Accelerated rendering of fractal flames

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

December 7, 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.

1.5 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.

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.

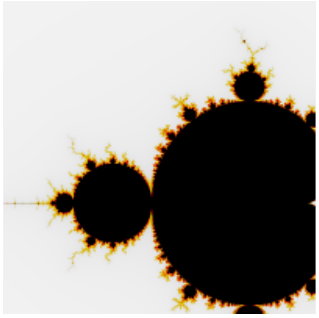


Figure 2.1: The Mandelbrot Set

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.

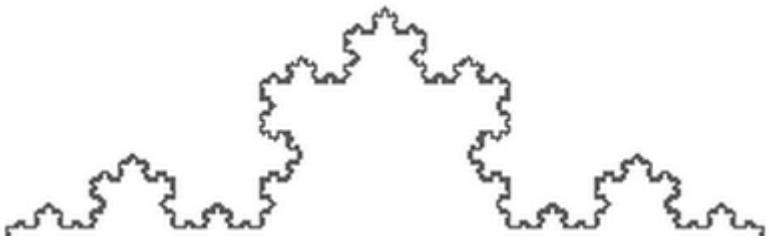


Figure 2.2: The Koch Curve

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!

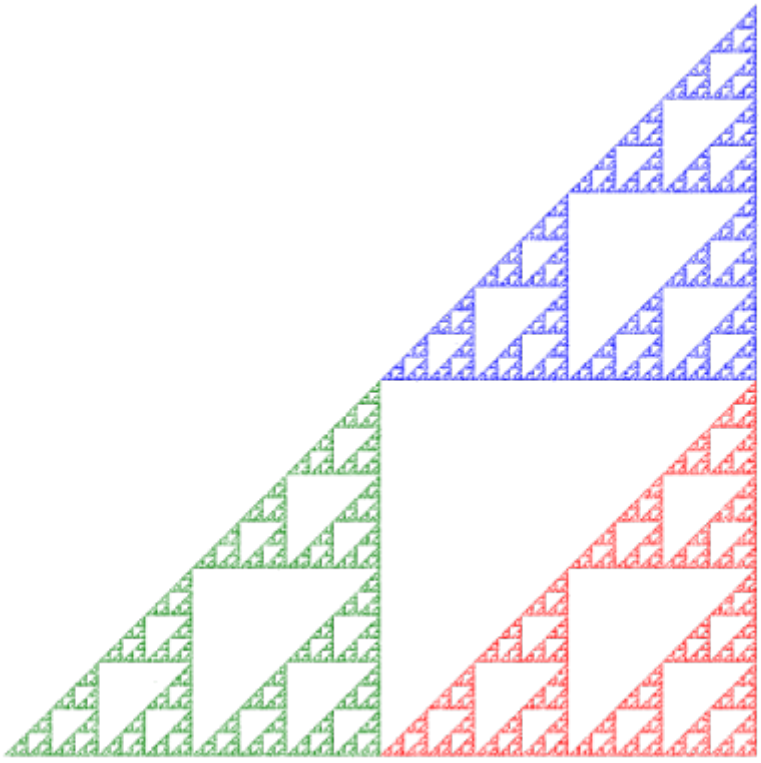


Figure 2.4: A visual of Sierpinski's Triangle which has a fractal dimensionality.

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.

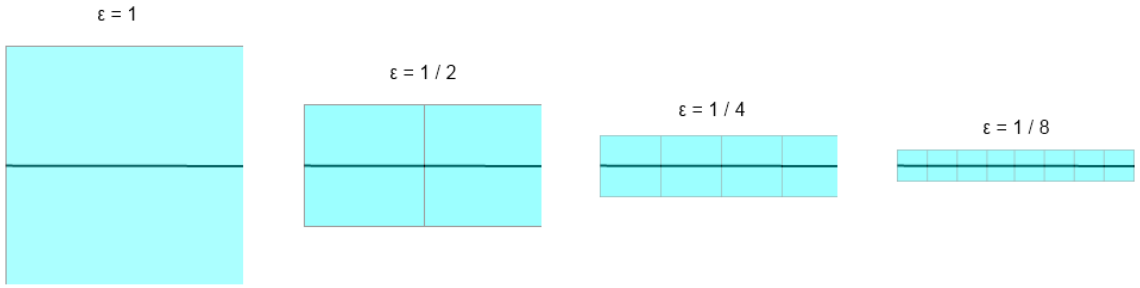


Figure 2.5: Box Counting Dimension Process For a Line

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.

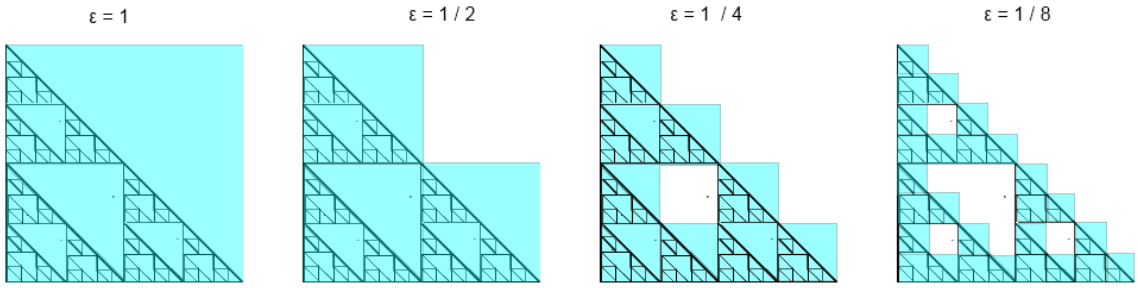


Figure 2.6: Box Counting Dimension Process For Sierpinski's Gasket

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.

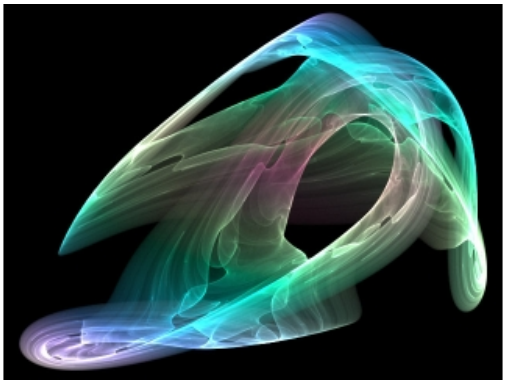


Figure 2.7: Image of a Strange Attractor

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.

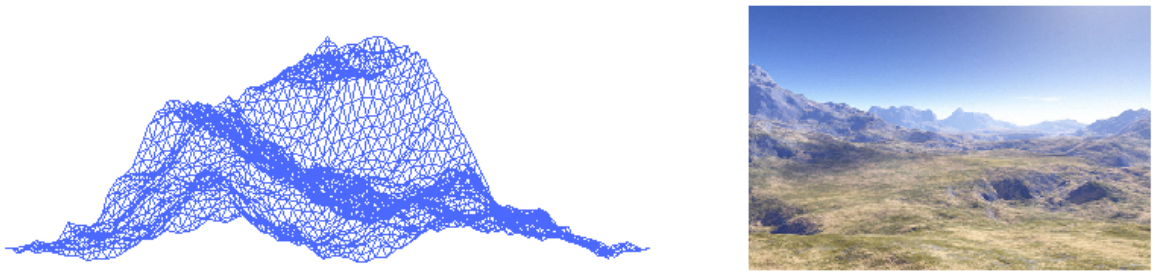


Figure 2.8: Image of a computer generated fractal landscape compared with a mountain landscape

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.

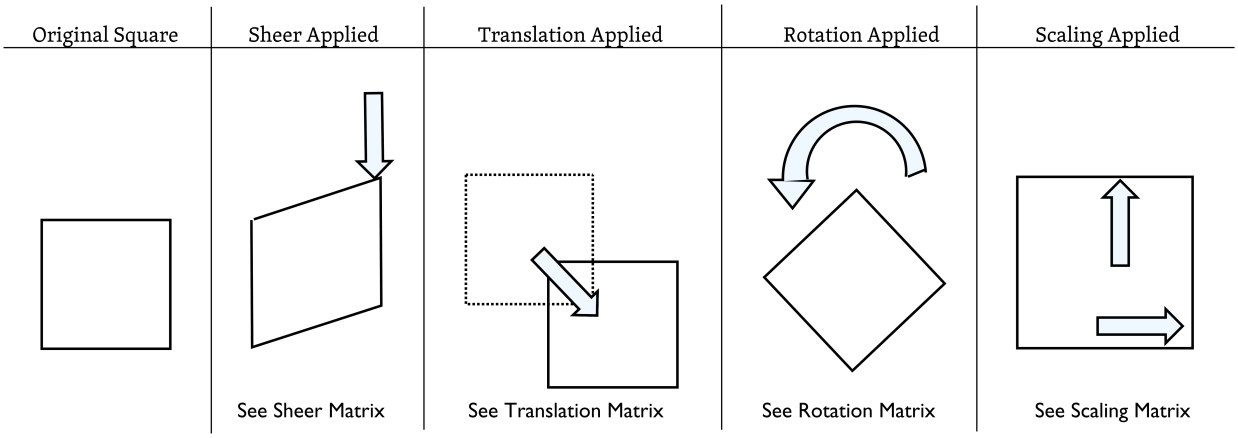


Figure 3.1: Visual representation of Shear, Translation, Rotation, and Scaling.

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.

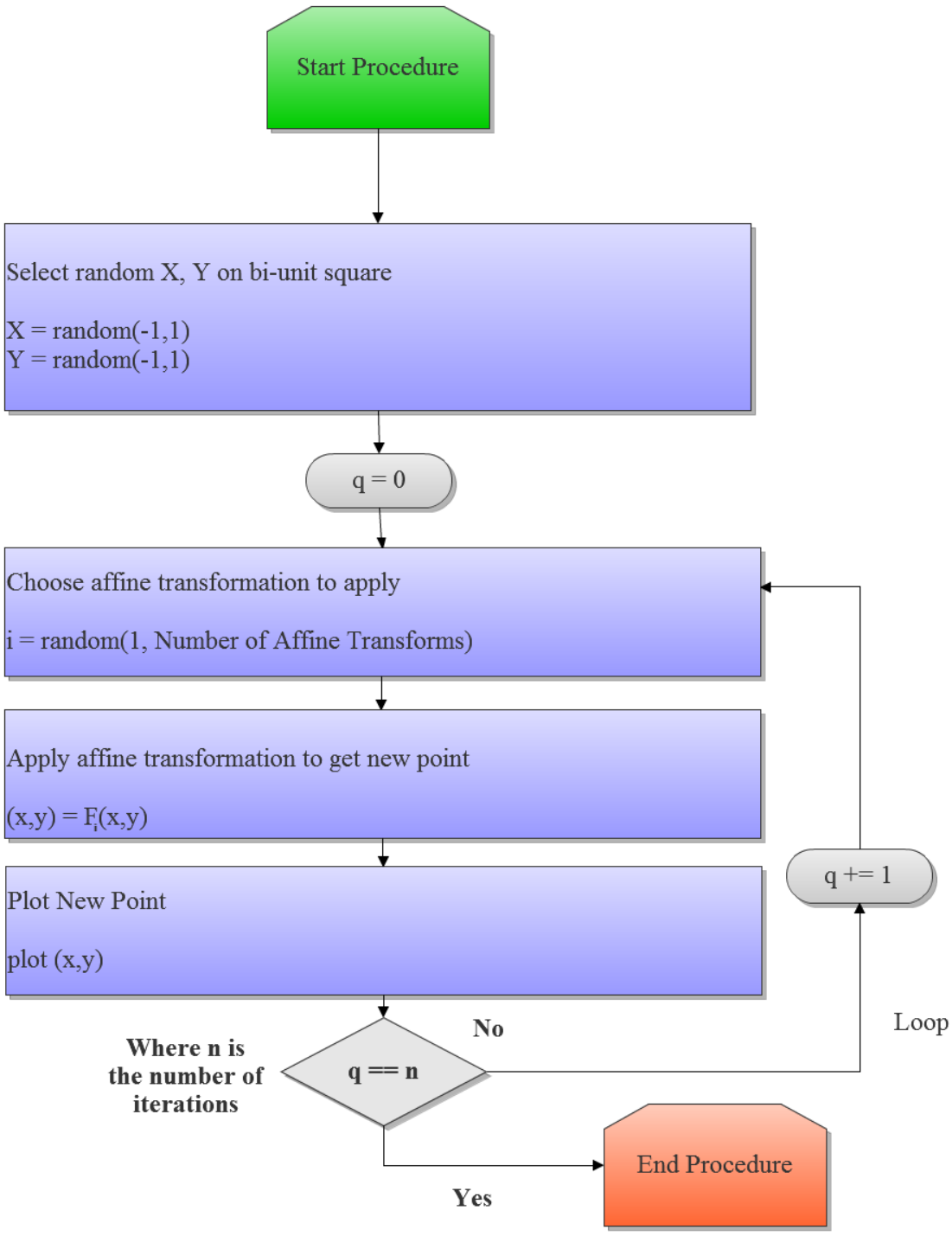


Figure 3.2: Flow chart of IFS Procedure

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:

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

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

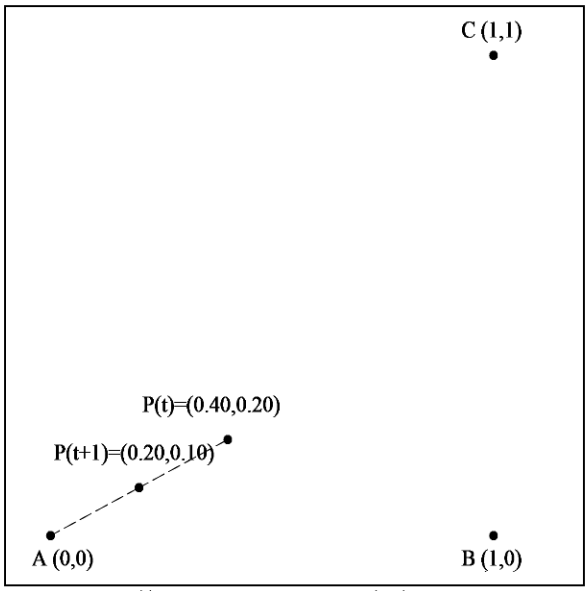
$$F_2 = \begin{vmatrix} \frac{1}{2} & 0 & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{vmatrix} \text{ selected with a probability of } \frac{1}{3}. \text{ (Pulls point towards Vertex C)}$$

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.

