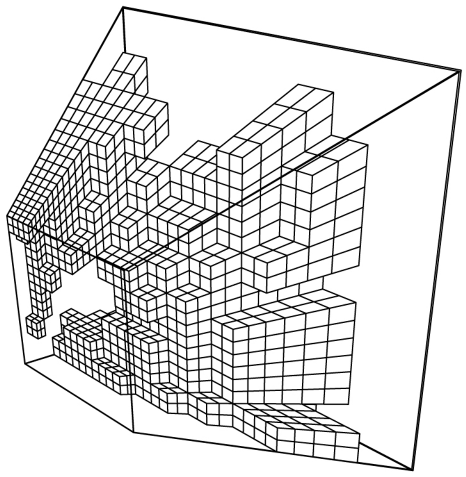


Figure 3.2: Frustum Sparse Grid Visualization from SIGGRAPH course notes 2010 [5]

memory efficient for the close up shot when the viewing data is only a small part of the scene data. The frame in my animation project used frustum grid in a resolution of 2000x1200x2000 has finer details than the one used double sparse grid in a resolution of 5000x5000x5000.

Aliasing issues may appear in Volume Rendering. If voxel size is not small enough, it may cause aliasing artifacts. Furthermore, the sampling frequency, that is ray marching step, needs to match the voxel size. Similar to ray tracing, a technique used for antialiasing in Volume Rendering is to shoot multiple rays for each pixel. Each ray would add a small offset to each voxel's sample position when ray marching.

In the design phase of this project, there were many variations of the parameters for 3D Fractal Flame Wisps. If the render programs were C++ main files, the code would be compiled each time before running, complicating the process. Python scripting of the design process made the parameter tweaking simpler and more effective.

A Python interface for the volume rendering code was generated through Simplified Wrapper and Interface Generator (SWIG). SWIG is an open-source software used to connect libraries or programs in C or C++ with scripting languages such as Python [15]. SWIG generates a shared library that later Python scripts can import. The Python scripts invoke the underlining C++

library.

To generate the Python wrapper for the volume code, the first step is to create corresponding SWIG interface (.i) files. For example, Camera.h file defines the camera class in the volume code. A corresponding camera.i file maps the camera class to the SWIG wrapper. In this file, the `%module` defines the name of the module that will be created by SWIG.

```
%module vr
%{
#include Camera.h
%}
%include Camera.h
```

The volume render (vr) module has multiple interface files that are collected together in a comprehensive interface.

```
%module vr
%include "std_string.i"
%include "boost_shared_ptr.i"
%include "Vector.i"
%include "Matrix.i"
%include "Camera.i"
%include "Image.i"
%include "Color.i"
```

From the interface file and C++ library for the volume code, SWIG generates two files: a shared object file and a Python module. Together they make the C++ functions and objects available in Python. For example, a block of C++ code in volume render test:

```
#include "DoubleSparseGrid.h"
#include "Vector.h"
DoubleSparseGrid dg;
dg.init(Vector(-80.0,-80.0,-80.0), Vector(80.0,80.0,80.0), 800,800,800);
DoubleSparseColorGrid cg;
dg.init(Vector(-80.0,-80.0,-80.0), Vector(80.0,80.0,80.0), 800,800,800);
```

can turn into a block of Python script:

```
import vr
dg = vr.DoubleSparseGrid()
dg.init(vr.Vector(-80.0,-80.0,-80.0), vr.Vector(80.0,80.0,80.0), 800,800,800)
cg = vr.DoubleSparseColorGrid()
cg.init(vr.Vector(-80.0,-80.0,-80.0), vr.Vector(80.0,80.0,80.0), 800,800,800)
```

To create 3D Fractal Flame Wisps, flame classes and a wisp class were built. A base flame class was created first and more variations of flames inherited this base flame class. The base flame class is shown as the following:

```
class Flame
{
public:
      Flame(){ iteration = 1;}
      virtual ~Flame(){}

      void setIteration(int i){ iteration = i;}
      virtual const Vector eval(const Vector& P){ return Vector(0,0,0); }

protected:
      int iterations;
};
```

Appendix A is a catalog of the 21 basic variation formulas I constructed and their render examples of 3D Fracal Flame Variations. Each variation was turned into a flame class in C++. The formulas were inspired by Scott Draves' 2D Fractal Flame and evolved from 2D to 3D. The 21 formulas are 3D forms corresponded to the appendix in Scott Draves and Erik Reckase's paper "The Fractal Flame Algorithm" [1]:

$$V_0(x, y, z) = (x, y, z)$$

$$V_1(x, y, z) = (\sin(x), \sin(y), \sin(z))$$

$$V_2(x, y, z) = \frac{1}{r^2}(x, y, z)$$

$$V_3(x, y, z) = (x\sin(r^2) - y\cos(r^2), x\cos(r^2) + y\sin(r^2), x\cos(r^2) - z\sin(r^2))$$

$$V_4(x, y, z) = \frac{1}{r}((x-y)(x+y), 2xy, (x-z)(x+z))$$

$$V_5(x, y, z) = (\frac{\theta}{\pi}, r-1, \frac{\theta_2}{\pi})$$

$$V_6(x, y, z) = r(\sin(\theta + r), \cos(\theta - r), \sin(\theta_2 + r))$$

$$V_7(x, y, z) = r(\sin(\theta r_1), -\cos(\theta r_1), -\cos(\theta_2 r_2))$$

$$V_8(x, y, z) = (\frac{\theta}{\pi}\sin(\pi r), \frac{\theta}{\pi}\cos(\pi r), \frac{\theta_2}{\pi}\cos(\pi r))$$

$$V_9(x, y, z) = \frac{1}{r}(\cos\theta + \sin r, \sin\theta - \cos r, \cos\theta_2 + \sin r)$$

$$V_{10}(x, y, z) = (\frac{\sin\theta}{r}, r\cos\theta, \frac{\sin\theta_2}{r})$$

$$V_{11}(x, y, z) = (\sin\theta\cos r, \cos\theta\sin r, \sin\theta_2\cos r)$$

$$V_{12}(x,y,z) = r(p_0^3 + p_1^3, p_0^3 - p_1^3, p_2^3 - p_3^3)$$

$$V_{13}(x,y,z) = \sqrt{r}(\cos(\theta/2), \sin(\theta/2), \sin(\theta_2/2))$$

$$V_{14}(x,y,z) = \begin{cases} (x,y,z) & x \geq 0, y \geq 0 \\ (2x, y, 2z) & x < 0, y \geq 0 \\ (x, y/2, z/2) & x \geq 0, y < 0 \\ (2x, y/2, z) & x < 0, y < 0 \end{cases}$$

$$V_{15}(x,y,z) = (x + b\sin(\tfrac{y}{c^2}), y + e\sin(\tfrac{x}{f^2}), z + h\sin(\tfrac{z}{i^2}))$$

$$V_{16}(x,y,z) = \tfrac{2}{r+1}(y,x,z)$$

$$V_{17}(x,y,z) = (x + c\sin(\tan 3y), y + f\sin(\tan 3z), z + g\sin(\tan 3x))$$

$$V_{18}(x,y,z) = (exp(x-1)\cos(\pi y), exp(x-1)\sin(\pi y), exp(y-1)\sin(\pi z)$$

$$V_{19}(x,y,z) = r^{\sin\theta}(\cos\theta, \sin\theta, \cos\theta_2)$$

$$V_{20}(x,y,z) = (\cos(\pi x)\cosh(y), -\sin(\pi x)\sinh(y), \sin(x))$$

where

$$r = \sqrt{x^2 + y^2 + z^2}$$

$$r_1 = \sqrt{x^2 + y^2}$$

$$r_2 = \sqrt{x^2 + z^2}$$

$$\theta = \tan(y/x)$$

$$\theta_2 = \tan(z/x)$$

$$p_0 = \sin(\theta + r)$$

$$p_1 = \cos(\theta - r)$$

$$p_2 = \sin(\theta_2 + r)$$

$$p_3 = \cos(\theta_2 - r)$$

(a,b,c,d,e,f,g,h,i) are the affine transform coefficients.

All those vairations can be generated as a non-linear function:

$$F_i(x,y,z) = V_j(a_i x + b_i y + c_i z + d_i,\ e_i x + f_i y + g_i z + h_i,\ k_i x + l_i y + m_i z + n_i)$$

Those variations are non-linear functions, which are different remapping functions of a 2x2x2 cube volume. The recognizable shapes are defined by the affine transform. Additional parameters can be added to the variations as the post transformation P_i.

$$P_i(x,y) = (\alpha_i x + \beta_i y + \gamma_i z + \delta_i,\ \epsilon_i x + \zeta_i y + \eta_i z + \theta_i,\ \kappa_i x + \lambda_i y + \mu_i z + \sigma_i)$$

to generate a parametric F_i function:

$$F_i(x,y,z) = P_i(V_j(a_i x + b_i y + c_i z + d_i,\ e_i x + f_i y + g_i z + h_i,\ k_i x + l_i y + m_i z + n_i))$$

In this way, 3D fracal flame can have countless variations after post transformation. To accommodate the post tranforms, a parametric flame class has been created as the following:

```
class Parametric3dFlame :  public Flame
{
public:
      Parametric3dFlame(Flame* flame, float a0,float b0, float c0,
            float d0, float e0, float f0, float g0, float h0, float i0,
            float j0, float k0, float l0):  elem(flame),a(a0),b(b0),c(c0),
            d(d0),e(e0),f(f0),g(g0),h(h0),i(i0),j(j0),k(k0),l(l0){}
      ~Parametric3dFlame(){}

      const Vector eval(const Vector& P)
      {
            Vector x = elem->eval(P);
            Vector newx;
            newx[0] = a*x[0]+b*x[1]+c*x[2]+d;
            newx[1] = e*x[0]+f*x[1]+g*x[2]+h;
            newx[2] = i*x[0]+j*x[1]+k*x[2]+l;
            return newx;
      }
};
```

In this way, flames can have multiple passes of post transformation which would greatly change the look of flames. In addition, translate, scale, and rotation flame classes have been created in order to have exact control of transformation.

