Accelerated rendering of fractal flames

Michael Semeniuk, Matthew Znoj, Nicolas Mejia, and Steven Robertson

November 28, 2011

CONTENTS

CHAPTER 1

EXECUTIVE SUMMARY

This document is provided as a technical manual describing all design considerations for the senior design project Cuburn, discussed herein.

1.1 Description

Cuburn is a completely software based project created for the purpose of creating visually appealing images and image sequences. More specifically, it is a GPU accelerated implementation of the flam3 algorithm for rendering fractal flames. The project is being created in the open source community and has the support of several developers currently working in flam3 related projects. The software being developed is being designed to be platform independent and to be usable as a substitute for the standard flam3 library. Fractal flames generated by Cuburn should be visually identitical to the human eye but will be rendered in a fraction of the time compared to flam3. The developers have pulled out all the stops to implement the latest cutting-edge technology whenever possible to help reach the goal of performing real time fractal flame rendering on a personal computer.

1.2 Significance

Many implementations of the flam3 algorithm already exist and have existed for many years. This project is significant because it is a modest improvement over all of the other implementations currently available at this time. It is a GPU implementation of the flam3 algorithm, designed to produce images equivalent to the CPU implemented flam3 software, something that other GPU implementations have yet to do. It should be noted that time moves quickly in the realm of software development and that there are others may be trying to accomplish today what is being described in this document. However, being that this software is being designed with the bleeding edge of technology in mind and with many optimizations being performed on all levels, it should proove difficult for another project to offer any modest improvement over this design.

1.3 Motivation

The team designing this project likes fractal flames, as do thousands of others. Fractal flames do not offer much of a practical purpose, they were only created for the mere entertainment only. It could be possible that they hold the key to unlocking the many mysteries of the universe, but for now, they just look pretty. Current software for creating these mesmorizing image sequences are relatively slow or of low quality.

There is no hope to use any currently existing software to incorporate fractal flames into real time applications such as music visualization. It is this condition that drives the motivation for this project. The authors are set out to create a high quality, high performance, fractal flame renderer that can generate exceptional flames in, or closer to, real time. This is not a trivial task, hence the reason it has not already been accomplished. The goal for real time rendering is an optimistic one, but all the stops are being pulled out so that if there is anything in the way of accomplishing this, it will only be the computationial resources limit of current hardware technology.

1.4 Goals and Objectives

The overall goal of this project is to create a piece of software that can render fractal flames of comparable quality to the original flam3 implementation that can do so in a fraction of the time. To reach this goal, the following objectives have been set:

- Independently implement a working version of the fractal flame algorithm.
- Develop a concrete and functionally complete understanding of GPU performance (for the particular architecture we select) through targeted microbenchmarking and statistical analysis.
- Using knowledge gained through microbenchmarking, rewrite the fractal flame algorithm for GPUs using the aforementioned dialect.
- Develop, implement, and test new optimization strategies to improve the speed of the renderer.
- Use statistical, graphical, and psychovisual techniques to improve the perceived quality per clock ratio.

Optionally:

- Add 3D support.
- Apply resulting renderers in real-world applications, including but not limited to:
  - Music visualization
  - Reactivity to environment
  - Real-time interactivity
  - Real-time evolution using genetic algorithms and user feedback
- Develop a ruthlessly optimized, composable, typesafe dialect of CUDA. Implement portions of a standard library with it.

1.5 Usage Requirements

Being that this project is a software application developed for use on personal computers, this section will outline what is required to run and use the software. It should be clear that this project relies very much on a specific hardware device, the GPU. Therefore, the designers have set strict requirements for this piece of hardware. Other hardware devices such as the CPU and memory have less strict requirements and are more or less presented as a recommendation. The required operating systems and software needed to run Cuburn are relatively easy to come by and available freely, but will nonetheless still be required. - NVIDIA CUDA-enabled GPU supporting Compute Capability 2.1

- 2GHz or faster CPU
- 2GB or more RAM
- CUDA compatible NVIDIA drivers

1.6 Research

The cutting-edge nature of this project requires that the latest and greatest software algorithms and hardware be used in order to obtain the highest performance possible. Much research has been put into realizing the high quality, high performance algorithms that take advantage of GPU hardware. These research topics include iterated function systems, psuedo-random number generators, coloring and log scaling, antialiasing, denoising, dynamic kernel generation, programming lanaguages, and more. Accelerating these standard algorithms for use on GPU's is key for this software to function optimally.

1.7 Design

The software will be broken up into a small collection of libraries that perform the operations necessary for rendering a fractal flame. The libraries that will be developed will be known as `cuburn`, `flam3-types`, and `flam3-hs`. `cuburn` will do the actual flame rendering, `flam3-types` contains flame genome datatypes and a basic flame parse, and `flam3-hs` provides Haskell bindings to the original flam3 library for the purpose of compatibility.

CHAPTER 2

FRACTAL BACKGROUND

2.1 Purpose of Section

The fractal flame algorithm draws upon concepts across many fields including: statistics, mathematics, fractal geometry, the philosophy of art and aesthetics, computer graphics, computer science, and others. One may become short of breathe just trying to read that entire sentence on one breathe of air. The point that is trying to made is that the fractal flame algorithm is arguably the most complex fractal process to date. The road ahead of us for not only optimizing but fundamentally changing the process for how fractal flames are rendered is not so clear and will require a solid knowledge as well as innovation.

The innovation is what the majority of this paper is about and as a guiding rule the words of Sir Francis Bacon are very true to the author's research process: *"When you wish to achieve results that have not been achieved before, it is an unwise fancy to think that they can be achieved by using methods that have been used before."*

As unwise as it would be to assume a solution to the current design challenge has already been solved, it would also be unwise not to draw from previous knowledge from the aforementioned fields. Therefore knowledge from mathematics, statistics, and graphics will be supplemented as needed when design decisions are presented later in the paper. However, before the paper transitions into the innovation aspect of this project, the need to present ample background information on two fields of which warrant attention is felt. These fields are fractal geometry and the aesthetic nature of fractal geometry.

The justification of presenting fractal geometry lies in the reasoning that the mathematics and properties behind it is not blatantly intuitive and key concepts cannot be hand waved later in this paper. Had the famous equation $z_{n+1} = z_n^2 + C$ been intuitive then humans would be able to visualize the Mandelbrot Set, seen in as seen in Figure 2.1, and understand its ability to scale infinitely without degradation, without the aid of computer graphics.

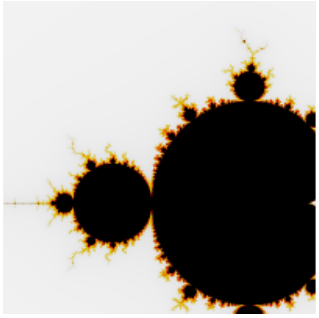


Figure 2.1: The Mandelbrot Set

This section will touch on these intriguing and sometimes counterintuitive fractal properties and also address their relevance in the project and what limitations they pose upon us for a GPU implementation or new approach. The different types of fractals and how fractal flames, a variant of the iterated function system, vary from the Mandelbrot set, shown above, will be explained. Unlike classical geometry, fractal geometry is a rather new field of geometry and the authors believe presenting a comprehensive knowledge of the field in context of the project is absolutely feasible.

The next area that will be articulated is an atypical one: the aesthetical nature of fractal geometry. The concept of beauty is something that has not been universally defined and one may often allude to the idiom: *"Beauty is in the eye of the beholder."* Besides perhaps art therapy and for visual appeal, flame fractals do not have an immediate real life application and therefore much of the justification for developing a GPU Fractal Flame Render lies upon their aesthetics, the idea of creating a process which allows artistic formation, and the wonder they bring. Excruciating detail is spared but major milestones are shown in history dating back to African civilizations who built their culture and art around self-similar repeating geometric figures. The point trying to be made is that there is a widely accepted attraction towards these shapes that penetrates different societies and cultures.

After understanding the background behind fractal aesthetics this will be furthered with additional visual concepts such as gamma correction, filtering, motion blur, and symmetry.

2.2 Origins: Euclidean Geometry vs. Fractal Geometry

Geometry has formalized the way humans talk about and perceive points, shapes of figures, and the properties of space. Up until the 19^{th} century geometry need not be prefixed with the specific type of geometry that it was referring to- it was assumed it was Euclidean, named after *Euclid* the Greek mathematician of Alexandria, Egypt. While teaching at the Alexandria Library, Euclid had transcribed a comprehensive set of 13 books in which he titled *Elements*. These books described Euclidean Geometry (and other topics) and included his own work along with other mathematicians including Thales, Pythagoras, Plato, Eudoxus, Aristotle, Menaechmus, and other predecessors.

Element's impact was dramatic. So much so that *Euclid* is often referred to as the "Father of Geometry". By the 20^{th} century Euclidean geometry was being taught globally in schools. Shapes such as: circles, triangles, and polygons are taught at an early age.

However as influential as the idea of Euclidean Geometry is its ideal shapes failed to describe the shapes that appear in nature. As stated in the opening paragraph of Benoît Mandelbrot's book, The Fractal Geometry of Nature[1] : *"Clouds are not spheres, mountains are not cones, and lightning does not travel in a straight line. The complexity of nature's shapes differs in kind, not merely degree, from that of the shapes of ordinary geometry."*

2.3 Fractal Geometry and Its Properties

This new geometry Benoît Mandelbrot writes about in his book, he calls fractals which come from the Latin work fractus meaning *"fractured"*. These new shapes exhibited different properties than classical Euclidean shapes. These shapes were rough and did not belong to an integer valued dimension. Fractals also exhibited self-similarity in which parts of the figure repeat themselves. Ideal fractals also did not degrade with scale either like other classical shapes or like a photograph. These new shapes had been investigated in the Western World previous to Mandelbrot and were already an accepted part of African art and culture before Mandelbrot had been observed and published his findings which lead to their widespread use and acceptance.

The properties in which Mandelbrot and his predecessors have found are summarized. They later will be freely referenced from this point forward when they are needed to explain additional concepts.

Self Similarity

Fractals contain the property of self-similarity. This self-similarity is classified into different types ranging from the strongest form which is called exact self-similarity to the weakest form called statistical or approximate self-similarity. The three classifications are below:

EXACT SELF-SIMILARITY: This type of self-similarity contains, as its name implies, exact copies of itself repeating at infinitely smaller scales. Classical examples include Sierpinski's gasket or the Koch Curve which can be seen in Figure 2.2.

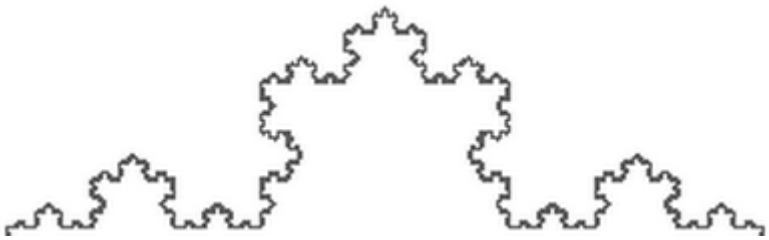


Figure 2.2: The Koch Curve

QUASI SELF-SIMILARITY: This type of self-similarity does not contain exact copies but rather distorted or degenerate forms of itself at infinitely smaller scales. Classical examples include the Mandelbrot set seen above in Figure 2.1.

STATISTICAL SELF-SIMILARITY: This type of self-similarity is the weakest and is the type often encountered in the real world. Statistical self-similarity refers to the fact that the object has numerical or statistical measurements that are maintained at different scales. When classifying shapes in nature as fractal-like this definition is being implied. For example, the self-similar aspects of how a tree branches are never found to be exact and sometimes deviate from their expected pattern but still exhibit self similarity in a sense. The definition of statistical self-similarity accounts for this and is important because the luxury is not always given to observe concepts in their ideal sense. We can attempt to observe this notion of statistical self-similarity in Figure 2.3 which shows a depiction of a leafless tree in order to exemplify the properties of the tree's branches.

Figure 2.3: Statistical self-similarity found in the branching of trees.

Another classical example is measuring a coastline such as Britain. When scaling the coastline it appears similar to at magnified scales. Additionally, what follows from this is the more accurately one measures the coastline (with a smaller base measurement) the more the length increases. This length increases without limit and contrary to intuition shows that the coastline of a country is infinite.

Fractal Dimensionality

Classical dimensionality is often expressed in whole number integer values. Lines have a dimensionality of 1, squares have a dimensionality of 2, and cubes have a dimensionality of 3. This however does not explain how completely a fractal fills a space. Does the Sierpinski's Triangle, seen in Figure 2.4, cover 1 dimension like a line or 2 dimensions like a triangle? The answer is actually that it contains a dimension that is between the two!

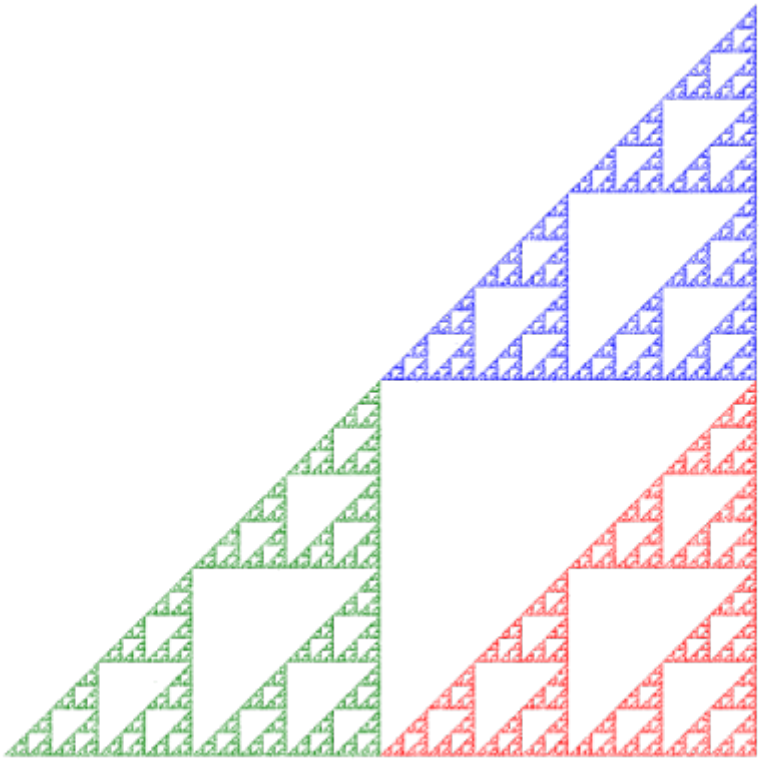


Figure 2.4: A visual of Sierpinski's Triangle which has a fractal dimensionality.

This can be shown using a variety of ways that formally define fractal dimensionality including: Hausdorff dimension, Rènyi dimension, and packing dimension. These theoretical definitions differ in their approach however all three attempts to explain the same phenomenon: real numbered dimensionality.

Fractal dimensionality will be explained in this section in an intuitive way rather than providing the reader with a heavy mathematical explanation. This will be done using the concept of a box-counting dimension which lends itself to ideas from the Rènyi dimension.

To calculate the dimensionality of an object, an equidistant grid is imposed upon the object and the number of boxes that are necessary to cover the object are counted. The process continues and the equidistant grid is

refined by decreasing the size of the grid. Again, the number of boxes that are necessary to cover the object are counted and the process repeats.

The formula used is:

$$\text{Dimensionality}_{box}(S) = \lim_{\varepsilon \to 0} \frac{\log N(\varepsilon)}{\log \frac{1}{\varepsilon}}$$

where $N(\varepsilon)$ is the number of boxes needed to cover the set, ε is the side length of each box, and S is the set to be covered.

For a line with a known dimensionality of 1 the box counting procedure is performed. The procedure will start with a side length of length 1 and continually half the side length until a recognizable pattern emerges which can be observed in Figure 2.5.

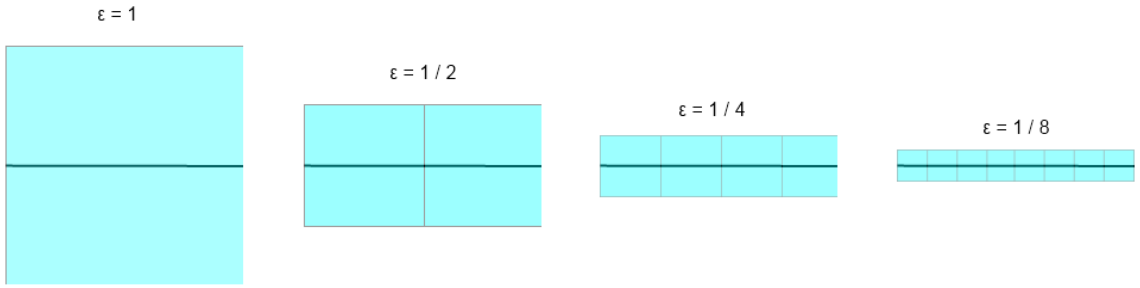


Figure 2.5: Box Counting Dimension Process For a Line

The box counting equation can be solved by completing the pattern that shows the rate at which the number of boxes in the grid grow compared to the number of boxes needed to cover the shape as the side length approaches 0. This is shown in Table 2.1.

BOX LENGTH: ε	NUMBER OF BOXES: $N(\varepsilon)$
1	1
$\frac{1}{2}$	2
$\frac{1}{4}$	4
$\frac{1}{8}$	8
...	...
ε	$\frac{1}{\varepsilon}$

Table 2.1: Box length (ε) and the number of boxes ($N(\varepsilon)$) as ε approaches 0.

From this table the following formula can be deduced by solving the pattern.

$$\text{Dimensionality}_{Line}(S) = \lim_{\varepsilon \to 0} \frac{\log \frac{1}{\varepsilon}}{\log \frac{1}{\varepsilon}} = 1$$

Our box counting procedure coincides with the view that a line has a dimensionality of one. We now use this same box counting procedure to calculate a shape of non integer value dimensionality. Sierpinski's gasket will be used as the example. The procedure will again start with side length of 1 and continually half it until a recognizable pattern emerges which can be observed in Figure 2.6.

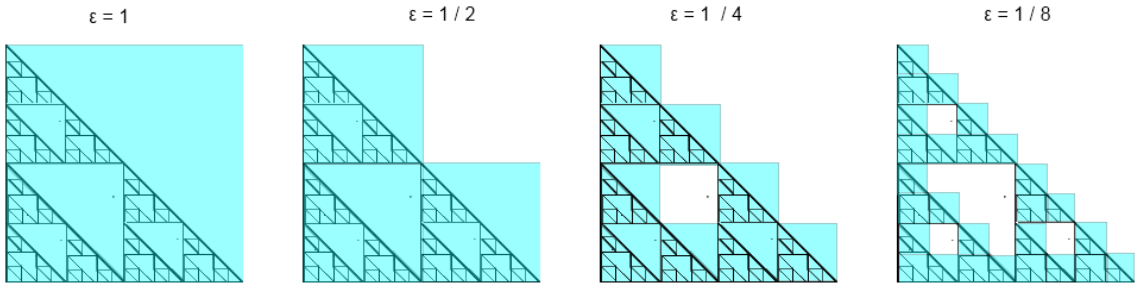


Figure 2.6: Box Counting Dimension Process For Sierpinski's Gasket

The results are rewritten in the form of powers to expose the pattern. This is shown in Table 2.2.

BOX LENGTH: ε	NUMBER OF BOXES: $N(\varepsilon)$
1	$3^0 = 1$
$\frac{1}{2^1} = \frac{1}{2}$	$3^1 = 3$
$\frac{1}{2^2} = \frac{1}{4}$	$3^2 = 9$
$\frac{1}{2^3} = \frac{1}{8}$	$3^3 = 27$
...	...
$\frac{1}{2^N} = \varepsilon$	3^N

Table 2.2: Box length (ε) and the number of boxes ($N(\varepsilon)$) as ε approaches 0.

From this table the following formula can be deduced by solving the pattern.

$$\text{Dimensionality}_{Sierpinski}(S) = \lim_{\varepsilon \to 0} \frac{\log 3^N}{\log 2^N} \approx 1.58$$

The concept of dimensionality is often referred to as ROUGHNESS which is a measure of a shape's irregularity.

Formation by Iteration

The method for constructing a fractal relies on an iterative process. Regardless if the fractal is a naturally occurring statistically self-similar fractal, a computer generated fractal, or even a mathematical calculation of a set that exhibits fractal-like properties they all rely on a process which involves multiple iterations of a specific process. This process could be for example in geometric fractals scaling shapes or in the case of algebraic computer generated fractals adjusting parameter values.

2.4 Fractal Types

When one gets their first taste of fractal geometry they notice the diversity of shapes and figures that encompass it. For the paper's purposes, fractals will not be classified by how they visually look but rather the process for creating them. This is done because given the nature of this project the focus is on the data structures and algorithms used to create the fractal. The shape and patterns that are merely the byproduct of the process. It is not always apparent which creation method was used to create a certain pattern. By classifying fractals by their creation method, the following information is gained:

1. Explain what this project is not
2. Draw similarities from closely related fractal systems
3. Compare the bottlenecks and difficulties between systems.

The major classifications of fractals by their generation methods are the 4 types presented in the following subsections.

Escape Time Fractals

This type of fractal relies on recursively applying an equation upon an initial point. The transformed point can either diverge past a certain bounds, set by the programmer, or can never reach the escape circumstance.T his bounds is called the escape circumstance. Different points reach the escape circumstance at different rates.

Output images of these images can be black and white denoting which points did not escape and which points did escape. This however is too simplistic and does not produce visually appealing image. A simple fix that greatly enhances the appearance is coloring the points depending on how fast each point escaped.

Classical examples of fractals include:

- Julia Set
- Mandelbrot Set
- Orbital Flowers

and many others.

Strange Attractors

Strange attractors, such as the one seen in Figure 2.7, are attractors whose final attractor set are that of a fractal dimension. An attractor is a set that a dynamical system approaches as it evolves. Dynamical systems are systems which describe the state of the system at any instant and contain a rule that specifies the future state of system.

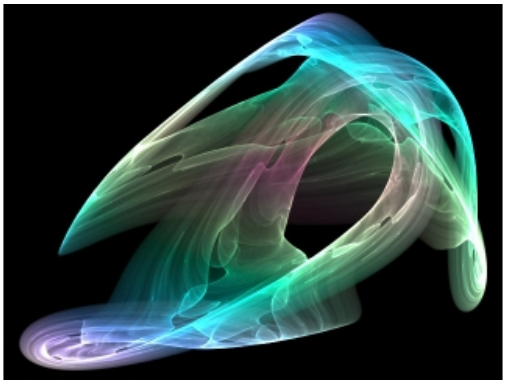


Figure 2.7: Image of a Strange Attractor

A difference of the strange attractor versus a traditional attractor is that strange attractors have a sensitive dependence on their initial conditions and often exhibit properties of chaos[1] which makes their behavior hard to predict.

Random Fractals

Random fractal's iterative process relies on a non-deterministic process for creation. By applying some process the resulting set or image exhibits fractal-like properties such as the two images seen in Figure 2.8. Many landscapes and plants in nature exhibit this property. For example, mountains are not formed by a deterministic process yet exhibit statistical self-similarity. Fractal landscape generation is a stochastic process which tries to mimic this stochastic process in nature.

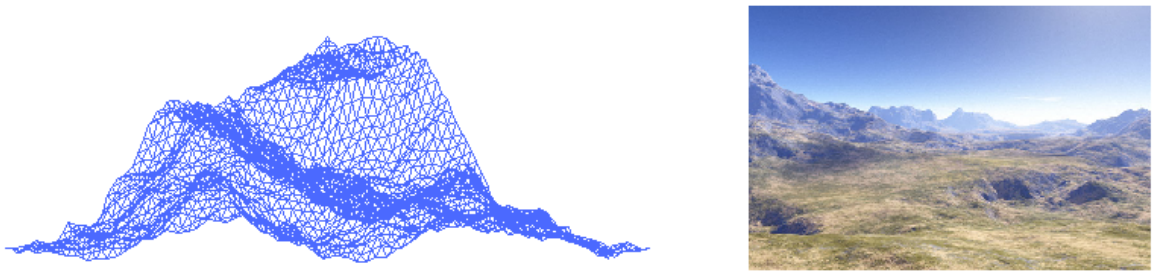


Figure 2.8: Image of a computer generated fractal landscape compared with a mountain landscape

Iterated Function Systems

This is the fractal system that the project will focus upon. Iterated function systems rely on performing a series of transformations stochastically (which are generally contractive on average[2]) to produce the output image. This stochastic process is called the CHAOS GAME. The CHAOS GAME starts with randomly choosing an initial point and then consecutively applying a randomly chosen transformation from the set of transformations that make up the iterated function system.

The entire iterated function system process and its intricacies will be articulated upon in Section 3.2.

2.5 Visual Appeal

The visual appeal of fractal geometry is far reaching and includes groups of people such as certain African societies, individuals who appreciate the fractal aspects of nature, and online fractal art communities such as Electric Sheep. Its universal appeal is of course subjective like any other art societies.

First and foremost, nature has is the most apparent in creating fractal-like features which can readily be observed. Examples are plentiful and include:

- The leaves of ferns and other plants
- Tree branching

[1] When the properties of chaos are referred to what is meant by them is the notation that a point which is close to the attractor will become separated at an exponential rate.

- Mountain landscapes
- Certain intricate rivers
- River erosion patterns
- Coastlines
- Electrical discharge patterns
- Romanesco (a broccoli-like plant)
- Hydrothermal springs
- Cloud-spiral Formations
- Virus and bacterial colonies
- Coastlines
- and numerous others

The wonder that nature brings individuals can partly be attributed to the idea of self-similarity and the complex shapes it produces.

Fractal Geometry has been a part of the African culture, social hierarchy, and art predating any formal western knowledge on fractals. Village architecture, jewelry, and even religious rituals all exhibit the concepts of self-similarity[3]. Recently with the advancement of computer aided image generation, the appreciation of fractals has spread to a wider community. For example, the application Electric Sheep uses distributed computing in order to evolve fractal flames which are displayed as screensavers to users. The community has membership of roughly 500,000 unique members [4] who appreciate viewing fractal flame images.

Hopefully this background information shows the general interest in fractal-like patterns and with that the project focuses on this last group of individuals who appreciate computer generated fractal images. The proposed GPU rendered fractal algorithm hopes to deliver the existing community with the opportunity to continue viewing these fractal flame images without the need for distributed computing to render them in real time- a major improvement.

2.6 Limitations of Classical Fractal Algorithms

Escape Time Fractals, Strange Attractors, and Random Fractals all have distinct methods of fractal generation however they lack several characteristics which limit the resulting images and videos that can be generated with them. Some of the limitations include:

- A generic process for combining multiple effects (whether they be matrix transformations, series of equations, or process steps) to create an increasingly complex fractal.
- The ability to structurally color each defined effects instead of coloring the entire result of all of the combined effects.
- Inherently, take on the task of image correction and color theory as part of the problem in order to provide higher quality and more accurate output.
- The ability to seamlessly interpolate between effects.

All of these bulletpoints above are accomplished using the fractal flame algorithm, a variant of the Iterated Function System fractal type. These additionally features allow beautiful interpolation between transformations, a heightened focus on color and image correction techniques, as well as more intricate shapes. Because of these additional features the flame algorithm has many advantages over classical fractal flame algorithms which is one of the governing reasons why this system was chosen for the project.

CHAPTER 3

THE FRACTAL FLAME ALGORITHM

3.1 Section Outline

This section provides an in-depth description of the fractal flame algorithm along with a primer on the Iterated Function System (IFS) in which the fractal flame algorithm is a variant of. This primer is provided to the reader in order to solidify the concept of the chaos game which is essential to understanding the flame algorithm because it builds heavily on upon the concepts that are used in the classical IFS.

Also included in this section is a brief history of the Flame algorithm from its birth in 1992 to the present day. As the algorithm is presented step-by-step references are also presented in which the topics in question are explained in more detail.

Finally, we end with a concluding section summarizing our current knowledge on the topic and describe how it influenced our proposed implementation for rendering fractal flames using the flame algorithm which is described in the following section.

3.2 Iterated Function System Primer

This primer aims to present the fundamental concepts of iterated function systems along with several classic examples that will visually and mathematically convey two important concepts:

1. The importance of random application of defined affine transformations on a random starting point in the plane
2. How affine transformations are used to transform[1] points to produce self-similar images such as Sierpinski's Triangle and Baransley's Fern.

These concepts are the building blocks of the flame algorithm. If the reader is already familiar with the concept of iterated function systems feel free to skip to Section 3.3 and begin reading about the fractal flame algorithm.

Definition

An ITERATED FUNCTION SYSTEM is defined as a finite set of AFFINE CONTRACTION TRANSFORMATIONS F_i where i= 1, 2, ..., N that map a METRIC SPACE onto itself.

[1] A transformation being an operator that can rotate, scale, translate, or provide shear to some vector space.

Mathematically this is [5]:

$$\{f_i : X \mapsto X\}, N \epsilon \mathbb{N}$$

A METRIC SPACE is any space whose elements are points, and between any two of which a non-negative real number can be defined as the distance between the points (e.g. Euclidean Space).

An AFFINE TRANSFORMATION from one vector space to another is comprised of a linear transform which gives either rotation, scaling, or shear following by a translation. Mathematically this is [6]:

These transforms can be represented in one of two ways:

1. By applying matrix multiplication (which is the linear transform) and then performing vector addition (which represents the translations).
2. By using a transformation matrix. To do this we must use homogeneous coordinates. Homogenous coordinates have the property that preserves the coordinates in which the point refers even if the point is scaled. By using the transformation matrix we can represent the coefficients as matrix elements and combine multiple transformation steps by multiplying the matrices. This has the same effect as multiplying each point by each transform in the sequence. This effectively cuts down the number of multiplications needed- this is worth noting as it will be utilized in our implementation. Figure 3.1 shows the operations in which the transformation can perform.

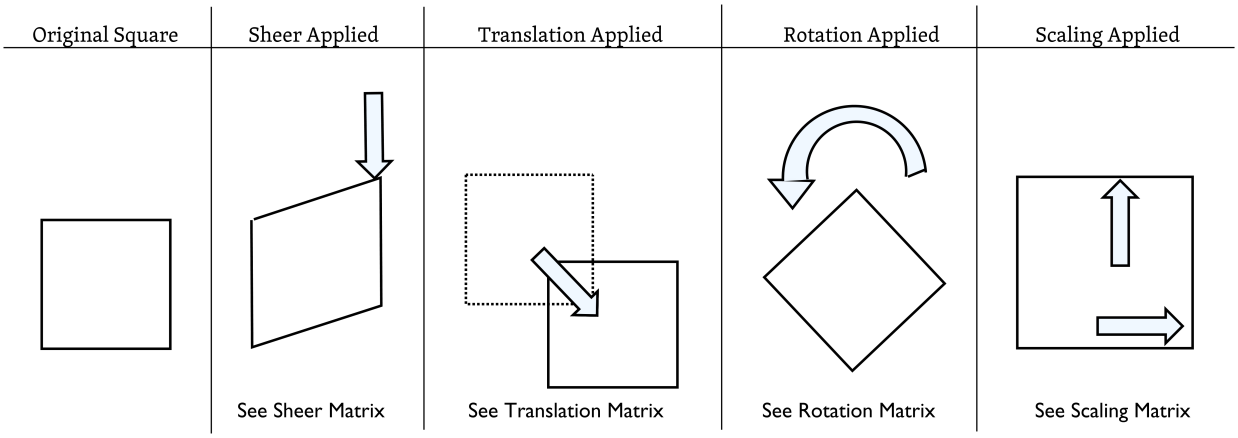


Figure 3.1: Visual representation of Shear, Translation, Rotation, and Scaling.

ROTATION MATRIX To perform rotation using the transformation matrix the matrix positions $A_{0,0}$, $A_{0,1}$, $A_{1,0}$, and $A_{1,1}$ should be modified (where A is the matrix). By using the transformation matrix below and setting θ you effectively rotate your vector space by θ degrees.

$$\begin{vmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

SHEAR MATRIX To perform shear using the transformation matrix the matrix position $A_{0,1}$ should be modified (where A is the matrix). By using the transformation matrix below and setting $Amount$ you effectively perform shear of value $Amount$ on your vector space.

$$\begin{vmatrix} 1 & Amount & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

SCALING MATRIX To perform scaling using the transformation matrix the matrix positions $A_{0,0}$ and $A_{1,1}$ should be modified (where A is the matrix). By using the transformation matrix below and setting $Scale\ Factor_x$ to the magnification you would like your x-axis and $Scale\ Factor_y$ to the magnification you would like your y-axis you effectively scale your vector space by that amount.

$$\begin{vmatrix} Scale\ Factor_x & 0 & 0 \\ 0 & Scale\ Factor_y & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

TRANSLATION MATRIX To perform translation using the transformation matrix the matrix positions $A_{0,2}$ and $A_{1,2}$ should be modified (where A is the matrix). By using the transformation matrix below and setting $Translation_x$ to the offset from your current x-point and $Translation_y$ to the offset from your current y-point you effectively translate your vector space by that amount.

$$\begin{vmatrix} 1 & 0 & Translation_x \\ 0 & 1 & Translation_y \\ 0 & 0 & 1 \end{vmatrix}$$

3. The term CONTRACTION MAPPING refers to a mapping which maps two points closer together[7]. The distance between these points is uniformly shrunk. This contraction will be seen when performing the classic Sierpinski Triangle problem . The properties above can be proved by the Contraction Mapping Theorem and because of this proves the convergence of the linear iterated function system presented in this section.

Chaos Game

The most common way of constructing an Iterated Function System is referred to as the *chaos game* as coined by Michael Barnsley. Our initial fractal flame algorithm will also use this approach. In the *chaos game* a random point on the plane[2] is selected. Next, one of the affine transformations to describe the system is then applied to this point and the resulting point is then plotted. The procedure is repeated for N iterations where N is left up to the user. Selection of the affine transformation to apply is either random (in the case of Sierpinski's triangle) or probabilistic (in the case of Barnsley's Fern). The more iterations you allow the chaos game to run for the more closely your resulting image resembles the iterated function system. A flow chart of this procedure is found in Figure 3.2.

[2]By plane we are refering to a biunit square where x and y values can have a minimum value of –1 and a maximum value of 1.