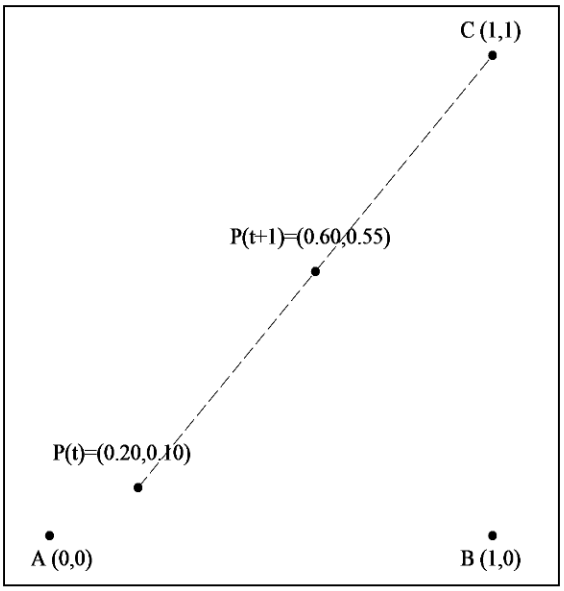
We now begin the *chaos game*. We first select a random point on the biunit square. In this case we have pseudorandomly selected x = 0.40 and y = 0.20. We then pseudorandomly pick transformations. The first four transformations shown are F_0, F_2, F_1, and then F_0. The application of these are shown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

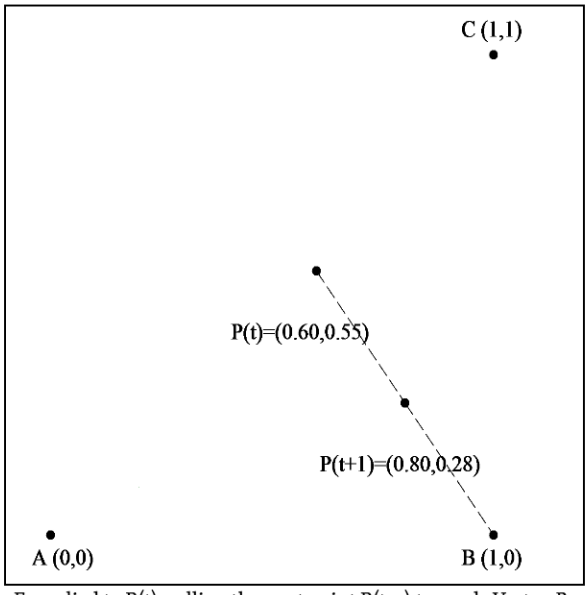
The more one stochastically samples, the closer the output image is to the solution of the Iterated Function System being computed.



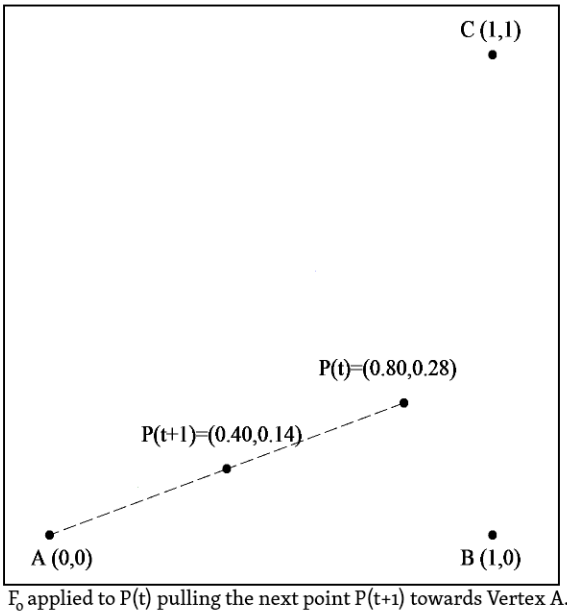
F_0 applied to P(t) pulling the next point P(t+1) towards Vertex A.



F_2 applied to P(t) pulling the next point P(t+1) towards Vertex C.

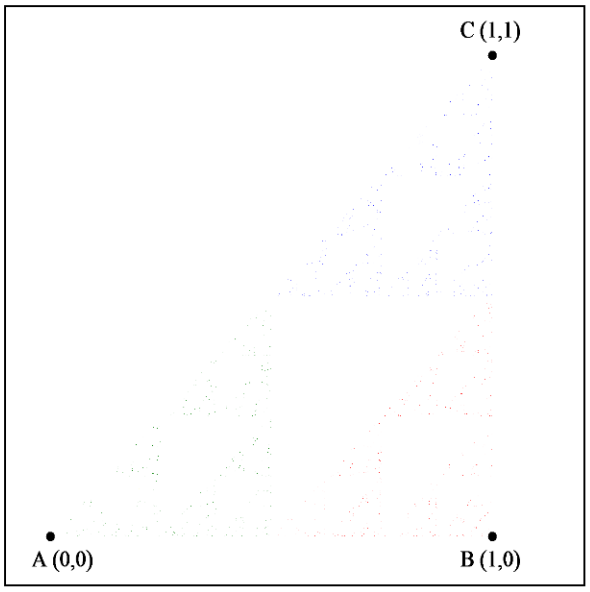


F_1 applied to P(t) pulling the next point P(t+1) towards Vertex B.

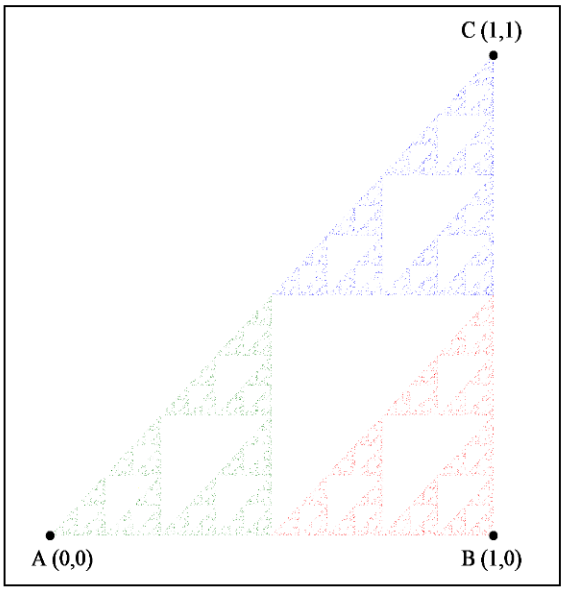


F_0 applied to P(t) pulling the next point P(t+1) towards Vertex A.

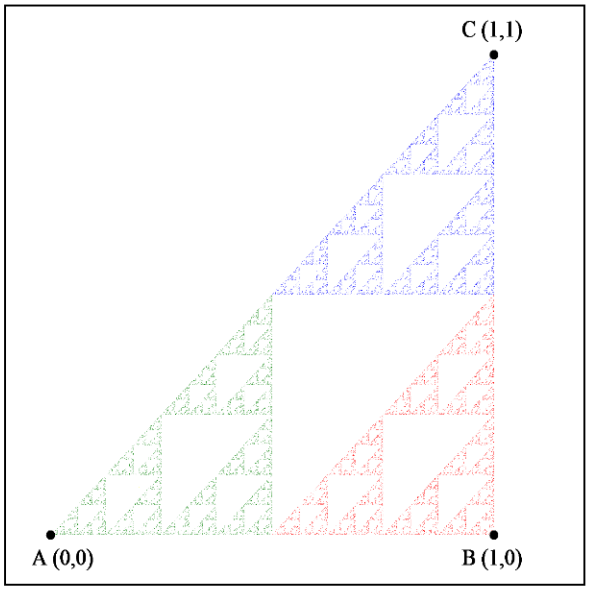
Figure 3.3: Illustrative process of the chaos game.



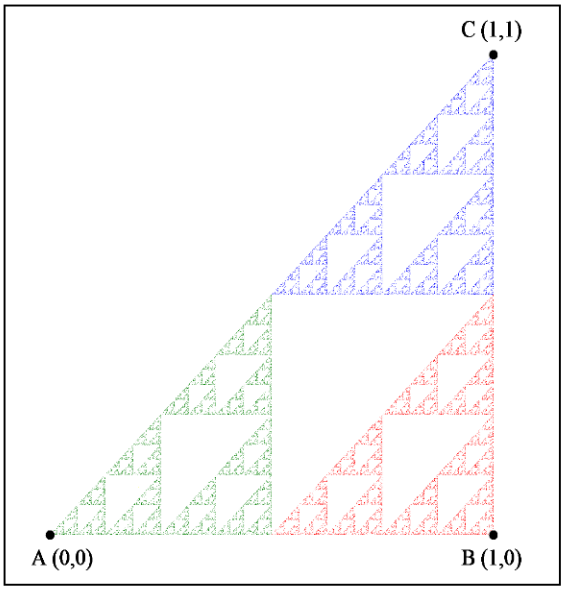
Sierpinski's Triangle after 1,000 iterations.



Sierpinski's Triangle after 2,500 iterations.



Sierpinski's Triangle after 15,000 iterations.



Sierpinski's Triangle after 25,000 iterations.

Figure 3.4: Sierpinski's Triangle after 1,000, 7,500, 15,000, and 25,000 iterations.

Classical Iterated Function System: Barnsley's Fern

As a more intricate example, the classical iterated function system called Barnsley's Fern is presented. This system was introduced by the mathematician Michael Barnsley in *Fractals Everywhere* [5]. This example is suitable to show all of the operations of an affine transform : shear, scale, rotation, and scaling.

To construct Barnsley's Fern using the chaos game we need to describe the affine transformations that will be used. Using the most basic version of an affine transformation (which use vecotr multiplication and vector addition), we can describe the system with the following 4 transformations seen below. As a note, the affine transformations of this system are not equally weighted and have their own probabilistic model associated with each [5].

$$A_0 = \begin{vmatrix} 0.00 & 0.00 \\ 0.00 & 0.16 \end{vmatrix} \; b_0 = \begin{vmatrix} 0.00 \\ 0.00 \end{vmatrix} \text{ selected with a probability of 0.01.}$$

$$A_1 = \begin{vmatrix} 0.85 & 0.04 \\ -0.04 & 0.85 \end{vmatrix} \; b_1 = \begin{vmatrix} 0.00 \\ 1.60 \end{vmatrix} \text{ selected with a probability of 0.85.}$$

$$A_2 = \begin{vmatrix} 0.20 & -0.26 \\ 0.23 & 0.22 \end{vmatrix} \; b_2 = \begin{vmatrix} 0.00 \\ 1.60 \end{vmatrix} \text{ selected with a probability of 0.07.}$$

$$A_3 = \begin{vmatrix} -0.15 & 0.28 \\ 0.26 & 0.24 \end{vmatrix} \; b_3 = \begin{vmatrix} 0.00 \\ 0.44 \end{vmatrix} \text{ selected with a probability of 0.07.}$$

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

$$F_0 = \begin{vmatrix} 0.00 & 0.00 & 0.00 \\ 0.00 & 0.16 & 0.00 \\ 0.00 & 0.00 & 1.00 \end{vmatrix} \text{ selected with a probability of 0.01.}$$

$$F_1 = \begin{vmatrix} 0.85 & 0.04 & 0.00 \\ -0.04 & 0.85 & 1.60 \\ 0.00 & 0.00 & 1.00 \end{vmatrix} \text{ selected with a probability of 0.85.}$$

$$F_2 = \begin{vmatrix} 0.20 & -0.26 & 0.00 \\ 0.23 & 0.22 & 1.60 \\ 0.00 & 0.00 & 1.00 \end{vmatrix} \text{ selected with a probability of 0.07.}$$

$$F_3 = \begin{vmatrix} -0.15 & 0.28 & 0.00 \\ 0.26 & 0.24 & 0.44 \\ 0.00 & 0.00 & 1.00 \end{vmatrix} \text{ selected with a probability of 0.07.}$$

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).

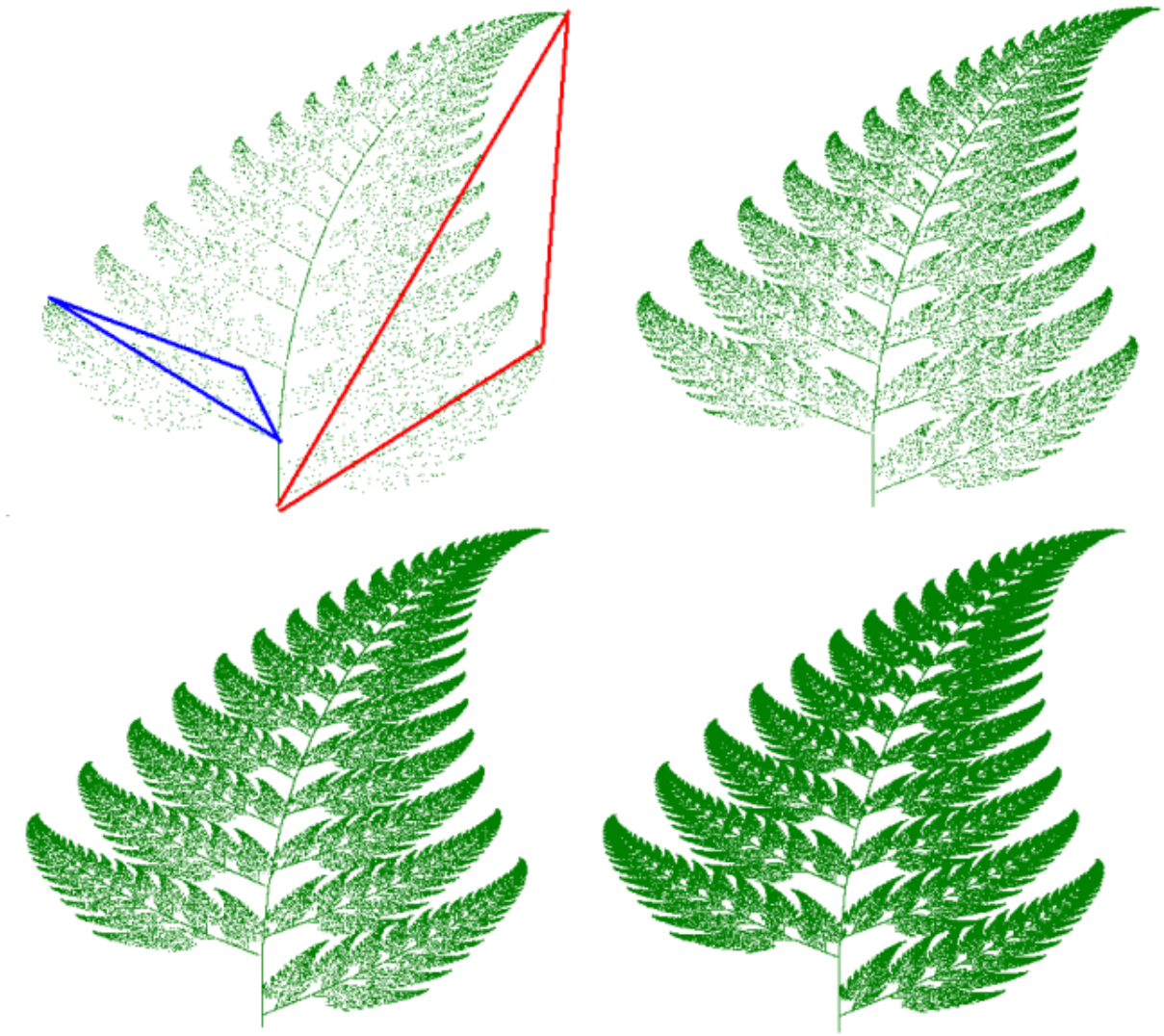


Figure 3.5: The formation of the Iterated Function System called Barnsley's Fern.

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

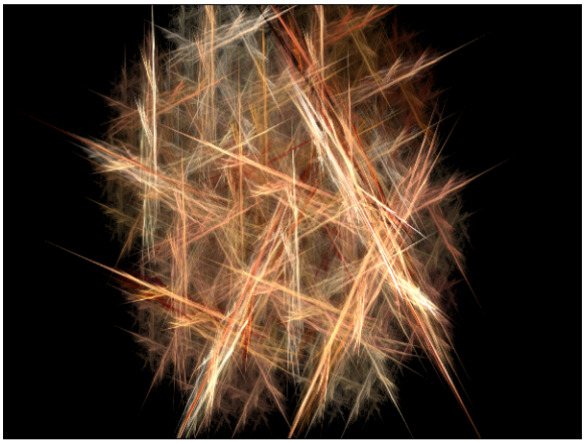
In the classical Iterated Function System, described previously, points were either members in the set or not. For every subsequent time the chaos game selected a point that was already shown to have membership in the set information was lost about the density of the points. To remedy this for the fractal 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.