The wisp class needs to have flames as its member variable. A fractalwalk function uses 3D fractal flame algorithms to generate the next point. For example in C++,

```
class Wisp
{
public:
      Wisp(){}
      ~ Wisp(){}
      void init(const Particle& p);
      const Vector fractalwalk(const Vector& p, int i);
      Color getColor(int i) { return colors[i]; }

      std::vector<Flame*> flames;
      std::vector<Color> colors;
private:
      Particle guide;
      Vector walkpos;
      Vector colorpos;
};
```

In the Wisp class, there is a flame pointer array to save multiple flame variations. And each flame has a corresponding color which is saved in the colors array. The fractalwak() function implement the 3D Fractal Flame Wisps algorithm in pseudo code as follows:

```
fractalwalk($P_0$(x,y,z))
{
      1.  fractal flame functions
          $\overrightarrow{X}$(x,y,z) = $f_i$($P_0$(x,y,z))
      2.  record the $\overrightarrow{X}$ for color
          $\overrightarrow{X}_{color}$(x,y,z) = $\overrightarrow{X}$(x,y,z)
      3.  displace $\overrightarrow{X}$ with FSPN
          $k =$ user-defined displacement coefficient
          $P_0 = \overrightarrow{X} + V(FSPN(\overrightarrow{X})) * k$
          $V(FSPN(\overrightarrow{X})) = (FSPN(\overrightarrow{X}), FSPN(\overrightarrow{X}+(0.1,0,0)), FSPN(\overrightarrow{X}+(0,0.1,0)))$
      4.  sample dot to grid
      5.  sample color to grid
          $c_i$ = flame color using getColor() in Wisp class
          color = $(FSPN(\overrightarrow{X}), FSPN(\overrightarrow{X}+(0.1,0,0)), FSPN(\overrightarrow{X}+(0,0.1,0)), 1.0)$
          color = (color + $c_i$)/2.0
          stamp color
}
```

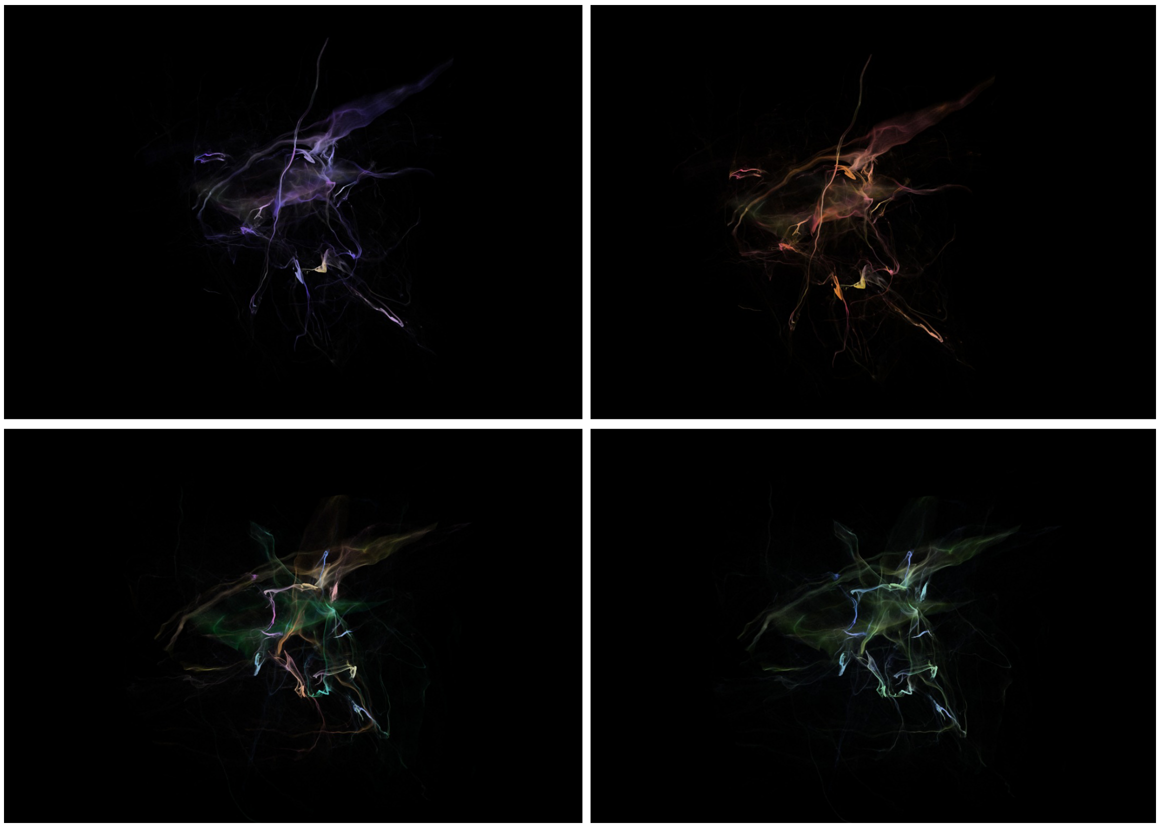


Figure 3.3: Wisp Color Design Test

There are two color contributions to the wisps' color. The first contribution is from the positions of wisp points. In pseudo code step two, the position is recorded for color. Later in step five, *color* is evaluated from the FSPN of the color position. This is a lower level of color control, displacement of the position would change the color. However, the changes from position to the color is not intuitive and the FSPN evaluation is not linear and hard to predict.

The second contribution is from the color assignment c_i of the fractal flames stored in the colors array. Therefore, the final color is: $(color + c_i)/2$. This is a higher level of control to the overall look. As shown in Figure 3.3, I tried to assigned four different color schemes to the same wisp group only change the second color contribution. It worked very well and is very artistic controllable.

Noise can greatly change the look of 3D Fractal Flame Wisps. In pseudo code step three, the FSPN is used as the displacement V of wisp points. Also, a noise coefficient k is added to the equation to have more control of the artistic look.

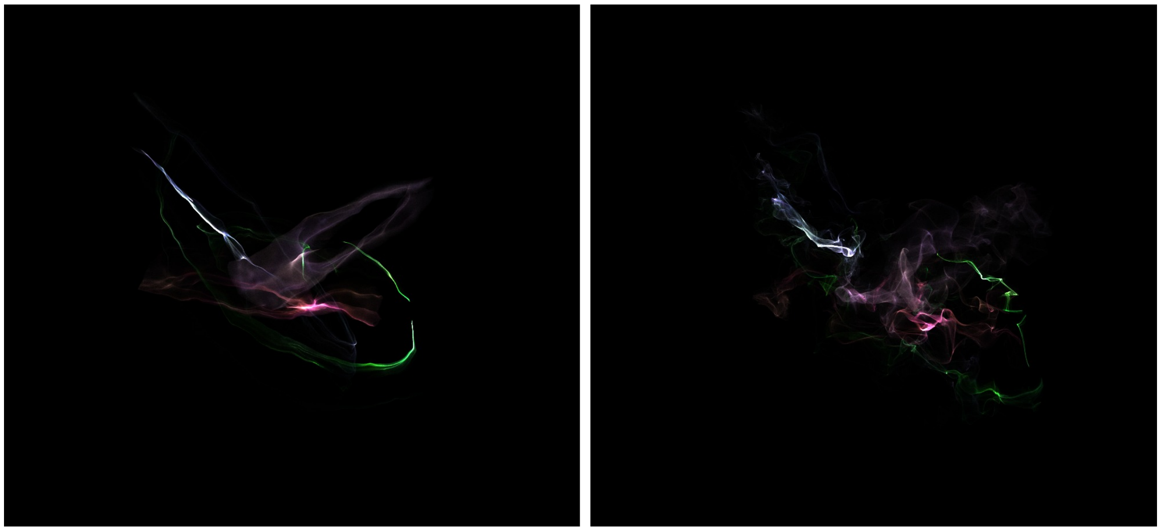


Figure 3.4: Example Image of Noise Parameter Wedge

$$pos = pos + k * V(FSPN)$$

As shown in Figure 3.4, the left image has a coefficient $k = 0.1$ and the right image has a coefficient $k = 0.6$. The displacement level drastically changes the look of the 3D Fractal Flame Wisps. This parameter gives artists a higher level control over the look depending on the animation context.

In addition, to have a more saturated look of 3D Fractal Flame Wisps, color scaling has been used in image processing. Because the final color is determined by the numerical accumulation from ray marching algorithm, the RGB values can easily go up much larger than one when wisp particles clustered together. This can wash out the highlights. When the particle numbers go up, the whole image tends to be overexposed. Therefore, in order to get a more saturated color range and prevent overexposure, an illuminance correction has been implemented as follows:

```
lmax = max(L.r, L.g, L.b)
L.r = L.r/(lmax+1)
L.g = L.g/(lmax+1)
L.b = L.b/(lmax+1)
```

The two images in Figure 3.5 show the comparison between the raw illuminance and corrected illuminance. The right image has been corrected compared to the left image. It is more

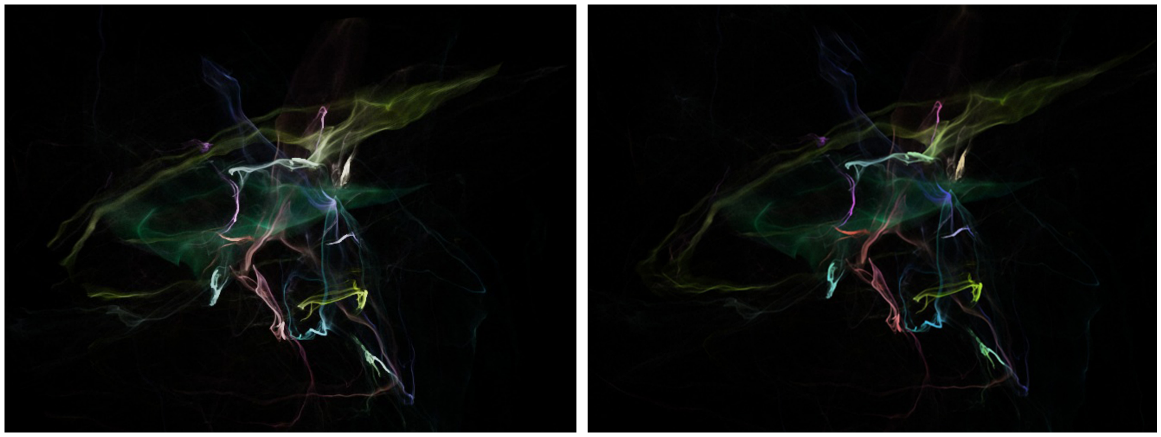


Figure 3.5: Wisp Color Illuminance Correction.
Before Correction (Left) and After Correction (Right).

saturated and closer to the artistic look I planned.

Figure 3.6 shows an example of 3D Fractal Flame Wisp images with all the elements discussed above in four different camera angles. These are the four images selected from a 360 turn table sequence. Different viewing angles demonstrates the volumetric shape of the 3D Fractal Flame Wisp. This wisp has 10 millions particles and 12 parametric flame variations. At this stage, I got lots of interesting patterns generated by the 3D Fractal Flame Wisps. Then I decided to apply those patterns to a galaxy animation project to add more artistic meanings to their representation.