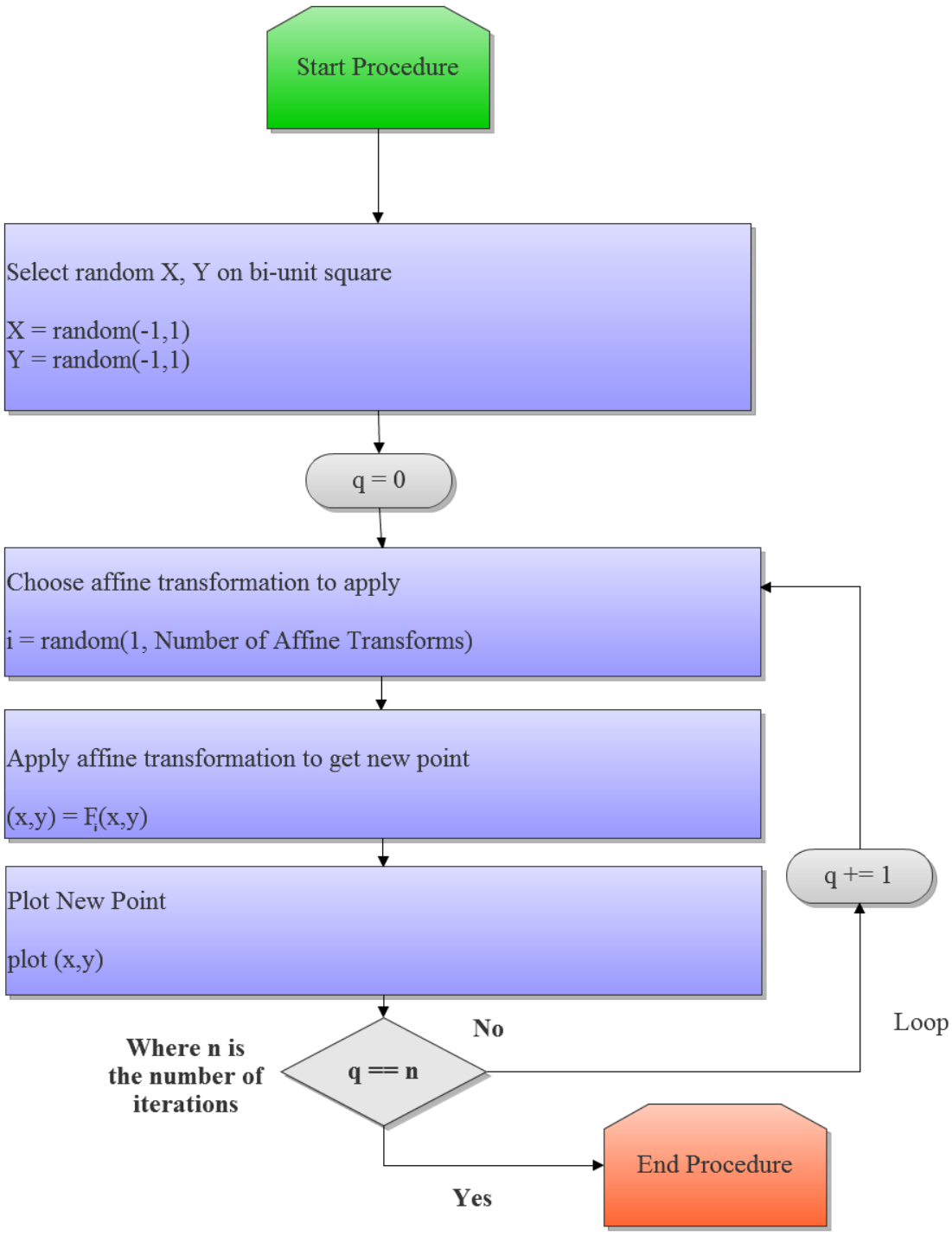


Figure 3.2: Flow chart of IFS Procedure

Classical Iterated Function System : Sierpinski's Triangle

Now that the algorithm has been explained an illustrative example known as Sierpinski's Triangle is presented for the reader This example is suitable to show how the fractal will begin to show itself after a certain number of iterations of the chaos game. This is also a suitable example to observe the contractive

nature of the affine transformations.

To construct Sierpinski's Triangle using the chaos game we need to describe the affine transformations that will describe the system. Using the most basic version of an affine transformation (which uses vector multiplication and vector addition), we can describe the system with the following 3 transformations:

$$A_0 = \begin{vmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{vmatrix} \; b_0 = \begin{vmatrix} 0 \\ 0 \end{vmatrix} \text{ selected with a probability of } \frac{1}{3}. \text{ (Pulls point towards Vertex A)}$$

$$A_1 = \begin{vmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{vmatrix} \; b_1 = \begin{vmatrix} \frac{1}{2} \\ 0 \end{vmatrix} \text{ selected with a probability of } \frac{1}{3}. \text{ (Pulls point towards Vertex B)}$$

$$A_2 = \begin{vmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{vmatrix} \; b_2 = \begin{vmatrix} 0 \\ \frac{1}{2} \end{vmatrix} \text{ selected with a probability of } \frac{1}{3}. \text{ (Pulls point towards Vertex C)}$$

Using the affine transformation matrix described previously we can equivalently write the transformations more succinctly as:

$$F_0 = \begin{vmatrix} \frac{1}{2} & 0 & 0 \\ 0 & \frac{1}{2} & 0 \\ 0 & 0 & 1 \end{vmatrix} \text{ selected with a probability of } \frac{1}{3}. \text{ (Pulls point towards Vertex A)}$$

$$F_1 = \begin{vmatrix} \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & \frac{1}{2} & 0 \\ 0 & 0 & 1 \end{vmatrix} \text{ selected with a probability of } \frac{1}{3}. \text{ (Pulls point towards Vertex B)}$$

$$F_2 = \begin{vmatrix} \frac{1}{2} & 0 & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{vmatrix} \text{ selected with a probability of } \frac{1}{3}. \text{ (Pulls point towards Vertex C)}$$

Each of these transformations pulls the current point halfway between one of the vertices of the triangle and the current point. F_0 performs scaling only. F_1 and F_2 perform scaling and translation.

We now begin the *chaos game*. We first select a random point on the biunit square. In this case we have pseudorandomly selected x = 0.40 and y = 0.20. We then pseudorandomly pick transformations. The first four transformations shown are F_0, F_2, F_1, and then F_0. The application of these are shown in Figure 3.3.

Notice how the next point is the midpoint between the vertex and current point. These mappings guarantee the convergence of the algorithm to the desired IFS. This process continues on with each point being plotted except for the initial 20 points that allow the system to settle. We have provided coloring for a visual representation of what transformation was responsible for each point. Points transformed by F_0 are labeled Green, F_1 are labeled Red, F_2 are labeled Blue. Iterations 1,000, 7,500, 15,000, and 25,000 are displayed in Figure 3.4.

The more one stochastically samples, the closer the output image is to the solution of the Iterated Function System being computed.

Figure 3.5 shows the procedure which results in the final system. This system resembles the Black Spleenwort fern [8]. This fern was not shown soley because it resembles a similar shape in nature but because of the explicit way the transforms were used to get the shape desired (which is often seen in the flame user community when creating intricate flames).

Figure 3.5: The formation of the Iterated Function System called Barnsley's Fern.

Below in Table 3.1 is an explanation of what each transformation conceptually does to produce the fern [5] [8].

Name of Transform	Conceptual Description
F_0	Maps to the base of the stem.
F_1	Maps inside the leaflet described by the red triangle in Figure 3.5.
F_2	Maps inside the leaflet described by the blue triangle in Figure 3.5.
F_3	Maps inside the leaflet represented by the blue triangle in Figure 3.5.

Table 3.1: Conceptual descriptions of each affine transformation of Barnsley's Fern.

3.3 Fractal Flame Algorithm

Differences from Classical Iterated Function System (IFS)

Fractal flames are a member of the Iterated Function System however differ from Classical Iterated Function Systems in three major respects [2]:

1. Instead of affine transformations presented in the previous section non-linear functions are used.
2. Log-density display is used instead of linear or binary.
3. Structural Coloring

On top of the core differences, additional pyschovisual techniques such as spatial filtering and temporal filtering (motion blur) give rise to more aesthetically pleasing images with the illusion of motion.

History

The flame algorithm was created in 1992. The algorithm was created by Scott Draves who is software and visual artist. Shortly after the creation of the algorithm the first implementation called `flame3` was made openly available in 1992. Drave's fractal flame software has allowed the process for artist creation by allowing the users to experiment with shapes, colors, and stylistic effects. More historical background can read about in Section 4.1.

Algorithm

Outline

The details of the algorithm as well as a detailed flow chart of the algorithm will be described in this section but will spare full-scale explanations for a specific reason: these will be saved for their own respective chapter in which we review each different concept of the algorithm and provide the existing implementation and then present the improved implementation. We do this merely to partition the large sections of the paper and to bring attention to the relevant new approaches that will be described.

The following will give a coherent understanding of the algorithm minus some of the implementation details.

Transforms

Unlike the classical IFS examples presented previously which apply one transformation to a set of points, the fractal flame applies multiple transformations. These transformations can be non-linear unlike their classical IFS counterparts. Additionally not all mappings are contraction mappings [5] however the whole system is contractive on average. There are some fractal flame systems which are degenerate and are not contractive; however, these are of no interest to us.

The multiple variations as well as their order of application on the initial point choosen at random are described below:

1. AFFINE TRANSFORMATION

The affine transformation we will be working with for the flame algorithm is of the form:

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

Again, this transformation makes it possible to provide rotation, scaling, and shear to the points. The information that is represented in this form is both space (x and y coordinates) as well as color which is explained in Section 3.3.

2. VARIATION

To provide the complex realm of shapes the algorithm can produce we introduce a non-linear functions called a variation.

The variation is applied to the affine transformed point resulting in the transformation being of this form:

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

Furthermore, multiple variations can be applied to an affine transformed point. Each point also is multiplied by a blending coefficient named v_{ij} which controls the intensity of the variation being applied. The expanded formula is the following:

$$F_i(x, y) = \sum_j v_{ij} V_j(a_i x + b_i y + c_i, d_i x + e_i y + f_i)$$

By applying variations, the resulting plane is changed in a particular way. Fundamentally there are 3 different types of variations in which can be applied. Variations are either simple remappings, dependent variations, or parametric variations.

SIMPLE REMAPPINGS: A simple remapping is one such that it simply remaps the plane. This could for example be remapping of the cartesian coordinate system plane to a polar coordinate system plane or some kind of sinusoidal plane.

DEPENDENT VARIATIONS: A dependent variation is a remapping of the plane such that the mapping is a simple remapping but additionally controlled by coefficients that are dependent on the affine transformation being applied.

PARAMETRIC VARIATIONS: A parametric variation is a remapping of the plane such that the mapping is a simple remapping but additionally controlled by coefficients that are independent of the affine transformation applied.

A baseline flame with purely an affine transform applied is shown side by side with both a simple remapping, dependent variation, and parametric variation in Figure 3.6. This will give you a good idea on what a single variation can do to shape the system and how intricate some of the variations can be. As an additional visual supplement please refer to the Appendix of the original Flame Algorithm Paper for an extensive collection of many catalogued variations [2].

3. POST TRANSFORMATION

After applying the variations which shape the characteristics of the system we apply what is known as a post transform which allows the coordinate system to be altered. This is done with another affine transformtion labeled Pi. By adding to our previous definition the definition for all of the collective transformations is:

$$F_i(x, y) = P_i(\sum_j v_{ij} V_j(a_i x + b_i y + c_i, d_i x + e_i y + f_i))$$

where P_i is equal to:

$$P_i(x, y) = (\alpha_i x + \beta_i y + \gamma_i . \delta_i x + \epsilon_i y + \varsigma_i)$$

4. FINAL TRANSFORMATION

Finally, because the image is eventually outputted to the user we apply the last transformation which is a non-linear camera transformation[3].

Log-Density Display of Plotted Points

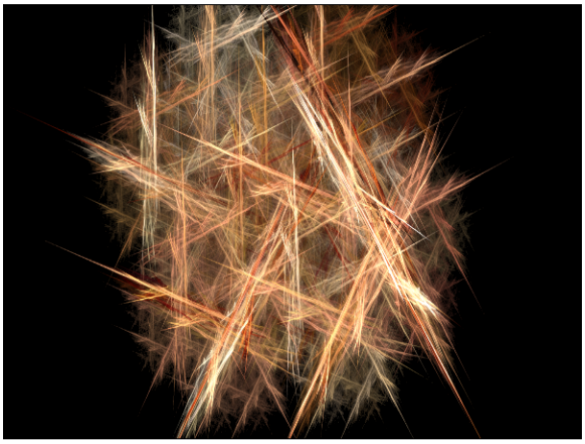
In the classical Iterated Function System, described previously, points were either members in the set or not. For every subsequent time the chaos game selected a point that was already shown to have membership in the set information was lost about the density of the points. To remedy this for the fractal flame algorithm we instead use a histogram for plotting the density of points in the chaos game. Given the density of points are now plotted onto the histogram we have several different methods we could go about plotting them into a resulting image which include:

1. BINARY MAPPING: As described before, this did result in the images we wished to produce but were not smooth and contained no shades of gray (black and white).
2. LINEAR MAPPING: A linear mapping of the histogram provides an improvement but the range of data is lost in the process. The linear mapping has problems differentiating large scales of range. For example, a point plotted 1 time, 50 times, and 5,000 times would be a great illustrative example. Compared a point of density 5,000 both point densities 1 and 50 appear to be of relatively same magnitude however there is a great different in them.
3. LOGARITHMIC MAPPING: This mapping proves to be superior to it's counterparts. The logarithmic function allows a great range of densities relationship to oneanother to be persered. This is the type of mapping the flame algorithm employs.

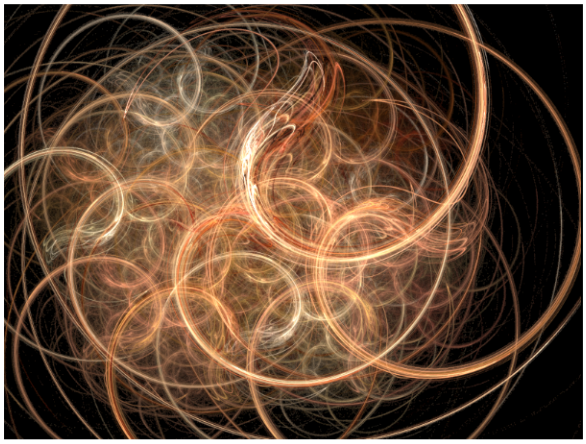
Visual representations of a flame using a binary, linear, as well as logarithmic mapping for the display can be seen in Figure 3 of Drave's original paper on the flame algorithm[2].

As a note to avoid confusion, the logarithmic mapping allows the image to now displayed in shades of grays and not as the vibrant colorful flames readily available to be viewed on flame gallery websites. Structural coloring, color correction and enhancement techniques, and tone mapping take care of these and are all seperate algorithmic processes.

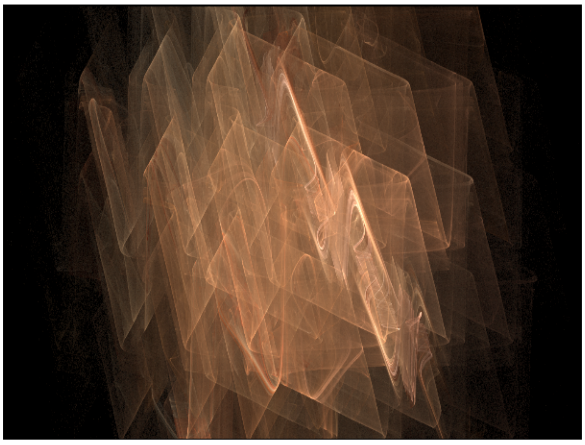
[3]This transformation isn't applied directly to the computational loop and is merely for visual output.



Flame without a variation applied,
shape is purely from an affine transform.



Flame with a simple remapping applied,
the remapping is the variation named 'Swirl'.



Flame with a dependent variation applied,
the dependent variation is the variation named 'Popcorn'
which is fully parameterized by the variables c and f.

Flame with a parametric variation applied,
the parametric variation is the variation named 'Julia Set'
which is fully parametrized by by the Julia Power and Julia Distance
variables.

Figure 3.6: 4 different variations applied to the same flame depicting the different types of variations and how they change the solutions characteristics.

Coloring (Tone Mapping)

Structural coloring is one of the elements that sets the flame algorithm apart from the classical iterated function system. Instead of mearly mapping grayscale (being the space from [0,1]) to a specific red, green, blue color space another approach is taken. A color mapping (presented in the previous sentence) is used however we further our definition of the affine transformation (F_i) and additionally include a color related to that transformation. After applying the transformation which looks like the following:

$$(x, y) = F_i(x, y)$$

We apply the color associated with the transformation. To do this we expand this transformation process and include the variable c which stands for the color (R,G,B) of that point. We average the current color with the color related to that transformation like so:

$$c = \frac{(c + c_i)}{2}$$

This has a major effect upon the color and allows the last applied transform to have the most significant effect. This application of affine transformation color helps structural coloring emerge in a similar way to how colors were applied to each transform in the Sierpinski's Triangle example. Additionally, a final transform also has a color associated with it. The final transformation (non-linear camera transformation) of the (x,y) point is inn the form of:

$$(x_f, y_f) = F_f(x, y)$$

Afterwards we also average the current color with the color related to that transformation:

$$c_f = \frac{c + c_f}{2}$$

Furthermore, when peforming log-density display we run into issues if we are only keeping information about the RGB values associated with each point. By logarithmically scaling each color channel we do not get the desired results. For more information on why please see Section 8.2 on brightness and that red , green, and blue wavelengths are not treated equal. The fractal flame algorithm remedies this by using RGB and also an additional variable called α which is the transparency value. This value is accumulated and scalled by $\frac{\log \alpha}{\alpha}$ at the end of the chaos game.

Gamma Correction and Company

Now that our flame is colored the process is complete, right? Wrong. Many complications still are not yet resolved. The next being the concept of gamma correction. To correctly display the flame image on a lower dynamic range (such as an LCD or CRT monitor or printer) we need to map our high dynamic range to the lower dynamic range. A full discussion of this topic can be seen in Section 8.2 and Section 8.2.

Additional color correction techniques can be applied to the flame. A full survey of what kind of color correction techniques are available and what kind of benefit they provide are mentioned in Section 8.5.

Symmetry

The fractal flame algorithm inherently supports the concept of self-similarity but also supports the concept of *symmetry* of two kinds:

- Rotational
- Dihedral

This added symmetry adds a new level of intricacy to the resulting flame. Symmetry is thought to be congenitally attractive to the human eye [9]. A description of how symmetry is added to the flame algorithm is as follows.

ROTATIONAL SYMMETRY is introduced by adding extra rotational transformations. To produce n-way symmetry you are essentially implying that you wish to have $\frac{360^\circ}{n}$ degrees symmetry. The set of transformations transformations necessary to add $\frac{360^\circ}{n}$ symmetry is:

$$\text{Rotational Transforms}_i = \left(\frac{360^\circ}{n} \times i \,|i = 1, 2, .., n \right) \text{ where n = number of way symmetry}$$

For example, To produce six-way symmetry the following 5 transformations would be needed:

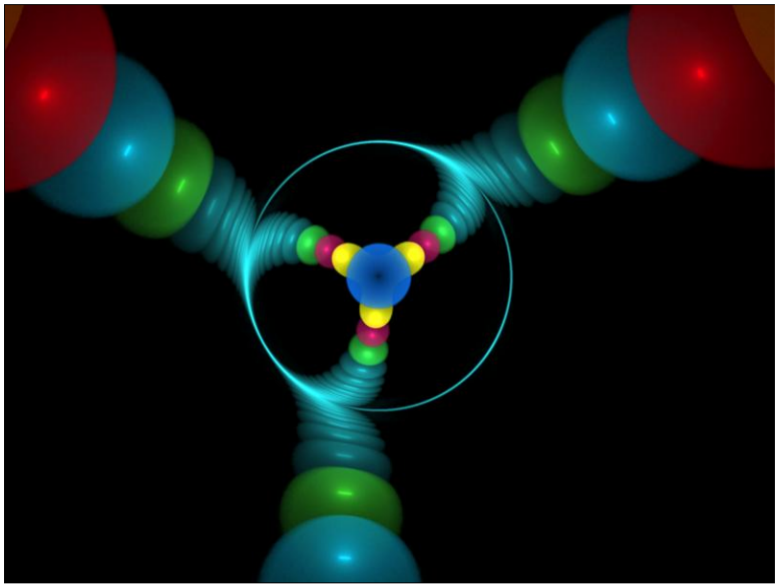
- $\text{Rotational Transforms}_1 = 60^\circ$
- $\text{Rotational Transforms}_2 = 120^\circ$
- $\text{Rotational Transforms}_3 = 180^\circ$
- $\text{Rotational Transforms}_4 = 240^\circ$
- $\text{Rotational Transforms}_5 = 300^\circ$

Each transformation is given an equal weighting, allowing the chaos game to realize the n-way symmetry the more it stochastically samples.

DIHEDRAL SYMMETRY is introduced by adding a function that inverts the x-coordinate or y-coordinate. This is a reflection of the axis. An equal weighting is given to this reflection function which allows the chaos game to realize the dihedral symmetry.

Both rotational and dihedral symmetry are shown in Figure 3.7.

The DIHEDRAL SYMMETRY and ROTATIONAL SYMMETRY are applied at the same step as the affine, variation, post, and final transformations. This is because the implementation of symmetry is defined in the form of a transformation and therefore is the most logical place to apply it.



Flame exhibiting 3-way rotational symmetry.

Flame exhibiting dihedral symmetry.

Figure 3.7: A visual depiction of what dihedral and rotational symmetry look like in a flame.

3.4 Filtering

After performing all aforementioned steps there is still several issues which still afflict our flame. Two of these are both noise and aliasing.

Aliasing is a common issue and occurs when a high resolution graphic maps to a lower resolution graphic. The result is that smooth edges or gradients are not represented correctly. To combat aliasing flame uses a method called supersampling. More information about aliasing and supersampling can be found in 9.1 and 9.1 respectively.

Noise in a flame occurs because of the stochastic nature of the iterated function system. While plotting the flame some seemingly random points may occur in our set. Supersampling the image takes care of the alias issues but does not take care of our noise issues. In order to correctly "blur" only noisy parts of the image we must blur selective regions of the image. In the case of noise, the flame algorithm performs a form of density estimation to address this image. More information on this can be found in Section 9.2.

The importance of both steps are paramount to providing an aethestically pleasing image as aliasing and noise are extremely noticable to the eye and can render even the most beautiful flame, atrocious.

A more in depth look at filtering can be found in Section 9. This section cover ant-aliasing methods, filtering methods, and more information on the `flame3` specific approach.

Motion Blur

Finally, we address one of the last issues. We have taken care of spatial aliasing but when the multiple images of flames 'in motion' are outputted we experience a new form of aliasing: temporal aliasing. Temporal aliasing can not be addressed correctly mearly by supersampling and one implementation that the flame algorithm uses is by using an extra buffer. The first buffer accumulates the histogram of points in a linear fashion. The second buffer accumulates logarithmitcally filtered histograms of each temporal sample from buffer one. At the end, the second buffer is filtered and presented.

Procedure

Initially presenting the fractal algorithm usually results in a lengthy discussion as seen above but is usually done at the sake of clarity of *why* and *how* each step is being done. Since these have already been explained, we recap the algorithm with a high level summary[4] in the same fashion we had provided the classical iterated function system algorithm. A flowchart diagram of the procedure can be found in Figures 3.8 through 3.10

The procedure is as follows: First, a random point is picked on the biunit square. The user picks the quality of the flame they wish to render and the program enters into a loop for Q iterations (where Q is quality). At each iteration a transformation is applied based on a probabilistic weighting similar to Barnsley's Fern in Section 3.2. This transformation will apply an affine transformation (which applies scaling, rotation, sheer, and translation), a variation transformation (to change the characteristics of the point), and a post function (to change the coordinate system). When the variation transform is applied a color associated with it is also applied which is explained in Section 3.3. After the new point and color are selected this vector is accumulated in it's respective histogram bin representing the density of each point. The points are not accumulated in the bins for the first 20 points in order to allow the system to settle.

This process happens until the final iteration. On the final iteration, many final processing steps happen. First, the histogram bins of point densities are log scaled. Next, filtering is performed. Supersampling

[4]For simplicty's sake we ignore the effects that Early Clip (Section 8.5) and Highlight Power (Section 8.5) have upon the algorithm and the way they reorder or modifications of core steps.

removes the aliases. Density Estimation allows the reduction of noise. As an added note, motion blur can also occur next and requires a second buffer of points to be filtered down. The final transform is now applied which is a non-linear camera transform. This final transform also applies a color associated with it. Now that the correctly filtered, log scaled and colored image has been produced color correction is now applied. This provides hue, brightness, gamma, and other corrections. The image is then written to a file and the procedure ends.

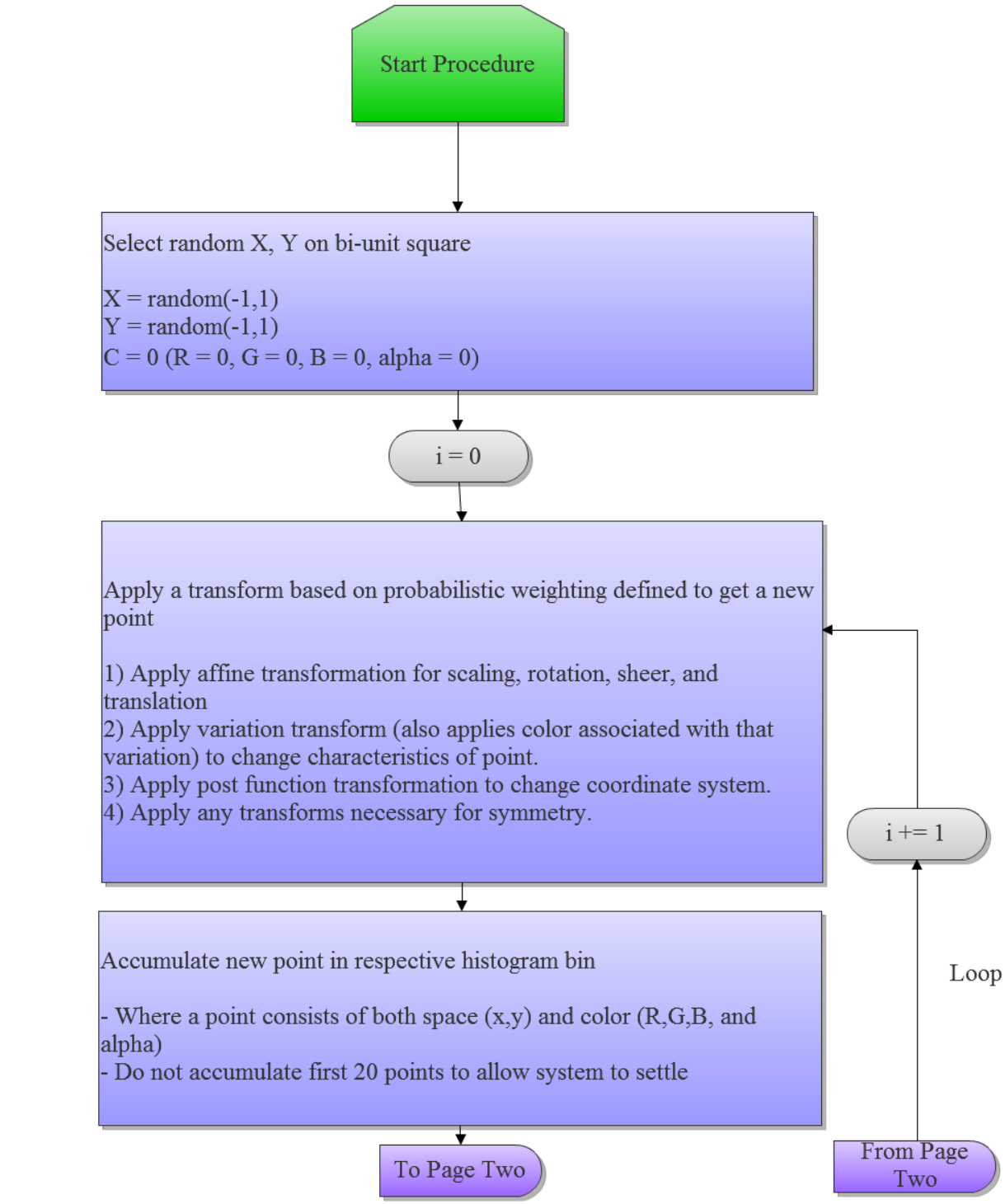


Figure 3.8: A flowchart describing the fractal flame algorithm.

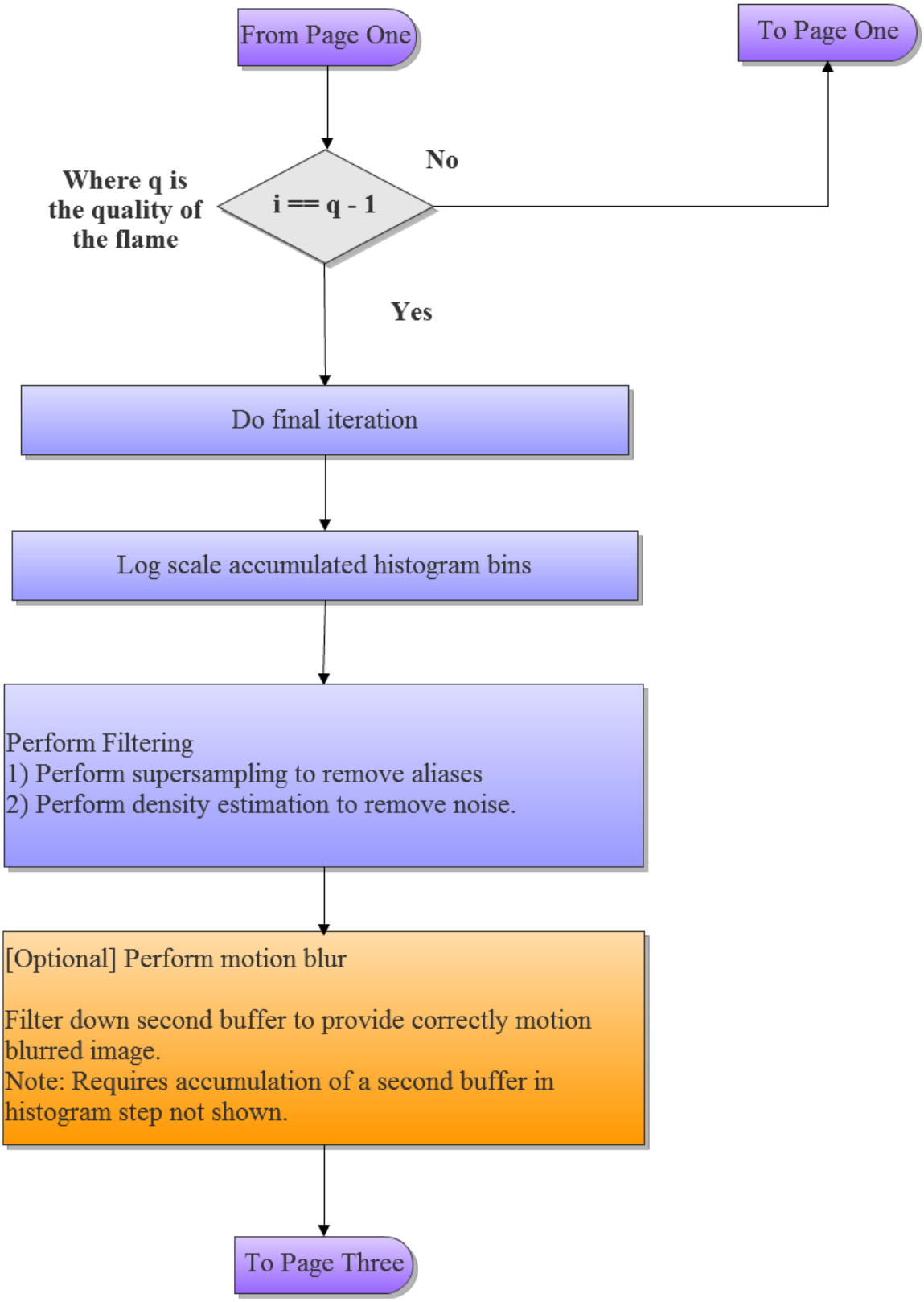


Figure 3.9: A flowchart describing the fractal flame algorithm.

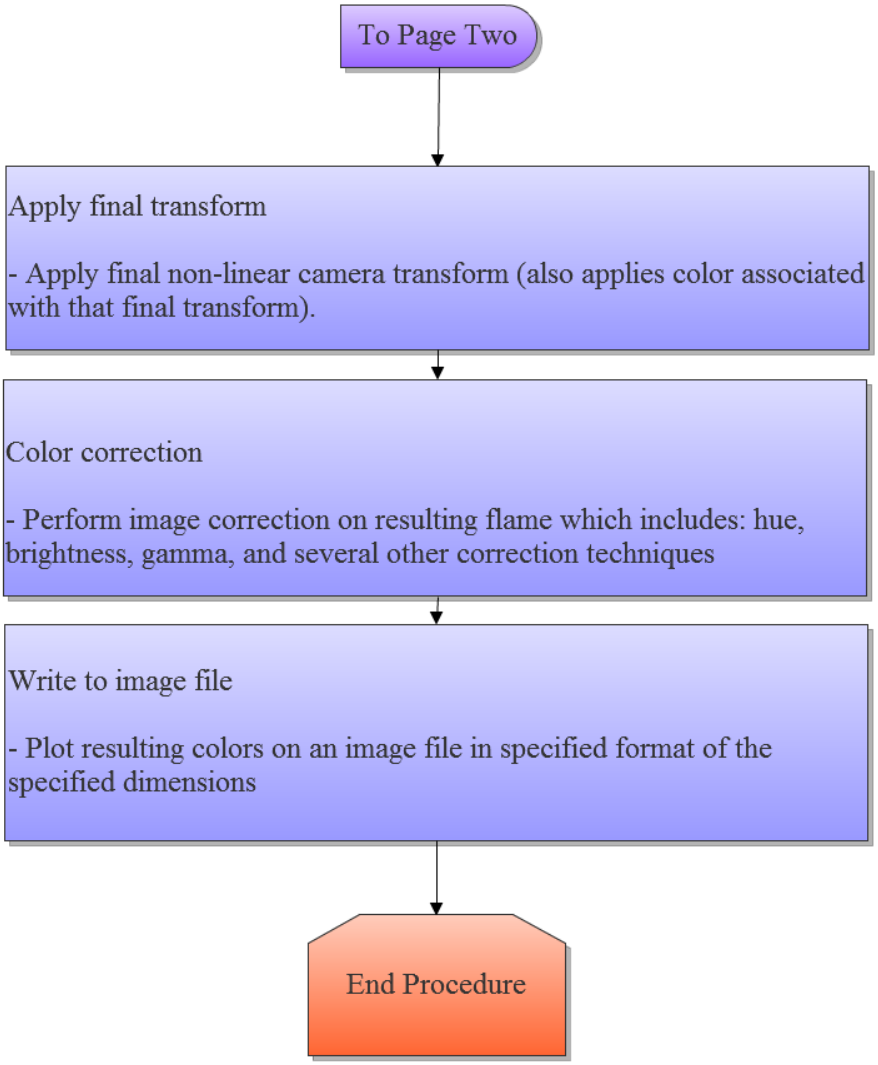


Figure 3.10: A flowchart describing the fractal flame algorithm.