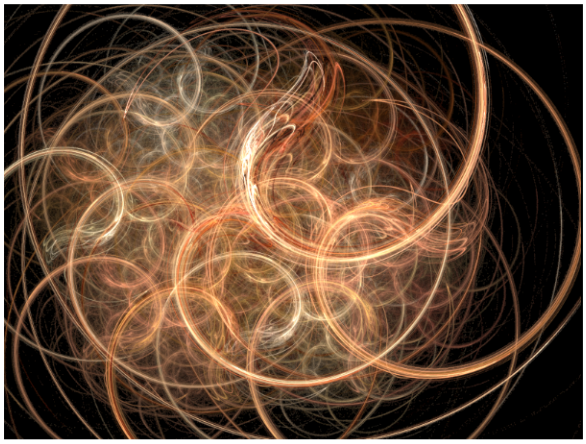
Visual representations of a flame using a binary, linear, as well as logarithmic mapping for the display can be seen in Figure 3 of Drave's original paper on the flame algorithm[2].

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.

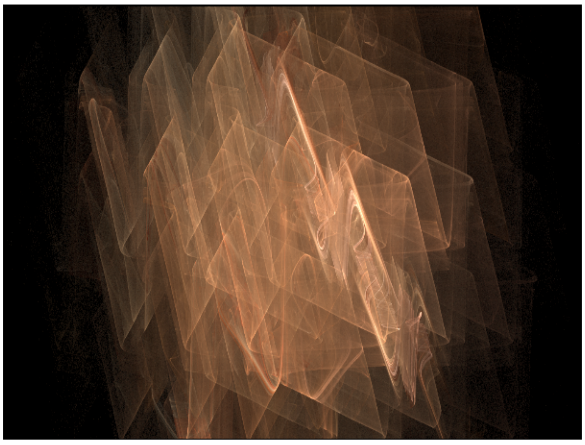
[3]This transformation isn't applied directly to the computational loop and is merely for visual output.



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 11.2 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 11.2 and Section 11.2.

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 11.5.

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:

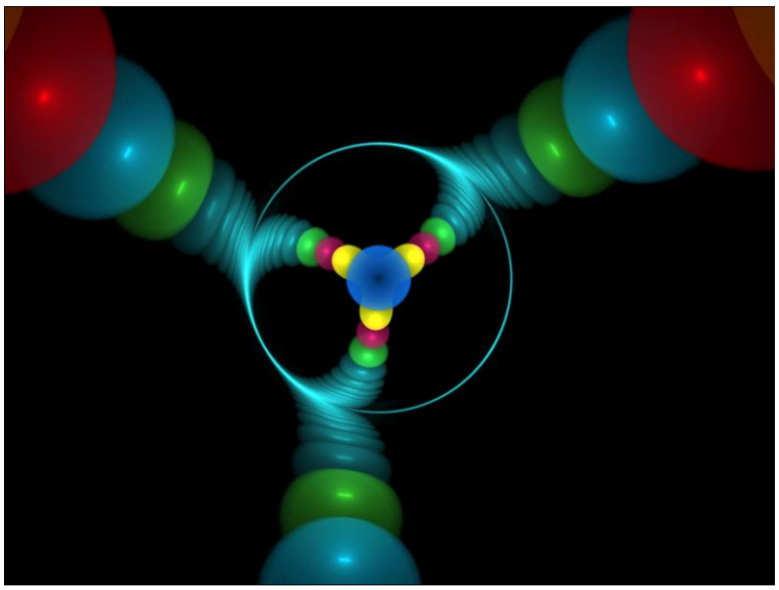
- $\text{Rotational Transforms}_1 = 60^\circ$
- $\text{Rotational Transforms}_2 = 120^\circ$
- $\text{Rotational Transforms}_3 = 180^\circ$
- $\text{Rotational Transforms}_4 = 240^\circ$
- $\text{Rotational Transforms}_5 = 300^\circ$

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 14.1 and 14.1 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 14.2.

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 Chapter 14. 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 11.5) and Highlight Power (Section 11.5) 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.

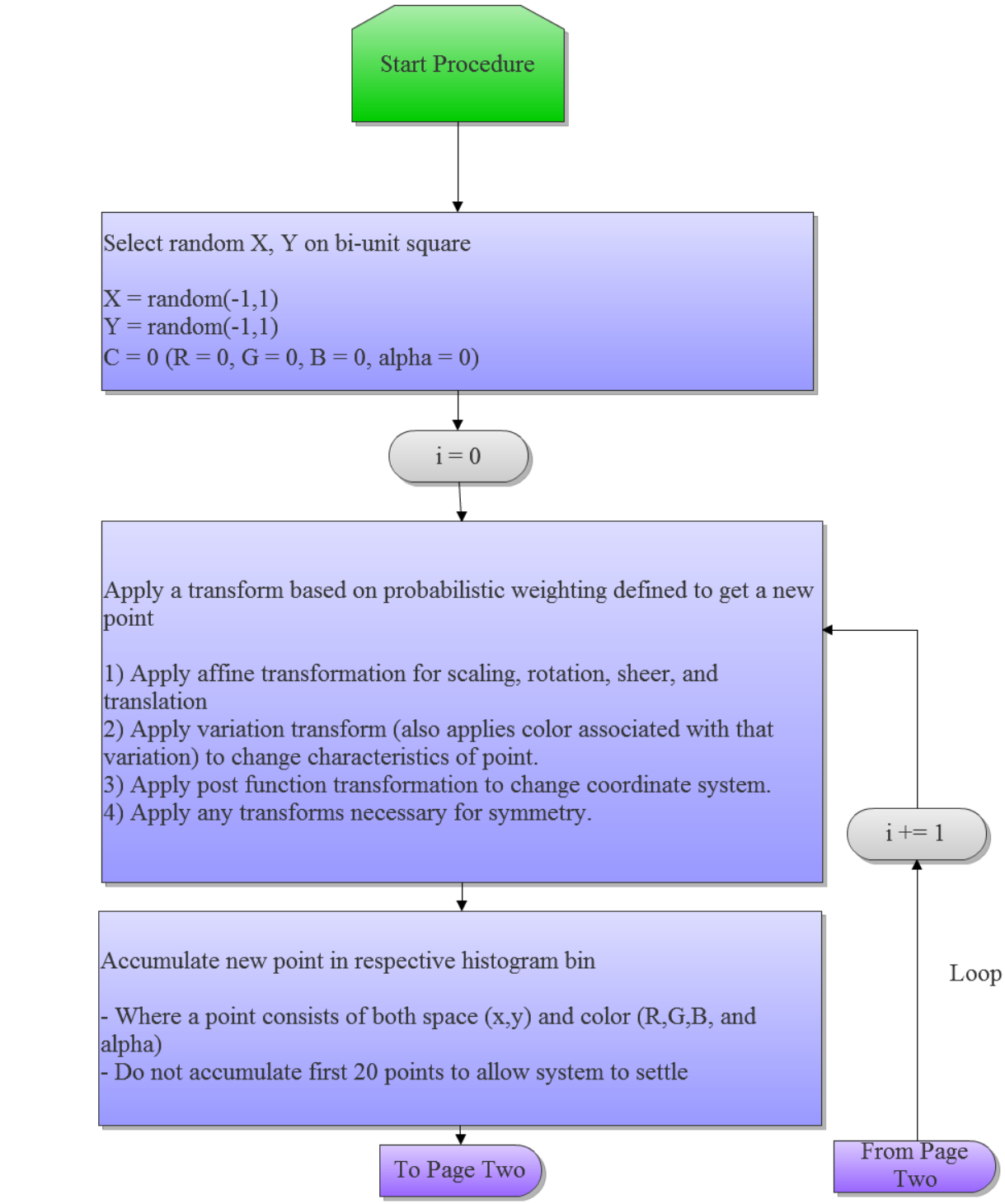


Figure 3.8: A flowchart describing the fractal flame algorithm.

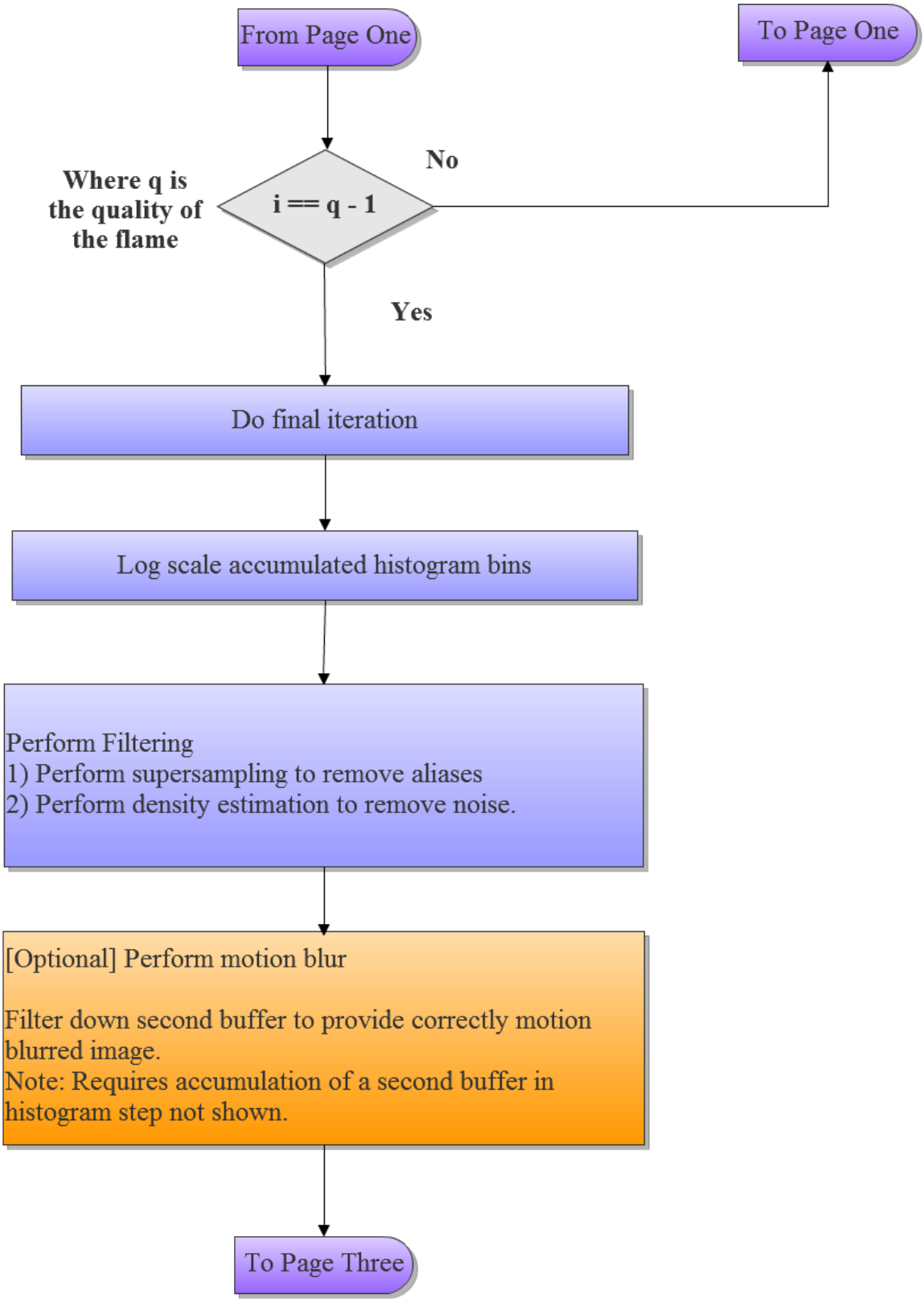


Figure 3.9: A flowchart describing the fractal flame algorithm.

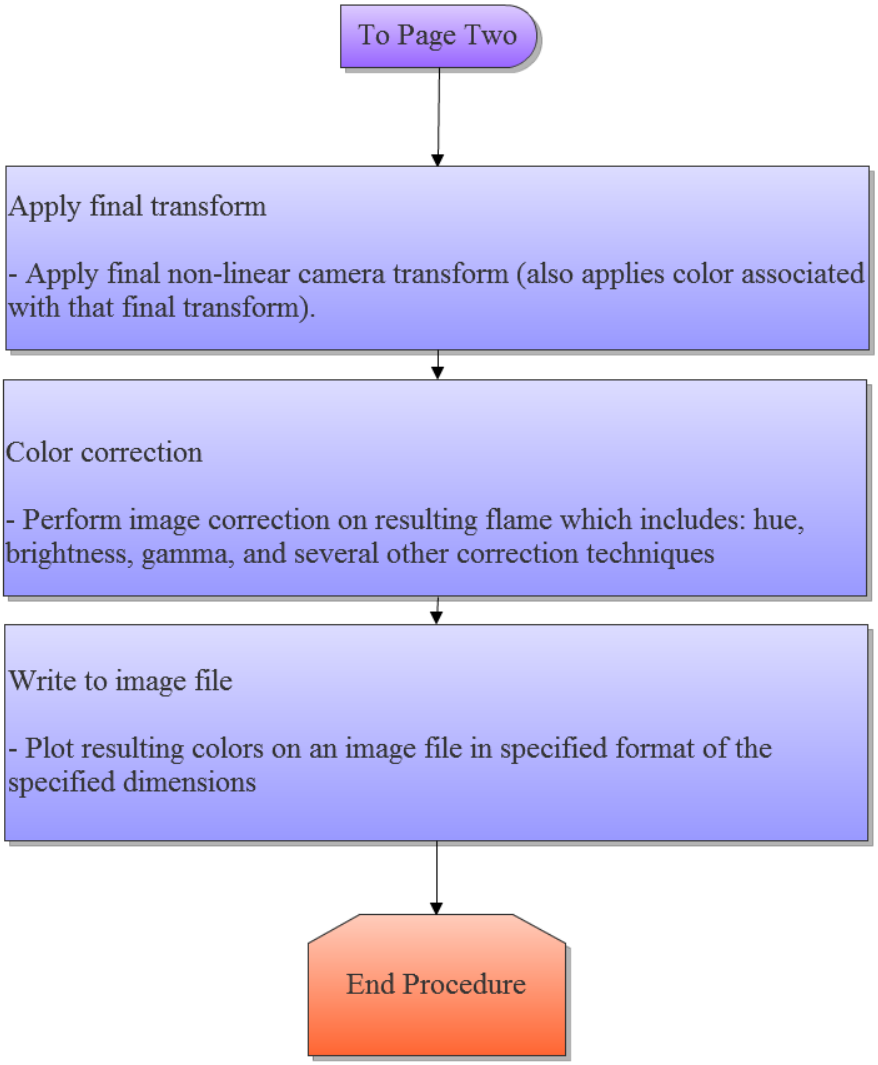


Figure 3.10: A flowchart describing the fractal flame algorithm.

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. However, since it is not an open product, we cannot directly adopt these techniques for our GPU-based implementation.

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].