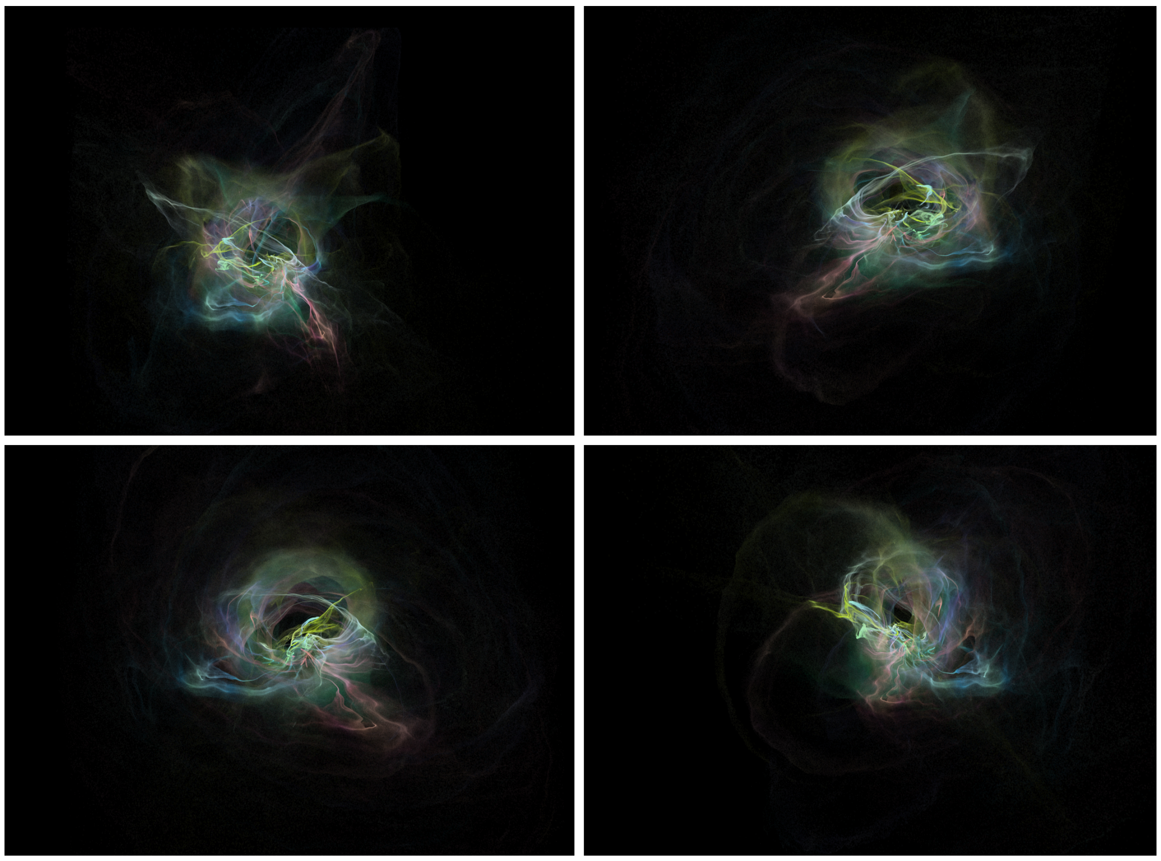


Figure 3.6: Example Image of 3D Fractal Flame Wisp With Different Viewing Angles

Chapter 4

Animation

3D Fractal Flame Wisps were integrated in the production pipeline to produce a two-minute sequence of galaxy animation. There were several tools that had been built to facilitate the FX production pipeline and artistic iterations. The process includes pre-visualization and rendering pipeline.

SideFX' Houdini is a valuable tool for the pre-visulization of the animation. It provides several programming interfaces, such as Python, C++ API and Hscript. In this project, Python was heavily used to set up volume rendering scripts and manage files.

Before the animation, simulation data of Milky Way galaxy and Sagittarius galaxy was provided by Dr. Lih-sin The in the Physics and Astronomy Department at Clemson University and Dr. Jeanette Myers in the Physics and Astronomy Department at Francis Marian University. The simulation is 275 frames in total. The information of particles is listed in Table 4.1. The Milky Way data has three elements: bulge, disk, and halo. The Sagittarius data has two elements: disk and halo. Each particle has a ID, position vector and velocity vector.

File Name	***Particle number***
milkywayBulge_particles.txt	320,000
milkywayDisk_particles.txt	320,000
milywayHalo_particles.txt	960,000
sagittariusDisk_particles.txt	100,000
sagittariusHalo_particles.txt	100,000

Table 4.1: Simulation Data Statistics

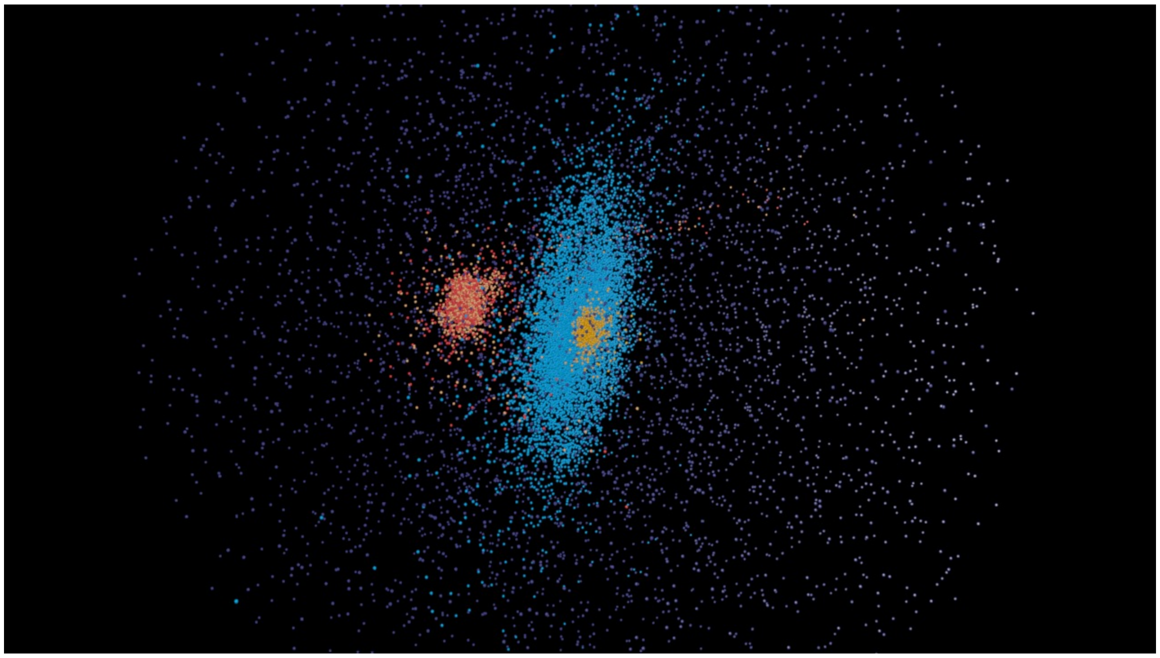


Figure 4.1: Galaxy Simulation Visulazation

The data was converted from text files to obj files and imported into Houdini. Python scripts were created to assist the process. In Figure 4.1, the galaxy simulation visualization illustrates galaxy particles' positions. The yellow color is the bulge of the Milky Way. The blue color is the main disk of the Milky Way. They purple color is the halo of the Milky Way. And the mixing red and orange are the disk and halo of Sagittarius galaxy. Only 5% of the data in this figure was used to be the starting point of the animation.

The simulation data has been linearly interpolated from 275 frames to 2740 frames to slow down the speed of movement. Also the Sagittarius galaxy has been copied to three groups and transformed to different positions to increase the visual complexity of the scene. As shown in Figure 4.2, a center disk galaxy and three smaller galaxy groups were animated in Houdini. The positions of galaxy groups were recorded and set to the starting position of the 3D Fractal Flame Wisps. The pre-visualization of animation has the advantage of a fast experiment with different staging, compositions, and camera movements. Once the camera movement was set, it was exported as caminfo.txt files frame-by-frame with three vector information: eye, view, and up. In addition, the positions of selected particles in each galaxy were exported in text files. As shown in Table 4.2, those particles were used as guide particles for 3D Fractal Flame Wisps.

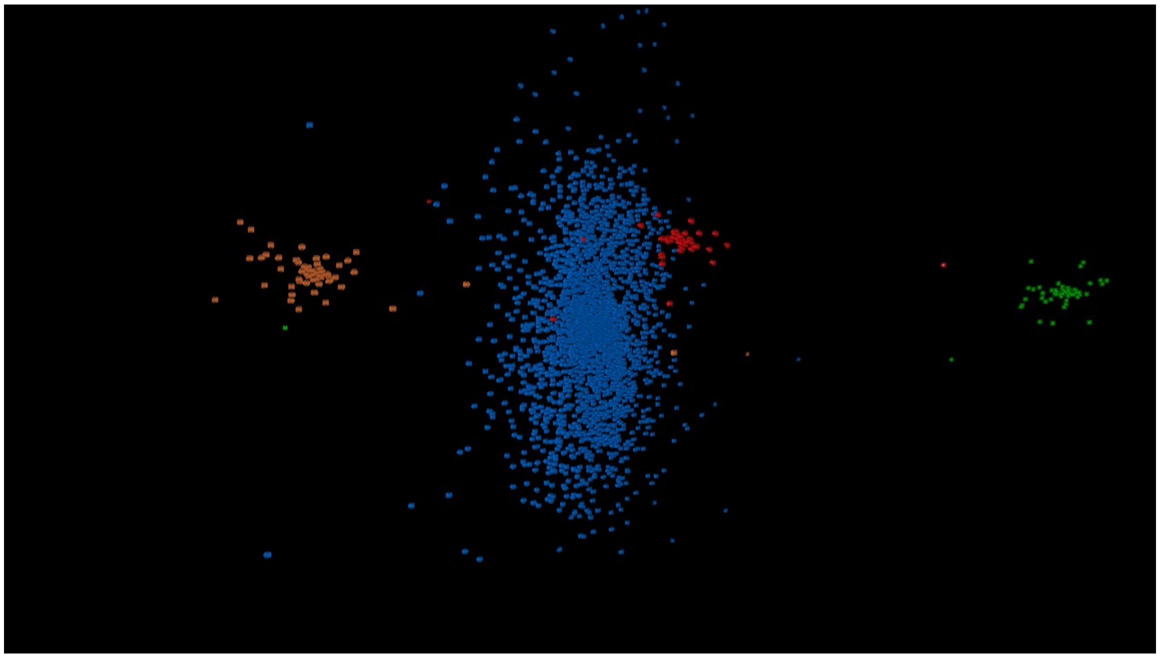


Figure 4.2: Houdini Galaxy Animation Pre-Visulazation

Object	***Guide Particle number***
Center Disk	1
Golden Ball	67
Purple Wisp	50
Magenta Wisp	61