CHAPTER 4

EXISTING IMPLEMENTATIONS

The fractal flame algorithm is relatively old, but unlike most antiquated image synthesis techniques, its output is still considered to be visually appealing today. As might be imagined for such a classic algorithm, there are several implementations available; a few even target GPUs. To ensure that our implementation provides a benefit, we must consider the strengths and weaknesses of each implementation, and carefully target our renderer to fill these gaps.

To that end, a brief survey of each publicly-available implementation is below.

4.1 `flam3`

Considered by most to be the "reference implementation" of the flame algorithm, `flam3` [10] was created in 1991 by Scott Draves, the creator of the fractal flame algorithm. In TK, Erik Reckase took over development, and continues to add features and release updates to this day.

Because of `flam3`'s status as a reference implementation, each new version is regression-tested against the output of previous versions to ensure it can still produce (nearly-)identical images. To retain this property while still accomodating new features, the code now includes a dizzying array of parameters, flags, and downright hacks. This makes it difficult to optimize and experiment with.

Since the Electric Sheep project uses `flam3` to produce all its images, however, scrapping this mess is not an immediate option. The Electric Sheep screensaver obtains its content from pre-rendered video sequences, and until an implementation fast enough to re-render the entire back catalogue of pre-rendered flames from scratch at sufficient quality is produced, backwards compatibility is needed to guarantee seamless transitions.

An implementation that could produce `flam3`-compatible images at high speeds would therefore be useful to the Electric Sheep project.

4.2 Apophysis

Apophysis [11] is an aging application for the interactive design of flames, and is one of the most popular tools to do so. Apophysis includes its own rendering backend, which has proved to be somewhat easier to modify than `flam3`; as a consequence, many variations now included in `flam3` started out as community experimentation within Apophysis, and more are yet being considered.

The Apophysis renderer lacks some of `flam3`'s newer visual-quality-oriented features, so while it remains a viable choice for users and interesting to watch, there is no particular need to fully support it.

4.3 flam4

One of the more complete implementations of the flame algorithm for GPUs, flam4 [12] nevertheless sits in an uncomfortable position in terms of its output: the implementation suffers from compromises necessary to allow reasonable performance on the GPU, reducing its perceptual output quality, yet it is not fast enough to render images for display on the fly. Since CPUs are fast enough to deliver offline renders at normal resolutions in reasonable size, there is little need for acceleration for the mid-range renders, as a bit of patience can usually accomodate most use cases.

Since `flam4` provides good acceleration at moderate loss of quality, we should not attempt to do the same. A novel implementation should target either acceleration without loss of quality, or fully real-time performance at an acceptable quality.

4.4 Fractron 9000

Fractron 9000 [13] is another accelerated renderer loosely based around the fractal flame algorithm. The software employs the same basic principles as `flam3` — that is, log-density accumulation of IFS samples, with nonlinear transform functions — but makes no effort to produce results that are compatible with the original software. It is also written against the Microsoft .NET framework, and is therefore not suitable for headless use.

4.5 Chaotica

Chaotica [14] is the only closed-source implementation of the flame algorithm known to the authors. The software's stated design goal is to produce images of superior visual quality to `flam3` in less time, which it does. Thomas Ludwig, Chaotica's author, is also a developer of Indigo Renderer, a professional ray-tracing application. Many performance and quality techniques employed in the field of ray-tracing are applicable to the rendering of fractal flames, and it is likely that newer advances in the field are being used in Chaotica. This is speculation, however, as the developer has been notoriously quiet about the nature of his improvements.

4.6 Our implementation

Given that no accelerated renderer explicitly targets `flam3` compatibility, despite the desire among the community for such a tool, it seems prudent to pursue that subfield of image compatibility. In addition to being able to compare images directly against the output of a CPU renderer, which simplifies testing, such a renderer would lower the operating cost of the Electric Sheep project and see widespread adoption as part of that software.

CHAPTER 5

A (NOT-SO-)BRIEF TOUR OF GPU COMPUTING

Graphics processing units began as simple, fixed-function add-in cards, but they didn't stay there. Over many generations, demand for increasingly sophisticated computer graphics required hardware that was not just faster, but more flexible; device manufacturers responded by spending ever-larger portions of the transistor budget on programmable functions. In 2007, NVIDIA released the first version of the CUDA toolkit, unlocking GPUs for straightforward use outside of the traditional graphics pipeline. Since then, "general-purpose GPU computing" has become a viable, if still nascent, market, with practical applications spanning the range from comsumers to the enterprise.

Don't let the words *general purpose* fool you, however. While the major manufacturers have shown interest in this market, it remains at present a fraction of the size of these companies' core markets [15]. Every transistor spent making GPGPU faster and easier to program may come at the expense of doing the same for games. Because the market for high-performance computing remains much smaller than for entertainment and professional imaging, GPUs remain primarily graphics-oriented.

Despite the "games first" approach which informs hardware designers, GPUs still provide the highest performance per dollar for most math-intensive applications. In general, porting algorithms to such devices can be a challenge, but for algorithms that fit naturally (or can be made to fit) into the computing paradigms available on current hardware, the performance benefit justifies the effort.

This section is intended to give a grounding in the concepts and implementations of GPU computing platforms. The OpenCL computing model subsets both NVIDIA's and AMD's hardware, and therefore forms a convenient location to start the discussion. While OpenCL mandates certain hardware features, many others appear across several GPUs as a consequence of their shared heritage; these features are also addressed. Finally, an in-depth analysis of certain unique hardware features on both NVIDIA's and AMD's flagship architectures provides background information that is built upon in later chapters.

5.1 OpenCL

The OpenCL standard for heterogeneous computing is managed by the Khronos Group, an industry consortium of media companies that also produces the OpenGL specification [16]. OpenCL provides a cross-platform approach to programming; while its execution model requires certain features of the hardware it executes on, the language is kept general enough so that almost all code can execute with reasonable efficiency on any supported architecture (via driver-provided just-in-time compiling).

Because it forms a common, abstract subset of the GPUs under consideration as platforms on which to implement this algorithm, OpenCL is a good starting place for our discussion. As much as we might like to

rely on an open standard alone to inform our algorithm design, however, the specification doesn't tell the whole story.

An editorialized history of the standard

OpenCL was developed by Apple, Inc. to provide a generic interface to high-performance devices like GPUs across their platform. At the time of development, Apple had standardized on NVIDIA GPUs across its desktops and laptops, and wished to expose the hardware's computational performance to developers, but did not wish to lock itself into NVIDIA's proprietary CUDA technology and in so doing weaken the threat of using AMD graphics products at the negotiating table.

While cooperation on standards had clearly served the two graphics firms in the past (with DirectX and OpenGL), NVIDIA's cards were far more flexible for computing than AMD's; any standard which would work seamlessly across cards would cripple NVIDIA's performance advantage. Naturally, Big Green wasn't keen on signing on to a standard that would necessarily eliminate its considerable head start in the compute market. But Apple provided leverage — ruthlessly, if history is any judge — and months later AMD and NVIDIA were showing off their new standard for compute together.

AMD was, at the time, shipping cards based on the much-derided R600 architecture, which did not meet even the limited requirements of OpenCL. While the company was preparing to include the necessary components in their next graphics architecture, full support did not emerge until two hardware generations later, with the competitive *Evergreen* family of GPUs.

On the heels of a very successful graphics architecture which was compatible with OpenCL from day one, NVIDIA invested even more engineering talent and die space into the *Tesla* architecture, which preceded AMD's Evergreen. Tesla formed an even broader super-set of features available in the base OpenCL spec, some which were simply inaccessable from the open standard. Rather than let these features go to waste, however, NVIDIA put them to work in their proprietary CUDA framework, which remains their primary development and marketing focus.

As of now, OpenCL is still at version 1.1, which (along with an extension or two) covers the functionality in AMD's *Northern Islands* family, their latest. NVIDIA's *Fermi* architecture provides yet another increase in compute features over OpenCL, and those features are again exposed through CUDA. We'll take a look at this situation a bit later; for now, let's turn to the OpenCL model for computation.

How to do math in OpenCL

OpenCL has something of a client/server model: a program running on an OpenCL *host* communicates with one or more *devices* through the OpenCL API. While the method of communication with the device is not fully specified, both NVIDIA and ATI post requests to a command queue on the device. The GPUs possess their own schedulers and DMA engines; after a command is handed to a device, the host is left to do little but poll for the task's completion.

By default, commands begin executing on the device as soon as the appropriate execution resources are available. Stricter ordering is possible; a *command-queue barrier* will stall until all previous commands in the queue are complete. A *stream*[1] provides a strict ordering for every task it contains, but multiple streams can execute concurrently.

There is no hardware mechanism for a strict interleaved ordering of both host and device code. The OpenCL API exposes apparently-synchronous execution of device commands in the host API, but this is implemented

[1] We borrow the CUDA notation here. OpenCL allows any command to wait on any other explicitly using *events*, which can be used to implement a stream, but has no term (or API call) for the ordered tasks as a group. It becomes a pain to talk about without a name.

via a spinloop which polls the device for task completion. This method is ineffecient and should be avoided in performance-critical code.

Hosts and devices must be assumed to have independent memory spaces. To provide data for execution, data must be explicitly copied to and from the device via an OpenCL API call. Memory operations are contained within commands, and are subject to the ordering constraints above; additionally, since memory commands are executed by the GPU's DMA engine, the host-side memory to be accessed may need to be page-locked to ensure that it is resident when accessed and that its location in physical RAM does not change. OpenCL devices may optionally support mapping a portion of device memory into the host's address space or vice versa, although such access is generally slower than bulk updates.

After the host has initialized the device's memory space, it may load a *kernel* onto the device. The kernel is a fixed bundle of device code and metadata, including at least one entry point for program execution. From the OpenCL API, the kernel's data is opaque on both host and device, so device-side run-time code modification is prohibited. After uploading a kernel, the host issues a command which sets up arguments for an entry point and begins executing it in one or more *device threads*[2].

As in a typical OS, a device thread is a set of data and state registers, including a program counter indicating the thread's position within the currently-loaded code segment. A thread can execute arithmetic instructions and store the result to its registers, perform memory loads and stores, and perform conditional direct branching to implement loops. However, OpenCL does not support a stack; all function calls must be inlined, and recursion is not allowed.

While a single thread executes instructions according to program flow, the order of execution between any two threads is generally undefined. It's possible to use global memory to do a limited amount of manual synchronization, but this is impractical, as global memory accesses typically carry high latencies, suffer from bandwidth constraints, have an undefined ordering, and heavily penalize multiple writes to the same location [17].

To facilitate inter-thread cooperation without mandating globally-consistent local caches, threads are collected into *work-groups*. A work-group is a 1-, 2-, or 3-dimensional grid of threads that share two important consistency features: a fast, small chunk of *shared memory*[3] accessible only to threads within that work-group, and *barrier instructions*, which stall execution of any thread that executes the instruction until every thread in the work-group has done so.

Work-groups themselves are arranged in a uniform grid of dimensionality ≤3. Every thread in a grid must execute the same kernel entry point with the same parameters. To obtain thread-specific parameters, each thread can access its index within its work-group (its *local thread ID*), as well as its work-group's index within its grid (the *global thread ID*); it may then use those IDs to load thread-specific parameters such as a random seed or an element of a matrix. This is the only means to differentiate between threads at their invocation. Aside from providing a global ID, the only feature provided by a grid of work-groups is the requirement that every thread terminate before the grid is reported as complete to the host.

In addition to global and shared memory, OpenCL also provides *private memory*, which is accessible only to a thread; *constant memory*, which has a fast local cache but can only be modified by the host; and *image memory*, which can only be accessed using texture samplers. The texturing pathway, a clear holdover from OpenCL's GPU origins, is a high-bandwidth but high-latency method of accessing memory which can only perform lookups of 4-vectors but offers a read-only cache and essentially free address generation and linear interpolation.

[2] We revert again to CUDA's terminology; this time, though, merely because "work-unit" is just a clumsy, unnecessary neologism.

[3] Another CUDA term. OpenCL calls this "local memory". Problem is, CUDA uses the term "local memory" to refer to what OpenCL calls "private memory". We choose the unambiguous name in both cases.

5.2 Common implementation strategies

The OpenCL standard was constructed to subset GPU behavior at the time of its ratification, but for portability reasons it omits implementation details even when techniques were used in both NVIDIA and AMD GPUs. While such details do not necessarily impact code correctness, they can have a considerable impact on the ultimate performance of an application.

Dropping the front-end

In modern x86 processors, only a small portion of the chip is used to perform an operation; more die space and power is spent predicting, decoding, and queueing an instruction than is spent actually executing it. This seems contradictory, but it is in fact well-suited to the workloads an x86 processor is typically used for. It's also a consequence of the instruction set; x86's long history and ever-growing set of extensions has made translation from machine code to uops a challenging and performance-critical part of a competent implementation.

Across the semiconductor industry, it has become clear that scaling clock speed alone is not a realistic way to acheive generational performance gains. To deliver the speed needed by graphics applications, both NVIDIA and AMD simply pack hundreds of ALUs into each chip. To avoid the gargantuan power draws associated with including a full x86-style front-end, the two hardware companies employ three important tricks.

The first of these is runtime compilation. In OpenCL, device kernels are stored in the C-like language which executes on the device, and are only compiled to machine code via an API call made while the program is running on the host; CUDA stores programs in an intermediate language, but the principle is similar. In both cases, this pushes the responsibility for retaining backward compatibility from the ALU frontend (where it would be an issue billions of times per second) to the driver (where it matters only once per program). Without needing to handle compatibility in hardware, the actual instructions sent to the device can be tuned for each hardware generation, reducing instruction decode from millions of gates to thousands.

Another considerable saving comes from dropping the branch predictor. On an x86 CPU, the branch predictor enables pipelining and prefetch, and a mispredict is costly. To axe the branch predictor without murdering performance, GPU architectures include features which allow the compiler to avoid branches. Chief among these is predication: nearly every operation can be selectively enabled or disabled according to the results of a per-thread status register, typically set using a prior comparison instruction. For many expressions, using the results of a predicate to disable writeback can be less costly than forcing a pipeline flush, especially when hardware and power savings are taken into account. Drivers also generally inline every function call; with thousands of active threads and hundreds of ALUs all running the same code, a single large instruction cache becomes less expensive than the hardware needed to make function calls fast. Perhaps most intuitively, both companies go out of their way to inform developers that branches are costly and should be avoided whenever possible.

The final technique used to save resources on the front-ends is simply to share them. A single GPU front-end will dispatch the same instruction to many ALUs and register files simultaneously, effectively vectorizing individual threads into an unit between a thread and a work-group. NVIDIA calls these units *warps*[4], with a vectorization width of 32 threads; AMD uses *wave-fronts* of 64 threads. Because each thread retains its own register file, this kind of vectorization is not affected by serial dependencies in a single thread. In fact, the only condition in which it is not possible to vectorize code automatically in this fashion is when threads in the same warp branch to different targets, whereupon they are said to be *divergent*. Not coincidentally, the same approaches used to avoid branches in general also help to avoid thread divergence.

[4] We follow what is now a tradition and adopt NVIDIA's term, though it does display a bit of whimsy on the part of Big Green.

While these techniques seem of only passing interest, the peculiarities of the fractal flame algorithm are such that a naïve implementation which did not heed these design parameters would suffer more than might be expected. We will need to make careful use of runtime compilation, predicated execution, and warp vectorization to write an efficient implementation.

Memory coalescing

The execution units aren't the only part of a GPU trading granularity for performance; memory accesses are also subject to a different kind of vectorization, called *coalescing*, that has extremely visible consequences for certain classes of tasks.

High-performance GPUs contain several front-ends. Because global memory is accessible from all front-ends, there is effectively a single, shared global memory controller which handles all global memory transactions[5]. Since each memory transaction must interact with this memory controller, and multiple front-ends can issue transactions simultaneously, this controller includes a transaction queue and arbitration facilities, as well as simplified ALUs for performing atomic operations.

To simplify and accelerate the memory controller, memory transactions must be aligned to certain bounds, and may only be 32, 64, 128, or 256 bytes wide (depending on architecture). Because of unavoidable minimums on address set-up time and burst width, GDDR5 devices can only attain rated performance with transaction widths above a certain threshold, and these minimums are reflected in the minimum transaction sizes on the other side of the memory controller.

A single thread can issue at most a 16-byte transaction (while reading a 4-vector of 32-bit values), and will more often simply use 4-byte transactions in typical code. On its own, this would result in most of each transaction being discarded, consuming bandwidth and generating waste heat. On traditional CPUs (and, to a limited extent, newer GPUs), caches are used to mask this effect. However, with so many front-ends on a chip, placing a large and coherent cache near each would be prohibitively expensive with current manufacturing processes, and even centrally-located caches would still require an enormously high bandwidth on-chip network to service a request from every running thread.

Since GPUs must issue wide transactions to reduce chip traffic and accomodate DDR latency, and temporal coherence is not enough to mitigate the memory demands of thousands of threads, hardware makers have instead turned to *spatial* coherence. As threads in a warp execute a memory instruction, the local load/store units compare the addresses for each thread. All transactions meeting certain criteria — falling within an aligned 128-byte window, for example — are coalesced into a single transaction before being dispatched to the memory controller.

On previous-generation architectures, use of coalescing was critical for good memory performance, with uncoalesced transactions receiving a penalty of an order of magnitude or more. Respecting coalescing is an easy task for some problem domains, such as horizontal image filtering. Others required the use of shared memory: segments of the data set would be read in a coalesced fashion, operated on locally, and written back. Unfortunately, the fractal flame algorithm supports neither of these modes of operation, and there is no way to create a direct implementation with sufficient performance on these devices.

Newer GPU architectures, such as NVIDIA's Fermi and AMD's Cayman, possess some caching facilities for global memory. The cache on these devices assists greatly in creating a high-performance implementation of the fractal flame algorithm, but remain far smaller than the framebuffer size at our target resolution. It is therefore clear that memory access patterns will be an important focus of our design efforts.

[5] Actually, there are typically several memory controllers connected by a crossbar switch, ring bus, or even internal packet bus, with address interleaving on the lower bits and any cache distributed per-core. But since each address block maps uniquely to one core, and typical access patterns hit all cores evenly, we ignore this.

Latency masking

Memory transactions, even when coalesced, can take hundreds of cycles to complete. Branching without prediction requires a full pipeline flush, as do serially-dependent data operations without register forwarding (another missing feature). Even register file access carries latency at GPU clock speeds. Without the complicated front-ends of typical CPUs, how do GPUs keep their ALUs in action?

The strategy employed by both AMD and NVIDIA is to interleave instructions from different threads to each ALU. In doing so, nearly every other resource can be pipelined or partitioned as needed to meet the chip's desired clockspeed. This technique increases the runtime of a single thread in proportion to the number of active threads, but results in a higher overall throughput. The mechanism for performing this interleaving differs between the two chipmakers, and is one of the more significant ways these architectures differ.

5.3 Closer look: NVIDIA Fermi

Fermi is NVIDIA's latest architecture, as implemented in the GeForce 400 and 500 series GPUs. The architecture represents a considerable retooling of the company's successful Tesla GPUs with a focus on increasing the set of programs that can be run efficiently rather than just on raw performance. This was done by adding some decidedly CPU-like features to the chip, including a globally-consistent L2D cache, 64KB of combined L1D and shared memory per core, unified virtual addressing, stack-based operations for recursive calls and unwinding, and double-precision support at twice the ops-per-clock rate of other GPUs.

As might be imagined, the chip was months late, and only made it out the door with reduced clocks and terrible yields. TSMC's problems at the 40nm node was partly responsible for the troubled chip's delay, but the impressive single-generation jump in the card's GPGPU feature set also had a hand. NVIDIA architects were not ignorant of this risk, but judged it a worthwhie one; an uncharacteristic move from a graphics company. What pushed NVIDIA to focus so much on compute?

In a word, Intel. Larrabee, the larger company's skunkworks project to develop stripped-down x86 CPUs with GPU-like vector extensions had the potential to grind away NVIDIA's enterprise compute abilities, not because of Larrabee's raw performance but because of its partial compatibility with legacy x86 code [18]. These chips would be far more power-hungry than any GPU, but NVIDIA felt backward compatibility and a simplified learning curve would woo developers away from CUDA, leaving them a niche vendor in the enterprise compute market. Worse, Sandy Bridge, the new CPU architecture, was to include a GPU on-die, potentially cutting out NVIDIA's largest-volume market segments. NVIDIA's response was to invest in Tegra, their mobile platform, and to make the first Fermi devices in the GTX 400 series an enterprise-oriented, feature-laden, unmanufacturable mess [15].

As it turns out, Larrabee was all but cancelled, Sandy Bridge graphical performace is decidedly lackluster, and TSMC got their 40nm process straightened out, leaving NVIDIA room to prepare the GF110 and GF114 architectures powering the GTX 500 series. These chips are almost identical to their respective first-generation Fermi counterparts at the system design level; tuning at the transistor level, however, greatly improved yield and power consumption, making these devices graphically competitive at their price level [19].

Shader multiprocessors

Each core[6] in GF110 contains a 128KB register file, two sets of 16 ALUs, one set of 16 load/store units, and a single set of 4 special function units. It also contains two warp schedulers, assigned to handle even- and

[6] NVIDIA actually refers to the smallest unit of independent execution as a shader multiprocessor. This is an example of what industry observers refer to as *absolute bollocks*. As with most GPGPU developers, we use traditional terminology.

odd-numbered warps, respectively. This area is partitioned so that the ALUs, SFUs, memory, and register file run at twice the rate of the warp schedulers and other frontend components. We refer to the clock driving the ALUs as the "hot clock", and likewise the "cold clock" for the rest of the chip.

Every thread in a warp executes together. At each cold clock, a warp's instructions are loaded by the scheduler and issued to the appropriate group of units for execution. Normal execution for all 32 of a warp's threads takes a single cold clock, followed by result writeback. This process is pipelined; it takes 11 cold cycles for a register written in a previous instruction to become available.

As mentioned previously, there is no register forwarding during pipelined instructions. In fact, every thread sees this delay between one instruction and the next, regardless of data dependencies. On NVIDIA architectures, this is hidden by cycling through all warps which are resident on the core and executing one instruction from each before returning them to the queue. This is done independently for each warp scheduler.

The SFUs, which handle transcendental functions (`sin`, `sqrt`) and possibly interpolation, are limited in number. When dispatching an instruction to that unit, it takes 8 hot clocks to cycle through all 32 threads of a warp. This stalls one warp scheduler for that duration, but doesn't interfere with the other; if the current thread in the other scheduler is waiting on access to that hardware, an NVIDIA-specific hardware component called the "scoreboard" marks the thread as unready and skips it until the required transactions complete.

This same scoreboarding approach handles the highly variable latency of memory instructions. Each load/store operation appears to take a single instruction to execute, wherein the resulting transaction is posted to a queue; when the result is returned, another cold cycle is spent in the load/store units to move the result from cache to the register pipeline. Some memory transactions, including L1D cache hits and conflict-free shared memory access, appear to complete in a single cold cycle.

Using thread-swapping is an elegant and simple way to hide latency, but it has an important drawback: the only way to avoid a stall is to always have a warp ready to run. Register file latency puts a hard lower bound on the number of threads required to reach theoretical performance, but memory access patterns can easily raise that number. Each of those threads, however, must contend for a limited register file, shared memory space, and bandwidth. Finding the right configuration to maximize occupancy without losing performance from offloading registers to private memory will be a theoretical and experimental challenge while developing our approach.

Memory architecture

The GF110 has a flexible memory model. Its most distinguishing feature among other GPUs is the large, globally-consistent L2D cache; at 128KB per memory controller across the Fermi lineup, GF110 has 768KB of high-speed SRAM to share across its cores. All global and texture memory transactions pass through the L2D, which uses an LRU eviction policy for its 128B cache lines, although an instruction can mark a cache line for discard immediately or upon being fully covered by write operations. The latter mode improves performance when threads perform sequential writes.

Each core has a 64KB pool of memory which can be split to provide 16KB of shared memory and 48KB of L1D cache, or 48 and 16KB respectively. All global reads must use this cache, although writes are handed straight to L2D, invalidating the corresponding cache line in L1D in the process. While the L2D is always globally consistent, L1D is only consistent across a single core; writes to global memory from one core will *not* invalidate the corresponding cache lines in a neighboring core. Volatile loads treat all lines in L1D as invalid, but do not actually invalidate those lines after a load is complete; inconsistent access with non-volatile loads may return the data cached when the volatile load was first issued.

Each work-group is assigned to a single core for the duration of its execution. Each thread acquires its registers and local memory as the work-group is assigned, and the work-group acquires shared memory in

the same manner. Resources are not released until the work-group is complete. As a result, every thread consumes its worst-case allocation at all times.

Atomic operations in Fermi are available on both global and shared memory. Shared-memory atomics are implemented using on-core ALUs, and operate at native speeds unless a bank conflict occurs. Global atomics make use of simple, dedicated ALUs in proximity to the L2 cache. In general, global atomics have higher latency and lower total throughput than local operations, and have lower peak throughput than coalesced read-modify-write cycles, but have higher throughput than any uncoalesced operation.

5.4 Closer look: AMD Cayman

Cayman is the latest GPU microarchitecture as implemented in the ATI Radeon 69xx graphics cards. It is the most significant change in AMD's GPU architecture since the RV770 architecture. The most notable change being the move from a 5-wide VLIW (Very Large Instruction Width) to a 4-wide symmetric VLIW [20]. AMD has stayed focused on graphics performance as opposed to general purpose computing but the Cayman architecture does make a modest step forward for AMD in the realm of GPGPU computing and presents a few evolutionary, not revolutionary, improvements for both general purpose computing and gaming.

The compute capabilities of Cayman GPU's can be accessed by one of two industry standard API's, OpenCL and Direct Compute [20]. While both of these API's have been embraced by all CPU and GPU vendors, they are both relatively young and do not offer the same features and performance that Nvidia's proprietary language CUDA does. However, support for these other two standards are increasing rapidly and are they are posed to dethrown CUDA as the API of choice for general purpose computing in the coming years.

System Architecture

Cayman's system architecture is largely similar to that of the previous generations Cypress. The architecture is split up into cores, or SIMD's, each having its own 8Kb of L1 texture cache and 512KB of shared cache. The biggest differences have all been made with respect to the cores. First of all, Cayman has a total of 24 cores while Cypress has a total of 20 cores with each core being a 16-wide SIMD processor. Each lane can process a VLIW instruction, which means 4 instructions at time for Cayman or (potentially) 5 instructions at a time for Cypress. 4 instructions x 16 lanes = 64 instructions per core for Cayman and 5 instructions x 16 lanes = 80 instructions per core for Cypress. 64 instructions per core x 24 cores = 1536 instructions per chip for Cayman and 80 instructions x 20 cores = 1600 instructions per chip for Cypress. Each lane executes the same operation over 4 cycles. Additionally, SIMD clock speed was increased to 0.88Ghz in Cayman (up from 0.85GHz in Cypress).

Memory architecture

The Cayman architecture uses a memory architecture which is OpenCL-compliant, and thus is not dissimilar from that used by Fermi: high-bandwidth but high-latency GDDR5 global memory is available for cross-device communication and long-term data storage, while fast, core-local shared memory under manual control is provided to store the working set and communicate across a core. L1 texture caches, which are small but achieve extremely high throughput, are located in each core, and a small constant cache accomodates the parameterization needs of graphics shaders.

Cayman differs considerably from Fermi in its treatment of cache. 512KB of L2 *read* cache serves to accelerate both texture and gather reads, while a separate 64KB cache serves to assist in consolidating writes to reach the minimum burst width of GDDR5 and avoid wasting bandwidth on masked bits. These caches operate independently, meaning that communication via global memory is never cached, and often requires

substantial delays to allow all writes to be flushed. The global data share, discussed in Section 6.1), is designed to offer a separate datapath for transactions that need immediate coordination.

CHAPTER 6

TOOLS AND COMPONENTS

GPUs attain extraordinary peak performance by sacrificing generalizability. Devices from NVIDIA and AMD present the same high-level model of massively parallel computing, but — as shown in the previous chapter — these architectures have significant differences at the implementation level. Standards-compliant OpenCL code which does not rely on vendor-specific extensions should run correctly on every compatible device without modification, including the newest GPUs from both manufacturers; the standard, however, offers no indication that the same code will achieve similar *performance* across multiple architectures [21].

The flame algorithm is, in one sense, an embarrassingly parallel problem, and thus fits well into the abstract model of computation offered by OpenCL. Yet, as the rest of the document makes clear, the actual implementation of this algorithm tests the limits of current-generation GPGPU hardware. Writing a fractal flame renderer for the GPU is straightforward; writing one with excellent performance is far more challenging, and requires a much deeper knowledge of each architecture.

The scope of this project is considerable, and the performance goals are near the theoretical upper bounds of current GPU architectures. Given the need to take advantage of architecture-specific features, this project's software is not likely to be portable between graphics architectures or compute platforms, making a late-stage change in those decisions expensive. Meeting the performance goals of this project without exceeding time or budget constraints will require carefully selecting the tools with which it is built.

This chapter is an overview of that selection process and its result. Because the optimizations required to implement the flame algorithm require support at every level of the toolchain, those optimizations are an important part of the selection process. However, the exact nature of those optimizations depends on the results of the selection process. Consequently, tool selection is an ongoing, iterative effort that may be subject to further change during initial implementation. For this document, we break the dependency cycle by presenting tool selection first and extensively referencing future chapters.

6.1 GPU architecture

The fastest compute devices available to consumers, at time of writing, feature AMD's Cayman or NVIDIA's Fermi architectures. Cayman devices have dramatically higher peak theoretical performance values, but as discussed in Section 5.4, it can be difficult to reach peak throughput due to the nature of the VLIW4 architecture used. In low-level optimization projects, it may be tempting to believe that hand-tuned code can beat even the best optimizing compilers; the pragmatic view, however, is that even if such an extraordinary hand-tuning effort were to produce faster code, architecture variations in the next GPU cycle would likely erase that performance gain. In other words, there is a practical limit to how low a project of this scope can delve for optimizations and still be successful. With that in mind, we accept the general

consensus on raw computing power — that Cayman and Fermi are generally well-matched, and the winner is workload-dependent [19] — and focus on other factors to choose an architecture.

AMD's architecture implements flow control using clauses which execute to completion; each clause specifies the next clause to execute in its terminating condition. Apart from indirectly enabling higher throughput by simplifying scheduling, clauses provide the advantage of temporary registers. NVIDIA cores allocate all local resources statically, requiring each thread to consume its worst case number of registers at all times, whereas Cayman and other AMD architectures allow non-persistent resources to be shared. This could significantly increase occupancy of AMD cores when the most complex variations are active, helping to hide latency. On the other hand, NVIDIA's solution — provde an enormous 128KB register file per core — tends to be sufficient to avoid this circumstance.

AMD executes clauses in wave-fronts of 64 threads, whereas NVIDIA uses a 32-lane warp. Both methods accomodate the producer-consumer relationships across vectorized execution units through work-group barriers, but Fermi takes advantage of its particular vector width by providing a number of instructions and virtual registers that enable intra-warp communication without using shared memory. Warp voting is not a common activity in graphics operations, but it is a required part of some of the optimizations described herein, and in such cases Fermi holds a 32× lead.

Another key differentiator between the two compute platforms is the use of cache in main memory access. Cayman devices have 512KB of read cache, and a separate 64KB of write cache; the latter is used primarily to extract spatial coherency from temporally-coherent data. The separation of concerns makes the cache a less costly addition to Cayman devices than Fermi's full-featured L2, but does little to accelerate random-access updates to values in global memory, and can increase the complexity required to ensure consistency of global values.

AMD's solution is the global data store, another 64KB chunk of memory shared across all cores. This structure is intended only for inter-work-group communication, providing fast and atomic access via a separate address space. This anomaly is a useful tool for coordinating access to complex data structures, but may simply be a stepping stone on the way to a full cache in future architectures [20]. For the complex addressing patterns needed to support full-rate accumulation, Fermi's L2 seems the more capable solution for inter-thread communication.

The company behind Cayman has a history of being more open than its competitors with technical information, a trend continued with its latest GPU offerings; technical documentation on the Cayman ISA and other architectural features is publically available. In principle, this is a big advantage over NVIDIA, who hides most instruction-level details behind PTX, their cross-platform intermediate language for GPU kernels. Unfortunately, for this project, the practical advantages of PTX make it the better option. The intermediate language provides access to nearly every feature of interest in NVIDIA's hardware while preserving forward compatibility, and is optimized at runtime by the driver to best fit each platform; writing a backend that emits PTX is therefore a relatively straightforward task. Generating assembly for AMD devices is more challenging, and a backend must target a set of primitives that changes with each hardware generation while performing device-specific optimization itself. A more realistic solution to take advantage of low-level instructions on AMD hardware is to precompile code for AMD hardware and monkey-patch in memory [22], but this task becomes much more challenging with dynamically-assembled code.

There is no substitute for profiling live code; conjecture on the performance of optimized code across multiple architectures is speculative at best. Given the need to standardize on a single architecture, the information available suggests that NVIDIA's Fermi is more likely to yield the highest performance without overwhelming optimization efforts.