6.3 Host language and intermediate language

The typical host language for CUDA development is C++. The CUDA toolchain includes compiler extensions and syntactic sugar to make many tasks simple, and the device code compiler supports a subset of C++ features, including classes and templates. Despite its name, however, the runtime API used for native C++ development with CUDA does not support run-time code generation, and is thus unsuitable for this project. No Cuburn host code lies in a performance-critical path, so without the tight integration offered by the CUDA toolchain, there is little incentive to use systems programming languages like C or C++.

In order to produce code at runtime in a structured, stable way, we would need a powerful intermediate language whose capabilities exceed those of, say, the C preprocessor by a considerable margin. Most of the host code will have similar structure regardless of language, so the differences that should be considered most closely are in language construction. From the set of languages with which the authors were familiar, two stood out as being suitable for this task: Python and Haskell.

Haskell is a lazy, pure, functional language with a remarkably expressive static type system and excellent support for both traditional domain-specific languages (with excellent native parsers such as Parsec and compile-time evaluation of native expressions using Template Haskell) and embedded DSLs (via infix operators, rebindable syntax, and other language features [23]). Haskell's purity and type safety make it an ideal host language to build run-time code generation facilities that feature compile-time analysis. However, Haskell also has a steep learning curve, which would place a significant burden on those developers not already familiar with it.

Python is a dynamic, interpreted programming language often cited as a counterpoint to Haskell (though truthfully these languages agree more than not on problem-solving approach, both in philosophy and implementation). Its rich object model, duck-typing of numerics, and monad-like ContextManager allow for the extraction of instruction streams from "pure" mathematical code, a technique employed by one of the authors in the PyPTX library for dynamic GPU kernel generation [24]. Experimentation with PyPTX, however, revealed shortcomings inherent in the expression of EDSLs in Python. Inside a code generation context, operations on PTX variables would trigger code generation, whereas normal operations would not;

[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.

the inclusion of a block of code in the output was contingent on whether that code was evaluated on the host. It became extremely difficult to separate both host and device flow, and complicated bugs would arise in edge cases along code generation paths which could not be detected in advance. For similar reasons, the backtracking context needed to provide type and data inference in the EDSL was complex and error-prone, and loops could not be tracked across host function call boundaries. In short, Python's flexibility in host code provided too many opportunities for improper code generation.

During initial development of cuburn, we elected to pursue a Haskell-based EDSL, relying on the language's own flexible static type checker to guarantee properties of the program. What we found in this approach is that the process of constructing these guarantees for our own program exceeded the time spent experimenting with the algorithms and hardware we were writing. Since our software is non-critical and offline, even severe bugs would have relatively low impact, so we decided to abandon the static guarantees of Haskell's type system and the tight integration with host code afforded by EDSL generation. With these considerations, the learning curve of Haskell outweighed its potential benefits, and we elected to use Python.

As an intermediate language, we chose a textual templating system with little semantic awareness but an ability to call on native code, and augmented that with Python code that echoed the monadic style of Haskell during code generation. This enabled the use of a hybrid of literal CUDA code inline template operators, much like typical HTML template engines, but in this case instead of accessing application data the templated statements actually create it.

CHAPTER 7

RUNTIME CODE GENERATION

The diversity of variations available in the flam3 and Apophysis implementations of the flame algorithm are a boon to fractal flame artists, but a curse upon implementors. While the selection of variation functions is somewhat standardized — by the simple expedient of declaring that whatever the latest version of flam3 supports is the standard — the functions themselves vary widely, from simple polynomial expressions to those that require thousands of CPU instructions and as many as ten parameters from the host.

In flam3, this is implemented by use of a cascade of `if/else` conditions, which check whether or not a variation is used and, if so, apply it. Even on CPU, this is not a terribly efficient way to handle the more than 100 variations; the branch structure requires tens of thousands of unused instructions from inlined variation functions to be brought into the instruction cache only to be dumped again thereafter. (A `switch` statement is more easily recognized by optimizing compilers, and more likely to be turned into a jump table or other efficient structure.) Nevertheless, the overhead for this dispatch technique is relatively small compared to the overall runtime of an iteration on CPU.

On GPU, however, the performance of such a structure is not so much suboptimal as farcical. This is in some part due to the hardware restrictions inherent in massively parallel devices; GPUs lack branch predictors, for instance, and the per-core instruction caches of Fermi GPUs are estimated to be on the order of 64 bytes. It is also partly due to a focus on floating-point hardware in current devices. Instructions in both the load and comparison classes execute at less than full throughput per clock — comparison being half of full throughput, load being at most a quarter, and typically much less — meaning that this conditional cascade, which must be executed billions of times per frame, would itself require more GPU cycles than all other components of the iteration kernel combined.

Even without the performance problems that arise from the conditional cascade itself, a branch-based solution to variation selection will *still* have a considerable negative impact on performance. GPUs do not perform on-the-fly instruction reordering, but can dual-issue instructions to shared functional units in some cases. The compiler performs instruction reordering to allow dual-issue to occur more often, and will also reorder instructions to reduce pipeline delays on dependent operations when there are insufficient active warps to hide pipeline latency. Instruction reordering can't cross basic block boundaries, however, limiting the impact these techniques could have on the computationally expensive variation functions. Common subexpression elimination, where multiple identical idempotent statements are replaced with a single invocation, similarly cannot cross basic blocks; this can be particularly costly since many variations use expensive trigonometric routines like `atan2`.

Additionally, no matter how it is scheduled to run, the mere presence of variation code within a kernel function is enough to reduce performance. This is due to CUDA's approach to register allocation. Each Fermi shader core can allocate a configurable number of registers from the 128KB register file to a thread. However, the number of registers used must be selected at compile-time, and must be the same for every thread. With

a pipeline depth of 22 cycles, Fermi requires at least that many warps to be ready to run in as many cycles in order to avoid a stall, and since thread swapping (discussed in Chapter 8) requires power-of-two block sizes and will regularly stall blocks for synchronization, avoiding stalls altogether requires 32 warps of 32 threads per shader core. With this many threads, an even division of the register file leaves just 32 registers per thread.

Without runtime register count changing, a kernel will use as many registers as that used in its most heavily-occupied point. Variation invocation occurs in the center of the inner loop, when the largest number of variables are in scope. As a result, total register use with the conditional cascade approach to variation selection is constrained by the register use of the most complex variation, and this number cannot be reduced by instruction reordering for the reasons noted above. After porting the full set of variations from `flam3`, register usage was so high that only half of the needed occupancy was obtainable.

The problems surrounding the variation functions were of such magnitude that we elected to pursue an approach which would be otherwise discouraged due to its complexity: we constructed a system to perform runtime code generation. Rather than distributing a compiled code object containing the kernels intended for GPU exeecution, we distribute an annotated repository of CUDA code snippets embedded into our Python library. When a flame is loaded for rendering, its genome is analyzed to determine which segments of device code are needed. These segments are gathered and joined into a consistent CUDA `.cu` file, which is then passed to PyCUDA's caching `nvcc` frontend to obtain a GPU code object in ELF format that can be loaded by the CUDA driver and copied to the device.

By selecting source code segments at runtime, we can eliminate the conditional cascade by building separate inlined functions for each xform that include the exact set of variation functions. We can also remove the conditional guards around those variations, allowing them to be merged into a single basic block and optimized more effectively. While this technique adds a considerable amount of complexity to the host side, the improved performance and flexibility in device code cannot be obtained otherwise.

While initially considered primarily to solve the variation problem described above, the runtime code generation system has found use in simplifying or improving many parts of cuburn's device code. In some cases, these improvements are small, yielding execution time changes of one or two percentage points; in others, the flexibility it offers has proved critical to reaching our performance targets. One such case is the method for interpolating and accessing control point parameters.

CHAPTER 8

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.

8.1 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.

8.2 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.

8.3 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

$$P(S_l) = P(n)^l = \frac{1}{N^l} \tag{8.1}$$

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 (8.1)	0.0156	$2.441 \cdot 10^{-4}$	$5.960 \cdot 10^{-8}$	$1.262 \cdot 10^{-29}$
$T=256$	No shuffle (8.2)	0.2314	0.1352	0.1218	0.1216
	Ring shuffle (8.3)	0.0556	$3.145 \cdot 10^{-3}$	$1.013 \cdot 10^{-5}$	$1.144 \cdot 10^{-20}$
	Full shuffle (8.4)	0.0535	$2.8658 \cdot 10^{-3}$	$8.213 \cdot 10^{-6}$	$4.550 \cdot 10^{-21}$
$T=512$	No shuffle (8.2)	0.0753	0.0608	0.0607	0.0607
	Ring shuffle (8.3)	0.0332	$1.1113 \cdot 10^{-3}$	$1.261 \cdot 10^{-6}$	$2.748 \cdot 10^{-24}$
	Full shuffle (8.4)	0.0317	$1.006 \cdot 10^{-3}$	$1.011 \cdot 10^{-6}$	$1.048 \cdot 10^{-24}$

Table 8.1: 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

$$P(S_l) = \frac{W-1}{T-1} + (1 - \frac{W-1}{T-1}) \cdot \frac{1}{N^l} \tag{8.2}$$

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

$$\begin{aligned} P(S_l) &= P(V) \cdot P(S_l|V) + P(\bar{V}) \cdot P(S_l|\bar{V}) \\ &= \frac{T/W-1}{T-1} \cdot \frac{1}{N^l} + \frac{T-T/W}{T-1} \cdot (\frac{W-1}{T-T/W} \cdot 1 + \frac{T-W-W/T+1}{T-T/W} \cdot \frac{1}{N})^l \end{aligned} \tag{8.3}$$

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

$$P(S_l) = (\frac{W-1}{T-1} \cdot 1 + (1 - \frac{W-1}{T-1}) \cdot \frac{1}{N})^l \tag{8.4}$$

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 8.1. 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 9

ANIMATING FRACTAL FLAMES

Fractal flames are uniquely interesting as animations. Well-designed fractal flames typically contain overwhelming amounts of what the human visual system perceives as objects in motion. Under proper viewing conditions, overloading the visual system in this way provides an almost hypnotic effect, and the word "mesmerizing" is often used to describe extended-length flame sequences. The coloration scheme in use by fractal flames also provides sub-pixel detail missing from other fractal rendering algorithms, allowing the human visual system to perform temporal interpolation to recover image detail at scales much finer than single-frame rendering or display.

Despite the distinctive visual qualities of fractal flame animations, animation has heretofore been a hands-off process. Interpolation for fractal flames is, in almost every case, handled by `flam3`'s genome tools, which leave extremely limited room for either artistic or programmatic exploration of the aesthetics and physiological impact of fractal flame animations. Since rendering was so expensive, this made sense; `flam3`, and the Electric Sheep project, were first written in an era where ordinary computers would take *hours* to render frames, instead of the minutes used by today's CPUs or the seconds it takes cuburn. However, with cuburn's ability to provide near-real-time feedback for animators and huge volumes of video for machine learning, the time has come to engineer more flexible animation tools.

9.1 Flocks

Flame animations are most commonly found as a flock. Flocks are generated from a set of still flames provided by users (and, in some cases, generated programmatically by blending two or more user-provided flames). Each still flame is animated to create a *loop* by rotating the primary affine transformation of each xform about that affine transformation's offset point. This rotation is performed at constant angular velocity over the duration of the loop. While this sometimes causes the resulting animation to give the perception of global frame motion, as if the "camera" or the "world" were spinning, it often results instead in a movement pattern which suggests that some of the "objects" in the frame are rotating across a fixed image grid. The motion of these "objects", when passed through variation functions, is key to providing an illusion of depth, and time-integration performed by our brains allows us to recover a sense of the "shape" of these variation functions.[1]

[1]The notions of objects passing through shapes in a world is entirely an artifact of the human visual system attempting to make sense of an astonishing amount of foreign information, and in a certain sense even the artists who design these flames are just poking at some numbers in a very peculiar spreadsheet. Nevertheless, humans' common heritage and neural structures allow an artist's intent to be received by the viewer intact, despite passing through an austere 4KB text file on the way. We can assure the reader that even years of careful study of fractal flame renders doesn't do a thing to shake the perception that a hundred thousand tiny snowflakes are dancing through crystal, or that a bolt of lightning just gave a glimpse into an alien landscape.

When creating a flame for a loop animation using XML tools, the artist may specify that certain xforms are not to be rotated, the number of frames to be rendered, and the width of the temporal multisampling as a ratio of a single frame's duration. When submitting to the Electric Sheep project, as most flame artists have done historically to get a loop, the duration and framerate are set by the server, so even that control is removed. Compared to the unbounded set of possibilities, this is a bit stifling.

After creating loops, `flam3` will add *edges* to a flock. An edge creates an animation that joins two loops "seamlessly" by interpolating every value in a flame (apart from those related to video playback, such as frame resolution and rate) smoothly between two loops. This typically results in a morphing effect, where the shapes of the source loop slowly distort into unrecognizability and then resolve into the destination loop. Occasionally, singularities, zero-crossings, or other irregularities will result in more unusual phenomena, such as vibrant bursts of color or simply black frames.[2]

The result is a directed graph with cycles. Each graph node is a point in time when two animations have identical values for all parameters. These points are made to occur either at the presentation time of the first frame in a pre-rendered video file containing an animation, or at the "end" of the last frame in such a file (presentation time plus display time). If two such animation files are properly concatenated,[3] they will appear to form a single, smooth animation. In this way, a playback engine can engage in a random walk of a flock, creating a continuous animation that uses variety in playback order to maintain novelty and user interest for far longer than sequential playback of all videos in the flock would suggest.

9.2 XML genome sequences

The format used by `flam3` to describe animations is a simple extension of the XML format used to describe still images. An animation genome contains a separate XML `<flame>` element for every frame in the animation, each bearing a `time` property describing the center of the display time for that frame (PTS $+ 0.5 \cdot$ DTS, essentially, although DTS is always equal to 1 and the first frame's PTS is always 0).

This presents a problem for smooth playback. Since the frame's center time is specified by the `time` parameter, the first frame of the sequence has an effective unscaled PTS of -0.5. Encoding this animation will shift this PTS to 0, since negative presentation times are not allowed. As a result, the edge parameter set used in the directed graph is never explicitly specified. Edges are generated against the last frame in the file, instead of this phantom parameter set, and therefore there is a small but noticeable discontinuity in position when transitioning between two sets.

Even if this discontinuity is corrected for, so that the position is continuous in time, the use of linear interpolation between nodes in a flock by `flam3` means that the velocity of objects will change abruptly when transitioning between a loop and an edge. With motion blur enabled, this does more than simply break the illusion of physicality lent to objects by continuous velocity curves during animation; the abrupt change causes visible distortions in motion-blurred shapes.

The interpolated parameter sets[4] are generated for a frame by performing linear or Catmull-Rom interpolation, depending on the number of surrounding frames available. Since the frames themselves are generated using linear interpolation, the use of Catmull-Rom interpolation to generate parameter sets between frames is questionable, and may result in additional velocity discontinuities.

[2] The Electric Sheep project relies on users to identify which edges are good and which are not. We're interested in accomplishing the same task algorithmically, and hope to do so given time.

[3] Few media containers can be concatenated at the bitstream level without remuxing. We have chosen one that can (MPEG-2 Transport Stream without Blu-Ray timecode extensions).

[4] These are called "control points" in `flam3` parlance, which is patently incorrect; the specified frames are the control points, not the result of interpolation. The authors have been sticking to `flam3` terminology when possible, including using "control point" in previous documents, but this is just too dang wrong to continue doing.