Table 4.2: Guide Particle Statistics

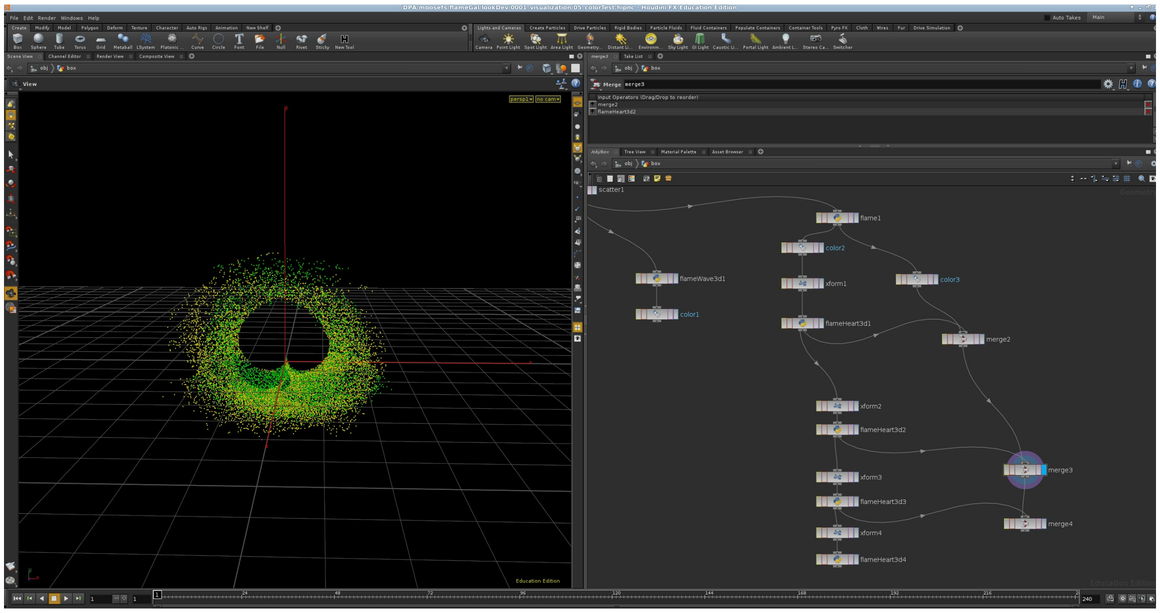


Figure 4.3: Houdini 3D Fractal Flame Wisps Pre-Visulazation

Another pipeline tool was developed to pre-visualize the 3D Fractal Flame classes. Houdini provides a programmable operator interface, inclulding SOPs (surface operators), POPs (particle operators), CHOPs (channel operators), COPs (composite operators), DOPs (synamic operators), SHOPs (shading operators), ROPs (render operators), and VOPs (VEX operators). For this project, Python SOPs were built to integrate Python scripts and visualize the 3D Fractal Flame Wisp dots. As shown in Figure 4.3, the custom SOPs invoked the volume code to evaluate 3D fractal flame wisp dots and display them as Houdini particles. In the figure, the flame SOP procedurally models the heart shaped fractal flame wisp in Houdini. This is an easy and efficient way to test the shapes of wisps.

After pre-visualization, camera and guide particle information was exported in text files for each frame. This allowed the automatic creation of render scripts for 2740 frames. This automation for the rendering was critical since the file management could be time consuming.

The rendering pipeline depended heavily on the Python scripts, which included the following steps:

1) Output particle and camera positions from Houdini

2) Generate rendering scripts

3) Submit jobs to the queue

For render scripts, templates need to be created first, which store the data that is shared by each frame. Then, other scripts parse the templates, the camera information, and guide particle positions to generate render scripts according to the frame number.

The Palmetto Cluster was used to generate this animation. It is a high-performance super computer cluster with a PBS queueing system [16]. Each frame was a job and there were 2740 jobs submited to the cluster at the same time. Usually, there can be up to 100 jobs running at the same time depending on how much resource is available. The automation of job management makes the pipeline more efficient.

The efficiency of the production pipeline is critical. The rendering time and memory usage were flags to the efficiency of the pipeline. Adding more particles in the wisps will consume more memory to construct the volumetric grid. To take down the memory usage and seek higher visual quality of resolution, rectangular grids were first upgraded to sparse grids, then to double sparse grids, and at last to frustum double sparse grids.

The result of this animation project is a two-minute long movie with a camera dolly into the center disk galaxy. Figure 4.4 and 4.5 are frames from the galaxy animation. Figure 4.4 is a far shot of four galaxies. Each galaxy has a distinct color range to differentiate from others and demonstrates various looks of 3D Fractal Flame Wisps. The look of those galaxies has taken the references from Hubble Telescope photographs in Figure 1.3. The three smaller galaxies have three layers of animation, including orbiting movement, self rotation, self transformation, and noise animation.

Figure 4.5 is a close up shot of the golden ball galaxy and the center disk galaxy. More wispy details are revealed from this angle. Furthermore, the artistic feel of the the center-disk galaxy is referencing the Andromeda Galaxy Nebula in Figure 1.3. The center-disk galaxy is purely animated by the noise parameters. Table 4.3 is the statistics of the particle counts in the rendering pipeline.

Object	***Particle number***
Center Disk	72,000,000
Golden Ball	3,000,000
Purple Wisp	1,920,000
Magenta Wisp	2,160,000
Total	79,080,000

Table 4.3: Animation Data Statistics

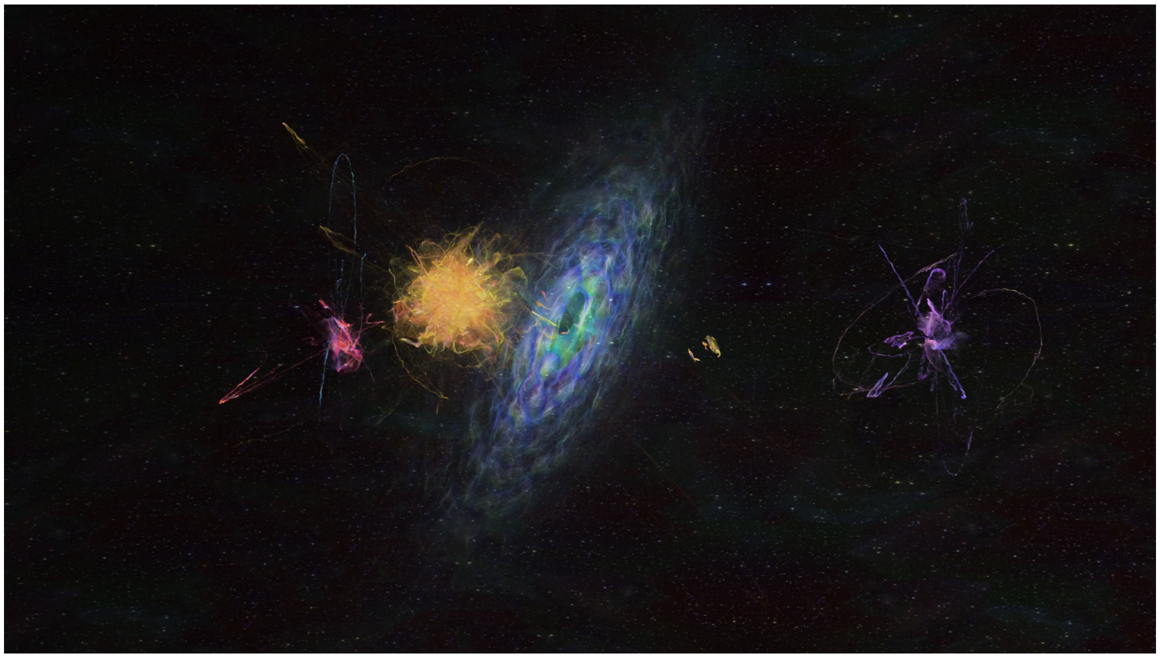


Figure 4.4: Galaxy Animation Screenshot Using 3D Fractal Flame Wisps

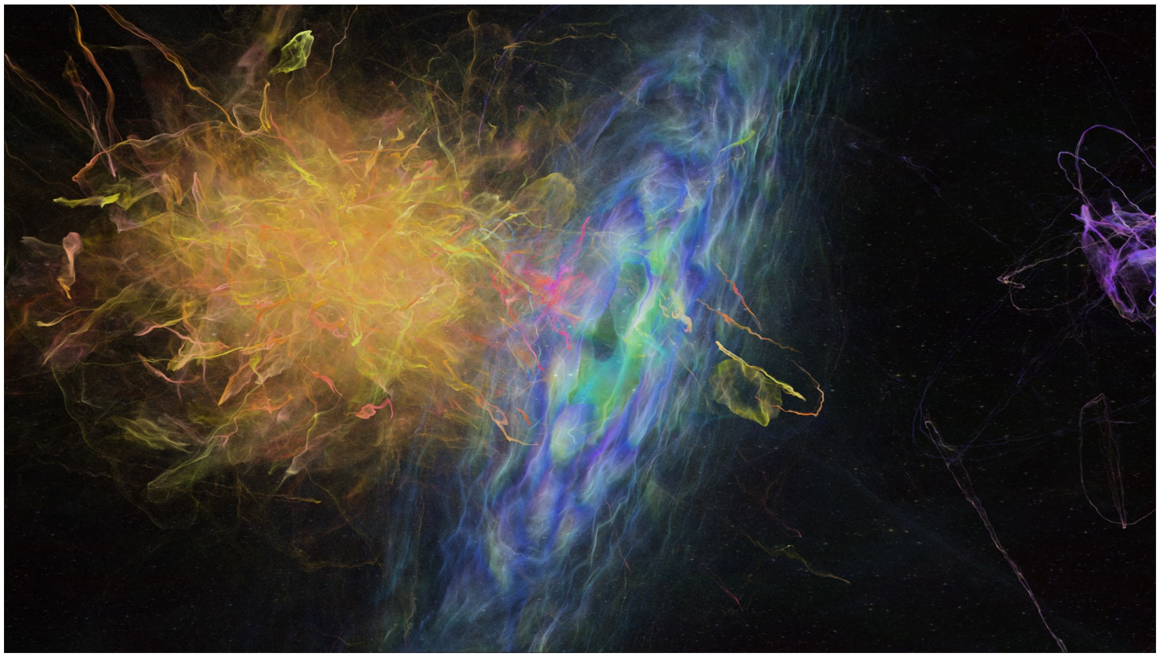


Figure 4.5: Galaxy Animation Screenshot Using 3D Fractal Flame Wisps

Chapter 5

Conclusions and Discussion

In conclusion, I have successfully constructed an algorithm to integrate Fractal Flames and Volumetric Wisps. Twenty-one basic patterns are shown in appendix A, as well as other patterns created through IFS, affine transformation and noise displacement. In addition, a custom visual effects production pipeline has been implemented to put the 3D Fractal Flame Wisps into a two-minute galaxy animation in HD quality. The animation starts from a far shot and steps into the galaxy to show much finer details in the mysterious and artistically stylized space.

There are three artistic aspects in which this animation could be improved. First, the camera angles and movements can be explored more and improved using cinematic techniques. The current two-minute animation is a single shot with a camera rotation in first 1000 frames followed by a dolly move. It is a nice way to illustrate the interesting features in those galaxies. However, more cinematic shots and camera movements might enhance the spectacular scenes.

Recently, I had two email interviews with two pioneers in Fractal Flame area, Josh Aller and J-Walt Adamczyk, who both worked at Imagesavant with Richard "Doc" Baily in the past. They mentioned that colors in their flames is handled differently than mine. In the future, I would like to try out other methods to explore more color schemes.

An important aspect of my 3D Fractal Flame Wisps artistic research is the great amount of experimentation required to find the look "gem" in 3D Fractal Flame Wisps. Because of the geometry originated with a physical simulation of galactic collisions, there was relatively little control of the geometric structure. Once a parameter is changed, the whole wisp is changed rather than a part of it.

I found two technical issues worth pointing out. First, the Python version is very sensitive to the pipeline since it is the glue of different tools and platforms. Houdini 12.1 uses Python 2.6 while my volume rendering code is using Python 2.7. This has caused me to go back and recompile the volume render library with a link to the Houdini Python 2.6 library.