6.2 GPGPU framework

OpenCL is, well, open. Its broad industry support, including stalwart backers Apple and AMD, and adoption in the mobile computing space make it likely to be the standard of choice for cross-platform development of high-performance compute software [16]. It also offers an extension mechanism, similar to the one used in OpenGL, to offer a clean path for a vendor-specific hardware or driver feature to become a part of the standard without breaking old code. OpenGL's history presents evidence that vendor support of these extensions is important in determining whether the standard will stay current and relevant.

Once again, however, NVIDIA's technological head-start in the GPGPU market is large enough to warrant ignoring ideological preferences. Kanter notes that OpenCL is "about two years behind CUDA ," a sentiment echoed by many industry observers and supported both by a simple comparison of the feature sets of both frameworks and by the authors' first-hand experience.

Due to the need to optimize the rendering engine to hardware constraints, particularly with regard to components such as the accumulation process (Chapter 12), porting this implementation across GPUs is expected to be difficult. As a result, compatibility and standards compliance is not a priority for this implementation. This implementation will therefore be based on the CUDA toolkit, rather than OpenCL[1].

[1]The authors are also planning an entirely new implementation which should not be quite so *fussy* about the hardware parameters. This implementation operates quite differently from the traditional flame algorithm, and we're still working out the necessary mathematics, so it is not documented here — but when it is ready to be implemented, we do intend to use OpenCL.

CHAPTER 7

RANDOM NUMBERS AND PSEUDO-RANDOM NUMBER GENERATORS

Random numbers are used in this project because of their importance in calculating and rendering fractals using Iterated Function Systems. This is a fundamental concept of an IFS and is known as the *chaos game* (See Chapter 3 for more detail). However, real random numbers are hard to calculate in a computer; in great part because they depend on time or because there isn't an infinite number of bit sized chunks for computation. Pseudo-Random Number Generators (PRNGs) are algorithms that simulate randomness in a computer, usually by using prime numbers as seeds because when they are used in a division, the output is an irrational number.The greater the prime number, the better quality numbers are outputted. In order to find the right PRNG for this project we will consider advantages and disadvantages of different well known PRNGs.

In selecting the right PRNG, it is common to look at its period (or numbers it outputs until it starts repeating itself), its speed and its spectral properties, the latter which determine its true randomness. For this project, we are looking for a simple and fast PRNG that meets out minimum needs.

7.1 Bias : An Illustrative Example

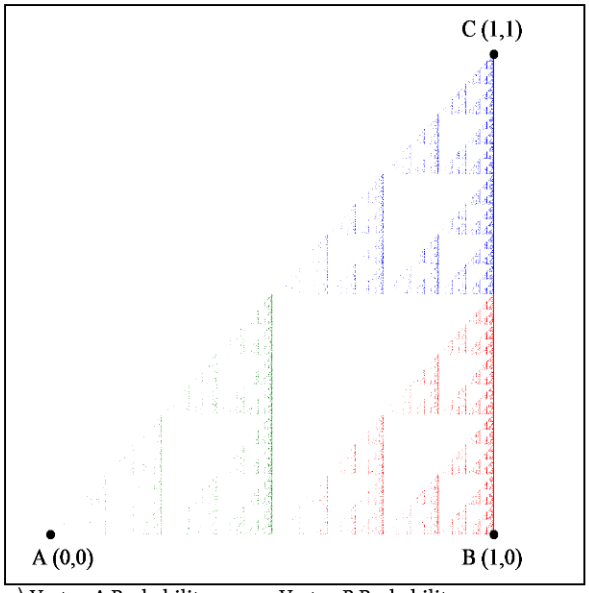
Before preceeding with selecting the proper PRNG for this project an illustrative example is shown. Normally in the Sierpinski's Triangle Iterated Function System each function has an equal probability of $\frac{1}{3}$ of being chosen (See Section 3.2 for the fully worked example with matrix transformation equations).

However, if the PRNG began showing bias, the results could be disastrous to our system.

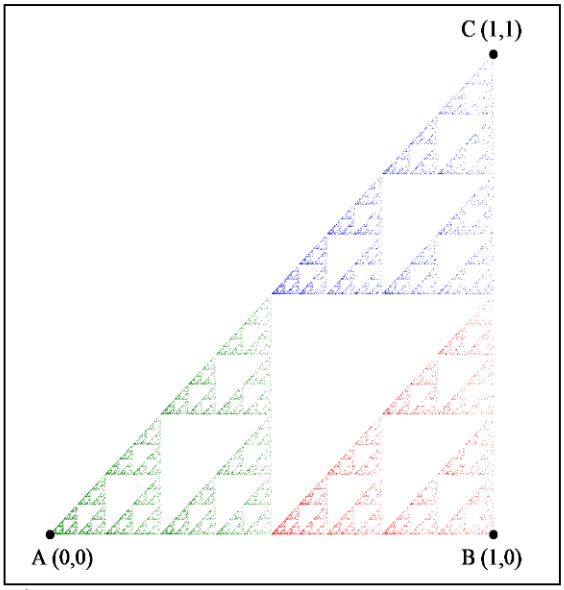
Shown in Figure 7.1 are 4 forms of extreme bias shown in an IFS. In (A), if the affine transform related to Vertex A was reduced to a probability of $\frac{1}{10}$ the resulting triangle would show a lack of detail at the green lower left region. This is because it must undergo subsequent transformations of the function relating to Vertex A in order to draw the bottom leftmost region of the image which is highly improbable with it's probability of being selected is $\frac{1}{10}$.

Conversely, if the probability of selecting the affine transform related to Vertex A was higher than the other two transforms (relating to Vertex B and Vertex C) an opposite effect happens (see picture (B) , (C), and (D)) in which the right region of the triangle is barely visible.

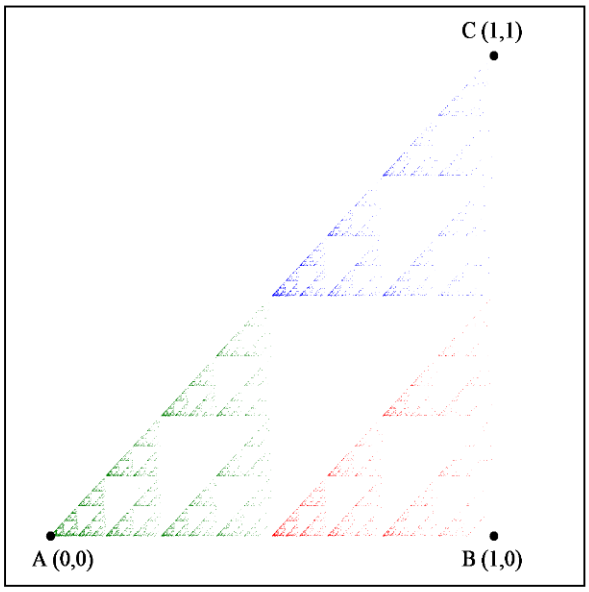
Obviously, this example was meant to show the extremes of PNG bias, however even subtle biases can propagate through the system and cause similar biases. This is why selecting a solid PNG is an important decision.



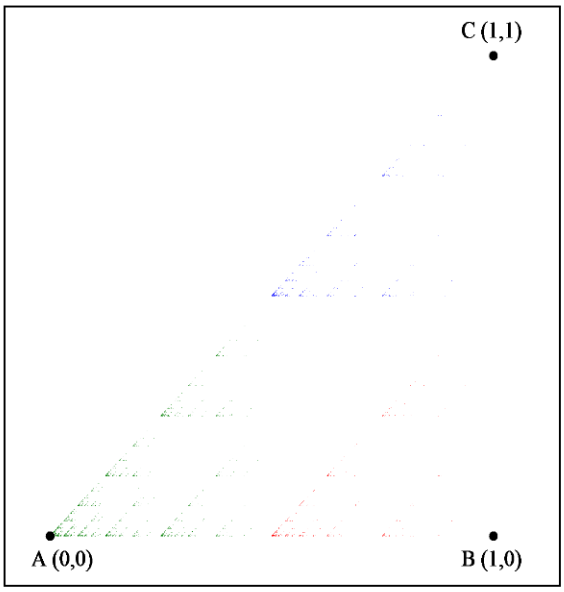
a) Vertex A Probability = 0.10 Vertex B Probability = 0.45
Vertex C Probability = 0.45



b) Vertex A Probability = 0.50 Vertex B Probability = 0.25
Vertex C Probability = 0.25



c) Vertex A Probability = 0.70 Vertex B Probability = 0.15
Vertex C Probability = 0.15



d) Vertex A Probability = 0.90 Vertex B Probability = 0.05
Vertex C Probability = 0.05

Figure 7.1: Sierpinski's Triangle shown with biased values of applying the function which pulls the points towards Vertex A.

7.2 Pseudo Random Number Generators

There are various properties that a PRNG can have, but for this project, we are looking for maximized speed and spectrum properties, and a PRNG that can be implemented in a GPU.

7.3 `rand()` and Linear Congruential Generators

The search for pseudo-random number generators begins with the most commonly used, win32's `rand()` function. The problem with this function is that its randomness is biased. Evaluation of the statement `x=rand()%RANGE;` returns any number represented by $[0, \text{RANGE})$ instead of $[0, \text{RANGE}]$. Assuming that `rand()` outputs a number $[0, \text{MAX}]$, RANGE should be able to divide by MAX $+ 1$ entirely in an ideal PRNG, however it doesn't in the `rand()` function and therefore the probability of choosing a random number X in $[(\text{MAX\%RANGE}), \text{RANGE}]$ is less than that of choosing it in $[0, \text{MAX\%RANGE}]$.

Another problem with `rand()` is that it is a Linear Congruential Generator (LCG). The way LCGs work is with the following basic formula:

$$X_{n+1} = (a \cdot X_n + c) \mod m$$

Where X_{n+1} is the next output and a and m must be picked by the user of the algorithm. Here, the problem is not only that to get decent randomness one needs to pick a and m carefully (with m closest to the computer's largest representable integer and prime) and a equal to one of the following values [23]:

m	a_1	a_2	a_3
549755813881	10014146	530508823	25708129
2199023255531	5183781	1070739	6639568
4398046511093	1781978	2114307	1542852
8796093022151	2096259	2052163	2006881

Table 7.1: Acceptable values for LCG modulus m and multiplier a.

There are other choices for m, with their respective values for a, but those sets also have rules and may not apply to certain computers if they don't have the required hardware.

7.4 ISAAC

An alternative that sounds like a better choice is ISAAC, it stands for Indirection, Shift, Accumulate, Add, and Count [24]. The way it works is by using an array of 256 4-byte integers which it transforms by using the instructions that define its name and places the results in another array of the same size. As soon as it finishes the last number, it uses that array to do the same process again. The advantages of this PRNG are that it is fast since it only takes 19 for each 32-bit output word, and that the results are uniformly distributed and unbiased[25]. The disqualifying disadvantage is that even though the GPUs; which we will use for this project, have enough global memory, they don't have the memory required to be able to have arrays of size 256.

7.5 Mersenne Twister

"Mersenne" in its name because it uses Mersenne primes as seeds (Mersenne primes are prime numbers that can be represented as $2p - 1$ where p is also a prime number). This PRNG uses a twisted linear feedback shift register (LFSR), which uses the XOR instruction to create the output, which then becomes part of the values that are being XORed. The "twist" in its name means that not only do values get XORed and shifted, but they also get tampered and there is state bit reflection.

It is a good choice for this project for several reasons; it is sufficiently fast for this project, it has a period of 2^19937 −1 (meaning the random numbers will not repeat for that many iterations), and it can be implemented on a GPU, however, it requires a large amount of static memory on the GPU, and it operates in batch mode, meaning that when the pool runs out of random bits, the entire pool must be regenerated at once. This can be handled with CUDA (NVIDIA's parallel computing architecture), but its not the fastest or simplest solution.

7.6 Multiply With Carry

This algorithm might seem similar to the typical one for a LCG, but it differs when it comes to how the new iteration values are chosen. To start, one chooses a numbers a, c, and m. A number b is also chosen such that $b = 2^{\text{half the size of the register}}$. First, $x_1 = (a_0 * x_0 + c_0) \mod m$, then, the quotient of the past calculation becomes the quotient for c and $x_n = (a_1 * x_{n-1} + c_{n-1})$ where $x_{n-1} = xn - 2 \mod b$ and $c_n - 1 = \lfloor (x_{n-1}/b) \rfloor$. This process gives Multiply With Carry (MWC) advantages over LCGs if the numbers are chosen carefully because by having c vary in every iteration, the randomness of its output can pass tests of randomness that LCGs can't.

It is important to note that MWC implemented in this form has a period that cannot be represented by a power of two, and depends on the size of the register used. The best values to choose in order to have a large period are when $ab - 1$ is a Safe Prime (a number that can be represented in the form $2p - 1$ where p is a prime number.) For a register of size 32, a can be chosen to be a number represented by 15 or 16 bits, if it is 15 bits, then the maximum number a can be is 32,718 and the period will be 1,072,103,423, if its 16 bits, then the maximum number a can be is 65,184 and the period will be 2,135,949,311. For a register of size 64, a can be chosen to be a number represented by 31 or 32 bits, if it is 31 bits, then the maximum number a can be is 2,147,483,085 and the period will be 4,611,684,809,394,094,079, if it is 32 bits, then the maximum number a can be is 4,294,967,118 and the period will be 9,223,371,654,602,686,463.

This algorithm can be used in the GPU for 4 main reasons; it is very elastic when it comes to limitations or requirements of register sizes, it is not over engineered or takes too many lines of code to implement, it passes the best known randomness tests, including the Diehard Tests, and its spectral properties meet the requirements for this project. The only thing that could be considered a disadvantage for this algorithm is that its most significant bits can be slightly biased, however, not enough as to make a difference in this project.

7.7 Spectral Distribution

The spectral distribution test is devised to study the lattice structures of PRNGs and especially that of LCGs. It is also famous in great part because it fails LCGs that that have passed other tests. It works by taking the outputs of PRNGs and finding where the numbers lie in s number of dimensions; it then takes that information and displays it as a lattice as seen in Figure R.2. Mathematically, overlapping vectors $L_s = x_n = (x_n, ..., x_{n+s} - 1)$ where $n \geq 0$ are considered, since they exhibit the lattice structure. An example of lattice structures can be seen in Figure 7.2.

However, without having to draw the dots, a conclusion about a PRNG can be made because of its mathematical properties; the spectral test determines a value y_k which determines the minimum distance between points in the s hyper-planes on which it tests. The formula is given by $y_k = \min(\sqrt{(}x_1^2 + x_2^2 + ... + x_k^2))$ Ideally, the minimum number from o to k will be a high value (in the thousands) and the PRNG will also have a high number of dimensions.

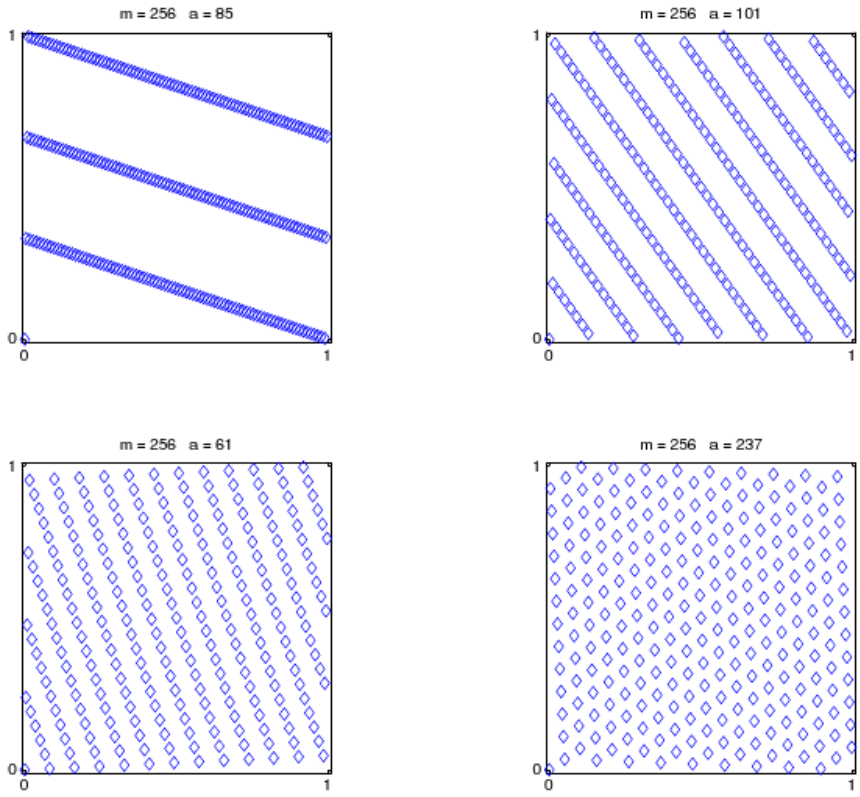


Figure 7.2: Example of lattice structure.

7.8 Monte Carlo simulations

The Monte Carlo methods are algorithms that use statistics to determine probabilities in systems and their properties. They are used in finance, physics, communications and even game design. In the context of this project, they are necessary measures of randomness that can be held as a standard that filters out PRNGs that don't meet the basic requirements. Using these methods, the spectral properties and periods of some PRNGs and their variations will be determined.

CHAPTER 8

COLORING AND LOG SCALING

8.1 Overview

The *chaos game* provides a way to plot whether points in the plane[1] are members of the iterative function system or are not. However, the resulting image appears in black and white (lacking color or even shades of gray). The application of color as well as making these images vibranat are their own processes in the algorithm which deserves much text for several reasons:

1. A flame is just simply not a flame without its structural coloring or if membership is binary (black and white) which results in a grainy image. Both of these shortcomings leave a lot to be desired but can be remedied.
2. Much of the new *implementation* relies on reworking details on how coloring is done.

Section 3.3 of the fractal flame algorithm chapter describes the application of log scaling, a scheme for structural coloring as well as certain color correction techniques however the implementation details were spared. This section explores the color correction techniques are implementatios in the context of the original fractal flame implementation called `flam3`. The inner algorithmic choices, data structures, and capabilities that the program has are analyzed. With that, in accordance to the challenge response style paper some of the difficulties with making improvements and transitioning the algorithm to the GPU are presented. Finally, the authors delve into the new *implementation* and the differences, similarities, and any relevant background information needed.

8.2 Relevant Applied Color Theory and Imaging Techniques

Introduction

In the case of the fractal flame algorithm when coloring is referred to what is meant is the act of tone mapping, structural coloring, any color theory techniques (such as colorimetry), and finally any imaging techniques (such as gamma correction) used. Luckily, all of these techniques have strong mathematical backgrounds and there is a vast information about each of them readily available because of advances in both computer graphics and digital photography. Additionally, because the flame algorithm's output is an image or series of images it often runs into the same complications which plague digital photographs such as

[1] Again by plane we are refering to a biunit square where x and y values can have a minimum value of –1 and a maximum value of 1.

color clipping and therefore these same image correction techniques are translated over to our domain and retrofitted to greatly improve the output image. A small detour is taken to visit all related techniques as one of the major requirements that must be adhered to for a new implementation is to produce images which are approximately visually equivalent. Without using some of these techniques, replicating flame would be increasingly more difficult.

High Dynamic Range (HDR)

A fundamental concept which the whole coloring and log scaling approach tries to achieve is a high dynamic range or simply abbreviated as *HDR*. High dynamic range means that it allows a greater dynamic range of luminance between the lightest and darkest areas of an image [26]. Dynamic range is the ratio between the largest and smallest possible values of changeable quantity (in our case light). Lastly, *luminance* being the intensity of light being emitted from a surface per some unit area.

The techniques that allow going from a lower dynamic range to a high dynamic range are collectively called high dynamic range imaging (HDRI). The reason HDR and HDRI imaging is mentioned is because the output of the flame attempts to give the appearance of an HDR flame while being restricted to Low Dynamic Range (LDR) viewing mediums such as computer monitors (LCD and CRT) as well as printers.

By observing common dynamic ranges of some typical mediums as well as various digital file formats we can begin to see why we are limited.

Both the file format technologies in which our images or videos and stored in as well as monitor or paper in which they are viewed on are interrelated limiting factors governing the dynamic range. Various typical contrast values that these scenes can emit or in the case of file formats are capable of representing are seen in Table 8.1[@Kolor].

Medium	Ratio	Stop
JPEG Image File	256 : 1	8
RAW Image File	1,024 : 1	10
HDR IMAGE FILE	approx. 32,768 : 1 to 1 : 1,048,576	approx. 15 - 20
Standard Video	45 : 1	5.49
Standard Negative Film	128 : 1	7
LCD TECHNOLOGY	500 : 1	8.96
CRT DISPLAY	50 : 1	5.64
GLOSSY PRINT PAPER	60 : 1	5.90
Newsprint	10 : 1	3.32
SUNLIT SCENE	approx. 100,000 : 1	16.60
HUMAN EYE	approx. 10,000 : 1	13.28

Table 8.1: Typical dynamic ranges of various scenes or typical dynamic ranges that able to be represented.

Where $Stop$ is defined as : $log_2(Ratio)/log_2(2)$.

Let's take a look at what this table really means in the case of imagery. The table shows that a *HDR Image File* can represent an impressive range of contrast - far higher than the eye can observe. We also note that it could approximately even capture a *Sunlit Scene* which contains extreme contrast between the brightness and darkest intensity values. However, if we look at our viewing technologies we notice their limits of displaying contrast. *LCD Technology* has a Stop value of approx. 8.96, *CRT Technology* has a Stop value of approx. 5.64, and *Glossy Print Paper* has a Stop value of 5.90. Compared to the *Human Eye* whose Stop value is approx. 13.28, these values are incapable of being on par with the level of contrast the human eye can observe and therefore will not accurately represent how the colors ideally should be observed.

Luckily, we can work within our imposed limitations and there are many imaging techniques that can be applied to attempt to remedy the situation. The following techniques described below are not only for aesthetics but also are some of the core techniques for representing HDR images on LDR mediums. This coincides with the goal the entire algorithm wishes to achieve and is paramount to fix our LDR dilemma.

A RGB Color Model: Hue, Saturation, and Brightness Value (HSV)

To attempt to mathematically define certain color concepts (e.g. brightness, saturation, vibrancy) a color model for how our colors will be represented spatially is chosen so the relationship between colors can be talked about.

All of the color definitions and concepts are in terms of the *Hue, Saturation, and Value (HSV)* model. It is explained in this model simply because `flam3` uses this concept and by using the HSV model it will save additionally explanation on how this model works. It should also be noted that there are alternative color models such as:

- Hue, Saturation, and Lightness (HSL)
- Hue, Saturation, and Intensity (HSI)

The HSV color model spatially describes the relationship of red, green, and blue according to these following components:

- Hue
- Saturation
- Brightness

It does this by representing these using a cylindrical coordinate system. The axis representations are the following:

- ROTATIONAL AXIS: The rotation axis represents *hue*. Hue refers to pure spectrum of colors - the same prism observed when splitting light. At 0° on the axis the primary color red is represented, at 120° the primary color green is represented, and at 240° the primary color blue is represented. The rest of the degrees are filled in according to the color spectrum.
- VERTICAL AXIS: The vertical axis represents brightness. Colors at the top of the spectrum have no brightness (value of 0 which would be the color black) and at the bottom have maximal brightness (value of 1 which would be the color white).
- HORIZONTAL AXIS: The horizontal axis represents saturation. Saturation is defined here as how prominent the hue is in the resulting color. The outer regions are that of the pure color spectrum whereas the inner regions are gray scale color where no hue is observed and the values depend purely on brightness.

Using the HSV model, seen in Figure 8.1, provides a simple way of representing the color space and describing the relationship between them. The calculations also require little computation. However, one of the major drawbacks is that the model gives no insight into color production or manipulation.

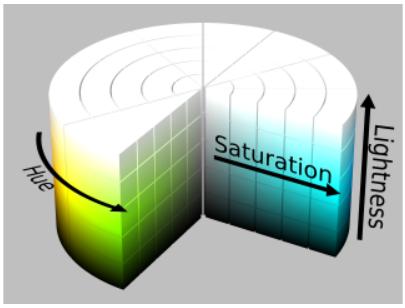


Figure 8.1: HSV Color Model

Hue

The term *hue* refers to the pure spectrum of colors and is one of the fundamental properties of a color. The unique hues are red, yellow, green, and blue. Other hues are defined relative to these. Looking at a spectrum of light (such as the rainbow) would represent the spectrum of hues.

Gamma and Gamma Correction

The term *gamma* refers to the amount of light that will be emitted from each pixel on the monitor in terms of in a fraction of full power (pixel being shorthand for the red, green, and blue phosphors of a CRT[2]).

The main concept we're interested in is *gamma correction*. The reason *gamma correction* is needed is the following:

1. CRT and LCDs displays do not display the light proportional to the voltage given to each phosphor. Therefore the image does not appear in the way it was expected to be viewed.
2. A typical consumer grade printer works upon 8 or 16 bit color and result in a relatively low HDR as seen above in Table 8.1.

To summarize, the RGB color system with red, green, and blue values ranging from 0 to 255 cannot be accurately represented. Some kind of correction must be performed in order to get the images that are expected to be seen rather than the images that are actually seen as output.

This concept of *gamma correction* can be applied at the hardware level however, but this varies depending on the vendor and hardware capabilities of the machine. For example:

- PCs typically do not implement gamma correction at the hardware level. A noteable exception is that certain graphics card may implement a gamma correction natively.
- Macintoshes typically provide a gamma correction at the hardware level of 1.4.

Besides being implemented at the hardware level, gamma correction can additionally be provided at the software level.

The formula for *gamma correction* is $b_{corrected} = b^{1/\gamma}$ where γ is the correction factor.

To understand the non-linearity of the gamma function 4 gamma correction values are applied to an image for visual depiction of the concept. The results are seen in Figure 8.2.

[2]For LCDs the relationship between signal voltage and intensity is very non-linear and a simple gamma value cannot describe it. A correction factor can be applied however and the concepts are similar.

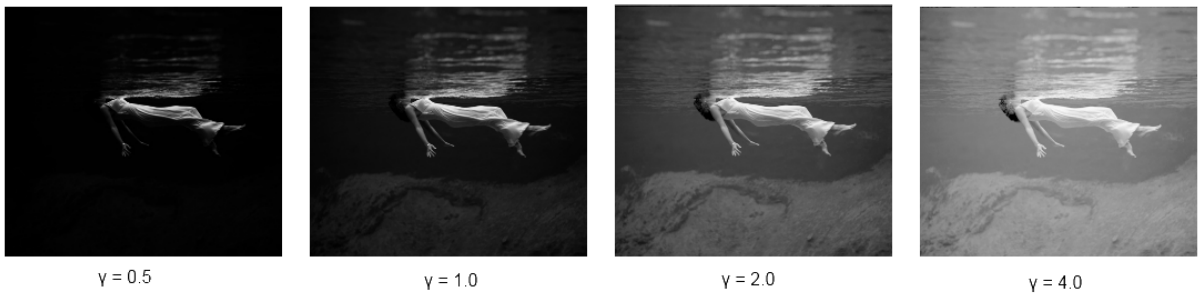


Figure 8.2: Comparison of gamma correction values.

The second image of Figure 8.2 does not undergo any gamma correction and can be used as a baseline comparison to the other images. By observing both the higher and lower bounds of the images presented we can see that images become either too dark or too light (and their is little contrast between the colors). The third image has a gamma correction value of 2. Normal gamma correction is roughly around 2.2 which explains why this image appears more natural and has a higher dynamic range then the others.

Brightness and Brightness Correction

The term *brightness* is a term that must be defined with great finesse. Unlike *luminance*[3] which is empirical, *brightness* is subjective. The subjectiveness comes from that brightness is according to the range of lumens that the eye can perceive. This attribute is often more qualitative than quantitative and can range from very dim (black) to very bright (white).

We can attempt to quantatively talk about brightness using the concept of a color model. There are many models and they usually compute brightness in one of the following two ways:

1. Give equal weights of each color component (R, G, and B)
2. Give weighted values of each color component (R, G, and B). This is referred to as perceived brightness.

An example of the first application would be a naïve approach that goes on the notation that if black is $Red = 0$, $Green = 0$, $Blue = 0$ and white is $Red = 255$, $Green = 255$, and $Blue = 255$ then the brightness can be simply $Red + Green + Blue$.

This flaw can be seen in Figure 8.3. The red, green, and blue components of a color have different wavelengths and therefore have a different perceived effect on the eye. A good brightness calculation attempts to model how the eye perceives color rather than treating each color component with equal weights. A common flaw of a color model for brightness is the under or over represent one of the color components.

[3] Again, *luminance* being the intensity of light being emitted from a surface per some unit area.

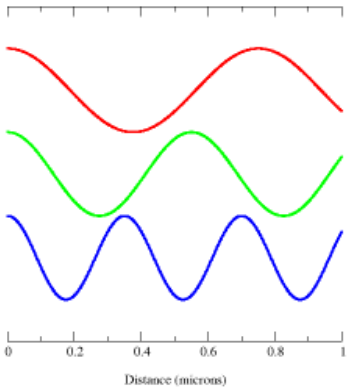


Figure 8.3: Wavelengths of Red, Blue, and Green

Some examples of weighted models to calculate brightness are below in Table 8.2.

height Model	Formula
Photometric/digital ITU-R	$0.299 \times R + 0.587 \times G + 0.114 \times B$
Digital CCIR601	$0.299 \times R + 0.587 \times G + 0.114 \times B$
HSP Color Model height	$\sqrt{0.241 \times R^2 + 0.691 \times G^2 + 0.068 \times B^2}$

Table 8.2: Weighted brightness calculations models[@Finley].

Later, the topic of *brightness correction* is of interest- the act of adjusting the brightness. Flame's brightness can be adjusted however care must be taken so that the minimum and maximum bounds are not exceeded.

With the addition of too much or too little brightness color clipping may occurs and the colors fall outside of representable realms which result in a loss of data. See Section 8.2 for additional detail.

Saturation and related terms

The concept this section intends of describing is that of *saturation* but as a building block concept it is felt necessary to talk first about the more broad concept in *color theory* which is the intensity of the color. There are different variations of measuring the intensity of the color. The three main terms as well as their distinctions between each other are below:

1. COLORFULNESS: The intensity of the color is a measure of the colors relative difference between gray[27].
2. CHROMA: The intensity of the color is a measure of the relative brightness of another color which appears white under similar viewing conditions[27].
3. SATURATION: The intensity of the color is a measure of its colorfulness relative to its own brightness rather than gray[27].

The term that is of importance is that of *saturation*.

Colors that are highly *saturated* are those closest to pure hues of color. Colors that have little saturation appear *washed out*. Also as a note, the changing of a color's saturation can be observed as linear effect.

Vibrancy

Now that saturation was explained, the term *vibrancy* can be explained. Vibrancy is similar to saturation however different in the following fashion:

Saturation is linear in nature whereas vibrancy works in a non-linear fashion. In vibrancy the less saturated colors of the image get more of a saturation boost than colors that already have higher saturation values. A simple non-linear saturation is applied to photograph shown in Figure 8.4.

Unsaturated Image

Non-linear Saturation making colors appear more vibrant.

Figure 8.4: Original image compared to a non-linearly saturated image.

Color Clipping

A problem that plagues images in digital photography and that of the flame algorithm is the concept of *color clipping*. Color clipping happens when color brightness values to be outputted to the image fall either below or above the maximum representable range.

In digital photography *color clipping* can happen from an improper exposure setting on the camera which results in effects, such as the lighting from the sun, overwhelming certain portions of the image. In the case of the flame algorithm, this concept can be viewed in a different context. One problem with flames is that certain areas of density in the histrograms from the *chaos game* can become so dense that their color setting exceed the bounds of representable brightness's.

When data exceeds the upper and lower bounds of representable brightness's a loss of data occurs. As a result, there is an inability to determine the differences between those regions of data as they default to the maximum or minimum brightness and appear uniform. A focus on the approach of the algorithm is to prevent this and preserve and be able to represent all contours of the image and their brightness's.

8.3 Log Transformation of Data

Data transformations in statistics are a common method of transforming data points in order to improve the interpretability of visualization of the output (e.g. graphs). Some common transformations include:

- Square Root
- Logarithm
- Power Transform

Transformations are in the form of deterministic functions. For the specific purposes of this paper the logarithmic transformation of data is studied.

The ultimate goal again of the coloring and rendering is preservation. If a non-transformed histogram of densities of a flame are plotted information is lost about the least and most dense areas of the histogram. The logarithm transformation helps preserve the relationship between points to provide a more accurate histogram.

8.4 Tone Mapping and Tone Operators

With a firm idea of High Dynamic Range (HDR) the concept of tone mapping is now described. What the process of tone mapping produces is a mapping from one set of colors to another that is applied to the image. This is heavily used in image processing. Because a flame is limited to a lower dynamic range when presenting images on monitors or printers tone mapping is applied in an attempt to closely resemble the appearance of an HDR image. This is one of the goals of tone mapping. The two typical application purposes of tone mapping are as follows:

1. Bring out all of the details of an image - or more specifically, maximizing image contrast. This approach focuses on producing realism and aims to render an image as accurately as possible.
2. Create an aesthetically pleasing image, often ignoring the realistic model that the first approach attempts to model but trying to create another desired effect. This effect is up to the person designing the tone operator which is applied to the image.

The method for applying this tone mapping is done via a tone mapping operator. The HDR image is processed by the tone mapping operator which provides one of the two above mentioned effects. There are two major classifications of tone mapping operators[28]:

1. GLOBAL TONE OPERATORS: In a global tone operator the mapping of one color set to another is uniformly applied to the image. This mapping is in the form of a non-linear function that is determined to be the desired mapping[28]. Gamma correction is an example of a simple global tone operator.
2. LOCAL TONE OPERATORS: In a local tone operator the mapping of one color set to another varies according to the local features of the image. The tone operator takes into the regions of changing pixels in the image.

8.5 `flam3` : Original Coloring and Log Scaling Implementation

Log Scaling of the Chaos Game

In the classical IFS membership in the system is binary however in the fractal flame algorithm one of the goals is to expose as much detail as possible. As mentioned before, every successive time a point gets plotted in binary representation information is lost about the densities of regions of the output flame. This is remedied with the concept of a *histogram*. At each iteration a variation is applied which adjusts the IFS color coordinate which represents the RGB color space and is from `[0,1]` as seen in Section 3.3. The density is also increased every time the point is plotted. At the end of each iteration the color coordinate looks up a RGB color from the palette which is the value which is accumulated. Upon the final iteration the triplet of color values (R, G, and B) become log scaled by the density.