Regardless of the technical limitations, the XML format is simply cumbersome to use. The genome file for an animation of reasonable length can be hundreds of megabytes in size, and the mix of implicit and explicit ordering within the file resists hand-editing, splitting, and merging of files. Animation files are almost never themselves edited, due to their size and that the output doesn't identify keyframes or transmit them intact. It is possible to design a tool which uses different interpolation strategies or exposes additional artistic options to the user, but the limits of the `flam3` tools mean that this format would either have to be interpreted and converted to `flam3`'s format on the rendering host, or exported beforehand, suffering all the indignations inflicted by the XML genome format.

9.3 Cuburn genome format

Instead of building a layer on top of `flam3`-style XML files, we have decided to create a new format for representing genomes. This format uses a JSON-compatible object model, and so may be embedded in any suitable container.

In cuburn's genomes, all parameters are represented by Catmull-Rom cubic splines. Animations are always represented as occurring from $t = 0$ to $t = 1$, with the start time corresponding to the start of the presentation of the first frame, and likewise for the end of the last frame. Spline knots may be inserted at any temporal value; cubic interpolation ensures that the value of any parameter at the time of a knot is exactly that of the knot. This enables graph nodes in a flock, or any other animation constraint, to be hit precisely.

Spline knots can also be inserted outside of the rendered time values. Our interpolator uses this to match velocity in addition to value at transition points between animations, creating seamless transitions. This system accounts for variable duration when matching velocity, lifting `flam3`'s implicit restriction that all loops and edges must be the same length to get smooth results.

Because the splines are encoded independently in a simple JSON list, they can be easily hand-edited. More importantly, those edits are not transformed. When using a frame-based format, edits would require resampling the entire curve and storing the samples; applying additional edits at a later time would either require having stored the unquantized curve separately or running error-prone inference to approximate the significant knot positions. With this format, users and editing software no longer have to keep two versions of the same file in sync.

The format is not yet finalized; we are considering extending the spline description to include specification of a domain in which to scale a parameter for interpolation, such as reciprocal or logarithmic. This will present more flexibility in transitions between very different values or values which behave non-linearly as they drop to zero.

This new flame format supports import of still images in XML format, and can support export of still images or animations to the legacy format.

9.4 Implementing interpolation on device

The new format allows us to store an entire genome on the device and interpolate parameter sets for rendering as needed, which is simpler and more efficient than relying on host-based interpolation driven by foreign function calls to `flam3`. This process is efficient and fast, but getting there was not trivial.

CUDA GPUs have a provision for loading parameter sets from memory, known as constant memory. This is device memory which is shadowed by a small read-only cache at each GPU core with its own narrow address space configured to map to global memory by the host. Accesses to constant memory can be inlined into dependent operations with no overhead, not requiring a separate load instruction or temporary register, but

only if that access follows certain restrictions, chief among these that the access must use a fixed offset from the start of the memory space. If a non-fixed index is used, the code makes use of the normal memory data path, which is considerably slower.

In order to run chaos game iterations on thousands of temporal samples, we need to be able to load data from a particular parameter set. Doing so with constant memory requires either performing a separate kernel launch, with corresponding constant address space configuration, for each temporal sample, or using indexing to select a temporal sample at runtime. The former method leads to ineffecient load balancing, and the latter forces constant memory accesses to take the slow path.

The most common alternative to constant memory is shared memory, which can be described as an L1 cache under programmer control. Static-offset lookups from shared memory are not quite as fast as inline constant memory lookups, but are faster than indexed lookups. However, another problem presents itself: when represented as a complete data structure, the full parameter set exceeds the maximum available 48KB of memory and far outstrips the 2KB maximum size required to obtain sufficient occupancy to hide pipeline latency on current-generation hardware.

To retain the benefits of static-offset lookups without requiring a static data structure, we augmented the runtime code generator with a data packer. This tool allows you to write familiar property accessors in code templates, such as `cp.xform[0].variations.linear.weight`. The code generator identifies such accessors, and replaces them with a fixed-offset access to shared memory. Each access and offset are tracked, and after all code necessary to render the genome has been processed, they are used to create a device function which will perform the requisite interpolation for each value and store it into a global array. Upon invocation, each iteration kernel may then cooperatively load that data into shared memory and use it directly.

Each direct property access as described above triggers the device-memory allocation and copying of the original Catmull-Rom spline knots from the genome file for that property. In some cases, it can also be useful to store something other than a directly interpolated value. To this end, the data packer also allows the insertion of precalculated values, including arbitrary device code to perform the calculation during interpolation. The device code can access one or more interpolated parameters, which are themselves tracked and copied from the genome in the same manner as direct parameter accesses. This feature is used both for precalculating variation parameters (where storing the precalculated version as a genome parameter directly would be either redundant or unstable under interpolation), as well as for calculating the camera transform with jittered-grid antialiasing enabled (described in Chapter 14).

The generated function which prepares an interpolated parameter set on the device performs Catmull-Rom interpolation many times. To control code size, it is important to implement this operation in as few lines as possible. One step in variable-period Catmull-Rom interpolation involves finding the largest knot index whose time is strictly smaller than the time of interpolation. To implement this, we wrote a binary search (given in Figure 9.1) that requires exactly $3 \log N + 1$ instructions. We suspect this is not a novel algorithm, but we have not seen it elsewhere.

```
    ld.global.u32   rv,     [rt+0x100];
    setp.le.u32     p,      rv,     rn;
@p  add.u32         rt,     rt,     0x100;
```

Figure 9.1: One round of cuburn's unrolled binary search. In this particular set of instructions, a load instruction brings a value from global memory into a register. The address of this value by adding an offset representing 64 array positions to the current index value, 'rt'. The addition is performed in-line by the memory units. The next instruction tests to determine if the value 'rv' is less than or equal to the reference value 'rn', storing the result in predicate 'p'. If 'p' is set, the last instruction simply advances the index by the offset. Each round repeats the same instructions, halving the offset size each time.

CHAPTER 10

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.

10.1 Bias : An Illustrative Example

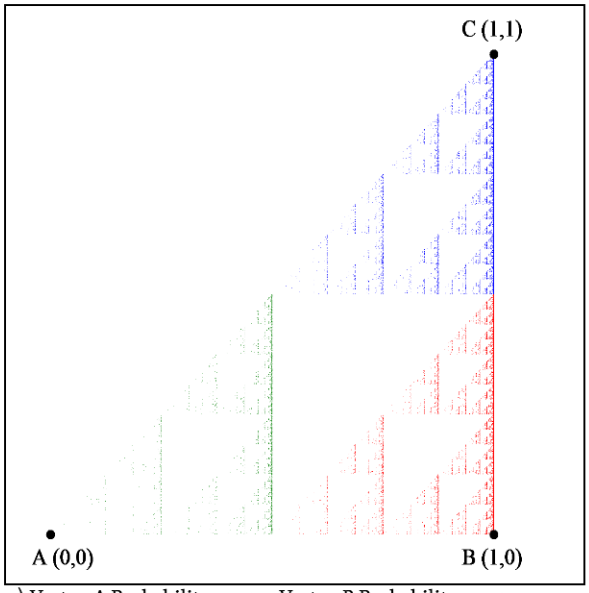
Before preceeding with selecting the proper PRNG for this project an illustrative example is shown. Normally in the Sierpinski's Triangle Iterated Function System each function has an equal probability of $\frac{1}{3}$ of being chosen (See Section 3.2 for the fully worked example with matrix transformation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

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

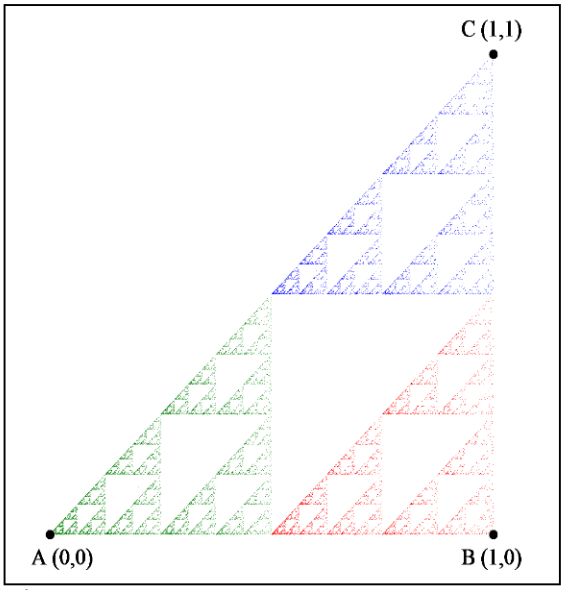
Shown in Figure 10.1 are 4 forms of extreme bias shown in an IFS. In (A), if the affine transform related to Vertex A was reduced to a probability of $\frac{1}{10}$ the resulting triangle would show a lack of detail at the green lower left region. This is because it must undergo subsequent transformations 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.

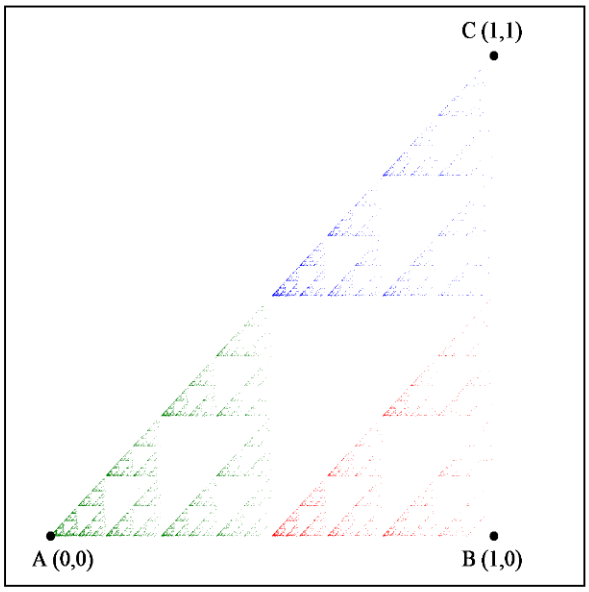
Obviously, this example was meant to show the extremes of PNG bias, however even subtle biases can propagate through the system and cause similar biases. This is why selecting a solid PNG is an important decision.



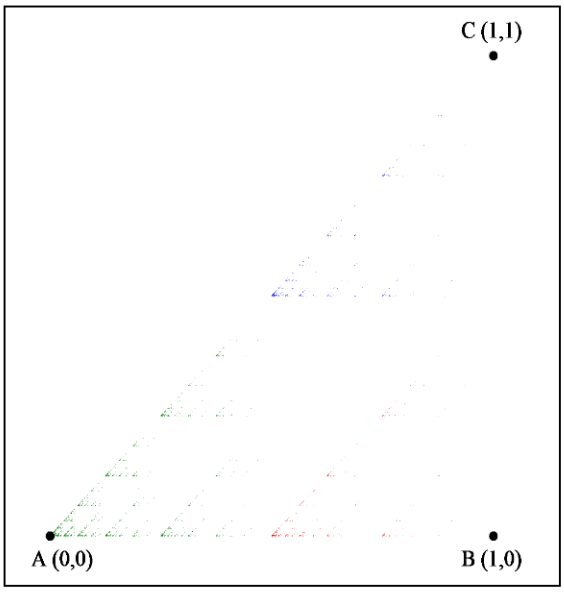
a) Vertex A Probability = 0.10 Vertex B Probability = 0.45
Vertex C Probability = 0.45



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

10.2 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.

10.3 `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 [25]:

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

Table 10.1: 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.

10.4 ISAAC

An alternative that sounds like a better choice is ISAAC, it stands for Indirection, Shift, Accumulate, Add, and Count [26]. 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[27]. 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.

10.5 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.

10.6 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.

10.7 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 10.2.

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.

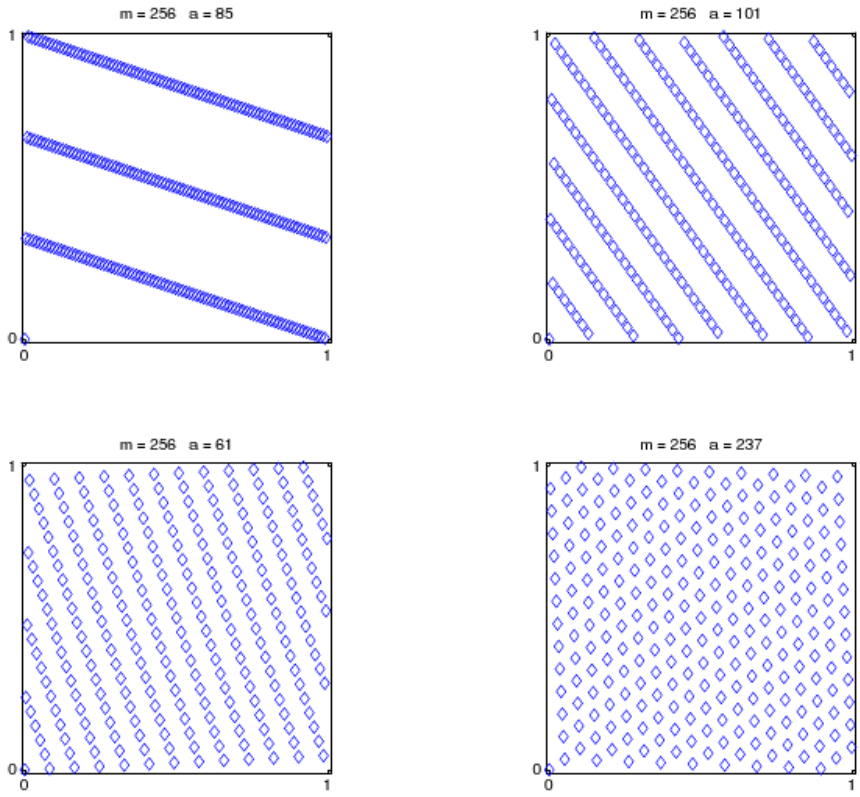


Figure 10.2: Example of lattice structure.

10.8 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 11

COLORING AND LOG SCALING

11.1 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.

11.2 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 [28]. 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 11.1[@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 11.1: 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 11.1, 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.

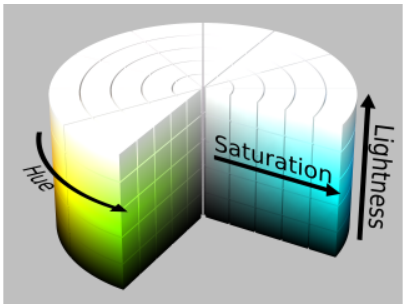


Figure 11.1: HSV Color Model

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 11.1.

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 11.2.

[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.

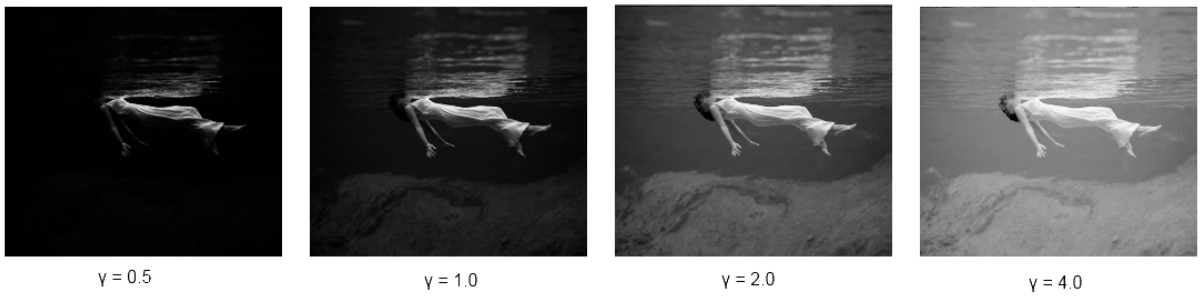


Figure 11.2: Comparison of gamma correction values.