Second, one must be very cautious about using pointers. C/C++ pointers are supported by SWIG when creating the Python interface for volume code. Python, however, has the automatic garbage collection for its memory management. It may deallocate the volumetric data before stamping it onto the grid. A safe way to avoid this segmentation fault is to use reference-counted pointers. This project used the Boost library, a shared_ptr class template, to store a pointer to an object.

An infinite number of patterns can be explored for 3D Fractal Flame Wisps. Only a small collection has been rendered for this thesis. Also, colors can greatly change the look of the wisps. Different color schemes will create more variations as well.

The current render time is about one hour per HD frame with two 2000x1200x2000 frustum double sparse grid. The memory requirement for each frame is 20gb on Palmetto. Optimizing the render time or memory usage would be worthwhile. There are two new directions that can be explored to further reduce the render time or memory usage.

One possible direction could be using OpenCL to massively parallelize the ray march rendering [17]. OpenCL provides parallel computing standard on graphics processing units. Each voxel can be paralleled as a computing kernel.

Furthermore, OpenVDB [18], recently released by DreamWorks Animation for use in volumetric applications, is an open-source library with a highly optimized hierarchical data structure and tools for the efficient storage and manipulation of 3D grids. It is also fully integrated into Houdini 12.5. It may help to reduce the memory usage.

Appendices

Appendix A Catalog of 3D Fractal Flame Wisp Variations

The 21 formulas are listed below with two example images each variation. On the left, the image is the single vairation of each formular; on the right, the image is the Wisp IFS with noise displacement.

All those vairations can be generated as a non-linear function V_j:

$$F_i(x, y, z) = V_j(a_i x + b_i y + c_i z + d_i,\ e_i x + f_i y + g_i z + h_i,\ k_i x + l_i y + m_i z + n_i)$$

where

$r = \sqrt{x^2 + y^2 + z^2}$

$r_1 = \sqrt{x^2 + y^2}$

$r_2 = \sqrt{x^2 + z^2}$

$\theta = \tan(y/x)$

$\theta_2 = \tan(z/x)$

$p_0 = \sin(\theta + r)$

$p_1 = \cos(\theta - r)$

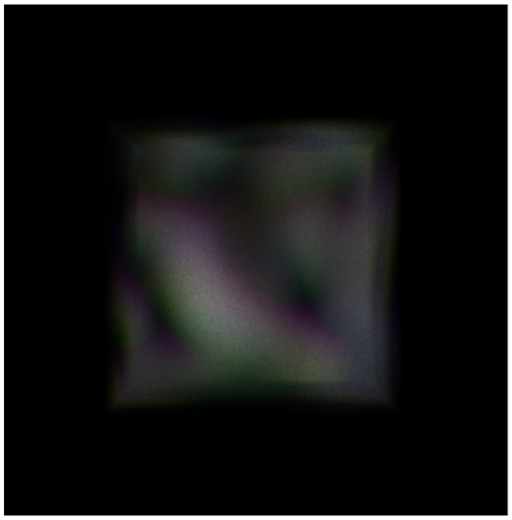
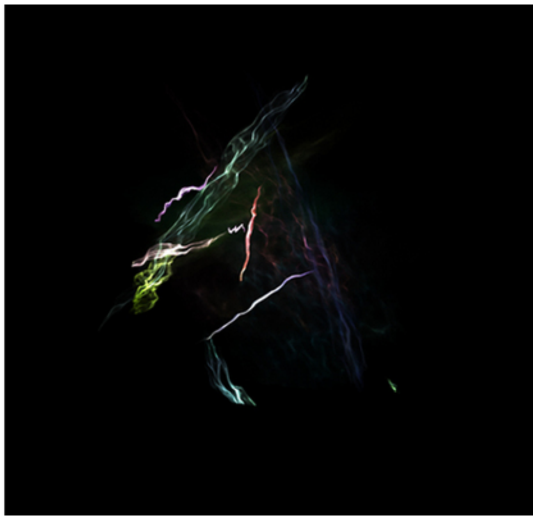
$p_2 = \sin(\theta_2 + r)$

$p_3 = \cos(\theta_2 - r)$

(a,b,c,d,e,f,g,h,k) are the affine transform coefficients.

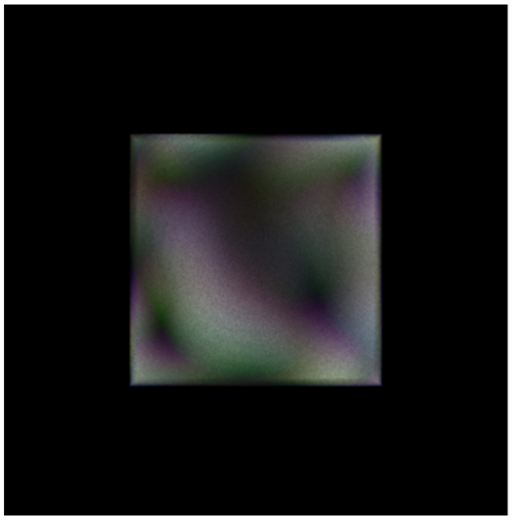
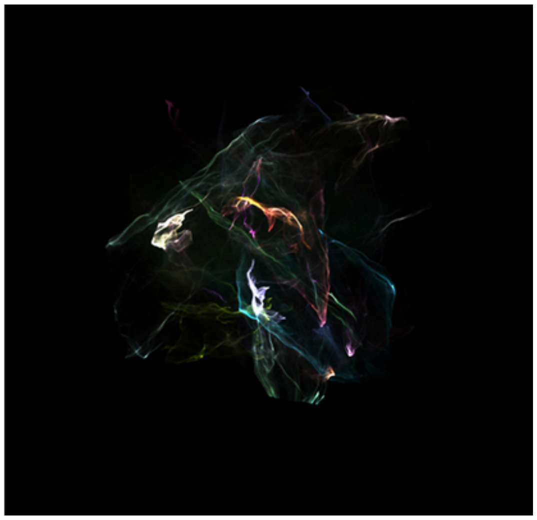
Variation 0 - Linear

$V_0(x, y, z) = (x, y, z)$

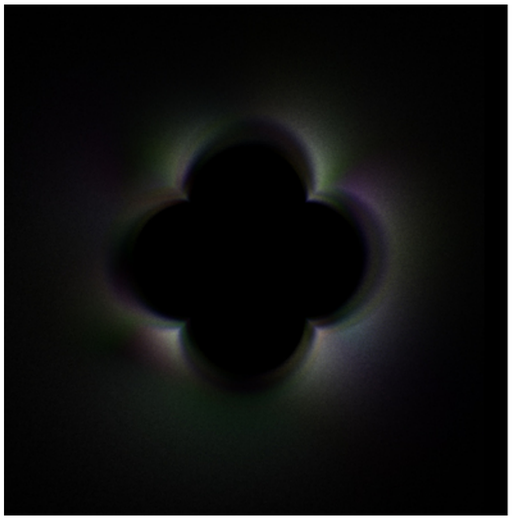
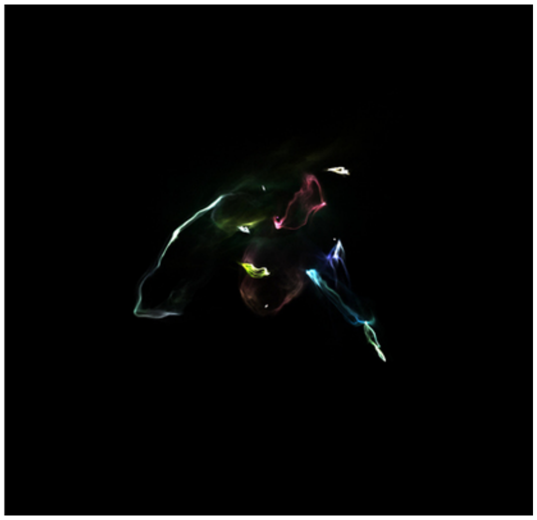
Variation 1 - Sinusoidal

$V_1(x, y, z) = (\sin(x), \sin(y), \sin(z))$

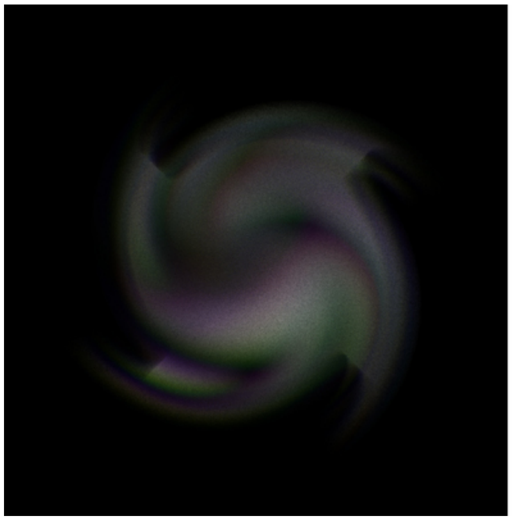
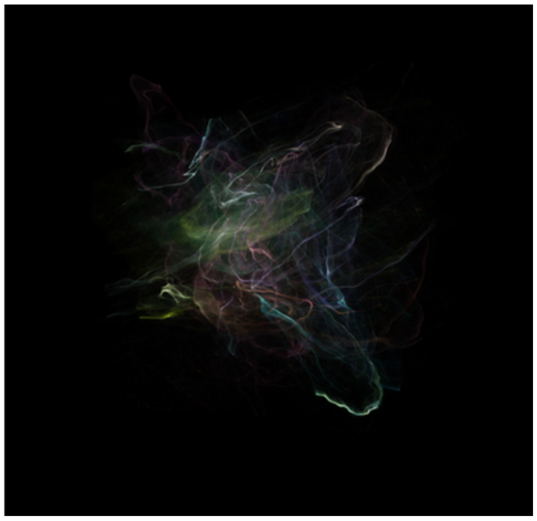
Variation 2 - Spherical

$V_2(x,y,z) = \frac{1}{r^2}(x,y,z)$

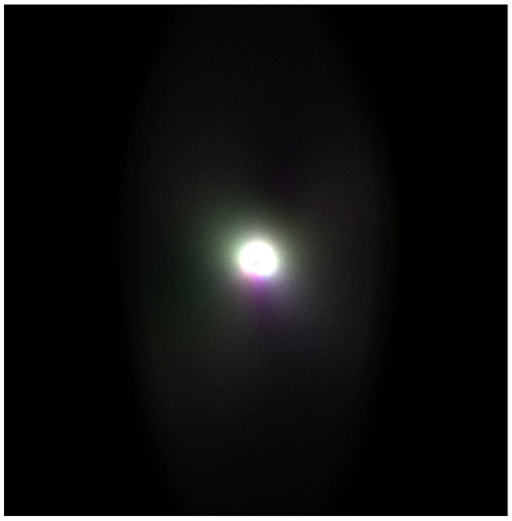
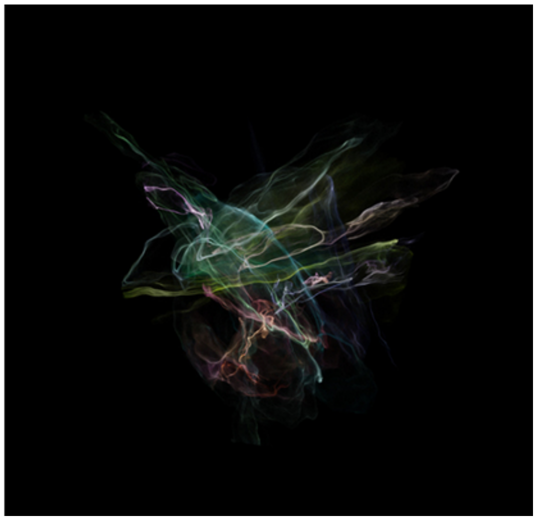
Variation 3 - Swirl