The log scaling performed in `flam3` coincides with the overall goal of approximating a high dynamic range flame. The method described above is a straightforward implementation although the naming convention of the `flam3` fractal flame algorithm needs articulation. No additional information is needed and the understanding of the benefits of log scaling and why `flam3` implements it should be found in the referenced sections above.

Ad-Hoc Tone Mapping and The Color Palette

As seen in the previous section, at the end of each iteration the color coordinate looks up the actual RGB color mapped to it inside the palette and applies that R,G,B triplet to the accumulator. `flam3` contains 701 color palettes packaged in the software however the user can define their own mapping. Some of these palettes will be explored later on in this section. In `flam3` each color palette is traditionally consists of 255 color entries. The reason behind this is that the mapping to the palette is in the form of a byte which contains 256 different bit choices.

Along with the application of color correction (gamma, vibrancy, etc.) this combination of mapping grayscale to a set of colors and then correcting them is a form of highly specialized tone mapping.

Coloring Capabilities

Overview

`flam3` provides not just structural coloring but also exposes a vast amount of functionality which allows the resulting flame to undergoing image correction and other altercations. The image correction and other altercations are done using a configuration file. The section visually inspects:

- Color Palette
- Gamma Correction
- Gamma Threshold
- Hue
- Brightness Correction
- Vibrancy
- Color Clipping

- Highlight Power.

After visually inspecting them as well as describing their purpose and how the output flames benefit from them, `flam3`'s implementation will be examined. Next, the authors discuss what features are essential for our task at hand and which color correction techniques could be omitted while still providing an essential subset of functionality.

Visual Inspection of the Baseline Image

For reference to the reader a baseline image of a detailed flame containing several transforms with a vivid default coloring scheme is provided in Figure 8.5.

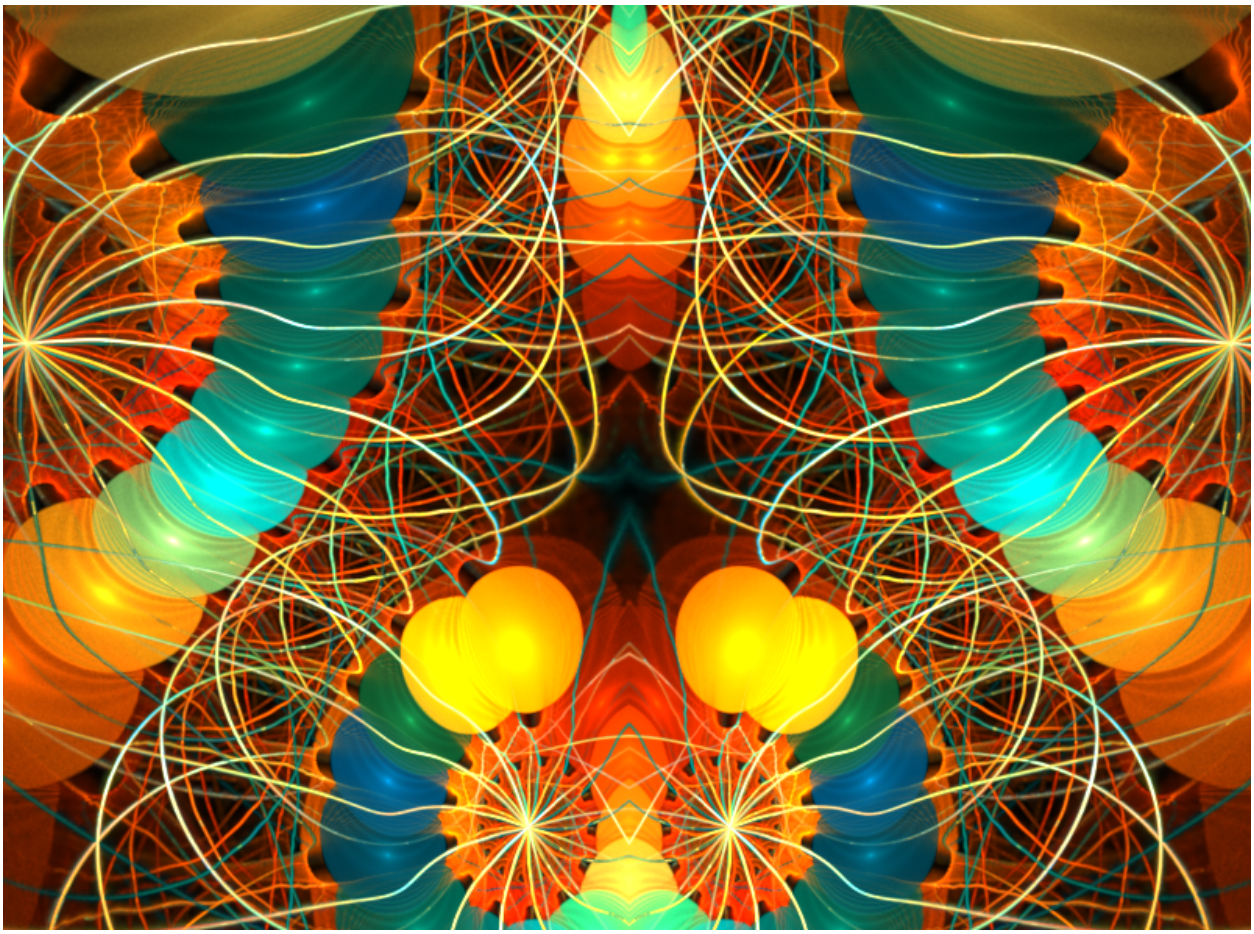


Figure 8.5: Baseline image of the flame whose parameters will be altered.

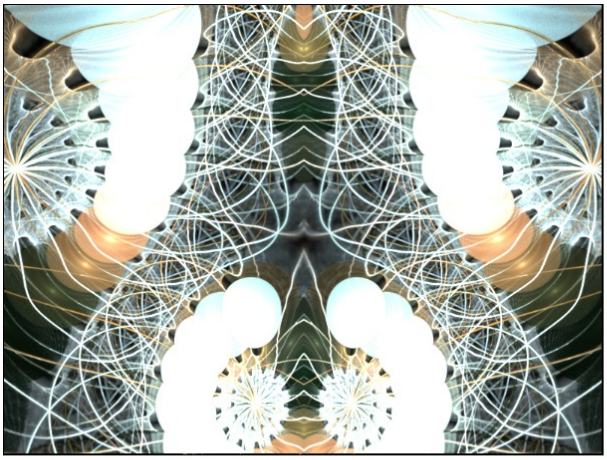
In the following sections adjustments are performed to one parameter of the flame while holding the others constant so that the parameter effect in question can be observed. The parameters that constructed the flame above are shown below in Figure 8.3.

Correction Technique	Default Value
Gamma Correction	3.54
Gamma Threshold	0.01
Brightness Correction	45.6391
Vibrancy	1.0
Early Clipping	Off
Highlight Power	0.0
Hue	0° Rotation to the Color Space
Color Palette	User Defined Palette

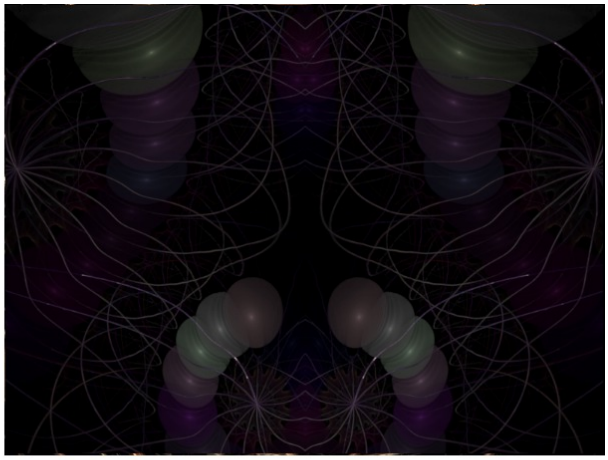
Table 8.3: Parameter values of our baseline image which modified versions of this flame will be compared to.

Color Palette Revisited and Explored

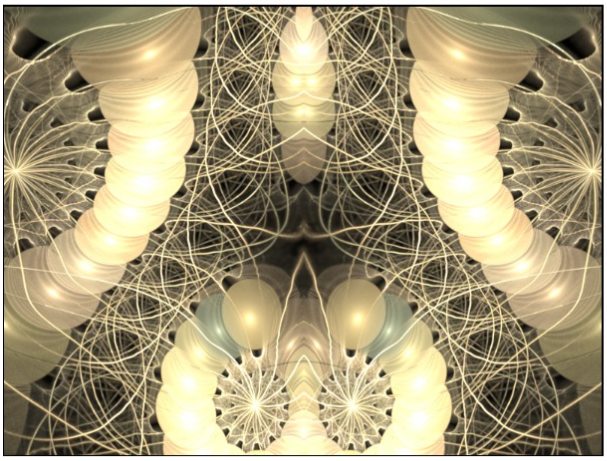
As mentioned in Section 8.5 : *Ad-Hoc Tone Mapping and The Color Palette* there are 701 standard palettes available. A minute amount of palettes are shown to give the reader an understanding of what a palette may look like. Figure 8.6 shows 4 different palettes applied to the baseline flame. By observing both palette number 1 and 5, you can see that colors become clipped and their is varying degrees of detail loss. There is a careful balance of setting tweaking between brightness, gamma, that must be maintained in order to preserve a higher dynamic range. This is one of the reasons these features exist.



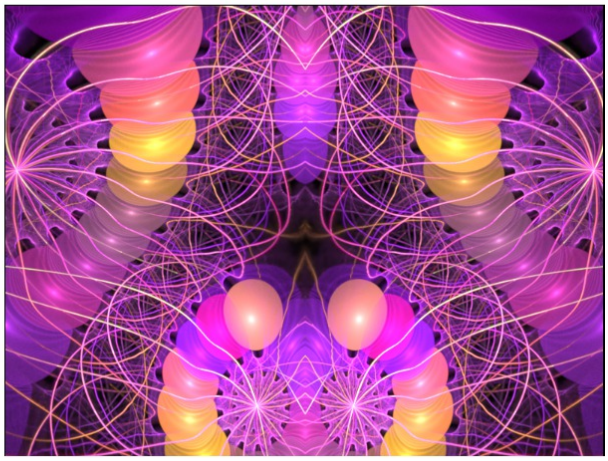
Flame colored with predefined palette number 1.



Flame colored with predefined palette number 5.



Flame colored with predefined palette number 82.



Flame colored with predefined palette number 300.

Figure 8.6: 4 different predefine color palettes applied to the baseline flame.

As mentioned in Section 8.2 *Gamma Correction Background* a non-linear function needs to be applied in order to produce an output flame that approximately replicates the expected image. The gamma correction formula's gamma seen in Section 8.2 is left to be set by the user and is of the *positive float* data type. The 12 different gamma correction values that were applied to the baseline image are shown in Table 8.4.

Flame Number	γ value
1	0.00
2	0.25
3	0.50
4	1.00
5	2.00
6	3.00
7	5.00
8	10.00
9	50.00
10	100.00
11	1,000.00
12	10,000.00

Table 8.4: Flame image numbers and their associated γ correction values.

The resulting images from the altered gamma corrections can be seen in Figure 8.7. The first several images show the effects of when the gamma is set to values that are too low and show the characteristic signs of low gamma which is that the image looks washed out. The last images in the series show the effects of when the gamma is set to values that are too high and show characteristic signs of high gamma which is that the image looks too dark.

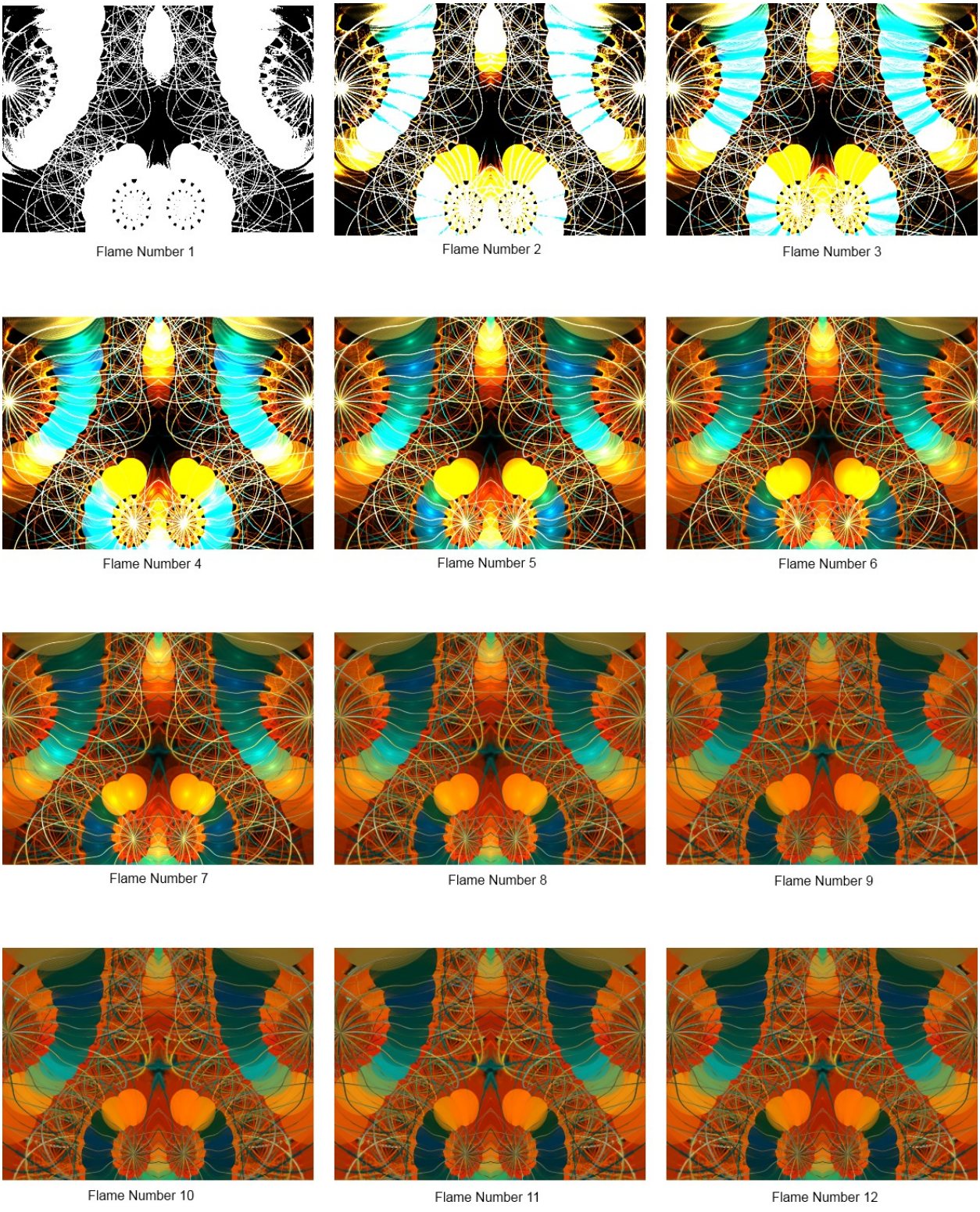


Figure 8.7: 12 different gamma values are presented one the baseline flame.

Gamma Threshold

Gamma Threshold is a parameter setting which controls the threshold for which colors recieve the non-linear gamma correction mentioned above. Colors brighter than the threshold receieve the non-linear correction and colors darker than the threshold receive a linear correction instead [29]. The threshold is a *float* data type value ranging from 0.00 to 1.00 (where 0.00 to 1.00 maps to the entire color space). This parameter can be used to linearly correct certain parts of an image and non-linearly correct others in attempts to produce a greater dynamic range or a stylistic affect. The 12 different gamma threshold values that were applied to the baseline image are show in Table 8.5.

Flame Number	Gamma Threshold value
1	0.00
2	0.05
3	0.10
4	0.20
5	0.30
6	0.40
7	0.50
8	0.60
9	0.70
10	0.80
11	0.90
12	1.00

Table 8.5: Flame image numbers and their associated gamma threshold values.

The resulting images from the altered gamma threshold values can be seen in Figure 8.8.

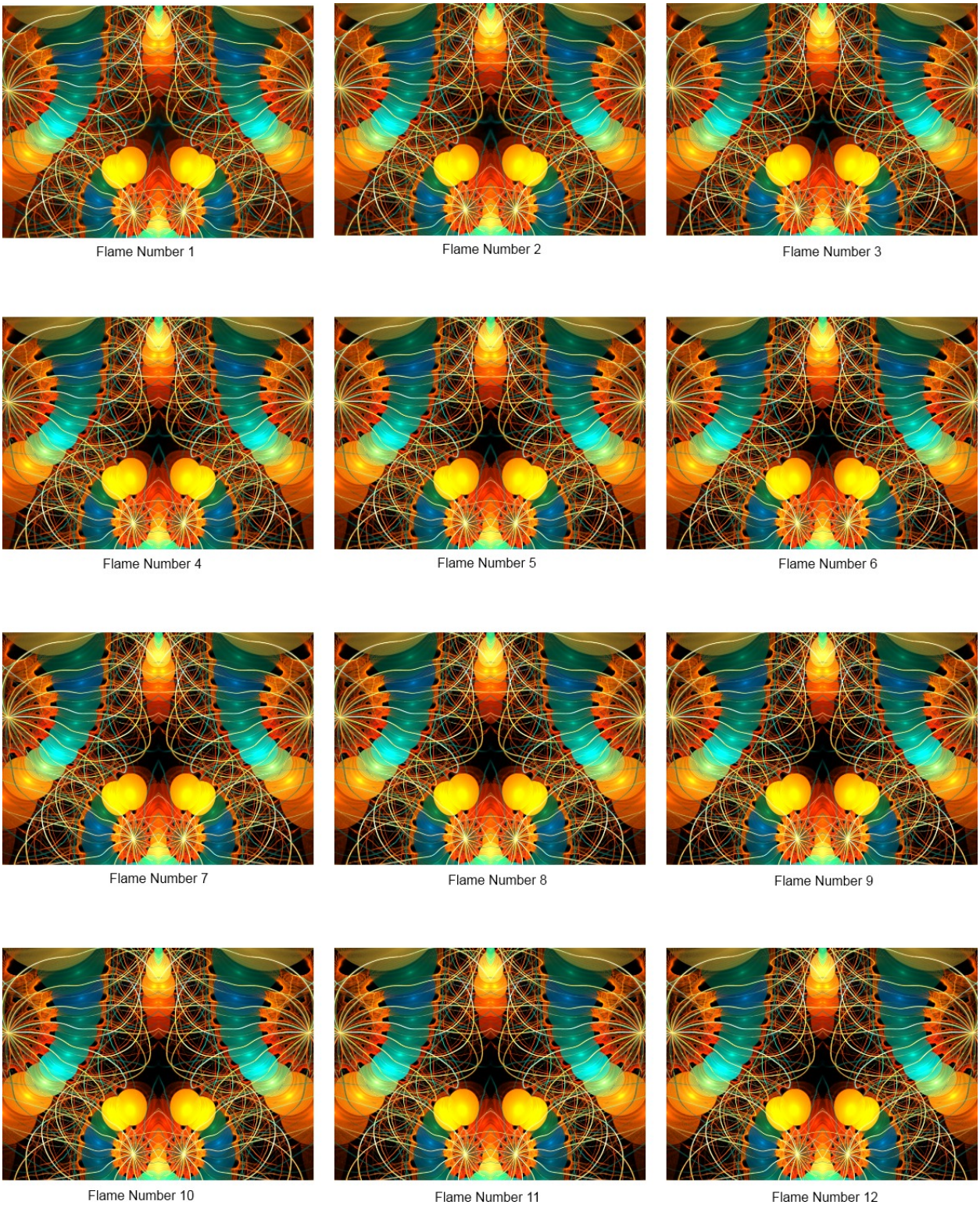


Figure 8.8: 12 different gamma threshold values are presented on the baseline flame.

Hue

Hue is a *float* data type value ranging from 0.00 to 1.00. 0.00 means that the color space is not rotated while 1.00 means there is a 360° rotation in the color space (which effectively is the same as 0.00). Any value in between rotates the color space by a certain degree. The 12 different hue values that provide rotation to the color space are shown in Table 8.6.

Flame Number	Hue	Rotates Color Space By
1	0.0000	$\approx 0^\circ$
2	0.0833	$\approx 30^\circ$
3	0.1666	$\approx 60^\circ$
4	0.2499	$\approx 90^\circ$
5	0.3332	$\approx 120^\circ$
6	0.4165	$\approx 150^\circ$
7	0.4998	$\approx 180^\circ$
8	0.5831	$\approx 210^\circ$
9	0.6664	$\approx 240^\circ$
10	0.7497	$\approx 270^\circ$
11	0.8330	$\approx 300^\circ$
12	0.9163	$\approx 330^\circ$

Table 8.6: Flame image numbers and their associated hue rotation values.

The resulting images from the altered hue values can be seen in Figure 8.9. To properly showcase hue, a light color palette has been applied which will be our new baseline image. This palette can be seen unmodified in Flame Number 1.

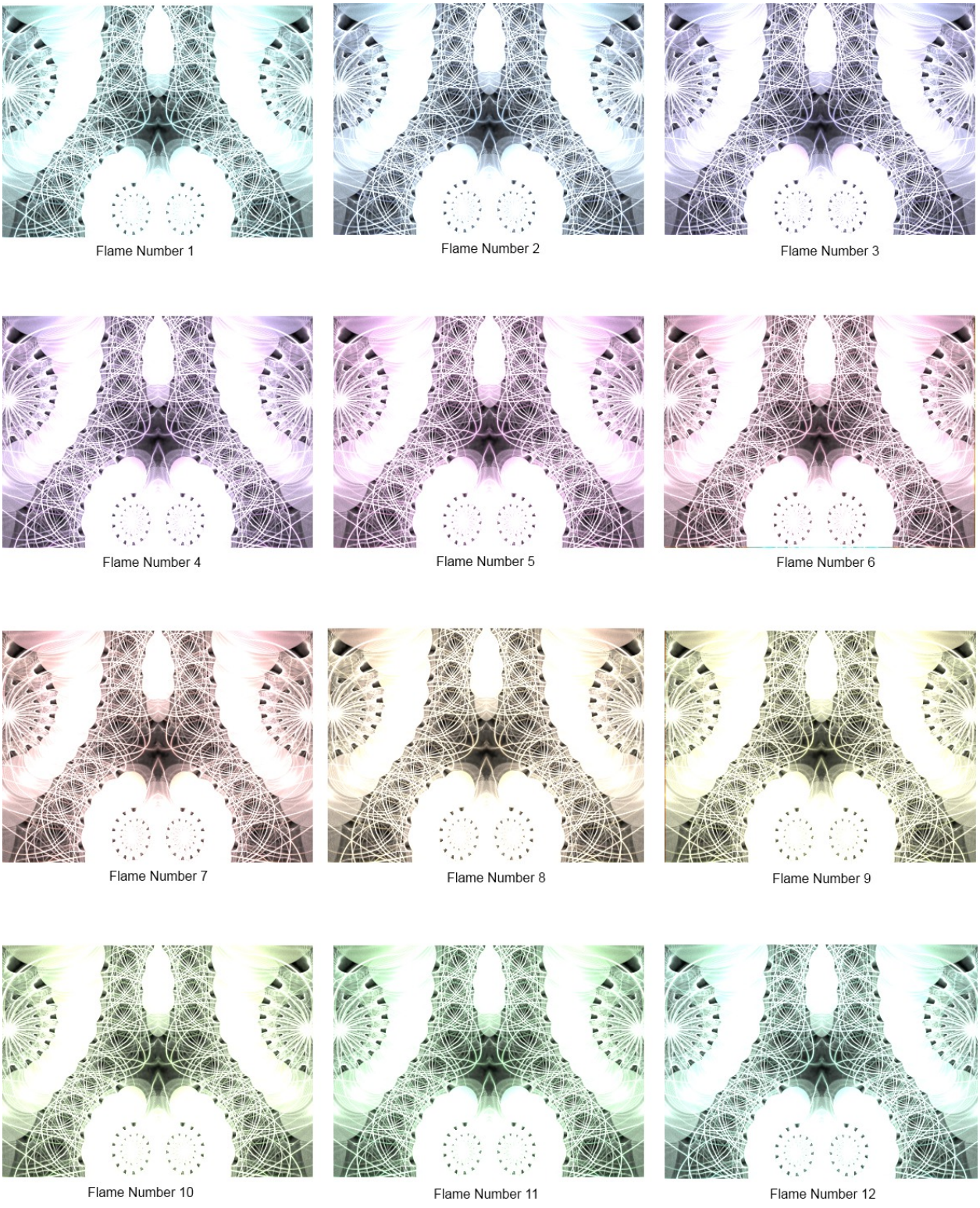


Figure 8.9: 12 different hue values are presented to the baseline flame with a non-vibrant color palette.

Brightness

Brightness correction is a function that changes the percieved intensity of light coming from the image can be enacted upon the image. Additional information is mentioned in Section 8.2: *Brightness Correction Background*. This percieved intensity can be set by the user and is a value of data type: *positive float*. The 12 different brightness correction values that were applied to the baseline image are shown in Table 8.7.

Flame Number	Gamma Correction value
1	0.00
2	0.25
3	0.50
4	1.00
5	5.00
6	10.00
7	25.00
8	50.00
9	100.00
10	1,000.00
11	10,000.00
12	100,000.00

Table 8.7: Flame image numbers and their associated brightness correction values.

The resulting images from the altered brightness corrections can be seen in Figure 8.10. Observe the first and last several images that color clipping occurs. There is absolute light in the flames with the highest brightness correction values (white) and there is an absense of light in the flames with the lowest correction values (black).

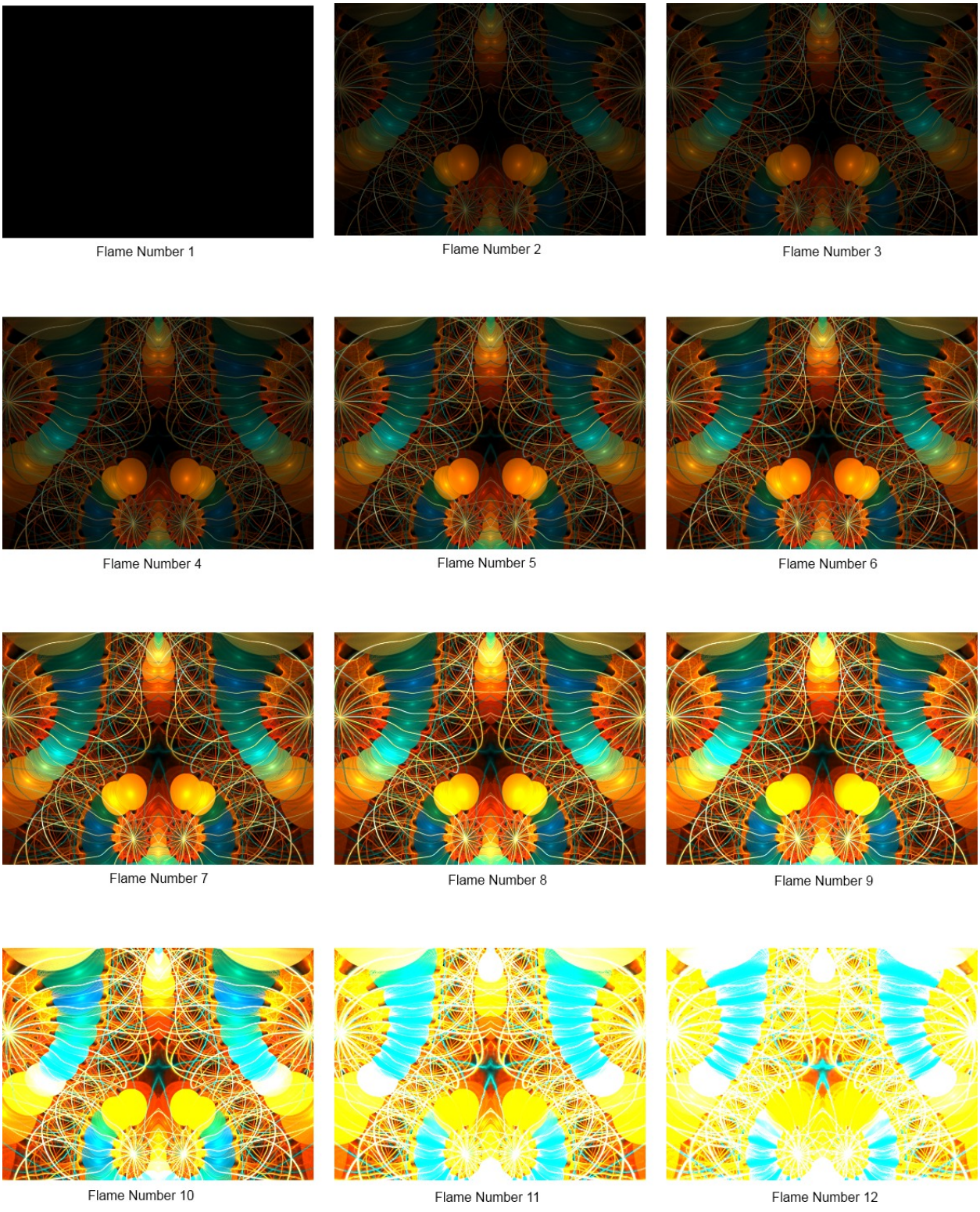


Figure 8.10: 12 brightness correction values are presented on the baseline flame.

Vibrancy

Vibrancy, as stated in Section 8.2: *Vibrancy Background*, provides saturation in a non-linear fashion. In the case of flam3, the actual implementation details to visually produce vibrancy are not found in the common literature. The concept that flam3 uses to alter vibrancy is by what factor the gamma correction should be applied (independently or simulatenously). Vibrancy is a setting in flam3 which the user defines and is a *float* from 0.0 to 1.0. A value of 0.0 denotes to apply gamma correction to each channel independently whereas a value of 1.0 denotes to apply gamma corrections to color channels simulatenously. Applying gamma correction to each channel independently results in *pastel* or *washed out* images of low saturation. Consequently, applying gamma correction to color channels simulatenously results in colors becomming saturated. The 12 different vibrancy values that were applied to the baseline image are show in Table 8.8.

Flame Number	Vibrancy Value
1	0.00
2	0.05
3	0.10
4	0.20
5	0.30
6	0.40
7	0.50
8	0.60
9	0.70
10	0.80
11	0.90
12	1.00

Table 8.8: Flame image numbers and their associated vibrancy values.

The resulting images from the altered vibrancy values can be seen in Figure 8.11.

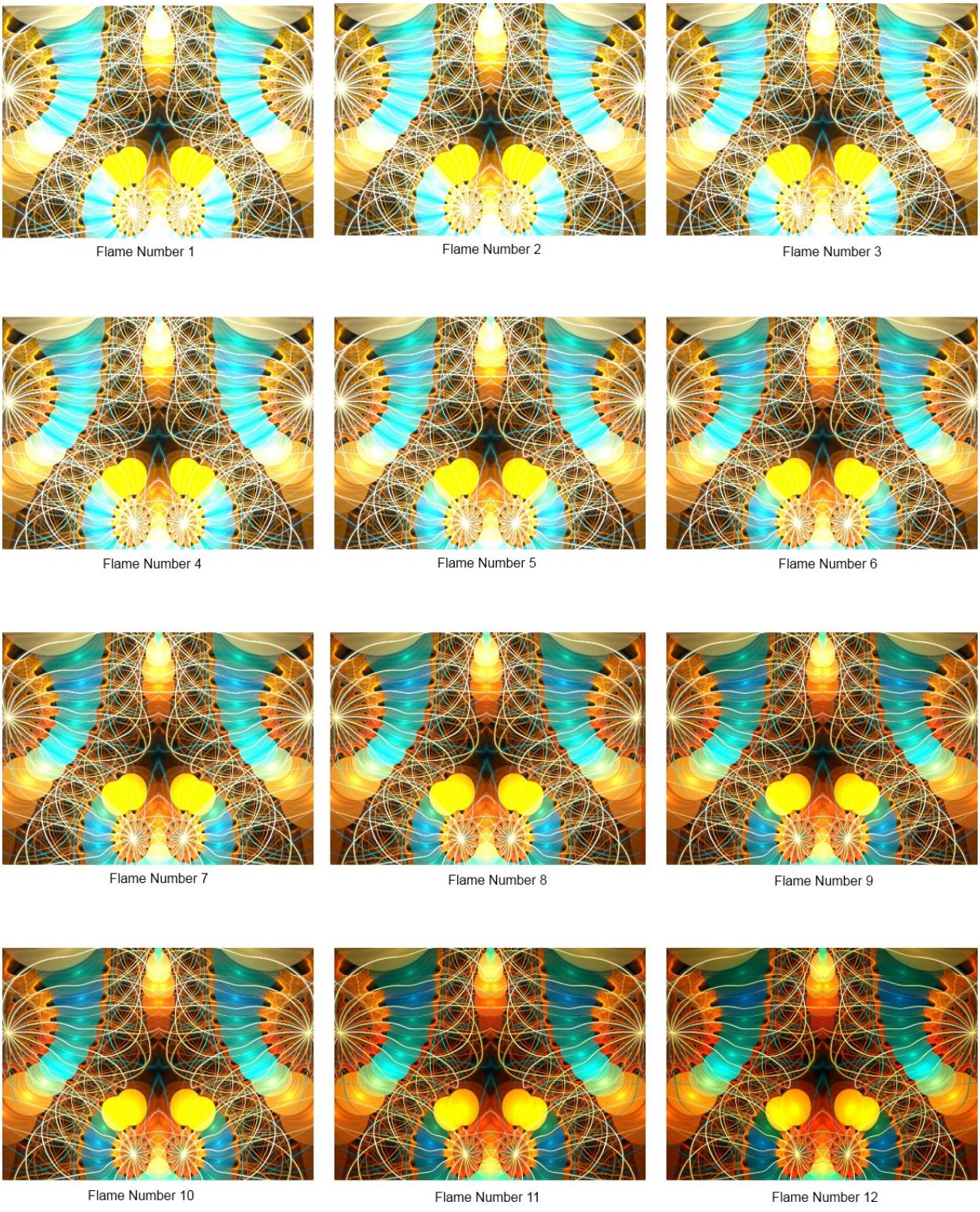


Figure 8.11: 12 different vibrancy values are presented on the baseline flame.