The second image of Figure 11.2 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 11.3. 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.

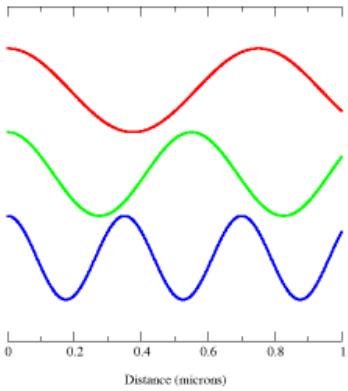


Figure 11.3: Wavelengths of Red, Blue, and Green

Some examples of weighted models to calculate brightness are below in Table 11.2.

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	$\sqrt{0.241 \times R^2 + 0.691 \times G^2 + 0.068 \times B^2}$

Table 11.2: 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 11.2 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[29].
2. CHROMA: The intensity of the color is a measure of the relative brightness of another color which appears white under similar viewing conditions[29].
3. SATURATION: The intensity of the color is a measure of its colorfulness relative to its own brightness rather than gray[29].

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 11.4.

Unsaturated Image

Non-linear Saturation making colors appear more vibrant.

Figure 11.4: 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.

11.3 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.

11.4 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[30]:

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[30]. 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.

11.5 `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 11.5.

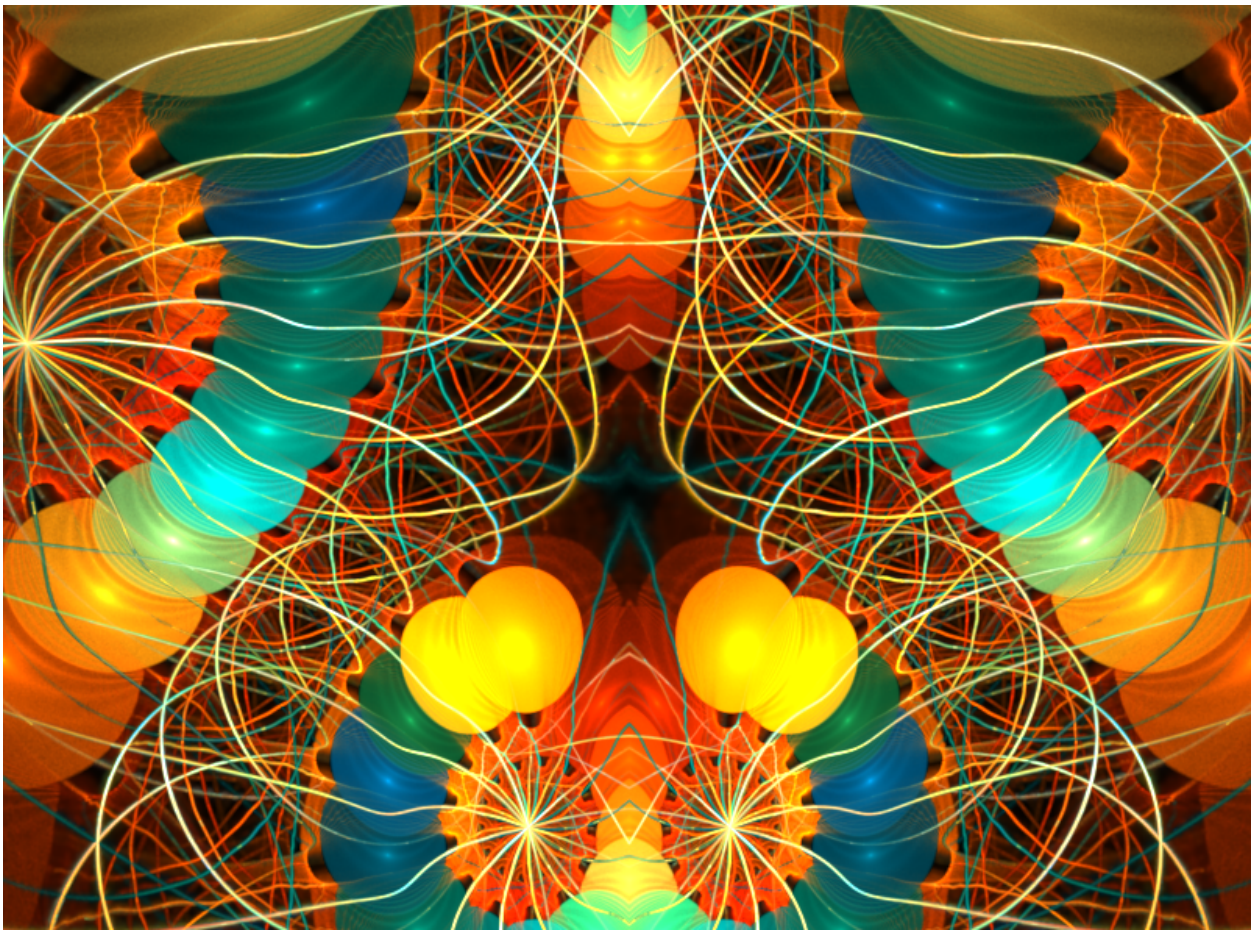


Figure 11.5: Baseline image of the flame whose parameters will be altered.

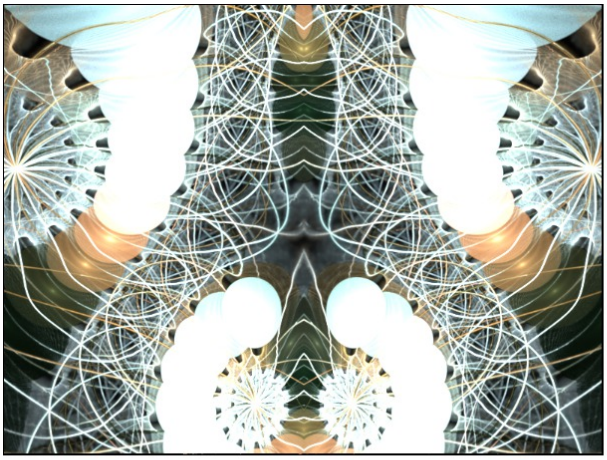
In the following sections adjustments are performed to one parameter of the flame while holding the others constant so that the parameter effect in question can be observed. The parameters that constructed the flame above are shown below in Figure 11.3.

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

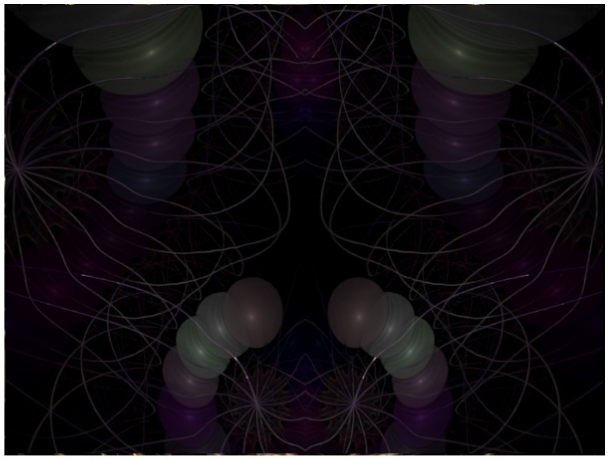
Table 11.3: Parameter values of our baseline image which modified versions of this flame will be compared to.

Color Palette Revisited and Explored

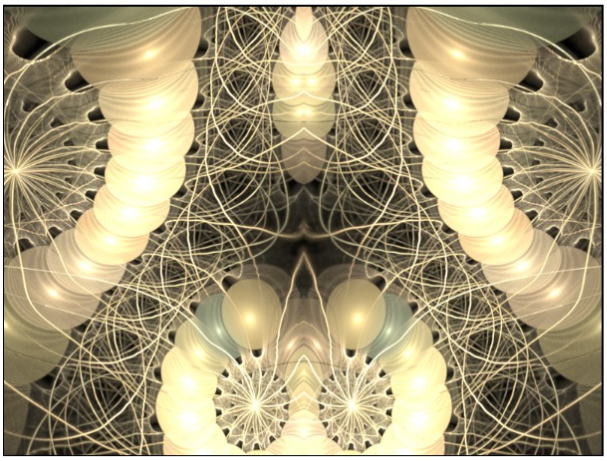
As mentioned in Section 11.5 : *Ad-Hoc Tone Mapping and The Color Palette* there are 701 standard palettes available. A minute amount of palettes are shown to give the reader an understanding of what a palette may look like. Figure 11.6 shows 4 different palettes applied to the baseline flame. By observing 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.



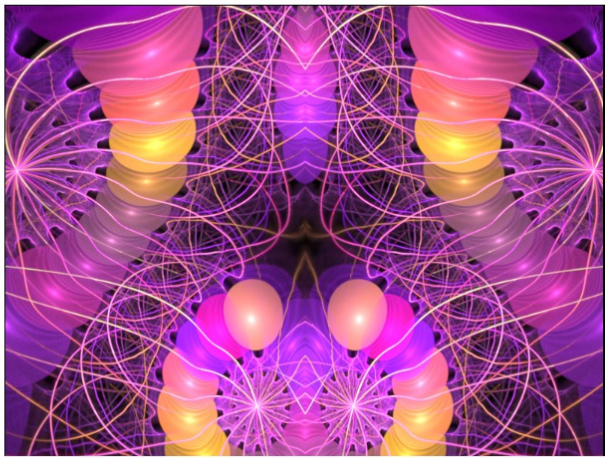
Flame colored with predefined palette number 1.



Flame colored with predefined palette number 5.



Flame colored with predefined palette number 82.



Flame colored with predefined palette number 300.

Figure 11.6: 4 different predefine color palettes applied to the baseline flame.

As mentioned in Section 11.2 *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 11.2 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 11.4.

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 11.4: Flame image numbers and their associated γ correction values.

The resulting images from the altered gamma corrections can be seen in Figure 11.7. 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.

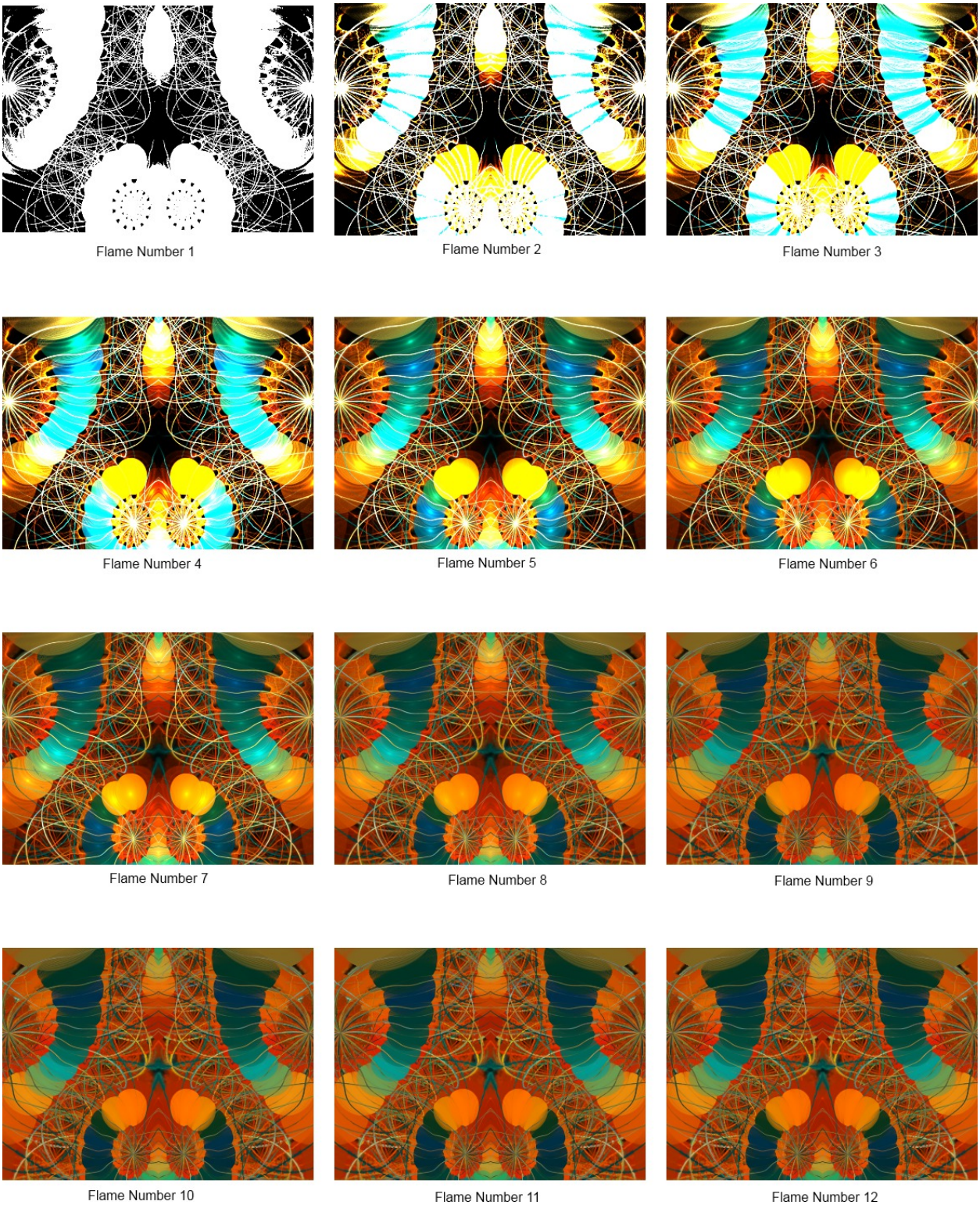


Figure 11.7: 12 different gamma values are presented one the baseline flame.

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 [31]. 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 11.5.

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 11.5: Flame image numbers and their associated gamma threshold values.

The resulting images from the altered gamma threshold values can be seen in Figure 11.8.