$V_3(x,y,z) = (x\sin(r^2) - y\cos(r^2), x\cos(r^2) + y\sin(r^2), x\cos(r^2) - z\sin(r^2))$

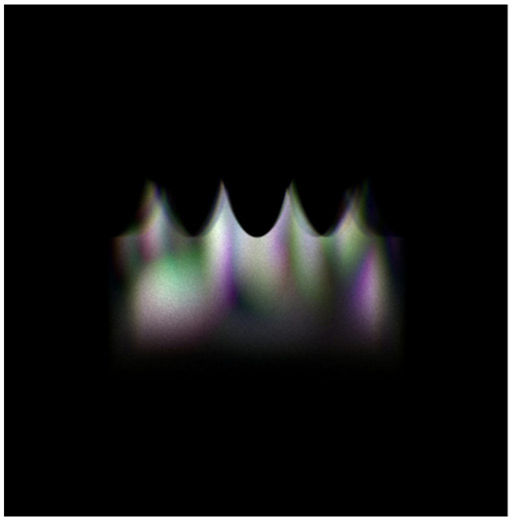
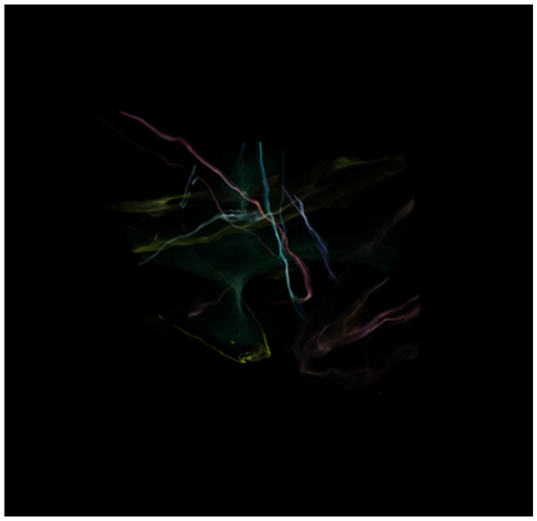
Variation 4 - Horseshoe

$V_4(x, y, z) = \frac{1}{r}((x-y)(x+y), 2xy, (x-z)(x+z))$

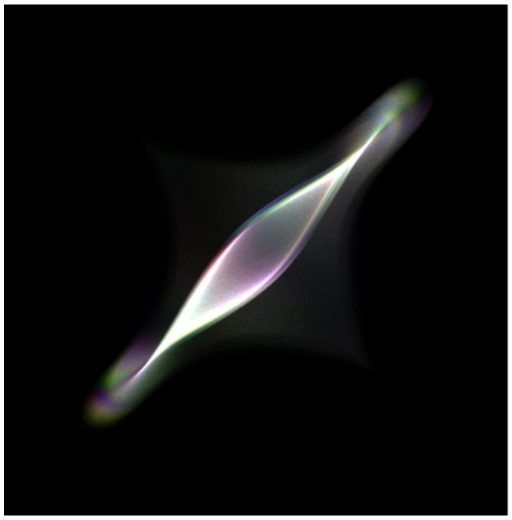
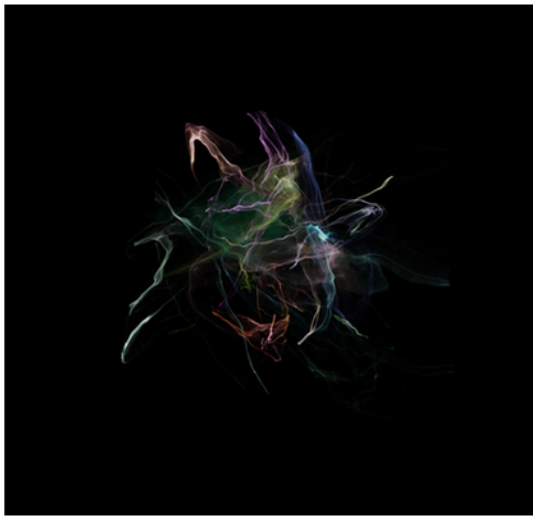
Variation 5 - Polar

$V_5(x, y, z) = (\frac{\theta}{\pi}, r-1, \frac{\theta_2}{\pi})$

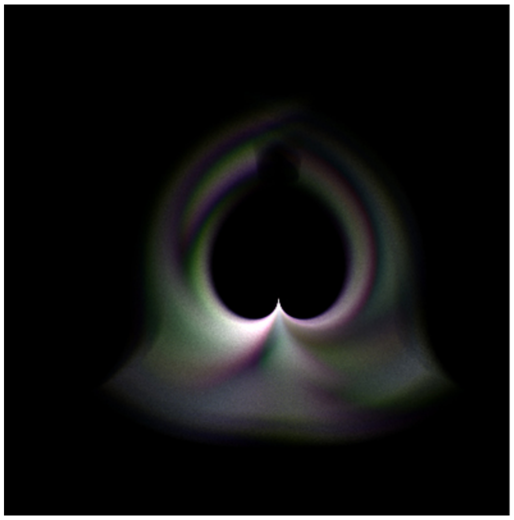
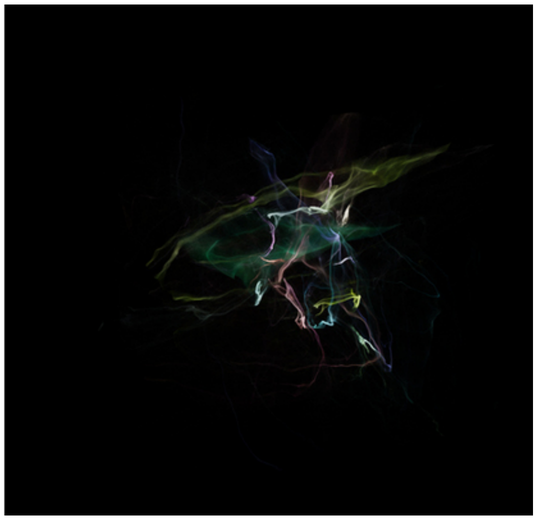
Variation 6 - Handkerchief

$V_6(x, y, z) = r(\sin(\theta + r), \cos(\theta - r), \sin(\theta_2 + r))$

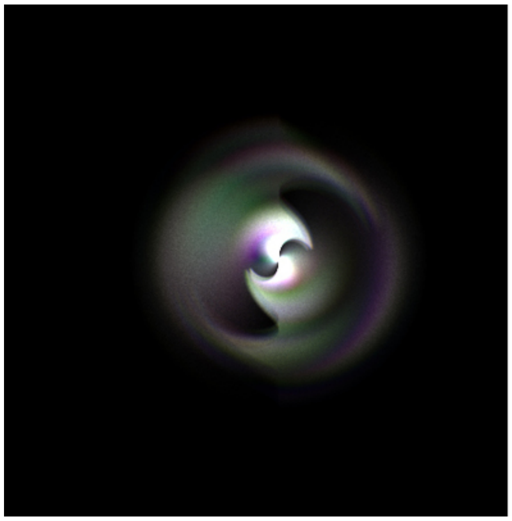
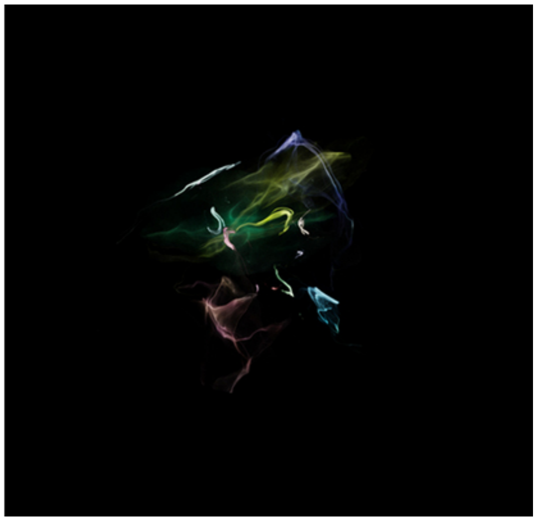
Variation 7 - Heart

$V_7(x, y, z) = r(\sin(\theta r_1), -\cos(\theta r_1), -\cos(\theta_2 r_2))$

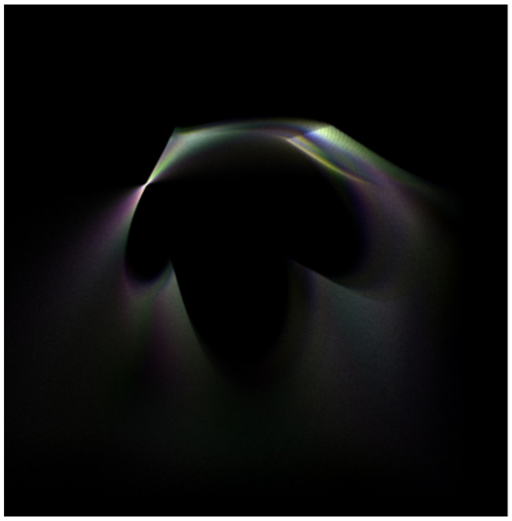
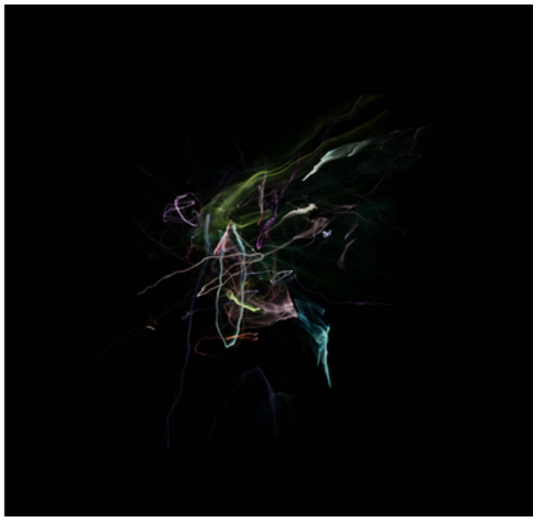
Variation 8 - Disc

$V_8(x, y, z) = (\frac{\theta}{\pi}\sin(\pi r), \frac{\theta}{\pi}\cos(\pi r), \frac{\theta_2}{\pi}\cos(\pi r))$