Early Clipping

Earlier it was discussed that the user may experience regions of the flame that become so dense that the colors to fall outside of the representable range of color. These creates regions of uniform density which results in a loss of detail. More background information on this can be found in Section 8.2: *Color Clipping Background*.

Early clip takes this idea of color clipping and provides a means to rectify the problem. The problem occurs because in the typical algorithm all of the log scaled histrogram of points is mapped to the RGB color space *after* applying the filter kernel. A potential problem that can happen is that the spatial filter can blur dense regions of the image and then when color correction techniques are applied these blurred regions can become saturated[29]. Visually, this produces regions that look smeared and more dense than it was intended to look. This deviation between the output image and what was intended is a form of detail loss. The rectification of this problem lies in clipping the RGB color material before applying the filter which fixes this issue. This setting can either be turned on or off and can be set by the user.

Highlight Power

Highlight power is a value (the data type is a *float*) which controls how fast the flame's colors converge to white. The visual effect of this is to blend areas that have drastic color differences that were caused by unintended side effects. The implementation works by keeping the color vector (RGB) pointed in the intended direction until it begins to saturation. When this happens the color starts getting pulled towards white as the iterations continue. A highlight power of 0.0 indicates that saturated colors will not converge to white whereas any value higher than 0.00 is the rate at which saturated colors converge to white[29]. The 12 different highlight power values that were applied to the baseline image are show in Table 8.9.

Flame Number	Highlight Power Value
1	0.00
2	1.00
3	2.00
4	3.00
5	4.00
6	5.00
7	10.00
8	50.00
9	100.00
10	1,000.00
11	10,000.00
12	100,000.00

Table 8.9: Flame image numbers and their associated highlight power values.

The resulting images from the altered highlight power values can be seen in Figure 8.12.

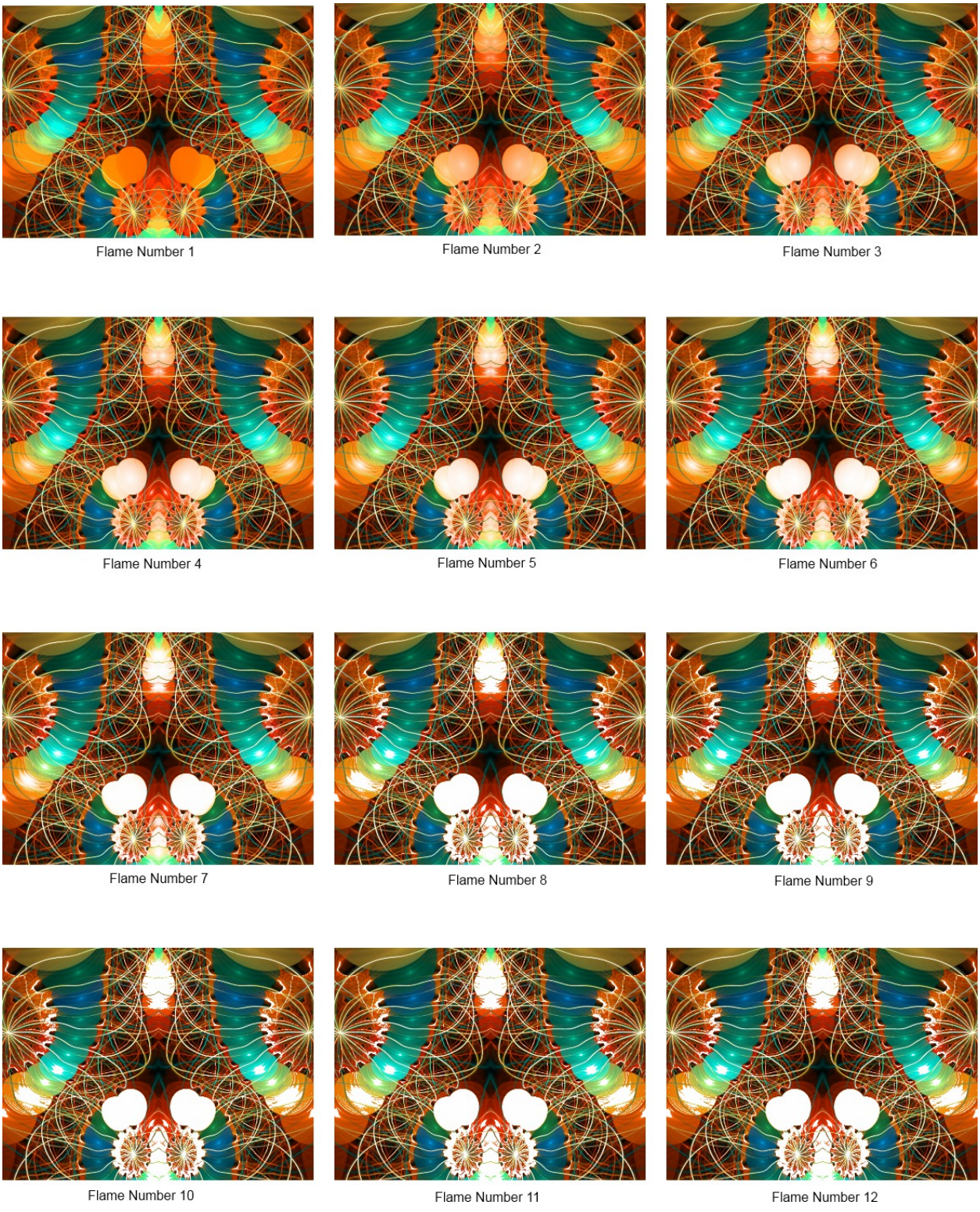


Figure 8.12: 12 different highlight power values are presented on the baseline flame.

8.6 Challenge

The challenge with deviating from the log-scaled and color correction process is that the main criteria of a new approach focuses mainly on approximating a high dynamic range. This is something that log-scaling and color correction has shown to do very well.

One of the complications of log-scaling is that an exponentially more amount of points will be needed in the bright areas than in the darker less dense areas. This results in high quality images needing a whoppingly high number of iterations which increases the run time greatly. This is one of the issues that needs to be addressed in the new implementation.

CHAPTER 9

FILTERING

Image filtering is the process of enhancing an image so that inaccuracies in the image can be corrected. Sources of inaccuracies can be from bad sensor measurements, extremely low or high data ranges, digital misrepresentations, and many other sources. These artifacts can be usually be corrected by blurring pixels together to filter out the inconsistencies. Since image quality is subjective due to human judgement, it may be difficult to determine a "best" filter. However, by identifying the type of inaccuracies and choosing filters suited to chosen criteria, the subjective nature of image enhancement can be decreased such that the resulting images will be a clear improvement over the original.

When rendering flames, there are two kinds of artifacts that need to be minimized in order to create more visually attactive flames: aliasing and noise. While these two problems and their solutions are related, they will need to be approached with different techniques. Much research has been done regarding these problems and their solutions and great improvements have been made over the past couple decades with respect to quality and performance. Just recently, with the increasing popularity of GPGPU computing, new solutions have been proposed to parallelize these algorithms so that significant performance gains can be had exploiting the highly parallel architecture of GPU's. These problems and their many solutions are discussed in more detail below.

9.1 Aliasing

Aliasing is the effect of high frequency signals, in high resolution graphics, being mapped and interpolated onto a lower resolution graphic such that the smooth edges and gradients in the original image can not be represented properly. It is usually observed as distortion or artifacts on lines, edges, and smooth curves. Aliasing occurs when high resolution graphics are mapped to a lower resolution that cannot support the smooth gradients in the original graphic, resulting in an image artifact colloquially known as *jaggies* [30]. See Figure 9.1 for an example.

Figure 9.1: Left: aliased image, right: antialiased image.

Graphical images are at the simplest level a collection of discrete color dots, or pixels, that are displayed on some graphic medium. These pixels are generated, or rendered, from collections of data called fragments.

The data contained in a fragment can include texture, shader, color, Z location, and other such data. Each pixel is made up of one or more fragments, with each fragment representing a triangle. Problems arise when the pixel is sampled from only one fragment in the pixel. This causes all the other data from the other fragments to be lossed and will result in an inaccurate image [31].

Visual image information

In the continouos domain, or at resolutions tending toward infinity, we can describe most flames perceptually as a collection of distinct (though often overlapping) objects with smoothly curved outlines. Our brains perform object recognition all the time - it's hardwired into our visual system - so it's natural that the most visually interesting flames are those which stimulate traditional object recognition pathways in novel ways, rather than, say, white noise.

2D object recognition in our brains depends on recognition of sharp discontinuities in images. Since so much of our neural hardware depends on discontinuities at object boundaries, they become important. However, our algorithm runs in the discrete domain; ultimately the results get sent to monitors. As a result, the perfect curves in the continuous domain must be sampled along the 2D grid of pixels used in raster graphics.

Spatial aliasing

In the flame algorithm, each generated (x, y) point is rounded to the nearest pixel, and then the color value is added to that pixel's accumulator. Effectively, each pixel represents the average value of the color function of the attractor across the area of that pixel. In other words, we sample the color values of the flame once per pixel.

In 2D image space, this means any function with a higher spatial frequency than a single pixel will be aliased by the sampling process. Image discontinuities, such as object edges, are instantaneous, and therefore have an infinite frequency response, though with finite total energy. As a result, object edges are aliased in the spatial domain.

The result is stair-step jaggies in images. Since our brain depends so heavily on detecting object discontinuities for object recognition, these artifacts are extremely noticeable, especially in motion. The solution is to downfilter the highest spatial frequency components below the sampling threshold. Unfortunately, downfiltering any image component, especially surface textures, results in a reduction of detail across the image. Our brains notice artifacts from aliasing at object borders, but also notice reduced detail apart from those regions.

Approaches to antialiasing

Supersample antialiasing

Supersampling is the most trivial method to solving the aliasing problem. It is a relatively naive algorithm and works well but is expensive in terms of resources. Aliasing distortion occurs when continuous objects cannot be represented correctly because of a relatively low sampling rate (resolution) used in the output medium. Supersampling solves this problem by rendering an image at a higher resolution and performing downsampling, using multiple points to calculate the value of a single pixel. Using the average value of multiple samples for one pixel leads to a more accurate color representation for that pixel. The sampling points all lie within the area of a pixel and their location is determined by the type of algorithm.

The number of sampling points is directly related to the desired quality as higher quality filtering will need more sampling points. This directly affects the performance of the filter and is the biggest factor of cost in

antialiasing. Turning on 4x SSAA (4 samples per pixel) will require four times as many samples to rendered, causing the fill rate to be four times longer (meaning animations will have a quarter of the original frame-rate [32]. Going back to sample locations, the specific supersampling algorithm will decide how these samples will be chosen. These algorithms are explained below.

The Ordered Grid algorithm is the most trivial supersampling method. It is the simplest and the fastest of the supersampling algorithms but also offers the least quality. It works by evenly diving a pixel into subsections (like a grid) and then taking the samples from the center of each subsection.

However, because of the samples being extremely regular and lying directly on the axis, the quality of this algorithm will suffer in certain cases. The Rotated Grid algorithm is a similar algorithm designed to offer higher quality filtering with the same performance of the Ordered Grid algorithm. In the Rotated Grid algorithm, the pixels are still evenly divided into regular subsections, but with the samples not lying directly on the axis. This algorithm is similar in performance to the Order Grid algorithm but with significantly improved filter quality.

Other supersampling algorithms exist that randomly chose sample locations with the goal of producing better quality images, but they all have a significant trade-off in regards to performance. There is the purely random algorithm that chooses every sample location randomly and is capable of very good quality, but there is the possible of having pixel locations not being uniformly distributed throughout the pixel area which wil cause inaccurate sampling. The Poisson and Jitter are algorithms that also use random placement while focussing on having uniform sample distribution.

The Poisson algorithm divides the pixel into subsections, similarly to how the Ordered Grid algorithm creates subsections, then chooses the samples by randomly selecting a point inside each subsection instead of always sampling at the center. The Jitter algorithm determines sample locations by choosing all samples purely by random, then looks for and throws out samples that are too close together, and then randomly chooses more samples until all samples are far enough away from each other [32]. See Figure 9.2 for visual depictions of these algorithms.

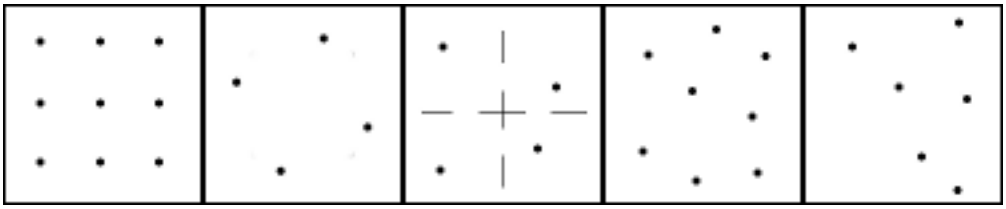


Figure 9.2: From left to right: Ordered Grid algorithm, Rotated Grid algorithm, Jitter algorithm, Poisson algorithm, Random algorithm.

Multisample antialiasing

MSAA, also known as full scene antialiasing, is a special case of supersampling where not all of the components of a pixel are supersampled. This algorithm can achieve near supersampling quality at a much higher performance. Pixels are generated using a collection of data called a fragments and may include raster position, depth, interpolated attributes, stencil, and alpha. Multisampling algorithms select only a few components of a fragment to "supersample" so that some of that computational cost can be shared between samples. Commonly, z-buffer, stencil, and/or color data is chosen to be the fully supersampled components [33].

Coverage antialiasing

CSAA is a special case of multisample aliasing, and therefore also a special case of supersample aliasing. The algorithm has been designed to further improve the performance of multisample antialiasing while keeping quality as high as possible. Multisample antialiasing will usually store only one value for texture and shader samples for an entire pixel. This is also true for coverage antialiasing but we take it a step further and limit the number of stored color and Z data samples. Coverage antialiasing can store more than a single value for the color and Z data, the point is to just hold less than multisampling. Usually, 4 or 8 color and Z data samples are used as opposed to 8 and 16, respectively. Holding more data constant allows for an even smaller memory footprint and less bandwidth [34].

Coverage sample points are boolean values that indicate whether or not a sample is covered by a triangle in the pixel. These samples are stored usually stored as 4 bit data structures with 1 bit representing the boolean value and with the other 3 bits used to index up to 8 color/Z values. The 8 bytes required for 16 samples will be much less then the memory needed for the color data so the extra overhead should be insignificant compared to the bandwidth reduction [34].

Morphological antialiasing

MLAA is a significantly different antialiasing approach. It does not rely on supersampling and it takes place post-processing. It works by blending colors after looking for and recognizing special pixel patterns in an image. The algorithm can be explained using the following steps [35]:

1. Look for discontinuities in an image. Scan through all adjacent rows and columns and store the lines where a disconituity is found. Edges of the images are extended so that unnecessary blending does not occur around the borders of the image.
2. Identify special pixel patterns. San through the list of discontinuous edges and identify crossing orthogonal lines. These locations will mark an area for one of three predefined pixel patterns: the Z-shaped pattern, the U-shaped pattern, and the L-shaped pattern.
3. Blend colors in pattern areas. The pixels that make make up the vertices of the identified patterns are sampled and blended together.

Notice that more samples do not have to be rendered when using morphological antialiasing. The computational resources required to do the above steps are far less than the resources needed to render 4, 8, or even 16 times as many pixels. Supersampling will generally produce slightly higher quality results but will not be worth the performance trade-off, especially if real-time rendering is needed [35].

9.2 Denoising

Antialiasing deals with the problems caused by approximating objects via sampling along a regular 2D grid. Denoising, by contrast, deals with the problems caused by approximating objects via random sampling. Image noise is one of the most common and studied problems in image processing. Noise occurs as seemingly random, unwanted pixel inaccuracies as collected by an image source (commonly a camera, in our case, approximating objects via random sampling). Most image denoising algorithms deal with this problem by treating noise the same as small details and then by removing all the small details with some form of blurring. This is done by replacing a pixel with a weighted average of all the nearby pixels [36].

The origins of noise

Sampling noise from Monte Carlo IFS estimation arises from two main sources: coverage limitations and accuracy errors.

- Because we don't know the shape of the attractor analytically, we can't sample it directly; we must follow it along the IFS. We use random sampling to approximate the IFS with Monte Carlo methods. This means that the IFS will jump around from location to location within the image in a generally unpredictable pattern. Because of this jumping, any errors in the image show up as point noise, rather than along contours as with aliasing.
- Again, since we don't know the shape of the attractor, we choose random points to start with. After picking a new random point, a thread runs a few iterations without recording any data so that the point can join the main body of the attractor. However, this number may sometimes be insufficient, leading to random points placed "outside" the attractor. Floating-point precision errors can similarly reduce the accuracy of generated points.

Visibility

Both of these sources of error have something in common, though: they show up a lot more in darker image regions. The image is log-filtered, meaning the brightest image regions are covered by hundreds or even thousands of times more samples than the darkest. In many images, log scaling parameters such as contrast, brightness, and gamma cause extreme sensitivity in dark regions, so that a single sample in the middle of an otherwise-black image region corresponds to a final, filtered pixel value whose value is an appreciable fraction of the total luminance scale of the final image. In instances where this noise is sparsely distributed, this effect can be extremely noticeable.

This phenomenon extends to all accumulated samples which are significantly amplified by the log scaling process. Generally, this only applies to samples with a value well below the mean value across the image. However, in cases where most image energy is concentrated in small sample regions, the mean value can be well above the median value, allowing this speckle to be distributed throughout a large portion of the image.

Denoising a flame

To remove noise, flam3 does density estimation filtering. This means that darker regions, or regions with fewer samples, are filtered with a smaller blur. This section covers denoising algorithms, including the Adaptive Density Estimation Filter employed by the standard flam3 implementation as well as new techniques for accelerating denoising algorithms on GPU's.

Adaptive Density Estimation Filter

The adaptive density estimation filter used by flam3 is a simplified algorithm of the methods presented in Adaptive Filtering for Progressive Monte Carlo Image Rendering [2]. The algorithm creates a 2 dimensional histogram with each pixel representing a bin. For each sample located in the spatial area of a pixel, the value for that bin is incremented. Kernel estimation is then used to blur the image, with the size of the kernel being related to the number of iterations in a bin.

Lower number of iterations in a bin (low sample density areas) lead to larger kernel sizes and increased blurring. Higher number of interations in a bin (high sample density areas) lead to smaller kernel sizes and decreased blurring [37]. Specifically, the kernel width can be determined by the following relationship:

$$KernelWidth = \frac{MaxKernelRadius}{Density^{Alpha}}$$

The $MaxKernelRadius$ and $Alpha$ values are determined by the user as they are properties of the flame. MaxKernelRadius tells the algorithm the maximum width that the kernel can be and the Alpha value determines the estimator curve to use. The ability to adjust the width of the kernel according to how many samples there are spatially increases the quality of the image by limiting the blur in the more accurate areas with higher sample density. [37]

Gaussian Convolution

Gaussian convolution filtering is a weighted average of the intensity of the adjacent positions with a weight decreasing with the spatial distance to the center position p. The strength of the influence depends on the spatial distance between the pixels and not their values. For instance, a bright pixel has a strong influence over an adjacent dark pixel although these two pixel values are quite different. As a result, image edges are blurred because pixels across discontinuities are averaged together [38].

Bilateral Filter

The bilateral filter is also defined as a weighted average of nearby pixels, in a manner very similar to the Gaussian convolution filter described above. The difference is that the bilateral filter takes into account the difference in value with the neighbors to preserve edges while smoothing. The key idea of the bilateral filter is that for a pixel to influence another pixel, it should not only occupy a nearby location but also have a similar value [38].

The bilateral filter is controlled by two parameters: σ_s and σ_r. Increasing the spatial parameter, σ_s, smooths larger features. Increasing the range parameter, σ_r, makes the filter approximate the Gaussian convolution filter more closely. An important characteristic of this filter is that the parameter weights are multiplied; no smoothing will occur with either of these parameters being near zero. [38]

Iterations can be used to generate smoother images similar to increasing the range parameter, except for being able to preserve strong edges. Iterating tends to remove the weaker details in a signal or image and is desirable for applications such as stylization that seek to abstract away the small details. Computational photography techniques tend to use a single iteration to be closer to the original image content [38].

Nonlocal Means

The nonlocal means (NL-Means) algorithm is a relatively new solution to the image noise problem. Unlike most other algorithms that assume spatial regularity, the nonlocal means filter looks for and exploits spatial geometric patterns. It will only use pixels that match the geometic correlation in the local area causing irregular image noise to be canceled out. This means a more accurate color selection for the pixel in question. [39]

Permutohedral Lattice

The permutohedral lattice is a data structured designed to improve the performance of high-dim-ensional Gaussian filters including bilateral filtering and nonlocal means filtering. It is a projection of the scaled grid $(d+1)\mathbb{Z}^{d+1}$ along the vector $\vec{1} = [1, ..., 1]$ onto the hyperplane $H_d : \vec{x}.\vec{1} = 0$ and is spanned by the projection of the standard basis for $(d+1)\mathbb{Z}^{d+1}$ onto H_d [40]. Each of the columns of B_d are basis vectors

whose coordinates sum to zero and have a consistent remainder modulo $d + 1$, which is how points on the lattice are determined; the lattice point coordinates have a sum of zero and remainder modulo $d + 1$.

Lattice points with a remainder of k can be described as a "remainder-k" point. The algorithm works by placing pixel values in a high-dimensional space, performing the blur in that space, then sampling the values at their original locations. These three steps are often referred to as splatting, blurring, and splicing, respectively [40].

Using a permutohedral lattice for n values in d dimensions results in a space complexity in the order of $O(dn)$ and a time complexity of $O(d^2n)$. According to Adams et al [40], algorithms based on using the permutohedral lattice are fast enough to do bilateral filtering in real time. There are four major steps in algorithms that use the permutohedral lattice.

First, position vectors for all the locations in high-dimensional space must be generated and stored in the lattice. Generating the position vectors for the lattice has a time complexity of $O(d)$. Secondly, splatting is performed by moving pixels onto the vertices of their enclosing simplex using barycentric weights. Splatting has a time complexity of $O(d^2n)$.

The next step is the blurring stage which convoles a kernel in each lattice dimension and is performed in $O(d^2l)$. The final step is the slicing stage which is similar to the splatting stage, except done in reverse order; barycentric weights are used to pull pixel values out of the permutohedral lattice. The entire algorithm has a time complexity of $O(d^2(n + l))$ [40].

Gaussian KD-Trees

The Gaussian filter, bilateral filter, and nonlocal means filters are non-linear filters whose performance can be accelerated by the use of Gaussian kd-trees. All of these filters can be expressed by values with positions. The Gaussian filter can be described as a pixel color being the value with coordinate position (x,y). The bilater filter can be describved as a pixel color with coordinate position (x,y,r,g,b). The nonlocal means filter can be described as a pixel color with position relative to a patch color around the pixel.

The Gaussian kd-tree algorithm treats these structures similarly in that it assigns all the values in an image to some position in vector space and then replaces each of the values with a weighted linear combination of values with respect to distance. By representing these images by a kd-tree data structure, the space and time complexity can be decreased significantly. These algorithms typically have a complexy of $O(d^n)$ or $O(n^2)$ whereas the kd-tree algorithm will have a space complexity of $O(dn)$ and a time complexity of $O(dn \log n)$ [41].

A kd-tree is a binary tree data structure used to store a finite number of points from a k-dimensional space [42]. Each leaf stores one point and each inner node represents a d-dimensional rectangular cell [41]. The inner node stores the dimension n_d in which it cuts, value n_{cut} on the dimension to cut along, the bounds of the dimension n_{min} and n_{max}, and pointers to its children n_{left} and n_{right} [41]. For this implementation of the kd-tree, n_{min} and n_{max} have been added in addition to the standard data structure.

There are two main steps associated with these accelerated Gaussian kd-tree algorithms. First, the tree must be built. Generally, the tree should be built with the goal of minimizing query time. In each leaf node is as likely to be accessed as any other leaf node, the kd-tree should ideally be balanced. Building a balanced tree can be accomplished by finding the bounding box of all the points being looked at, finding the diagonal length of the box, and if that length is less than the standard deviation, a leaf node is created and a point is set for the center of the bounding box. If the length is not less than the standard deviation, split the box in the middle along the longest dimension and continue recursively. The building of a tree is expected to have a time complexity of $O(nd \log m)$ with m being the number of leaf nodes.

The second step in the algorithm is querying the tree. Queries are used to find all the values and their weights given a position. To be specific, a query should take in the pixel location, a standard deviation

distance, and the maximum number of samples that should be returned. The query will then find and return all the values and weights of pixels around that pixel, up to the standard deviation and maximum number of samples. The complexity of performing queries is expected to be $O(dn \log n)$ [41].

The advantage of using Gaussian kd-trees to improve these algorithms is that not only is it faster serially but can have portions of it parallelized over a GPU. The tree building portion of the algorithm relies on recursion which is not ideal for GPU's because of having no stack space, though it can be converted to an interative algorithm but that will not give us any more performance. But, the querying portion of the algorithm — where most of the computation time comes from — is highly parallelizable.

9.3 Spatial Filtering

A *window function* is a mathematical function that is zero-valued outside of some chosen interval while manipulating the values inside that interval. The simplest window is the rectangular window. It simply takes a chunk the portion of the signal fitting inside in the window leaving discontinuities at the edges (unless the signal is entirely within the limits of the window). Filter shapes available in flam3 are the Guassian (default), Bell, Blackman, Box, Bspline, Hamming, Hanning, Hermite, Mitchell, Quadratic, and Triangle [10]. See Figure 9.3 for fractal flame image without filtering and Figure 9.4 for fractal flame image with Gaussian filtering.

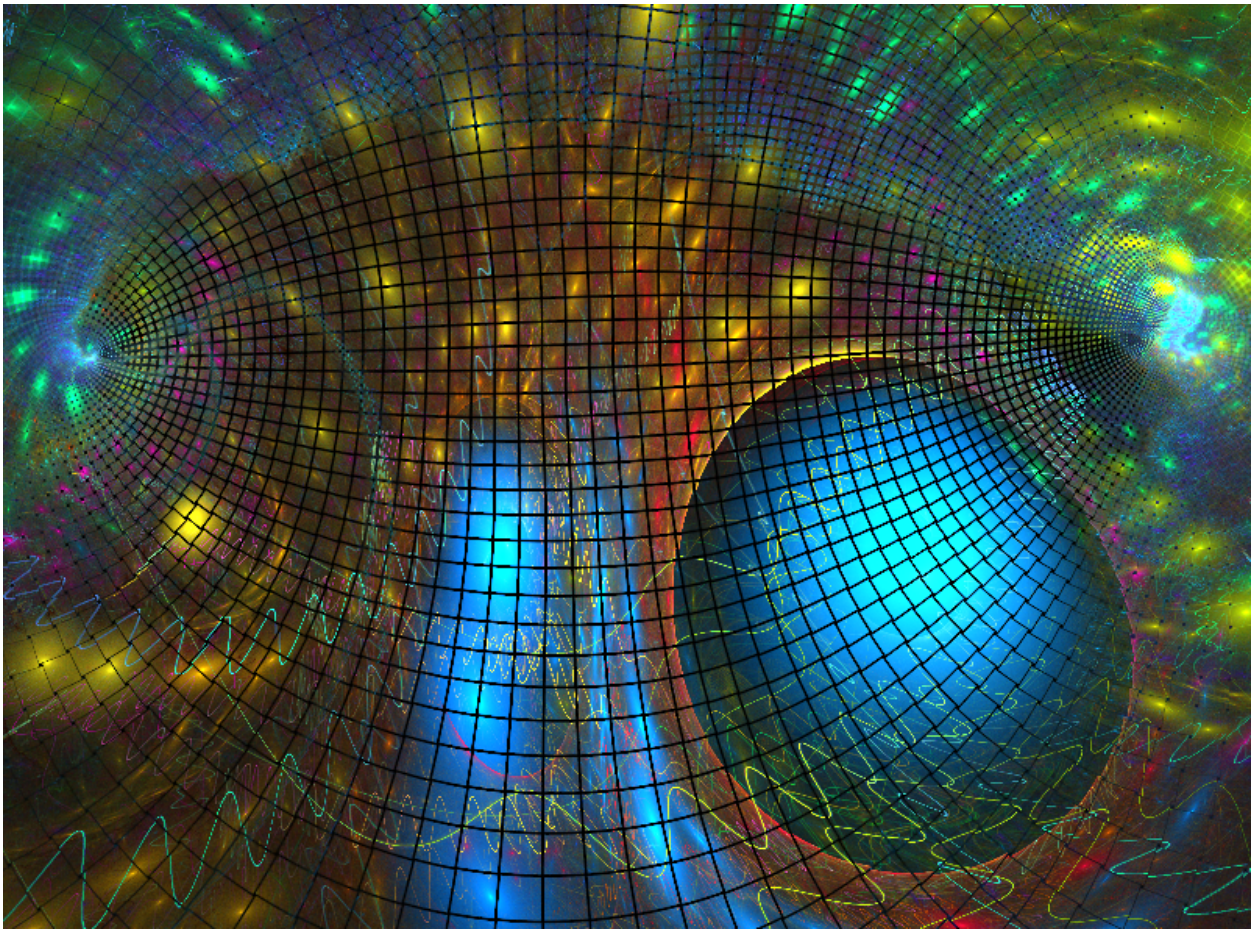


Figure 9.3: Electric Sheep 244 36724 fractal flame with no filtering.

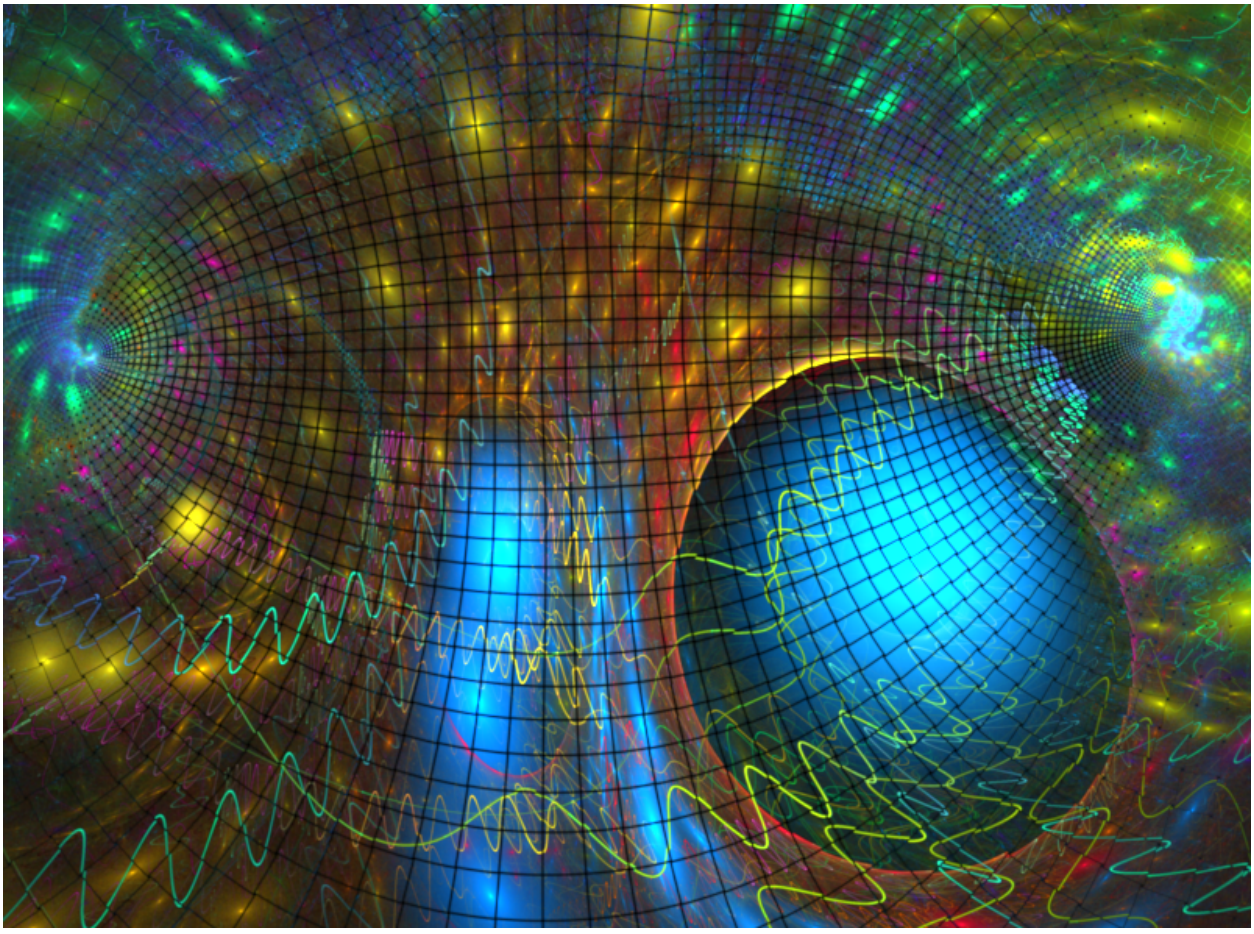


Figure 9.4: Electric Sheep 244 36724 fractal flame with Gaussian filtering.

9.4 Motion Blurring

Motion blur is the visual effect that occurs when a moving object is caputered as an image over a certain period of time rather than a single point in time. It appears as smear or blur in the direction of movement. Motion blurring occurs in all physically captured images to some degree, although it's effect can be significantly diminished with extermely low shutter speeds. It occurs naturally in human eyesight as well; try waving your finger back in forth in front of your face, does it look blurry? Yes, it does.

This is the way we percieve the world and it doesn't usually occur to us consciously, our brain takes care of deblurring the image for us. In fact, it is so much a part of the way we percieve the world such that if we didn't see a motion blur, the moving object would appear jerky in motion. This is a problem with animated, computer generated sequences. Each frame is rendered at a single, definite point in time and contains absolutely no blurring due to motion. Even at relatively high frame rates, this can cause the animation to look jerky.

CHAPTER 10

DYNAMIC KERNEL GENERATION

The current version of `flam3` has nearly 100 variations. A transform function is composed of an initial affine transform, followed by application of each of the variation functions to the result of the initial affine transform. A weighted sum is then performed on the results of the variation applications to get the transformed point.