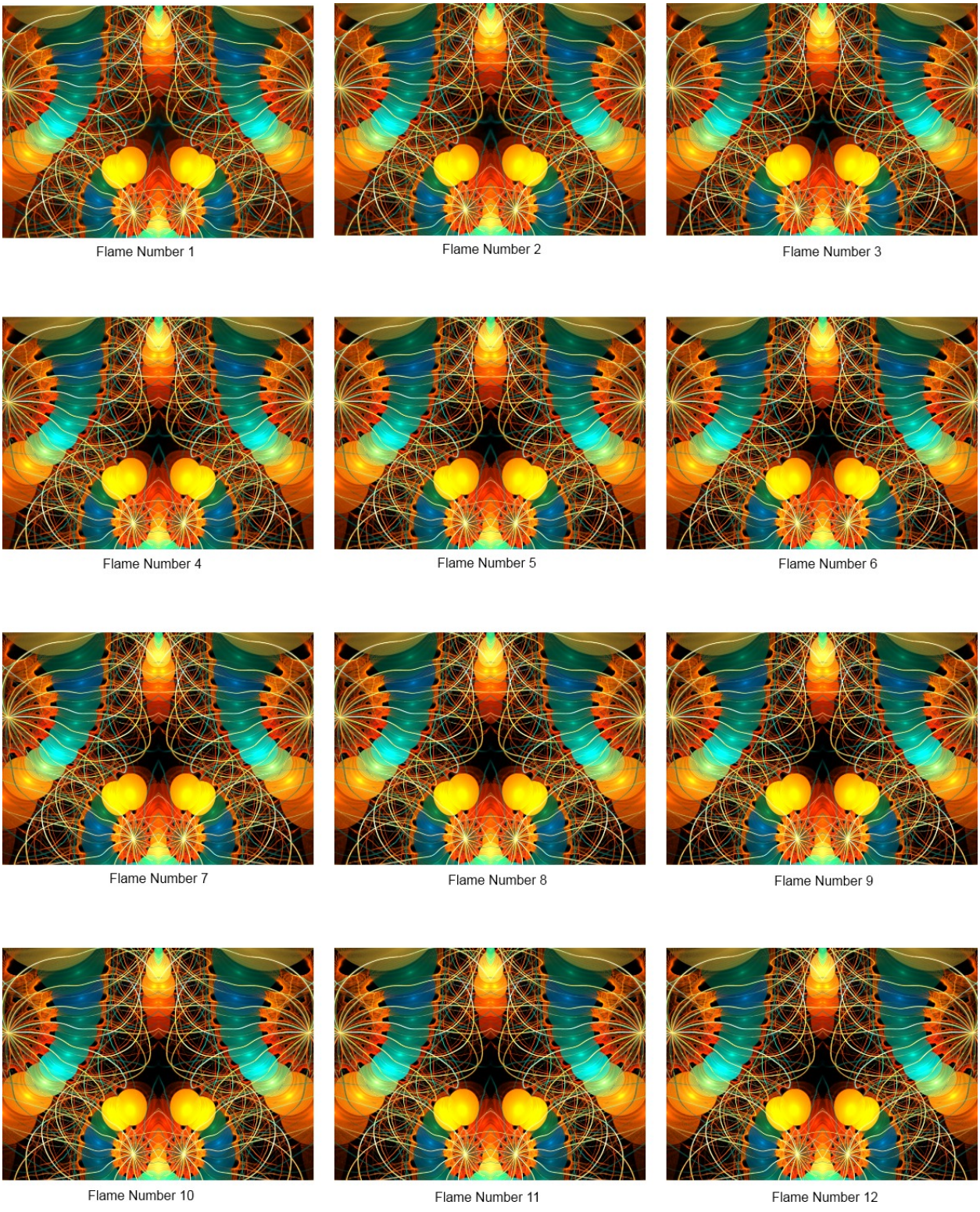


Figure 11.8: 12 different gamma threshold values are presented on the baseline flame.

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 11.6.

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 11.6: Flame image numbers and their associated hue rotation values.

The resulting images from the altered hue values can be seen in Figure 11.9. 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.

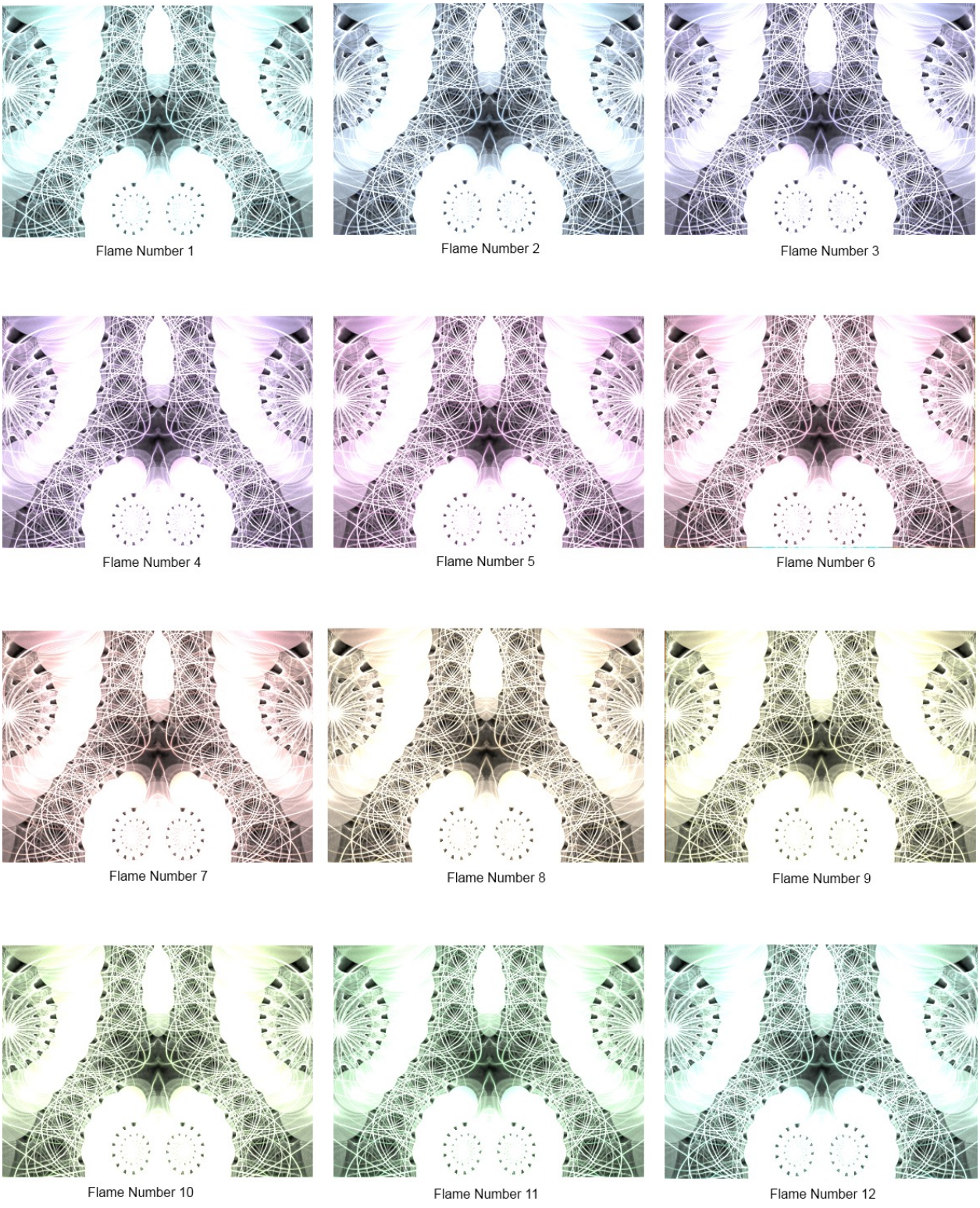


Figure 11.9: 12 different hue values are presented to the baseline flame with a non-vibrant color palette.

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 11.2: *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 11.7.

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 11.7: Flame image numbers and their associated brightness correction values.

The resulting images from the altered brightness corrections can be seen in Figure 11.10. 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).

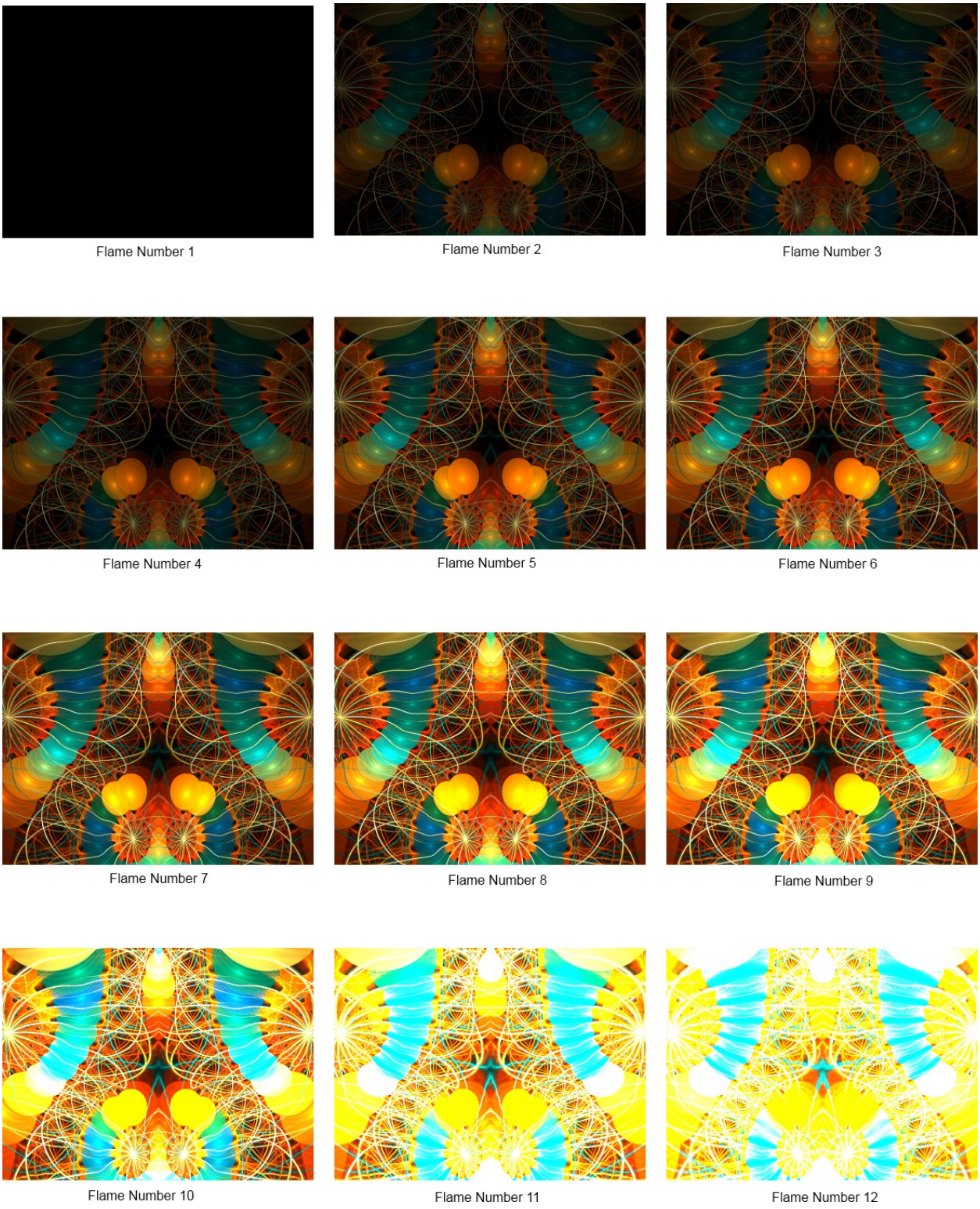


Figure 11.10: 12 brightness correction values are presented on the baseline flame.

Vibrancy

Vibrancy, as stated in Section 11.2: *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 11.8.

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 11.8: Flame image numbers and their associated vibrancy values.

The resulting images from the altered vibrancy values can be seen in Figure 11.11.

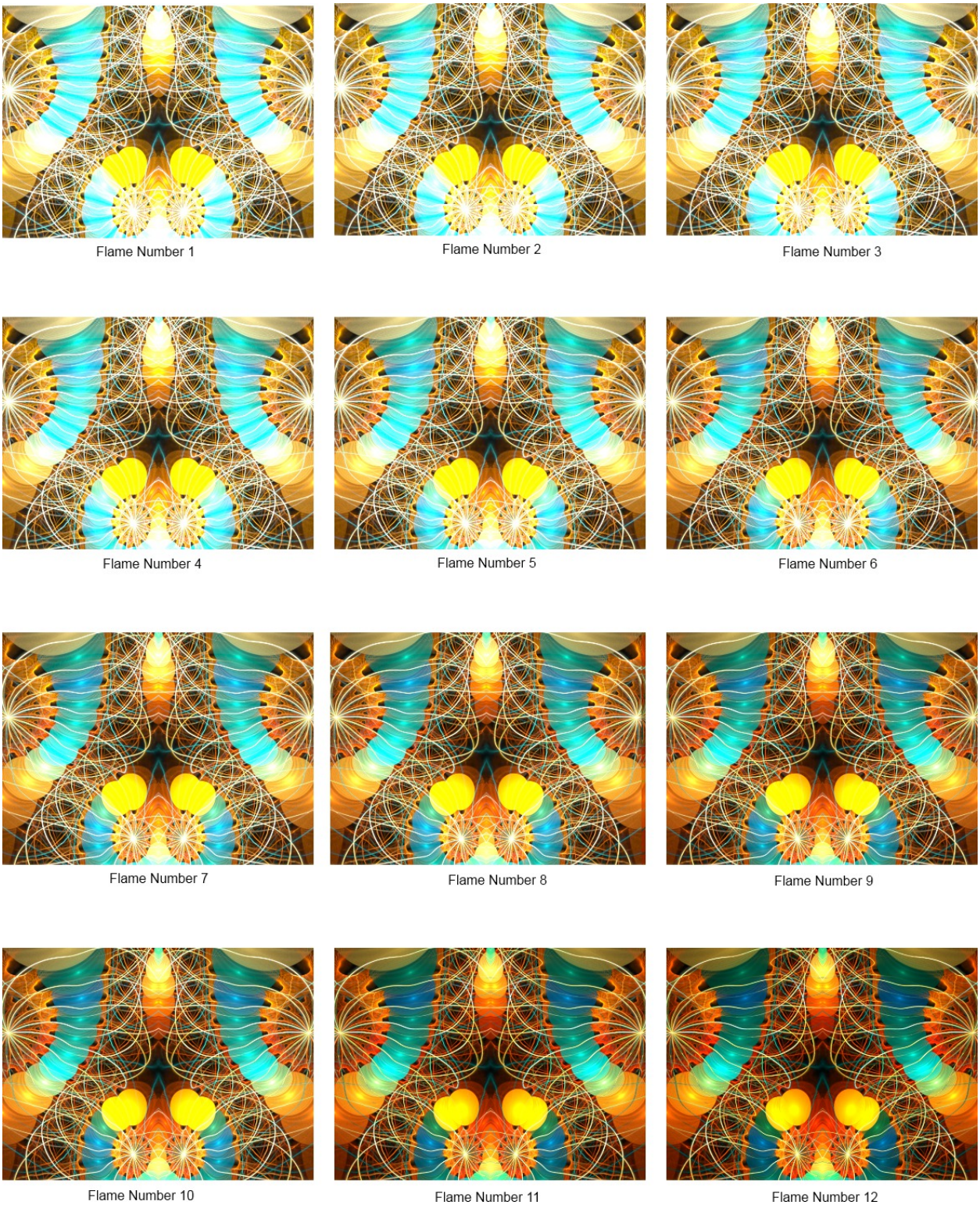


Figure 11.11: 12 different vibrancy values are presented on the baseline flame.

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 11.2: *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[31]. 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[31]. The 12 different highlight power values that were applied to the baseline image are show in Table 11.9.

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 11.9: Flame image numbers and their associated highlight power values.

The resulting images from the altered highlight power values can be seen in Figure 11.12.