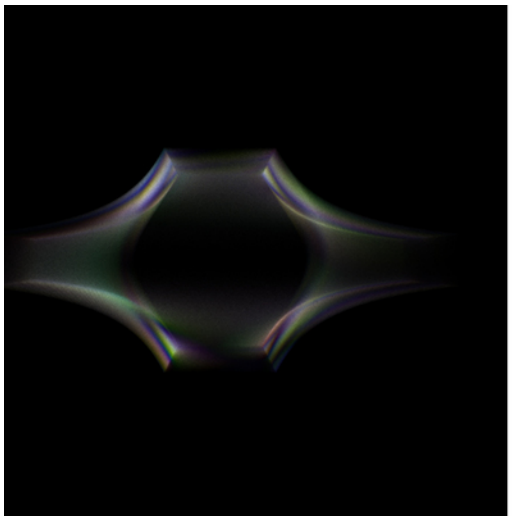
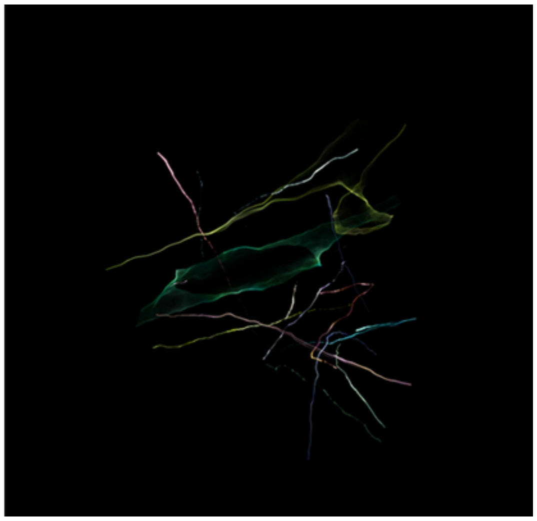
Variation 9 - Spiral

$V_9(x, y, z) = \frac{1}{r}(\cos\theta + \sin r, \sin\theta - \cos r, \cos\theta_2 + \sin r)$

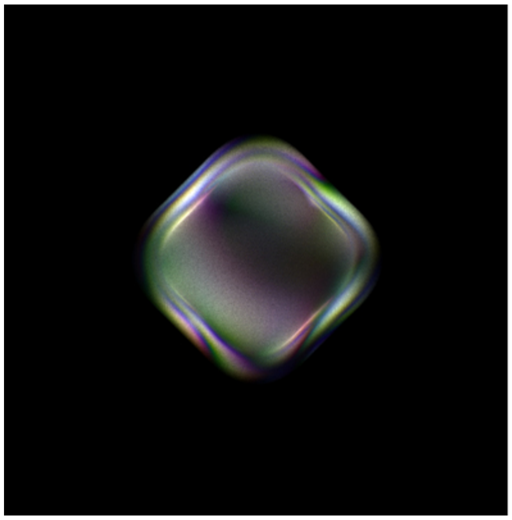
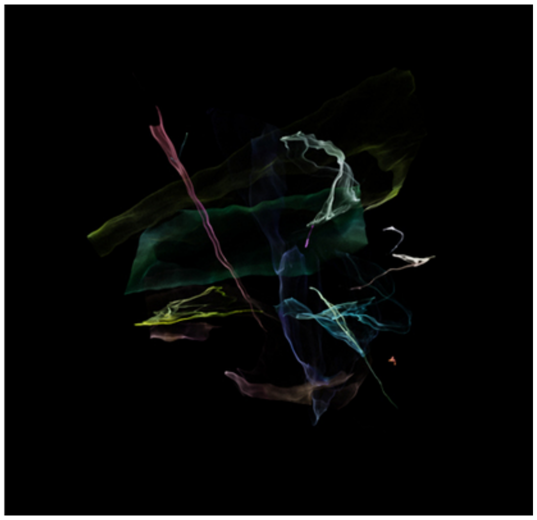
Variation 10 - Hyperbolic

$V_{10}(x,y,z) = (\frac{\sin\theta}{r}, r\cos\theta, \frac{\sin\theta_2}{r})$

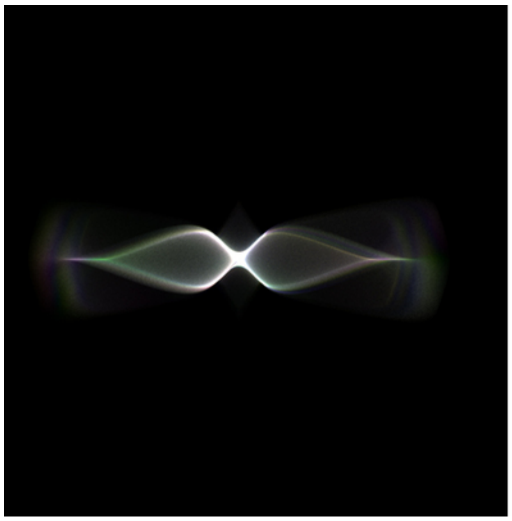
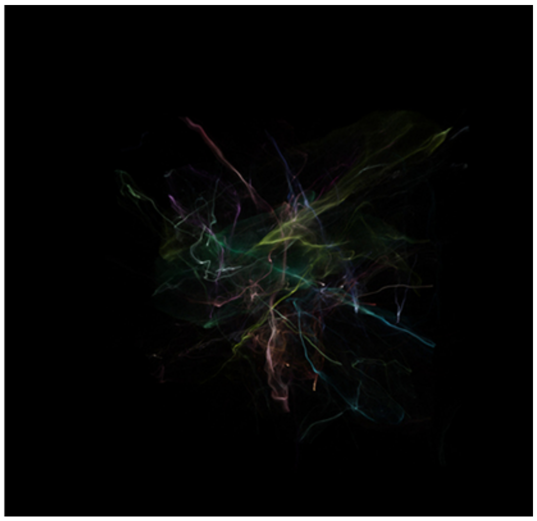
Variation 11 - Diamond

$V_{11}(x,y,z) = (\sin\theta\cos r, \cos\theta\sin r, \sin\theta_2\cos r)$

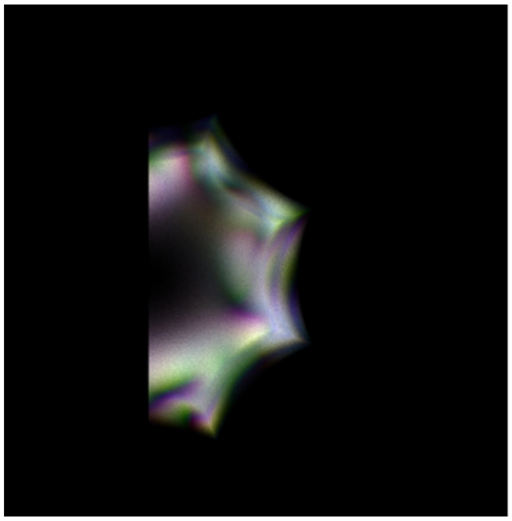
Variation 12 - Ex

$p_0 = \sin(\theta + r)$
$p_1 = \cos(\theta - r)$
$p_2 = \sin(\theta_2 + r)$
$p_3 = \cos(\theta_2 - r)$
$V_{12}(x, y, z) = r(p_0^3 + p_1^3, p_0^3 - p_1^3, p_2^3 - p_3^3)$

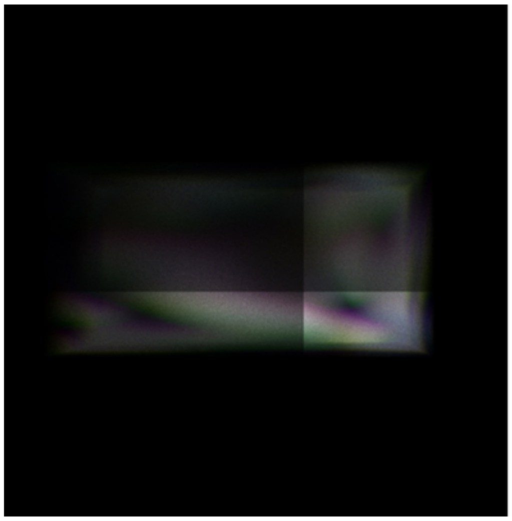
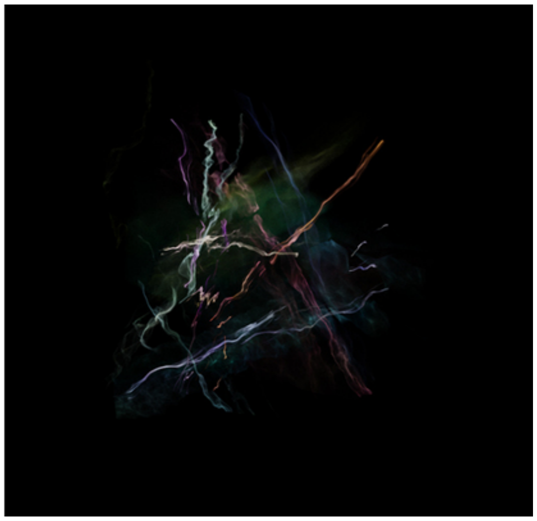
Variation 13 - Julia

$V_{13}(x, y, z) = \sqrt{r}(\cos(\theta/2), \sin(\theta/2), \sin(\theta_2/2))$



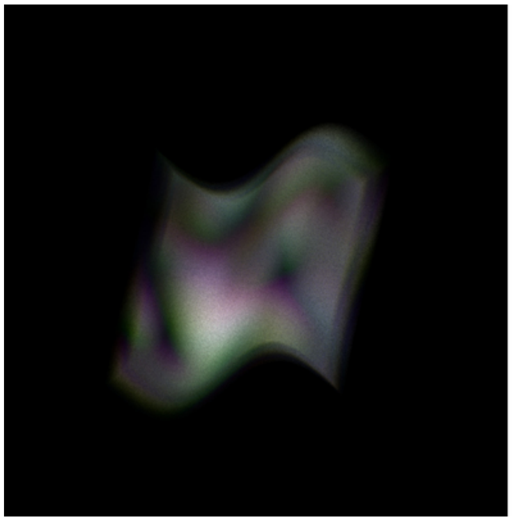
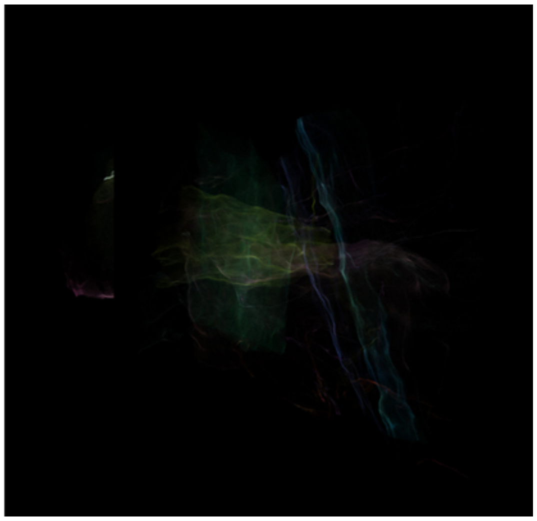
Variation 14 - Bent

$$V_{14}(x,y,z) = \begin{cases} (x,y,z) & x \geq 0, y \geq 0 \\ (2x,y,2z) & x < 0, y \geq 0 \\ (x,y/2,z/2) & x \geq 0, y < 0 \\ (2x,y/2,z) & x < 0, y < 0 \end{cases}$$