In `flam3`, a list of those variations with non-zero weights is generated before iteration begins, and only those variations are computed. This is implemented in-loop by a very large `if-else` block. Some compilers will optimize this `if-else` block by turning it into an indirect branch situation, and most modern CPUs will perform branch prediction to accelerate the fall-through of this structure.

The current version of NVIDIA's CUDA platform, however, does not fully implement indirect branching, so a switch statement cannot be used. Direct branching, even in non-divergent situations, is still expensive; it takes time to perform instruction fetches and update the scoreboard for each warp after a branch. With the entire set of variations in place, the single most computationally expensive component of an iteration for most flames would actually be this `if-else` cascade.

The transforms aren't the only feature guarded by conditionals. In fact, the data structure which controls which features are enabled and disabled for a particular flame can grow to over 64KB in size. Motion blur in `flam3` is handled by interpolating a default of 1000 control points in a tight temporal neighborhood around the current frame's timestamp in an animation; these control structures can easily occupy 64MB of memory on the GPU. At that size, keeping the relevant parts of a control structure in local cache is unlikely.

The `flam3` iteration loop just has too many options to operate quickly on GPU. Previous efforts to port the flame algorithm to GPU have all encountered this problem, although solutions vary: `flam4CUDA` and Fractron 9000 simply omit parts of the algorithm which are required for full compatibility, whereas `flam4OCL` attempts to construct the source of a kernel which is tuned for the flame at hand.

This project requires full `flam3` compatibility and high performance; the often-disabled parts of the algorithm cannot be omitted, but the performance hit from having dead code in a kernel cannot be ignored. Dynamic creation of a rendering kernel seems the most promising option.

10.1 Just-in-time compilation

A profusion of rarely-enabled legacy code, experimental hacks, compatibility tweaks, and alternate approaches — a result of more than a decade of enthusiastic participation by the fractal flame community — complicates and slows rendering. Fortunately, between environment configuration and the genomes themselves, it is possible to detect these features' presence before rendering begins.

Animations in `flam3` are composed from sequences of *loops* and *edges*. A loop is a frame sequence where a subset of the parameters are rotated over time, returning to their original values at the end of the rotation. Edges are rendered by interpolating every parameter between two visually distinct flames. The desired effect of an edge is a slow morphing from one shape to another.

The interpolation functions exported by the `flam3` library provide several guarantees about which features will be enabled in one of these kinds of animations. While some parameters, such as image size and filter support width, require special handling, the general rule is that every frame of an animiation will use exactly the union of the sets of features used by both end-points. If instead of providing a rendering kernel with the host code, a template for producing a kernel is provided, an example control point taken from near the center of an interpolated sequence can be used to to tune the templated code and produce a kernel optimized for every frame in the animation.

The task of compiling such a kernel is not as complex as it may sound. To make it possible for new GPU designers to enter a market with devices capable of running existing software, and for established manufacturers to produce new hardware without needing to support their own legacy instruction set architectures, OpenGL requires that shaders be shipped in the same C-like syntax in which they are typically written and compiled on the host system. OpenCL preserves this requirement for its more general kernels. This has led to the inclusion of extremely fast, high-quality compilers within major GPU manufacturers' drivers.

CUDA's model is somewhat different. Because CUDA kernels can take advantage of certain C++ features, a small and fast compiler is much harder to obtain. As a result, CUDA kernels are stored in PTX, a RISC-like intermediate language, and assembled for the target architecture on the host. This has the advantages of being faster than OpenCL's subset of C to compile, and offering additional hardware control; however, because it is very low-level, it is difficult to write large amounts of code directly in PTX.

In either case, the compiler and assembler is located on the client system with a simple, standard API; there is no need to include a development environment with the resulting binary or write our own assembler.

10.2 Dynamic code, static types

Both OpenCL and CUDA provide mechanisms for loading dynamically-generated code, which is used to exclude unused features from the final kernel. On its own, however, this does not exclude unused data from the control point structure; this information will still be sparsely distributed throughout the control point, spread over many additional cache lines, and is thus more likely both to evict needed data and be evicted itself.

To use cache lines more efficiently and avoid costly misses, the data may be packed according to the pattern of its use. For dynamically generated code, a simple means of doing this is to use a stack-like structure, where a pointer advances or rewinds in a memory region with code flow, and each block of code carries the total size of its data so that it may properly reset the pointer at entrance and exit points.

This mechanism works well for small dynamically-generated systems, but is challenging for larger ones: The host logic which packs the stack must be kept precisely in line with the control flow of the device code, and since the contents of the stack can affect program flow, any bugs on the device which result in mis-positioning of the stack pointer can be exceptionally hard to isolate. Therefore, fixed address offsets into the stack are preferred. This simplifies debugging on the device and adds reliability, but still requires that the front-end follow the flow of device code.

Instead of attempting to replicate the control flow of the templated code following feature analysis, it would be more efficient to combine both stages; have the information needed to perform memory packing on the

host in the same location within the program source as the description of the device-side operations to load it. Shard was written as part of this project to provide this ability.

Shard is an embedded domain-specific language for the dynamic creation of GPGPU programs, written in Haskell. As an EDSL, Shard uses native language syntax to record operations rather than perform them. Consider the statement `y = 4 * x + 1`. If `x` is a number, this statement would instruct the Haskell compiler to emit code that stores the value of $4 \cdot x + 1$ in the memory reserved for variable `y`. As a Shard variable, however, this statement instructs the compiler to store a data structure representing the calculation in `y`, which can itself then be used as another Shard variable. In this way, each new expression adds to a tree of operations that can later be evaluated to produce device code.

Shard includes expressions that assist with dynamic memory management. A `loadHost` statement in Shard takes a base pointer and a host-side expression which returns a value. When the Shard data structure is evaluated, all such statements are analyzed to determine the offsets of each value in the stack, so that the actual code emitted on the device contains the appropriate offset. To pack the stack for use on the GPU, each host-side expression is applied to the corresponding data structure automatically. Because the host and device commands are contained in the same expression, Shard eliminates the risk of code drift between memory packing and loading.

Haskell's strong static typing provides additional guarantees on the correctness of this approach. In many languages, the two arguments of the `loadHost` function would be indistinguishable from one another and from any invocation. Shard uses phantom types to avoid this; the base pointer, closure, and function result must all be consistent. Despite representing a primitive on the device, the result of this expression preserves its type from the host code, so that a program will refuse to compile even if two values with different meanings but the same OpenCL or CUDA type are used together. This checking happens entirely at compile-time, and carries no run-time overhead.

10.3 Testing — or lack thereof

Haskell's type system is not limited to ensuring variables are not erroneously switched; it can provide deeper guarantees of code correctness and interchangeability.

Because the number of independent selectively-enabled code segments required for a full implementation of `flam3` is so large, it is effectively impossible to test even a fraction of every permutation of features. This enormous parameter space compounds the similarly massive parameter space of the genomes themselves.

When it is not possible to fully cover code with holistic tests, unit testing is often used, wherein each component is exercised separately, and to use strict modularity to ensure that side effects from function calls are contained. However, this modularity is expensive, particularly on GPUs, and would preclude certain optimizations that are not localized to particular regions of code. Without guarantees of modularity, unit testing cannot discover the insidious bugs arising from interactions between different combinations of parameters.

For this reason, Shard requires explicit and granular denotation of impure code in function type signatures. Certain operations which have side effects — that is, have the potential to modify the operating environment of the device — have their return types wrapped in such a way that the results can only be accessed inside of functions that are themselves similarly wrapped. Because this is embedded in the type system, these checks are evaluated at compile-time, meaning that it is enforced in all code that has the potential to be enabled on the device. This prevents hidden interactions against shared state for all permutations of the rendering kernel without having to test them discretely.

This renderer is designed to hew closely to `flam3`'s visual output. To provide a holistic test, a selection of flames are rendered on CPU and GPU and compared; if no significant deviations are encountered, the code is

working as expected. It is reasonable to attain complete code coverage in this way, wherein every segment of device code is exercised at least once. Such testing cannot provide assurances on hidden interaction errors, however, and is no subsitute for strong type safety.

Note that Shard remains under heavy development; as a result, syntax examples and implementation details are not yet available. Several fully-working prototype implementations, as well as libraries that use similar methods in different domains, provide strong evidence that the concept is viable. Further revisions are focused on finding the correct balance between strictness of type system verification and conciseness of expressed code.

CHAPTER 11

FUNCTION SELECTION

The GPU relies on vectorization to attain high performance. As a result, divergent warps carry a heavy performance penalty; even one divergent thread in a warp can double the amount of time evaluating the results of an operation that includes branching. Avoiding unnecessary branches within a warp is therefore an important performance optimization.

For each iteration of the IFS, one function of the flame is randomly selected to transform the current point. This poses a problem: if the algorithm relies on the random selection of transforms for each point, threads may select different transforms and therefore become divergent. With a maximum of 12 variations per flame, this leads to a worst-case branch divergence penalty of an order of magnitude on the most computationally complex component of an iteration.

11.1 Divergence is bad, so convergence is... worse?

The trivial solution to this problem is to eliminate divergence on a per-warp basis. The typical design pattern for accomplishing this is to evaluate the branch condition in only one lane of the warp, and broadcast it to the other threads using shared memory; in this case, have the thread in lane 0 broadcast the `xform_sel` value generated from the RNG. Each thread in a warp will then proceed to run its own point through the same transform.

This neatly resolves the problem of warp divergence, bringing the samples-per-second performance of flames with large numbers of transforms closer to that of simpler transforms. However, inspect the output of this modified engine and it becomes clear that visual quality suffers; in fact, subjective measurement shows that this change actually *decreases* overall quality-per-second[1]. The illustrated change has no effect on the transform functions themselves, or on any other part of an individual thread — from the perspective of a sample passing through the attractor, both variants are identical. Where, then, is this drop in quality coming from?

Recall that a necessary condition for stability in a traditional iterated function system is that each transform function is *contractive*, and that this is at least approximately true for the flame algorithm as well. Each successive application of a contractive function to a point reduces the distance between the point and the function's fixed point. In the system modified to iterate without divergence, each thread continues to select a new transform each time it is chosen, and this behavior prevents the system as a whole from converging on a fixed point.

[1] This information was gathered by one of the authors using the earliest GPU implementation, which no longer runs on current hardware, so example images are not available until our renderer is complete.

However, since each thread in a warp applies the same transform, each application brings every point in the warp closer to the same fixed point, and therefore to the other points in the warp. It doesn't matter that the next transform will have a different fixed point; the same effect will happen. While the points won't converge to a single fixed point across the image, they will quickly converge on one another. The precision of the accumulation grid is relatively low, even with FSAA active, so that after only a few rounds each of the 32 threads in the warp is effectively running the same point. Despite computing each instruction across all threads, the vectorized hardware produces no more image information than a single thread.

While a sequence of contractive functions applied to any two disparate points will cause those points to converge, the amount of convergence depends both on the length of the sequence and the contractiveness of those functions. Because the images have limited resolution, any sequence which reduced variability between disparate points below the sampling resolution[2] of the image grid would effectively "reset" the point system each time it was encountered, resulting in an image with substantially reduced variation. Since short-length subsequences of transforms are likely to be encountered in a high-quality render, we can reason that flame designers typically reject genomes whose transform sequences are overly contractive.

It is therefore not necessary to ensure that every instance of the system under simulation receive an entirely independent sequence of transforms; rather, it is sufficient to limit the expected frequency of identical transform subsequences across threads. Fortunately, there's a simple and computationally efficient way to do this while retaining non-divergent transform application across warps — simply swap points between different warps after each iteration.

11.2 Doing the twist (in hardware)

There is no efficient way to implement a global swap, where any thread on the chip can exchange with any other; architectural limits make global synchronization impossible, and an asynchronous path would further burden an already overworked cache (see below). Instead, data can be exchanged between threads in a work-group by placing it in shared memory, using barriers to prevent overwriting data.

To conduct such a swap on a Fermi core, each warp of a work-group issues a barrier instruction after writing the results of an iteration to the global framebuffer. The barrier instruction adds a value to an architectural register, then adds a dependency on that register to the scoreboard, preventing further instructions from that warp from being issued until the barrier register reaches a certain value, after which it is reset. Multiple registers (up to 8) are available for implementing complex, multi-stage synchronization, but as with all addressable resources in a Fermi core, they are locked at warp startup, so overallocation will reduce occupancy.

After reaching this barrier, all threads write one value in the state to a particular location in shared memory, then issue another barrier instruction. Once the next barrier is reached, indicating that values have been written across all threads, each thread reads one of the values from another location. If further data must be exchanged, another barrier is issued and the process repeats; otherwise, each warp proceeds as usual.

The choice of location for each read and write is, of course, not arbitrary, and depends on implementation factors as well as software parameters. One important constraint on location arises from the arrangement of shared memory into 32 banks, with a 4-byte stripe. Fermi devices have 64KB of local SRAM which can act as L1D or shared memory, indicating a 16-bit address size for accessing this memory. Bits $[1:0]$ of each address identify position within a 4-byte dword, and are handled entirely by the ALU datapaths. The next five bits, $[6:2]$, identify which of the 32 bank controllers to send a read or write request to, and the remaining nine

[2] It is possible to construct generally contractive functions with exceedingly large local derivatives, which would allow the extraction of visible structure from points ordinarily too close to be seen; in this case, the lower bound is actually determined by the precision of the floating point format in use. However, these systems tend to be highly unstable under interpolation and are not often found in practice.

upper bits `[15:7]` form the address used by an individual bank's memory controller[3]. Each memory controller can service one read or write per cold clock. A crossbar allows 1-to-N broadcast from every bank port on read operations and 1-to-1 access to any bank port for write operations, all handled automatically.

This memory architecture is flexible and fast, and most shared memory operations can be designed to run in a single cold clock across all 32 threads. However, there are some addressing modes which trigger a *bank conflict*, requiring thread serialization and a corresponding stall. These conditions arise whenever two or more operations access the same bank at different addresses — that is, when bits `[6:2]` are the same but `[15:7]` are not. Because barriers are required for synchronization and code in this section is essentially homogeneous across warps, warp schedulers cannot hide latency as efficiently while waiting for these transactions to complete, so stalls while swapping may be compounded in other warps and should be avoided.

A simple way to prevent bank conflicts is to constrain each thread to access the bank corresponding to its lane ID, such that bits `[6:2]` of every shared memory address are equal to bits `[4:0]` of its thread ID. We follow this pattern — with an inner loop this complex, simplicity is something we're pretty desperate for — and thereby keep the problem of determining read and write locations in a single dimension of length equal to the number of warps in a work-group.

Within that dimension, we must still find a permutation of bank addresses for both read and write operations. Shuffling both read and write order provides no "extra randomness" over shuffling just one, so we allow one permutation to be in natural thread order; since registers cannot be traced on the GPU, reads are more challenging to debug, and so we choose to only shuffle the write orders.

To further simplify matters, we fix the write offset against the bank address as a modular addition to the warp number. The resulting write-sync-read, therefore, turns each memory bank into a very wide barrel register. This scheme can be accomodated with, at most, a single broadcast byte per bank, one instruction per thread and no extra barriers. A more complex permutation would require considerable amounts of extra memory, a multi-stage coordination pass, and a lot of extra debugging; it is the latter which most condemns a full permutation. We'll examine the impact of this simulation a bit later in this section.

In the end, the entire process resembles twisting the dials on a combination lock: a point can move in a ring around a column, but can't jump to another row or over other points in a column.

11.3 Shift amounts and sequence lengths

Under this simplified model for swap, there is one free parameter for each lane of a warp, shared across all warps. Methods for choosing these parameters include providing a random number per vector lane and using the lane ID. We wish to determine how effective each method is at minimizing the length of repeated sequences in comparison with best- and worst-case arrangements.

For a flame with N transforms of equal density, the probability of selecting a given transform n is $P(n) = \frac{1}{N}$. For two independent sequences of samples, the probability that one stream would have the same transform at the same index as the other stream is therefore $P(S) = P(n) = \frac{1}{N}$; the probability of having a sequence of identical selections of length l is

$$P(S_l) = P(n)^l = \frac{1}{N^l} \tag{11.1}$$

In any work-group using independent selection of transforms, any two pixel state threads $t_1, t_2 \in T,\ t_1 \neq t_2$ will also be independent, and therefore the probabilities do not depend on the

[3]Some details in this subsection are conjecture. The described implementation is consistent with publicly disclosed information from NVIDIA and benchmarks run by the authors and third parties, but has not been confirmed by the company.

		$l=2$	$l=4$	$l=8$	$l=32$
	Independent (11.1)	0.0156	$2.441 \cdot 10^{-4}$	$5.960 \cdot 10^{-8}$	$1.262 \cdot 10^{-29}$
$T=256$	No shuffle (11.2)	0.2314	0.1352	0.1218	0.1216
	Ring shuffle (11.3)	0.0556	$3.145 \cdot 10^{-3}$	$1.013 \cdot 10^{-5}$	$1.144 \cdot 10^{-20}$
	Full shuffle (11.4)	0.0535	$2.8658 \cdot 10^{-3}$	$8.213 \cdot 10^{-6}$	$4.550 \cdot 10^{-21}$
$T=512$	No shuffle (11.2)	0.0753	0.0608	0.0607	0.0607
	Ring shuffle (11.3)	0.0332	$1.1113 \cdot 10^{-3}$	$1.261 \cdot 10^{-6}$	$2.748 \cdot 10^{-24}$
	Full shuffle (11.4)	0.0317	$1.006 \cdot 10^{-3}$	$1.011 \cdot 10^{-6}$	$1.048 \cdot 10^{-24}$

Table 11.1: Probability of encountering identical transform sequences of length l with different shuffle types.

work-group size T. This is the optimal case which corresponds to an efficient approximation of the attractor.

For a work-group using warp-based branching without a swap, any two threads in different warps are essentially independent, and so $P(S_l|\bar{W}) = P(n)^l$. Threads in the same warp will always have the same transform, giving $P(S_l|W) = 1$. For a warp size W, the chance that any thread t_2 shares a warp with a particular thread t_1 is $P(W) = \frac{W-1}{T-1}$, yielding a combined probability

$$P(S_l) = \frac{W-1}{T-1} + (1 - \frac{W-1}{T-1}) \cdot \frac{1}{N^l} \tag{11.2}$$

The shuffle mechanism modifies $P(W)$, introducing dependencies on vector lanes. Since two threads in the same vector lane can never appear in the same warp, they are independent. Vector lanes are shared with $P(V) = \frac{T/W-1}{T-1}$; as a result, $P(S_l|V) = P(n)^l$. The probability of any t_2 being in the same warp as t_1 is $P(W) = P(\bar{V}) \cdot \frac{1}{V}$. Threads sharing a warp will always have the same state at a given sequence index, but because threads in other vector lanes may now be swapped, each stage is independent.
$P(S_1|\bar{V}) = P(W) \cdot 1 + P(\bar{W}) \cdot P(n)$, and so

$$\begin{aligned} P(S_l) &= P(V) \cdot P(S_l|V) + P(\bar{V}) \cdot P(S_l|\bar{V}) \\ &= \frac{T/W-1}{T-1} \cdot \frac{1}{N^l} + \frac{T-T/W}{T-1} \cdot (\frac{W-1}{T-T/W} \cdot 1 + \frac{T-W-W/T+1}{T-T/W} \cdot \frac{1}{N})^l \end{aligned} \tag{11.3}$$

In one sense, this model also extends to the case of fixed modular offsets; however, for cases where $W < T/W$ — that is, where the warp size is larger than the number of warps per work-unit — each lane equal under the modulus of the number of warps will never swap with respect to each other, which violates the assumptions of independent events and increases the expected length of identical sequences. We solve this by applying a different columnar rotation of each repeated section in the read pattern, which respects banking and thus adds little overhead.

For reference, we also find the expected probability of a common sequence for a full shuffle, which whe have not implemented on the device. In this case, $P(W) = \frac{W-1}{T-1}$, and there are no independent values, so

$$P(S_l) = (\frac{W-1}{T-1} \cdot 1 + (1 - \frac{W-1}{T-1}) \cdot \frac{1}{N})^l \tag{11.4}$$

To compare the efficacy of each shuffle method to the independent case, we show the results of calculating these probabilities for a few configurations and lengths in Table 11.1. Fixed values of $N = 8$ and $W = 32$ are used.

The results display a strong preference towards higher efficiency at larger work-group sizes; this is an important and challenging constraint on launch parameters, as more effort is required to avoid stalls and inadequate occupancy of shader cores when using large work-groups. It's also clear that the simple and efficient ring shuffle methods work nearly as well as a full shuffle. Less clear, however, is how well the ring shuffle works as compared to completely independent threads. While the probability of a chain decays asymptotically to zero, as it does in the independent case, the ring shuffle algorithm does not do so as quickly. So, does it do so quickly *enough*?

Alas, the answer is image-dependent, and not amenable to easy statistical manipulation. The probabilities derived are a good way to gain insight about different strategies for swapping points without an implementation — we discarded several mechanisms that proved too slow or complex for the relative gain in statistical performance in this manner — but there is no way to apply this information. We will simply have to implement and compare.

If a ring-shuffled implementation loses little or no perceptual quality per sample due to point convergence on test images, we will be satisfied. However, in the unexpected event that it is not, the best solution may simply be to allow threads to diverge. This will cause extra computation to be done, but in the end may not significantly impact rendering speed; as it turns out, the bottleneck on current-generation GPUs is likely to lie in the memory subsystem. This issue is discussed further in Chapter 12.

CHAPTER 12

ACCUMULATING RESULTS

The simplest transform functions can be expressed in a handful of instructions; with careful design of fixed loop components, many common flames may require an average considerably less than 50 instructions per iteration. Single-GPU cards in the current hardware generation can retire more than $750 \cdot 10^9$ instructions per second[1] [15]; it's not unreasonable to expect normal consumer hardware to be able to calculate more than 15 billion IFS samples in a single second. Each of these samples needs to make it from a core to the accumulation buffer. Can the memory subsystem keep up?

Each element in a typical CPU implementation's accumulation buffer is 16 bytes wide, holding one single-precision floating point value representing density and three representing the unscaled sums of the red, green, and blue color values for the points within that accumulator's sampling region. With full utilization of the 150 GB/s or so of global memory bandwidth in current-gen hardware, one device may be expected to perform about 5 billion of the 16-byte read-modify-write cycles required to accumulate a sample.

A three-fold drop in performance due to memory limitations is easy to accept for an offline renderer. In fact, such a limitation might simplify the entire project; an externally-imposed performance cap would limit the need for more complex optimizations and provide a concrete stopping place for ongoing optimization efforts. Unfortunately, the iteration rate limit imposed by accumulation speed is far more severe than the 3× penalty implied by the raw bandwidth.

12.1 Chaos, coalescing, and cache

The flame algorithm, and the chaos game in general, estimates the shape of an attractor by accumulating point information across many iterations. While visually interesting flames have well-defined attractors, the trajectory of a point traversing an attractor is chaotic, jumping across the image in a manner that varies greatly depending on the starting state of its thread. As a result, there is little colocation of accumulator addresses in a thread's access pattern over time. Spatial coherence is also unattainable, due to the need to avoid warp convergence discussed in the previous section.

Each accumulation, therefore, is to an effectively random address. While the energy density across an image is not uniformly distributed, most flames spread energy over a considerable portion of the output region. Accumulation buffers can be quite large; for a 1080p image rendered using 2× supersampling, the framebuffer is over 100MB[2]. This access pattern would challenge even traditional CPU caches, which tend to be spacious and include advanced prefetching components; it renders the small, simple caches found in GPUs useless.

[1] The FLOPs figures commonly cited by graphics manufacturers are twice this value, as they count multiplies and adds as separate operations in an FMA.

[2] $(1920 \cdot 1080)$ accumulators $\cdot\ 2^2$ accumulators/sample $\cdot\ 16$ bytes/accumulator $= 132710400$ bytes

With each cache miss, a GPU reads in an entire cache line; each dirty cache line is also flushed to RAM as a unit. In the Fermi architecture, cache lines are 128 bytes. If nearly every access to an accumulator results in a miss, then the actual amount of bus traffic caused by one accumulation is effectively eight times higher than the accumulator size suggests — and consequently, the peak rate of accumulation is eight times lower.

To make matters worse, DRAM modules only perform at rated speeds when reading or writing contiguously. There is a latency penalty for switching the active row in DRAM, as must be done before most operations in a random access pattern. This penalty is negligible for sustained transfers, but is a considerable portion of the time required to complete a small transaction; when applied to every transaction, attainable memory throughput drops as much as 50% [43].

A 3× performance penalty may be accepted; a 30× penalty *must* be addressed to meet this project's stated performance goals. An improvement of more than an order of magnitude, however, is rarely trivial, and no single solution will remove this bottleneck entirely. Over the course of this chapter, several improvements will be introduced, each providing incrementally higher memory performance at the cost of increasing complexity. In concert, these techniques form an accumulation stage that, while arcane, is fast enough to keep up with iteration.

12.2 Tiled accumulation

The most immediate problem in the writeback stage is that of a cache miss, which will happen often due to the random distribution of write locations and the poor fit of the accumulation buffer into the L2 cache. Little can be done about the write patterns without fundamentally restructuring the flame algorithm[3], suggesting that it is the cache fit which should be optimized.

This is not a novel problem. Implementations of the Reyes algorithm for ray-tracing, such as Pixar's RenderMan, handle limited cache sizes by splitting the global geometry in a scene along the boundaries of small rectangular regions that tile the destination framebuffer; the contents of each tile are then drawn separately and discarded, allowing the information in a single image area to remain in the cache [44]. A similar technique is employed by graphics chips designed for low-power applications, such as those in the PowerVR architecture [45].

As accumulation regions cannot be bounded *a priori* in the flame algorithm, dynamic point queues must be maintained throughout the rendering process. The feasibility of maintaining these queues depends on many parameters, and exactly one combination of parameters has been found which meets all necessary criteria. The techniques used to attain this particular configuration are presented below, followed by a description of the integrated algorithm. Because of the sensitivity of this approach to configuration changes, extreme measures are described which provide seemingly insufficient benefit for their cost, until they are understood in the context of the entire approach.

Representing an iteration

An accumulation requires adding a density and three color values to an accumulator located on a two-dimensional grid. Representing this information as a six-dimensional tuple using floating point values requires 24 bytes. Queueing the image information for rendering requires reading and writing it multiple times. If each value consumes 24 bytes of memory, the implementation will encounter space and bandwidth limitations; encoding this information more efficiently is desirable.

The accumulator is identified along a rectangularly-sampled 2D grid. In this arrangement, there is an isomorphic map between image regions and accumulator indices, so representing the location in

[3] The authors intend to explore such a restructuring after the initial rendering is built.

intermediate stages by accumulator index loses no information as compared to normal accumulation. Use of an integer format for representing location is also useful. Typically, flames are rendered using resolution and supersampling settings which require a number of accumulators in the neighborhood of 2^{24}, indicating that the index can be fully represented as a 25-bit integer.

The color values in an IFS accumulation are calculated from the IFS color coordinate by performing a linear blending between the two closest samples of the active control point's color palette, followed by a multiplication against the transform visibility. Storing this information, rather than the results of this calculation, can lead to additional savings. A typical flame uses 1000 control points to produce motion blur, so a palette index can fit in a 10-bit integer. Both the color coordinate and the visibility transform are bounded on $[0, 1]$, so a fixed-point representation can achieve a full-precision representation with 22 bits per coordinate.

Together, this results in a total of 79 bits of information. On an architecture designed entirely around 32-bit word sizes, this size is awkward and ineffecient. Fortunately, this algorithm is a Monte Carlo simulation with a very high sample count; since all samples in a given image region are averaged, sample contributes relatively small amounts of information to the final image. Thus, considerably smaller intermediate representations are possible without loss of detail — but only if dithering is used to avoid quantization bias.

Dithering

Consider a two-entry color palette, where the tristimulus value at index 0 is $C(0) = (0, 0, 0)$, and at index 1 is $C(1) = (1, 1, 1)$. With linear blending, a lookup for an arbitrary value takes the form $C_l(i) = P(\lfloor i \rfloor) \cdot (\lceil i \rceil - i) + C(\lceil i \rceil) \cdot (i - \lfloor i \rfloor)$, such that $C_l(0.4) = C(0) \cdot 0.6 + C(1) \cdot 0.4$. The average value of a number of linearly-blended lookups with $i = 0.4$ would then be $(0.4, 0.4, 0.4)$. On the other hand, if the index was first truncated to integer coordinates, the average value of any number of samples would be $C_l(\lfloor 0.4 \rfloor) = C_l(0)$, or $(0, 0, 0)$. Many flames use automatically-generated palettes which feature violent color changes between coordinates, so such truncation can result in a substantial amount of error.

Dithering applies noise to signal components before quantization to distribute quantization error across samples, enabling more accurate recovery of average values [46]. For the fixed quantization of the IFS color coordinate, dithering is accomplished by adding a value sampled from a uniform distribution covering the range of input values which are quantized to 0 — for integer truncation, this is $[0, 1)$, whereas round-to-nearest-integer would use $[-0.5, 0.5)$ — to the floating-point color sample before quantization.

To continue the example, dithering the index value $i = 0.4$ before integer truncation would provide an expected value uniformly distributed in the range $[0.4, 1.4)$. Over many samples, this value would be expected to be quantized to a value of 0 as $P(i_q \leq 1) = 0.6$, and 1 as $P(i_q > 1) = 0.4$. Applying those values to the palette lookup functions, we have $C(i_q) = C(0) \cdot P(i_q \leq 1) + C(1) \cdot P(i_q > 1) = C_l(i)$, so that over many samples the added noise has actually eliminated the quantization error in the average.

A flame's palette contains 256 color samples. Linear blending occurs between palette samples, but no such guarantee exists from sample to sample; as a result, subsampling the palette would result in aliasing, even when dithering is applied. The minimum size of the color index component, therefore, is 8 bits. This offers a potent optimization: if the information in the other coordinates can be transmitted without adding additional bits to a logged point, the log of each point can fit in a single 32-bit word. As it turns out, side channels exist for both control point time and visibility.

Each work-group conducts all iterations required to complete a time step before moving on to the next in order to keep control point data in cache. Because most work-groups will proceed at approximately the same pace, a single global counter indicating the current control point can be shared across all work-groups. Across the narrow time range of a single animation frame, the variation of a palette is typically imperceptible, so this approximation is acceptable.

As for visibility information, there is another channel from which an additional bit can be extracted: whether or not the point is written to a queue. Discarding half of the points that were computed to provide an accurate long-run density intuitively seems much more severe than discarding half of the bits to recover an accurate density value, yet the principles are the same; dithering works just as well with one bit as with eight. In some circumstances, the consolidation queue may remove enough redundancy from the address lines to allow more traditional storage of this information in the upper bits of the address. Since the framebuffer configuration is available during code generation, the appropriate scheme can be selected automatically.

Even at a mere 32 bits per sample, queueing every sample of a high resolution flame would exhaust a GPU's memory[4]. As a result, output queues must be managed in real-time by threads running on the device, rather than in an offline fashion by the CPU. To remain within the bounds of this algorithm, the flushing process must operate over as large a range as possible, a task made possible by the techniques of the next two sections.

Color subsampling

Most rendered flames are compressed with the JPEG image codec or the H.264 video codec. These codecs, and many others, reduce the size of transmitted frames by sampling image channels containing color information at a lower spatial resolution than the luminosity channel, and by quantizing the subsampled result more severely [47]. That these codecs do so without objectionable degradation of the image is evidence of the human visual system's reduced sensitivity to perception of color, as compared to brightness. Since most of the color information in the accumulation buffer is discarded before ever reaching the user, subsampling color during the accumulation stage reduces the size of the accumulator with little or no loss in visual quality.

In video and image applications, RGB tristimulus values are first transformed to the YUV colorspace[5] via a simple matrix multiplication. The resulting value separates luminosity information, in the Y component, from the chromaticity information in the other two components. It is the latter two components which are subsampled in the resulting image. However, due to transform color selection and the behavior of the density-based log scaler, the luminance component of the resulting image will display very strong correlation with density values. As a result, this implementation will subsample all three components in the accumulation process, exploiting redundant information in the density buffer to perform accurate reconstruction.

Sampling image components at different spatial frequencies breaks the isomorphism between the image-space coordinate grid and buffer element coordinates, requiring more complicated addressing schemes. In most systems which use channel subsampling, the components are interleaved so that the values for any image-space coordinate are contained within the same cache line (regardless of system). On the other hand, interleaving complicates addressing, requiring rearrangement in two dimensions according to a predetermined pattern. This restricts the set of subsampling ratios which are valid, and makes certain filtering operations less efficient.

Because we are explicitly targeting a system which will keep all data being accessed for accumulation in cache, there is no significant benefit to having all image components share a cache line, and we are free to choose a simpler solution. One such solution is to simply store each image component, or set of image components, with a unique sampling rate in a separate plane. Assuming an efficient data alignment exists, this scheme uses no more memory than an interleaved format, and retains the benefits of linear addressing.

[4] $(1920 \cdot 1080)$ pixels $\cdot$ 2000 samples/pixel $\cdot$ 4 bytes/sample $= 16588800000$ bytes

[5] YUV is more accurately described as a "hodgepodge of ideas" than a colorspace. We refer here to linear, invertible, continuous, full-range encodings such as YC_BC_R.

Efficient representation

The log-scaling process performed to compress the high-dynamic-range flame into the display image's dynamic range is explicitly designed to remove the information in the lower bits of each image sample. In nearly every case, the final display format of fractal flames is an image format using 8 bits for each of its three channels for 24 bits total; in most cases, the intermediate subsampled encoding represents flames with 12 bits per pixel before compression. At 128 bits per sample, traditional accumulators store more than ten times as much information as the final image.

A problem confounding the selection of intermediate encodings is the need to store a large dynamic range. Images with high sample counts, aggressive gamma curves, and very small image features can require accumulators to store thousands or millions of points, requiring excessively large fixed-point and integer representations. The solution to such a problem is often to use floating-point numbers, which is what has been done previously.

Modern GPUs are optimized to work quickly with standard single-precision floating-point numbers, and they serve as an excellent compromise between speed and precision during calculation. When storing accumulators, however, the exceptionally high cost of a cache miss makes footprint reduction worth using slower, more complicated instructions. For such purposes, GPUs include support for converting to and from IEEE 754 half-precision floating-point formats. Unfortunately, support for these 16-bit values is missing from the atomic instruction set; flushing a tile would require performing read-modify-write over the inter-chip bus, which would cause it to flood [6].