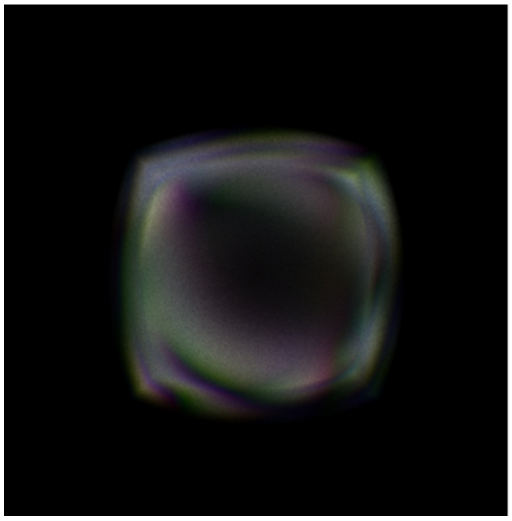
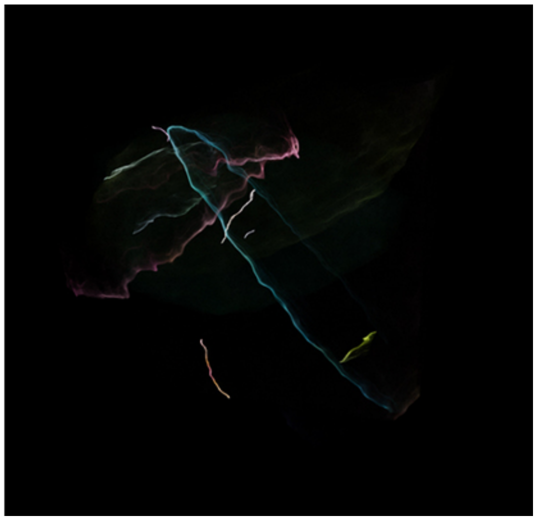
Variation 15 - Wave

$$V_{15}(x,y,z) = (x + b\sin(\tfrac{y}{c^2}), y + e\sin(\tfrac{x}{f^2}), z + h\sin(\tfrac{z}{i^2}))$$

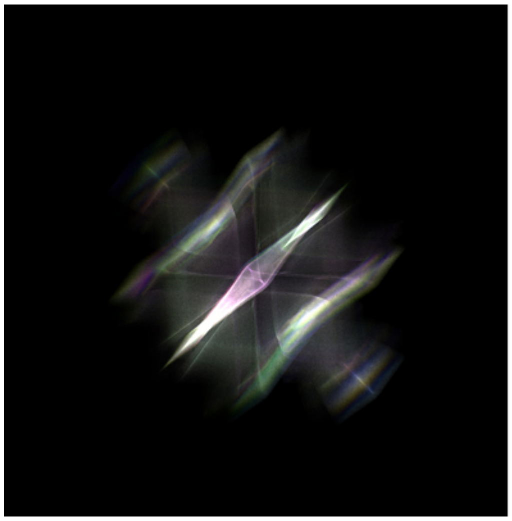
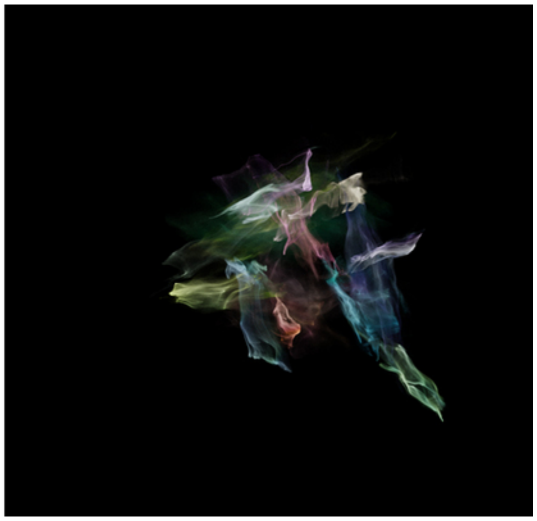
Variation 16 - Fisheye

$V_{16}(x,y,z) = \frac{2}{r+1}(y,x,z)$

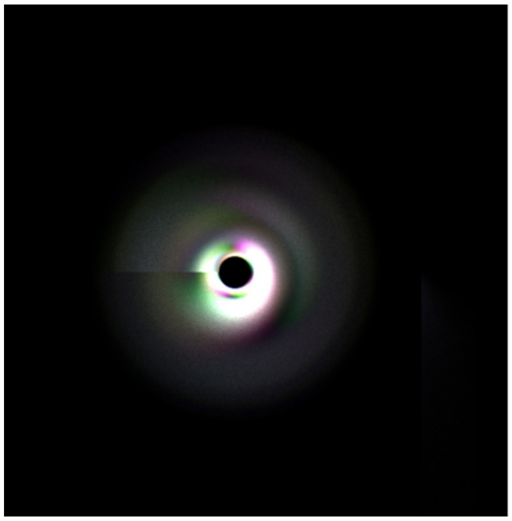
Variation 17 - Popcorn

$V_{17}(x,y,z) = (x + c\sin(\tan 3y), y + f\sin(\tan 3z), z + g\sin(\tan 3x))$

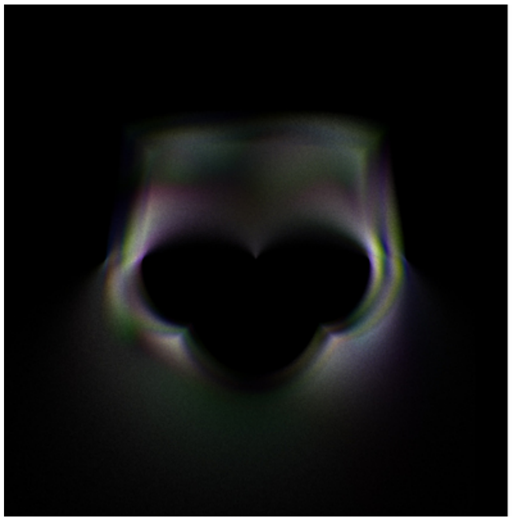
Variation 18 - Exponential

$V_{18}(x,y,z) = (exp(x-1)\cos(\pi y), exp(x-1)\sin(\pi y), exp(y-1)\sin(\pi z)$



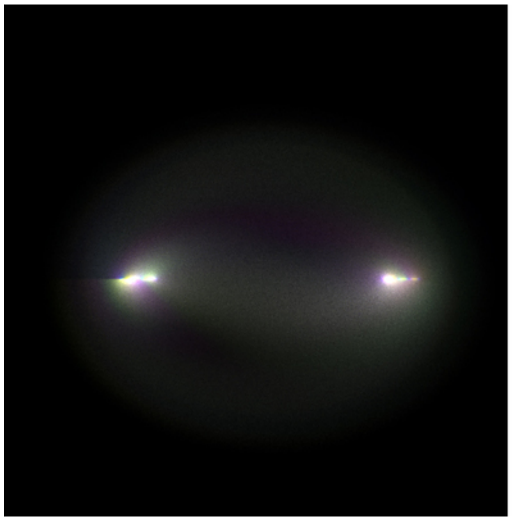
Variation 19 - Power

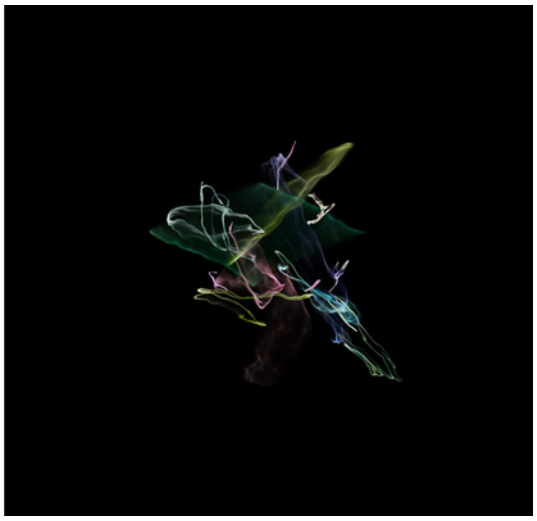
$V_{19}(x,y,z) = r^{\sin\theta}(\cos\theta, \sin\theta, \cos\theta_2)$



Variation 20 - Cosine

$V_{20}(x, y, z) = (\cos(\pi x)\cosh(y), -\sin(\pi x)\sinh(y), \sin(x))$





Bibliography

[1] S. Draves and E. Reckase, "The fractal flame algorithm." http://flam3.com/flame_draves.pdf, 2008.

[2] "IMDB." http://www.imdb.com/name/nm0047665/.

[3] J. Aller. Email Interview, May 2013.

[4] R. Baily, "Image savant spore." http://www.imagesavant.com/info.html, 2005.

[5] M. Wrenninge, N. Zafar, J. Clifford, G. Graham, D. Penney, J. Kontkanen, J. Tessendorf, and A. Clinton, "Volumetric methods in visual effects," (Los Angeles, CA, USA), SIGGRAPH course, July 2010.

[6] J. Tessendorf and M. Kowalski, "Resolution independent volume," in *Production Volume Rendering 2*, (Los Angeles, CA, USA), SIGGRAPH course, July 2011.

[7] "Wisp image in Lord of The Rings: Fellowship of The Ring." http://wpc.4846.edgecastcdn.net /804846/www//r/dd/filfo/44/64/4464/Lord_of_the_Rings_G_07.jpg.

[8] "Andromeda galaxy." http://www.scienceinthebible.net/KNOWLEDGE_BIBLE/andromeda_big.jpg.

[9] "Elliptical galaxy hercules a." http://www.dailymail.co.uk/news/article-2254802/Dazzling-collection-Hubble-Telescope-photographs-released-year-captures-countless-swirling-stars-sparkle-space.html.

[10] "Nasa." http://www.nasa.gov/mission_pages/hubble/science/hercules-a.html.

[11] J. Hutchinson, "Fractals and self-similarity," *Indiana Univ. Math. J.*, vol. 30, no. 5, pp. 713–747, 1981.

[12] M. Barnsley, J. Hutchinson, and O. Stenflo, "V-variable fractals and superfractals," *eprint arXiv:Math/0312314*, 2003.

[13] J. Tessendorf, "Volume modeling and rendering." CPSC 819 Notes, 2012.

[14] R. Wagner, "Mersenne twister: A random number generator since(1997/10)." http://www.math.sci.hiroshima-u.ac.jp/ m-mat/MT/emt.html, 2009.

[15] "SWIG." http://www.swig.org/.

[16] "Palmetto Cluster." http://citi.clemson.edu/palmetto/.

[17] B. Pelfrey, "A mathematical framework for volume modeling and simulation," Master's thesis, Clemson University, August 2012.

[18] "OpenVDB." http://www.openvdb.org/.