Due to extreme cache pressure and a limited number of atomic instructions, the most efficient solution which accomodates the large dynamic range of intermediate image formats in the minimum number of bits is the implementation of a software floating-point format.

The 32-bit datapath of a GPU makes using values which fit evenly into that width. Since the density value scales all channels in the output image, at least 8 bits of precision are required. Only widths of 10 and 16 bits possess enough space for an 8 bit mantissa while dividing evenly into a word. Typical flame images will easily exhaust the upper bound of a two-bit mantissa[7], so we consider a width of 16 bits as the minimum choice for independent storage of density information. Within these 16 bits, a five bit mantissa is the smallest value providing comfortable headroom.

Apart from the missing sign bit, this floating point format differs from standard IEEE 754.2008 half-precision floats in one important respect: the exponent bias is set at -01_h, rather than $0F_h$. Subnormal numbers are therefore represented by the exponent $1F_h$, where the exponent is treated as if it is also represented in two's complement. This notational difference implies that a string of zero bits does not represent a value of zero, which is inconvenient. The extra effort is justified by an extremely simple addition algorithm.

To add a value to a memory location, first read the exponent value as E. Multiply the addend by $2^{-(E+1)}$, and round to the nearest integer using dithering. Finally, perform an atomic addition to the value in global memory.

This algorithm can be implemented in as few as ten operations, including the generation of a random number for the purposes of dithering. Fermi devices can also read directly from shared memory in certain circumstances, shaving another operation off the average time required. Compared to an atomic, 32-bit floating point addition, this is still a considerable penalty; however, using 32-bit numbers provides insufficient cache coverage, as noted below, so this option is not acceptable.

[6] Fermi's L2 provides low-latency acceleration of instructions, but does not provide significant bandwidth amplification over global memory; microbenchmarks pin it at less than twice the bandwidth of the memory it covers. On-core RMW cycles to global memory require an immediate transfer and invalidation of lines in L1, making the transactional efficiency even lower than for an L2 miss, so we hit the ceiling at about the same rate.

[7] $2^8 \cdot 2^{2^2} = 4096$. Adding another bit to the mantissa provides a heartier range of $2^7 \cdot 2^{2^3} = 32768$, but this is not enough to be generally applicable.

Since a more compact notation is required, the only available options are converting to and from half-precision floating point values, which has reduced precision, is susceptible to bias as a value grows, and requires at least five operations; half-precision with dither, which eliminates the bias but brings the total number of operations to twelve; and half-width integer, which requires complicated overflow handling. Half-width integer accumulation is a reasonable choice for density accumulation, but is a poor choice for color values, where a choice must be made between frequent overflow handling and significant loss of data in darker image regions.

Sharing shared memory

The most efficient method for modifying data using a heterogeneous write order is manual management in shared memory. While shared memory and L1 share the same SRAM and local controllers, accesses to L1 use a different data path than accesses to shared memory which incurs a several cycle penalty. Further, written values are discarded from L1 immediately, and each L1 miss requires a 128-byte read from L2, which incurs a delay even when that data is in L2. Atomic writes to L2 also have reduced throughput and increased latency compared to shared memory.

Using the 16-bit software floating-point method, as well as 2× subsampling of color information, each accumulator requires an average of 3.5 bytes of storage. To enable fast, radix-based binning, the number of bins in the final accumulation step must be a power of two. Less than 24K of shared memory is available, as will be discussed later. Under these parameters, we find that a suitable size for the shared-memory accumulation buffer is 4,096 entries, consuming 14K of shared memory.

CHAPTER 13

DESIGN SUMMARY

This implementation of the fractal flame algorithm is split into several Haskell libraries, with a small front-end application for demonstration purposes. Together, these programs build, run, and retrieve the results of a CUDA kernel. The actual rendering is done entirely on the CUDA device.

This chapter provides an overview of the method of operation of the software produced as part of this project. It also includes background information and rationale on those portions of the project which have not been covered elsewhere in the document.

13.1 Host software

This project will produce a platform-independent console application that will accept one or more flames in the standard `flam3` XML genome format and save a fractal flame image for each input flame. More elaborate applications, such as a long-running process with socket communication for use in quasi-real-time applications such as the `fr0st` fractal flame editor, are expected to be produced, but these applications fall outside the scope of this document. The high-level flow of the rendering application is depicted in Figure 13.1.

The host application parses command-line arguments to determine operating mode, and then loads a `flam3`-compatible XML stream, either through a specified filename or through standard input. This information is then passed to the `flam3` library via the `flam3-hs` bindings for parsing and validation. The resulting flames are returned in a binary format that can be unpacked to a genome type from `flam3-types`.

A control point near the center of the requested animation range is selected as the prototype control point. This information, in addition to information about the device gathered from the CUDA driver and from the running environment, is passed to the `cuburn` library, where it is used to generate the CUDA kernel and context for uploading to the device, as well as the functions needed to pack control points for use by this algorithm.

A CUDA context is initialized in a bound thread, and the generated kernel is loaded, where kernel-specific initialization is performed as needed by the `cuburn` library. For each control point to be rendered, the program creates a final image framebuffer, performs data-stream compaction on the interpolated family of control points needed to render with motion blur, and schedules a rendering with another `cuburn` command.

This rendering thread continues performing these steps until all images have been scheduled. Once any render is complete, its final framebuffer is copied from the device, and an output thread is spawned to save the contents of this image to disk.

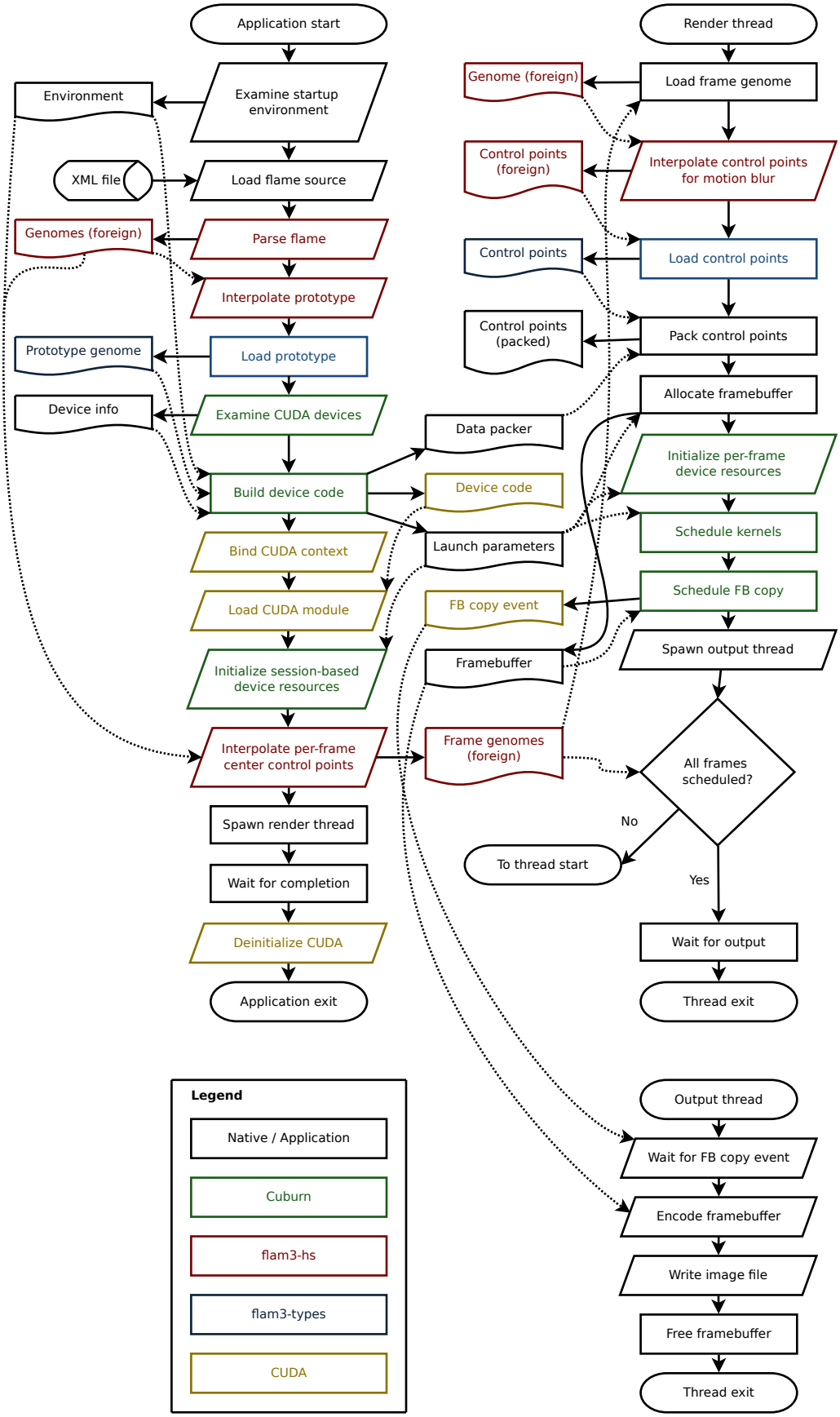


Figure 13.1: The host-side workflow of the example application.

`flam3-hs` and `flam3-types`

The `flam3` library is written in C, and is designed to be used as a dynamic library. It uses POSIX facilities, and some features are only enabled when compiling and linking with GCC. Cross-platform support is partially implemented, allowing the library to be built on Windows with MSVC++, but the process is not trivial. Additions of new features, such as variations, filtering modes, and empirical algorithm tunings, require modifications to the extremely large data structures used to store genome information. These binary-incompatible changes require foreign function interfaces to be tightly coupled to particular versions of `flam3`.

To ensure that code produced by this project is cross-platform and not strictly dependent on particular `flam3` versions, the bindings to the flame library are split across two libraries. `flam3-types` defines Haskell datatypes which are capable of expressing a genome, as well as a basic parser for flames. The genome datatypes are specifically crafted to allow maximum compatibility without breaking type safety, so that later feature additions do not require changes to existing code where-ever possible. `flam3-types` does not depend on the `flam3` library, so that it is maximally portable.

For the purposes of interpolation and compatibility, `flam3-hs` provides Haskell bindings to `flam3`. This requires linking to the `flam3` library, which presents a number of practical problems on Windows; as a result, it is possible to use the `cuburn` library without `flam3-hs`, although some features will be disabled. This FFI library maintains forward compatibility with data structures by preventing them from being serialized from their `flam3-types` counterparts; while the raw data can be maintained for fast consecutive function calls, the only means of turning a native Haskell genome into one compatible with `flam3` is to use XML as an intermediate format.

Shard and `cuburn`

The `cuburn` library contains the actual code used to render fractal flames. The code is written in the Shard language for GPU programming, which is being developed as part of this project. As an embedded domain-specific language, Shard allows developers to build code at runtime for the GPU using ordinary, pure Haskell expressions. This allows for complex analysis and optimization efforts to take place without sacrificing functional purity or type safety.

`cuburn` also includes ancillary functions to help with generating and running device code. It intentionally omits "all-in-one" rendering functions which encapsulate the rendering process from start to finish, so that focus can be placed on ensuring the interface for full control is straightforward enough for common use. Management of CUDA state is also omitted; operations on CUDA contexts are effectful, and a developer making use of this library must that `cuburn` does not interfere with other CUDA activity in the same process.

The full API is not difficult to use; while the full flow presented in Figure 13.1 may appear to be intimidating, it lends itself to concise expression under idiomatic Haskell.

13.2 Device software

Code on the device is split into either two or three kernels. Together, these kernels form an implementation of the algorithm outlined in Chapter 3. Each of these kernels is dynamically generated, for reasons outlined in Chapter 10, so the workflow may differ from that presented below.

Threads running the rendering kernel — henceforth simply a rendering thread — run for the duration of a render. After loading initial data, the thread will begin the iteration phase of the flame algorithm, computing an IFS sample for the indicated control point. The generated points are efficiently recorded to global memory buffers, and after a short intra-work-group communication, the threads continue with another iteration.

Once all iterations are complete, the thread exits, allowing the device to begin computing subsequent commands.

Accumulation threads run alongside rendering threads, using memory hardware while rendering threads remain mostly arithmetic in their operations. An accumulation thread loads point record produced by rendering threads and performs a sorting pass to group point records by address. When a group becomes full, the accumulation threads then process it and add the accumulated results to the global buffer.

After all rendering and accumulation threads complete on the device, the hardware thread scheduler dispatches the filtering threads on the device. Each filtering work-group cooperates to apply the steps necessary to convert the accumulated color and density information into a final image. When these threads exit, the device signals to the host that the image is complete, copies the final framebuffer to the device, and begins rendering the next image.

Rendering

Upon invocation, a rendering thread uses its global thread ID to load a unique random state. This state is kept resident for the duration of the thread's execution, and is updated in global memory by the thread as it exits. This action is also performed by the accumulation and filtering kernels upon a distinct set of states. Other rendering-specific actions include setting the consecutive bad value flag to trigger point regeneration and initial point fusing, acquiring output queues, and clearing the buffers in shared memory.

Threads in a work-group operate on a single control point at a time. This control point is selected through atomic access to a global counter, which acts as an index into a global array of control points; once the index reaches a maximum value, all control points have been processed, and the rendering thread exits. The control point select index is a fixed multiple of the number of control points in an image, allowing multiple work-groups to operate on the same control point simultaneously. This increases the efficiency of the global cache and balances workload across cores.

The iteration loop begins with a test of the bad value flag to determine whether a new random point needs to be generated, and a divergent branch to do so if needed. The first thread in a warp then generates the transform select value and broadcasts it to the rest of the warp. Threads then apply the selected transform function to the current position and color values to obtain a new IFS state.

If the bad value flag is not less than zero — indicating that the point is currently joining the attractor, and should not be written — and the point is within image boundaries, the point is quantized by its thread into the record format described in. The top six bits are used to select an output queue, and the record is appended to the corresponding consolidation queue, using the resolution algorithm described in to handle consolidation flushes. The bad value flag is reset to zero.

If the point is outside of the image domain or joining the attractor, the bad value flag is incremented. If the flag is originally less than zero, this may cause it to move to zero, indicating that the point's trajectory has joined the attractor and is ready to be written in subsequent threads. If it is larger than a determined amount, indicating several consecutive invalid values, it may have entered an expansive region of the function system, and will be rest at the next iteration.

Once a sufficient number of valid points have been generated, the iteration loop exits. The control point select index is tested to see if additional work is queued; if more is available, the thread restarts, otherwise it exits.

Accumulation

The process of moving samples from first-stage output queues to final accumulators is handled by accumulation threads. A portion of the first warp in an accumulation work-group performs an infrequent

poll of a corresponding portion of the global queue tree to determine when new work is available; when it is, the entire work-unit wakes, takes ownership of the corresponding queue buffers, and begins processing the first queue.

The accumulation threads perform two functions, depending on the queue being processed. First-stage output queues contain point records over an address range of as many as 17 bits. The output stage requires a range no larger than 10 bits, so there may be as many as 128 second-stage output queues per first-stage queue. The accumulation thread is responsible for reading points from the first-stage queue and writing them to the second-stage queue. This sorting pass is almost identical to that performed by the first-stage output, save for using a larger number of consolidation queues and obtaining points by reading output queues instead of iterating directly.

Once a first-stage output queue is read, its buffers is returned to the main buffer pool. After reading the contents of all first-stage output queues, the second-stage queues within an accumulation work-group are examined to determine if any are approaching a sufficient length for writeback to begin. If any are, the accumulation work-group enters its second mode.

Processing a second-stage queue begins by taking ownership of all flushed output buffers in the output queue, and clearing the internal accumulation values in shared memory. Each value in the queue is then read and added to the appropriate shared accumulator. As each buffer in the queue is read, it is freed and returned to the global pool. After a queue has been fully processed, the work-group once again waits for data by polling global memory.

Filtering

The filtering kernels perform the log scaling process described in Chapter 8, and one of the filtering mechanisms described in Chapter 9. The log scaling information is applied per-pixel, and maps cleanly to the traditional threading dispatch model employed in GPU shading; while the particular method has considerable impact on the appearance of the final image, the implementation is straightforward. On the other hand, efficient implementation of filtering algorithms on GPU architectures can be challenging, and almost always involves optimizations particular to the kind of algorithm chosen, so there is little opportunity for a general description. In both cases, full explanations of the processes involved are available in the appropriate chapters.

APPENDIX A

GLOSSARY

Between GPU computing, the fractal flame algorithm, signal processing, and multiresolution analysis, describing this project requires a considerable amount of often conflicting terminology. This reference may help resolve and disambiguate unfamiliar terminology.

In cases where terminology is nonstandard or conflicting across the fields of study involved by this project, the source of a term is provided. Only the terms used in our project have entries; some terms may have alternate meanings in other fields of study, but these are not listed here.

ACCUMULATION BUFFER The grid of accumulators used to store the results of a simulation. The accumulation buffer may be of a higher resolution than the output buffer to accomodate FSAA.

ACCUMULATOR An element of the accumulation buffer, storing density and color information. In `flam3`, these are called "buckets". In some IFS literature, these are called "histogram bins".

ACCUMULATED SAMPLE The value of an accumulator after all iterations have been performed.

ANIMATION (flam3) A series of frames, where each frame is generated from a different time step along a particular interpolation between two flames.

BARRIER (CUDA) An instruction which will stall the warp which issues it until a condition is met, used for synchronization. OpenCL term is "work-group barrier", where OpenCL's "command-queue barriers" are used to build streams.

COMMAND (OpenCL) A task which a device must complete.

CORE The smallest unit of a device capable of completely executing a single instruction. Our usage explicitly conflicts with marketing material from both AMD and NVIDIA, which refer to each vector lane as a core, but is in line with industry parlance. (In some architectures, a core may be able to dispatch more than one instruction at a time to shared hardware resources; under this definition, it is still a single core, as the functional unit cannot be divided further.)

DECIMATION (Multiresolution analysis) A reduction in the number of samples in a signal. In our algorithm, as elsewhere, the term is assumed to refer to octave-band decimation, where a signal is downsampled by a factor of 2 in all dimensions.

DEVICE (OpenCL) A hardware unit which may execute commands, and which appears to run asynchronously to the CPU. In our case, one of the GPUs available in a system.

EDGE (flam3) An animation involving interpolation between two visually distinct flames, so named because they are attached to the edges in the graph used to resolve playback order in the Electric Sheep screensaver.

FLAME The abstract notion of a particular class of chaotic attractor. Flames are described by their genomes, and visually approximated using the fractal flame algorithm.

GENOME (flam3) The set of parameters describing a flame, or the concrete data structure containing this information. May also include information about aspects that affect the rendering only, rather than the underlying attractor.

IFS ITERATION An application of one transform function from an iterated function system to an IFS point to produce a new point.

IFS POINT The vector resulting from a number of applications of transform functions to a starting vector.

IFS SAMPLE The information about the shape of the attractor gained by performing an iteration.

KERNEL An entry point for a device thread; the code associated with a single device invocation.

LOOP (flam3) An animation of a rotation interpolation, which modifies a single flame in such a way that the final frame is identical to the first.

STREAM (CUDA) A strictly ordered series of device commands. Ordinarily, devices may dispatch commands as soon as execution resources become available to do so; a command in a stream, on the other hand, is not started until the previous command in the stream completes.

TRANSFORM FUNCTION A member of an iterated function system, as described by an xform.

WARP (CUDA) A group of threads that must execute the same instruction at the hardware level. Hardware and compiler tools allow a programmer to overlook warps without compromising code correctness, but optimal performance requires careful consideration of warps. This term technically applies only to NVIDIA devices, where AMD uses notionally similar but technically different “wave-fronts”, and its use in this document is a compromise between correctness and clarity.

WORK-GROUP (OpenCL) A collection of threads which share a global ID. Work-groups are the largest structure that can access a common slice of shared memory or enter a barrier. CUDA equivalent term is “block”.

VECTOR LANE An element of a vector.

XFORM (`flam3`) The data structure associated with each function of an iterated function system.

APPENDIX B

LICENSING AND PERMISSIONS

FIGURE 2.2: Reprinted with permission under the Creative Commons Attribution-ShareAlike 3.0 license.

- *Source*: Self made based on Java Appplication (http://to-campos.planetaclix.pt/fractal/koch.html).
- *Date*: 15 May 2007
- *Author*: António Miguel de Campos
- *Description*: 7 first steps of the building of the von Koch curve in animated gif. Notice the parallel corresponding diameters present in the inner rhomboids.

FIGURE 2.3: Reprinted with permission from Vlado from FreeDigitalPhotos.net a royalty-free site.

- *Source*: http://www.freedigitalphotos.net/images/Trees_and_Shrubs_g75-Cherry_Tree_In_Winter_p33977.html
- *Terms*: http://www.freedigitalphotos.net/images/help/acknowledgement/terms.php
- *Permission*: http://www.freedigitalphotos.net/images/help/-acknowledgement/index.php?photogname=Vlado&photogid=1836
- *Author*: Vlado
- *Description*: llustration of a cherry tree in winter.

FIGURE 2.7: Reprinted with permission under the Creative Commons Attribution-Share Alike 3.0 Unported.

- *Source*: http://www.chaoscope.org/gallery.htm
- *Date*: 5 March 2007
- *Author*: Nicolas Desprez
- *Description*: Attractor Poisson Saturne.

FIGURE 2.8 (LEFT): Reprinted with permission under the Creative Commons Attribution-ShareAlike 3.0 license.

- *Source*: Self made based on Java Appplication (http://en.wikipedia.org/wiki/File:Animated_fractal_mountain.gif).
- *Date*: March 2006
- *Author*: António Miguel de Campos
- *Description*: Animated fractal mountain.

FIGURE 2.8 (RIGHT): Reprinted with permission under the Creative Commons Attribution 3.0 Unported.

- *Source*: Terragen.
- *Date*: 2002
- *Author*: The Ostrich
- *Description*: n example of a fractal landscape, generated using my own program and rendered using Terragen.

FIGURE 3.5: Reprinted with permission under the Creative Commons Attribution-ShareAlike 3.0 license.

- *Source*: Own work, using model written by Mike Borrello (http://www.vissim.com/node/199)
- *Date*: 6 January 2010
- *Author*: DSP-user
- *Description*: Barnsley's fern illustrates the use of affine translations in an iterated function system (IFS) to create a fractal. In Table III.3 of Michael Barnsley's book, the IFS code for the four affine transformations for the Barnsley leaf is given as a table of values for the coefficients a, b, c, and d, the constants e and f and the probability percentage factor of p as follows:

FIGURE 3.7 (TOP): Reprinted with permission under the Creative Commons Attribution 3.0 license.

- *Source*: http://sheep.arces.net/generation-243/dead.cgi?id=1490
- *Date*: July 28 2008
- *Author*: ReFa

FIGURE 3.7 (TOP): Reprinted with permission under the Creative Commons Attribution 3.0 license.

- *Source*: http://sheep.arces.net/generation-243/dead.cgi?id=243
- *Date*: June 28 2008
- *Author*: BrothaLewis

FIGURE 8.1: Reprinted with permission under the Creative Commons Attribution-ShareAlike 3.0 license.

- *Source*: http://en.wikipedia.org/wiki/File:HSV_color_solid_cylinder_alpha_lowgamma.png

- *Date*: March 22 2010
- *Author*: SharkD
- *Description*: The HSV color model mapped to a cylinder. POV-Ray source is available from the POV-Ray Object Collection.

FIGURE 8.2: Reprinted with permission under the Creative Commons Attribution-ShareAlike 3.0 license.

- *Source*: http://en.wikipedia.org/wiki/File:GammaCorrection_demo.jpg
- *Date*: September 14 2010
- *Author*: X-romix and Rubybrian
- *Description*: A demonstration of the effect of gamma correction on images.

FIGURE 8.3: Reprinted with permission under the Creative Commons Attribution-ShareAlike 3.0 license.

- *Source*: http://en.wikipedia.org/wiki/File:GammaCorrection_demo.jpg
- *Author*: UC Davis ChemWiki by University of California
- *Description*: Different Wavelengths and Frequencies

FIGURE 8.5, FIGURE 8.6, FIGURE 8.7, FIGURE 8.8, FIGURE 8.9, FIGURE 8.10, FIGURE 8.11, FIGURE 8.12: Reprinted with permission under the Creative Commons Attribution 3.0 license.

- *Source*: http://v2d7c.sheepserver.net/cgi/dead.cgi?id=11148
- *Date*: January 28 2011
- *Author*: BrothaLewis

**Figure 7.1

- *Source*: http://random.mat.sbg.ac.at/tests/theory/spectral/img13.gif
- *Date*: April 17 2011

> Dear Nicolas,
>
> thank you very much for your interest in our research.
>
> The images are the scientific property of Karl Entacher, now at Fachhochschule Salzburg (Salzburg Polytechnical University). Please contact Karl at the address karl.entacher@holztechnikum.at or, alternatively, entacher@cosy.sbg.ac.at
>
> Best regards
>
> Peter

Peter Hellekalek Dept. of Mathematics University of Salzburg Hellbrunner Str. 34 A–5020 Salzburg

tel.: +43-(0)662 8044 5310 fax : +43-(0)662 6389 5310 web : http://random.mat.sbg.ac.at

---

FIGURE 9.1: Reprinted with permission under the Creative Commons Attribution-ShareAlike 3.0 license.

- *Source*: http://upload.wikimedia.org/wikipedia/commons/8/88/Aliasing_a.png
- *Date*: July 29 2009
- *Author*: Mwyann
- *Description*: Aliasing example of the "A" letter in Times New Roman.

FIGURE 9.2: Reprinted with permission from Dorbie after being released into the public domain.

- *Source*: http://upload.wikimedia.org/wikipedia/en/3/32/GridSS.png
- *Source*: http://upload.wikimedia.org/wikipedia/en/1/16/RandomSS.png
- *Source*: http://upload.wikimedia.org/wikipedia/en/e/e3/PoissonSS.png
- *Source*: http://en.wikipedia.org/wiki/File:JitterSS.png
- *Source*: http://upload.wikimedia.org/wikipedia/en/3/30/RotGridSS.png
- *Date*: September 14 2007
- *Author*: Dorbie

Important Note: All other illustrations and figures have been generated by the authors of this document.

APPENDIX C

BIBLIOGRAPHY

[1] B.B. Mandelbrot, *The Fractal Geometry of Nature*, W. H. Freeman, 1983

[2] S. Draves and E. Reckase, "The fractal flame algorithm," 2003, pp. 1-41. http://www.flam3.com/flame_draves.pdf

[3] R. Eglash, "Ron Eglash on African fractals," *TED Conferences, LLC*, 2007. http://www.ted.com/talks/ron_eglash_on_african_fractals.html

[4] P. Perry, "Special Post: Scott Draves," 2011. http://www.triangulationblog.com/2011/01/scott-draves.html

[5] M.F. Barnsley, *Fractals Everywhere*, Academic Press, 1988

[6] E. Weisstein, "Affine Transformation," *MathWorld*. http://mathworld.wolfram.com/AffineTransformation.html

[7] K. Conrad, "The Contraction Mapping Theorem," 2010, pp. 1-9. http://www.math.uconn.edu/\~kconrad/blurbs/analysis/contractionshort.pdf

[8] E. Weisstein, "Barnsley's Fern," *MathWorld*. http://mathworld.wolfram.com/BarnsleysFern.html

[9] F. Charles, "Looking Good: The Psychology and Biology of Beauty," *Journal of Young Investigators*, vol. 6, 2002. http://www.jyi.org/volumes/volume6/issue6/features/feng.html

[10] E. Reckase and S. Draves, "flam3," 2011. http://flam3.com

[11] R. Hordijk, P. Borys, and P. Sdobnov, "Apophysis," 2011. http://www.apophysis.org

[12] S. Brodhead, "flam4," 2011. http://sourceforge.net/projects/flam4/

[13] M. Thiesen, "Fractron 9000," 2011. http://fractron9000.sourceforge.net/

[14] T. Ludwig, "Chaotica," 2010. http://www.indigorenderer.com/forum/viewtopic.php?f=6\&t=10205

[15] A. Voicu, "NVIDIA Fermi GPU and Architecture Analysis," *Beyond3D*, 2010. http://www.beyond3d.com/content/reviews/55

[16] D. Kanter, "Introduction to OpenCL," *Real World Technologies*, 2010. http://www.realworldtech.com/page.cfm?ArticleID=RWT120710035639

[17] S. Robertson, "CUDA atomics: a detailed analysis," *strobe.cc*, 2009. http://strobe.cc/cuda_atomics/

[18] D. Kanter, "Larrabee 1 Defers Graphics, Bins Rendering," *Real World Technologies*, 2009. http://www.realworldtech.com/page.cfm?ArticleID=RWT120409180449

[19] S. Wasson, "Nvidia's GeForce GTX 590 graphics card," *The Tech Report*, 2011. http://techreport.com/articles.x/20629

[20] D. Kanter, "AMD's Cayman GPU Architecture," *Real World Technologies*, 2010. http://www.realworldtech.com/page.cfm?ArticleID=RWT121410213827

[21] A. Munshi, "OpenCL Specification," *Khronos Group*, 2010

[22] M. Bevand, "Whitepixel," 2010. http://whitepixel.zorinaq.com/

[23] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, *Numerical recipes*, Cambridge University Press, 2007. http://www.nr.com

[24] R. Anderson, D. Gollmann, B. Preneel, C.S.W. (1993), FSE., and W. on Fast Software Encryption, *Fast software encryption: proceedings ; ... international workshop, FSE ..... Cambridge, UK, February 21 - 23, 1996*, Springer, 1996. http://books.google.com/books?id=yngAFrKFsd4C

[25] B. Jenkins, "ISAAC: a fast cryptographic random number generator." http://www.burtleburtle.net/bob/rand/isaacafa.html

[26] E. Reinhard, G. Ward, S. Pattanaik, and P. Debevec, *High dynamic range imaging: acquisition, display, and image-based lighting*, Morgan Kaufmann, 2006. http://portal.acm.org/citation.cfm?id=1208706

[27] M.D. Fairchild, "Color Appearance Models: CIECAM02 and Beyond," 2008

[28] P. Ledda, A. Chalmers, T. Troscianko, and H. Seetzen, "Evaluation of tone mapping operators using a High Dynamic Range display," *ACM Transactions on Graphics*, *ACM*, vol. 24, 2005, p. 640. doi:10.1145/1073204.1073242

[29] S. Draves and E. Reckase, "flam3 Wiki Page," 2011. http://code.google.com/p/flam3/w/list

[30] L. Yang, D. Nehab, P.V. Sander, P. Sitthi-amorn, J. Lawrence, and H. Hoppe, "Amortized supersampling," *ACM Transactions on Graphics*, vol. 28, 2009, p. 1. doi:10.1145/1618452.1618481

[31] M. Schwarz and M. Stamminger, "Multisampled Antialiasing of Per-pixel Geometry," *Eurographics*, 2009, pp. 21-24. http://www.mpi-inf.mpg.de/\~mschwarz/papers/msaappg-eg09.pdf

[32] K. Beets and D. Barron, "Super-sampling Anti-aliasing Analyzed," *Beyond3D*, 2000, p. 22. http://www.beyond3d.com/content/articles/37/

[33] M. Segal and K. Akeley, "The OpenGL Graphics System: A Specification (Version 1.5)," 2003. http://www.opengl.org/documentation/specs/version1.5/glspec15.pdf

[34] P. Young, "Coverage Sample Aliasing Technical Report," 2002. http://tinyurl.com/3jyq6g3

[35] A. Reshetov, "Morphological antialiasing," *Proceedings of the 1st ACM conference on High Performance Graphics - HPG '09*, *ACM Press*, 2009, p. 109. doi:10.1145/1572769.1572787

[36] A. Buades, B. Coll, and J.M. Morel, "On image denoising methods," *SIAM Multiscale Modeling and Simulation*, 2005, pp. 490-530. http://www.cmla.ens-cachan.fr/fileadmin/Documentation/Prepublications/2004/CMLA2004-15.pdf

[37] F. Suykens, K.U. Leuven, and Y. Willems, "Adaptive Filtering for Progressive Monte Carlo Image Rendering," *Eighth International Conference in Central Europe on Computer Graphics, Visualization and Interactive Digital Media*, Plzen, Czech Republic: 2000. http://graphics.cs.kuleuven.be/publications/DESCREEN/descreen.pdf.gz

[38] S. Paris, P. Kornprobst, J. Tumblin, and F. Durand, "Bilateral Filtering: Theory and Applications," *Foundations and Trends in Computer Graphics and Vision*, vol. 4, 2008, pp. 1-75. doi:10.1561/0600000020

[39] K. Huang, D. Zhang, and K. Wang, "Non-local means denoising algorithm accelerated by GPU," *Proceedings of SPIE*, SPIE, 2009, pp. 749711-749711. doi:10.1117/12.833025

[40] A. Adams, J. Baek, and M.A. Davis, "Fast High-Dimensional Filtering Using the Permutohedral Lattice," *Computer Graphics Forum*, Wiley Online Library, 2010, pp. 753-762. doi:10.1111/j.1467-8659.2009.01645.x

[41] A. Adams, N. Gelfand, J. Dolson, and M. Levoy, "Gaussian kd-trees for fast high-dimensional filtering," *ACM Transactions on Graphics (TOG)*, *ACM*, vol. 28, 2009, pp. 1-12. doi:10.1145/1531326.1531327

[42] A. Moore, *A tutorial on kd-trees*, 1991. http://www.autonlab.org/autonweb/14665/version/2/part/5/data/moore-tutorial.pdf?branch=main\&language=en

[43]"Datasheet: 2G bits GDDR5 SGRAM EDW2032BABG," *Elpida Memory, Inc.*, 2011

[44] R.L. Cook and L. Carpenter, "The Reyes Rendering Architecture," *Computer Graphics*, vol. 21, 1987, pp. 95-102. http://graphics.pixar.com/library/Reyes/

[45]"POWERVR MBX Technology Overview," *Imagination Technologies, Inc.*, 2009, pp. 1-17

[46] B. Furht, *Encyclopedia of multimedia*, Springer-Verlag New York Inc, 2008

[47]"CCITT Recommendation T.81," *International Telecommunication Union*, 1992