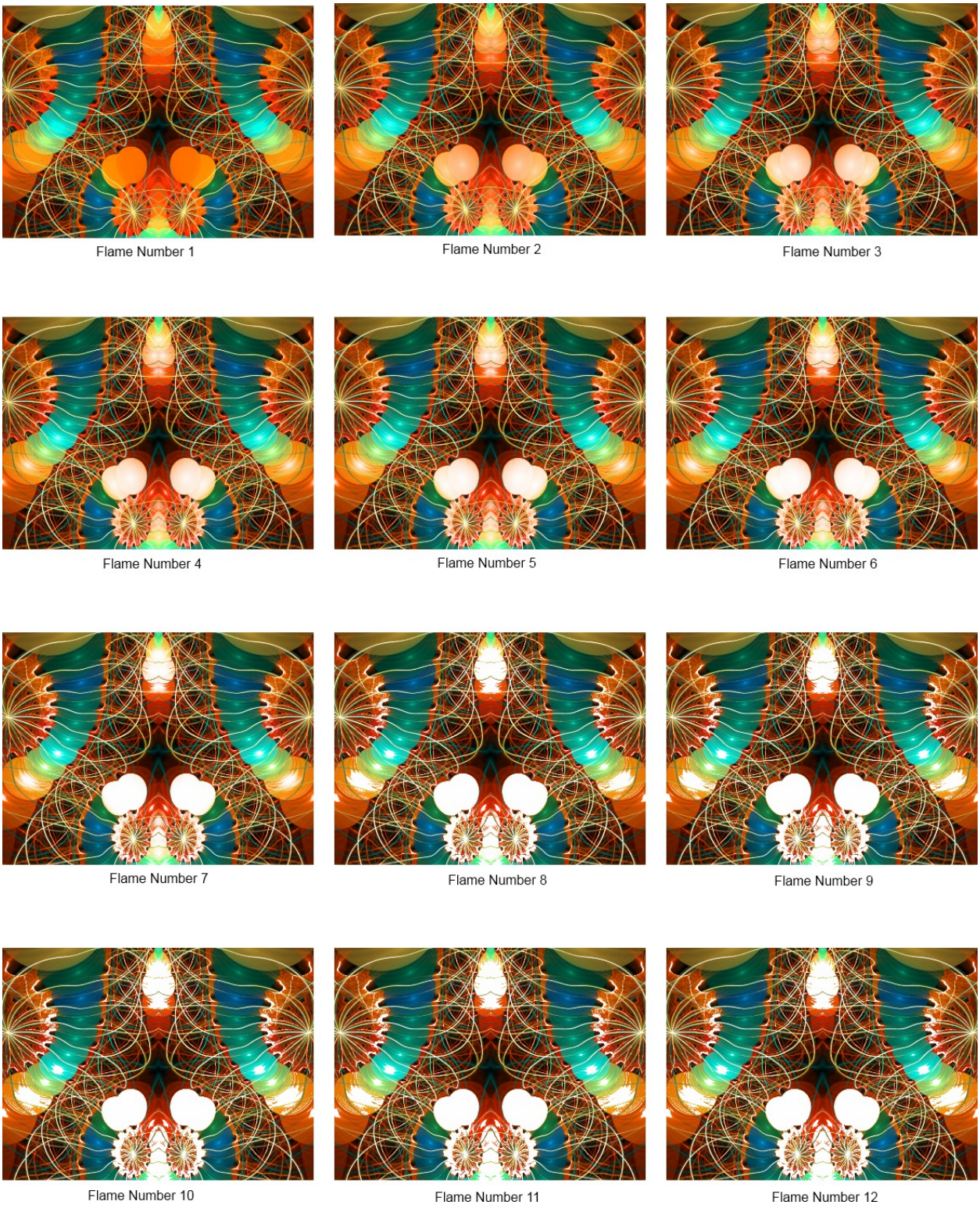


Figure 11.12: 12 different highlight power values are presented on the baseline flame.

11.6 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 12

SAMPLE ACCUMULATION

After each iteration, the color value of a point within the bounds of the accumulation buffer — a 2D region of memory slightly larger than the final image, with gutters on all sides for proper filter behavior at edges — must be added to its corresponding histogram bin. This process is simple, both conceptually and in its basic implementations, yet it is, by nearly an order of magnitude, the most time-consuming part of the fractal flame rendering process on GPUs.

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 Chapter 8.

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. Since the accumulation buffer necessarily uses full-precision floating-point values, it is not small; for a 1080p image, the framebuffer is over 31MB[^fbsize]. Random access to a buffer of this size renders even CPU caches useless, although CPUs are bottlenecked by iteration speed, so this is not a concern. On GPU, however, the effects are amplified by simpler and smaller caches, such that nearly every transaction may be expected to incur a cache miss.

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, as compared to the accumulator cell size of 16 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% [32].

Improving performance requires addressing this step of the flame algorithm carefully and thoroughly. In this chapter, we'll compare the three implementations of the writeback stage currently avaiable in cuburn.

12.2 Atomic writeback: perfectly slow

When running in multi-threaded mode, `flam3` uses atomic intrinsic functions where available to perform histogram accumulations. These compiler-provided intrinsics perform three operations — load a value from memory, add a second value from a live register, store the updated value — in such a way as to appear to all running threads like the operation happened instantaneously. On x86 CPUs, this is typically accomplished using a compare-and-swap spinloop, which is not exactly fast but far from the most egregious ineffeciency in `flam3`. This technique allows for very simple multithreaded writeback code.

To establish a point of reference, we replicated this behavior in cuburn using CUDA's atomic intrinsics. Global memory atomics on Fermi are implemented as a single instruction, which is forwarded to the appropriate memory controller along the packetized internal bus as an independent transaction[1]. At the memory controller, the cache line containing the address is loaded and then locked, so that it cannot be evicted or operated on by another context. One of a small set of local ALUs reads the relevant word, performs an operation on it, and writes it back. The line is then unlocked and allowed to be flushed to DRAM when evicted.

This architecture makes individual atomic writes for which the issuing thread does not require a return value execute faster than a read-modify-write cycle, from the perspective of that thread: the SM dispatches a single memory operation which gets appended to the memory controller queue, and is then immediately free to move on with other tasks. However, the number of atomic ALUs is limited, as is the queue depth for atomic operations, and the cache-line-based locking mechanism stalls consecutive atomic transactions to the same cache line. Under load, these quickly saturate the memory transaction queue. This is handled by signaling each shader core to hold requests until the queue has room, ensuring data integrity at the cost of performance.

The performance hit is substantial, as expected. Atomic writeback is at least ten times slower than direct writeback on our hardware. It remains available in cuburn as a reference point and a debugging aid, since this technique leads to deterministic output and has no precision loss, but is impractical for production renders.

12.3 Direct writeback

The direct writeback strategy performs a read-modify-write cycle against global memory from the shader core, rather than at the atomic units. For the reasons discussed at the start of this chapter, the random access pattern used by this pattern performs poorly. Another important problem that needs to be considered when using direct writeback arises from the incoherent nature of the caches across the device.

A cache line loaded into an L1 cache to take part in a read-modify-write cycle on a shader ALU is vulnerable to inconsistency from two sources. In a local conflict, one thread can perform a read of a memory location after another thread has read that location but before this other thread issues an instruction to overwrite the location with an updated value. Both threads will base their updates on the original value, causing one update to be silently lost. This can happen either as a result of two warps interleaving instruction issues in the normal process of latency hiding, or as a result of two threads within the same warp reading the same location within the same instruction. The latter case is particularly venomous, and results in siginficant image quality degradations when point swapping between threads (as described in Chapter 8) is omitted.

A cross-core conflict occurs because L1 caches are not synchronized across the device. If two separate shader cores request a cache line, that line will be copied to both cores' L1 caches. A subsequent write from one core will not invalidate the cache line in the other core. Depending on access patterns, this allows invalid data to

[1] Anecdotal evidence suggests no coalescing is performed, but we have not confirmed this with direct testing

remain cached for substantial durations. It is possible to effectively skip L1 via cache control operations, but in general the access patterns cuburn exhibits makes this particular form of inconsistency far less frequent, and so we do not take steps to avoid it.

These inconsistencies occur in a framebuffer, rather than in data that is used for execution control, and therefore are closer in effect to a kind of sampling noise than an outright bug. Fortunately, these collisions will tend to happen in the image regions which have the highest density, where the relative error for each lost sample will be considerably smaller and further reduced by log filtering.

Characterizing the error at the pixel level is difficult, as out-of-order execution makes exact reproduceability unlikely and inserting code to atomically write to a separate buffer within one pass will increase time between read-modify-write cycles and therefore underestimate the prevalence of the problem. Indirect estimates suggest that the total error is small, and most flames rendered using this mode were found to be indistinguishable from those rendered using atomic writeback in a still-image, simultaneous presentation subjective A/B test. However, this result is contingent on device timings and flame characteristics. We have constructed pathological flames which exhibit this flaw much more severely. As a result, we were interested in a strategy which would not suffer these inconsistency problems — or at least suffer them in a quasi-deterministic way that could be compensated for — and meet or perhaps exceed the limited performance of direct writeback.

12.4 Deferred writeback

A common strategy for avoiding the performance limits imposed by random memory access patterns in graphics applications is tiling, where image elements are rendered alongside others which are near it in screen-space. 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 the scene-space projection 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 [33]. A similar technique is employed by graphics chips designed for low-power applications, such as those in the PowerVR architecture [34].

Implementing tiling in cuburn's output stage would allow for the use of shared memory to perform accumulation, as tile sizes could be chosen to fit within available shared memory bounds. Shared memory supports atomic transactions with much greater aggregate throughput than global memory, so this could allow for reasonable performance without transaction loss. However, tiling requires that we be able to subdivide data by region of interest, while the chaos game requires that points cross the entire image area. It is not possible to perform accurate chaos game sampling within a particular region of interest, save by simply discarding all points outside of that region, which would be tremendously ineffecient.

Since it is not possible to obtain chaos game samples by region of interest directly, it is necessary to separate sample generation from sample handling. The deferred writeback method does precisely this.

Storing chaos game samples

Storing a point to the accumulator using atomic or direct writeback involves adding three color values to a particular accumulator index, and incrementing the corresponding density count. The color tristimulus value is obtained by performing a palette lookup, using color coordinate and control point time delta to perform a texture fetch with bilinear filtering enabled. This value is then scaled by an opacity value, depending on xform, if the flame makes use of the xform opacity feature. The accumulation index is determined by applying the camera transform to the 2D (x, y) coordinate pair, and multiplying y by the image row stride value.

```
|                            | Sort key (9 bits)        |  | Shared key (12 bits)              |
| Color (8 bits)        |     | Accumulator address (22 bits)                                        |
 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0

|                                        | Sort key (7 bits)  |                                  |
|                                                          | Shared key (12 bits)              |
| Color (8 bits)        | XOI |          | Accumulator address (18 bits)                        |
 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
```

Figure 12.1: Top: the log format used for a 2560x1600 image which does not use xform opacity. Note that each memory cell in the sorted log is scanned twice in this configuration, since bit 12 is not sorted; each pass ignores log entries outside of its boundary. (Sorting will be described in Chapter 13.) Bottom: The log format used for a 640x360 image. To avoid entropy reduction, bit 11 is included in the sort.

For deferred writeback, we wish to store this information as efficiently as possible in such a way that it can be reconstructed for accumulation at a later time. A compact scheme is presented in Figure 12.1. 8 bits are used to represent the color index, 2 bits represent the xform index when opacity is used, and up to 22 bits (or 24, for flames which do not use xform opacity) encode the accumulation buffer index after camera rotation and clipping.

Faithful representation of the color index in 9 bits, instead of the 32 used to represent that value as a floating-point coordinate, is possible by using 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 [35]. 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, which we use.

The accumulator index is also in need of dithering under this scheme to suppress the quantization error resulting from truncation of floating-point values to linear memory addresses. Due to the 2D nature of this process, it is often easier to use a signal-processing framework to analyze and correct these errors, rather than a statistical one; we do so in Chapter 14.

A 22-bit upper bound on the image address size places an upper bound on accumulator buffer size just larger than a 1080p region would occupy. Since the fractal flame algorithm is statistically fractal, allowing for detail even at very fine image scales, most flames can be scaled well past 1080p for applications such as print

material or art installations. However, consumer hardware cannot decode video at these resolutions; in fact, H.264/AVC High profile, as used in Blu-Ray, is insufficient to capture the image detail available in even 1080p streams without visible blurring of fine structure. For rendering animations, this limit is an acceptable compromise.

Even with dithering, two bits does not afford ample precision for storing xform opacity. Rather than directly encode the floating-point opacity from the interpolated control points, the value represents an index into a time-varying array of those opacities, precomputed in a separate step. This array can represent four opacity values exactly, which is often sufficient for most use cases — although flames can have more than four xforms, typical uses of xform opacity often involve the same value on multiple xforms. When more unique xform opacities are in use, combining this lookup table with dithering enables a DXTC-like encoding scheme which is capable of more accurately representing opacity values, even at low sample counts, than a linear mapping.

Both color and xform opacity depend on knowing the time value at which the source control point was interpolated to determine which look-up table to use. Fortunately, no bits are needed to encode this value: the index position in the output array is determined by the block index of the thread block which generated the point, which also determines the control point used to create the points. The time value can therefore be inferred from the index of the point in the output buffer.

This information is packed and stored consecutively from the chaos game iteration thread in efficient, coalesced transactions, which proceed asynchronously and do not require confirmation of completion. As a result, memory bottlenecks are absent from the iteration thread.

Accumulating stored samples

The accumulation buffer is divided into tiles, with each tile's size based on the interaction between accumulation buffer addressing patterns and shared memory size.[2] After a certain number of iterations have been performed, the samples in the iteration log are sorted into tile groups based on a key extracted from a portion of the global address. Whenever possible, each tile group contains exactly one tile, such that the tile group address is equal to the tile address prefix, but this is not always the case, for reasons explained in Chapter 13.

The accumulation kernel is launched with a thread block to service each tile. A block looks up the start and end indices of the tile group in the sorted array, clears its shared memory buffer, and begins loading log entries from the tile group. If a log entry's tile address prefix does not match the current block's tile address, it is discarded, and the next loaded. When a log entry is in the current tile, the color, local address, and xform opacity index are extracted from the entry, and the control point time value is inferred by the entry's index within the tile group. Extracting the time from sorted data is not necessarily as accurate as extracting it from the full array index, but the error is subthreshold in every flame we have tested.

In traditional in-loop accumulation, the color value is determined by performing a texture lookup into a palette texture with bilinear interpolation. This enables compact, cache-efficient, and very accurate blending based on both color coordinate and time value to retrieve an appropriate color four-vector (of which one channel is currently unused) that can then be scaled if needed to account for xform opacity. The use of clamped-integer values with dithering to store the necessary coordinates offers an alternative possibility with equivalent results: provide a pre-blended texture palette which does not use texture unit interpolation. When using a four-vector of floats to store the texture, this strategy would not result in a large performance improvement; its usefulness is derived from the accumulation buffer format.

[2] For simplicity, we use linear address segments as tiles, rather than rectangular areas, but the principle, and thus the term, remains the same.

The process of shared-memory accumulation can be relatively expensive. Shared-memory atomics are implemented on Fermi via load-lock and store-unlock instructions, with the former setting a predicate indicating successful lock acquisition; conflicts are handled by spinning until all threads in a warp have successfully completed their transaction. Certain flames will encounter a high number of collisions in certain tiles, and minimizing the number of collisions improves performance. Storing accumulation buffers in planar format within shared memory helps — with a stripe size of 4 bytes, interleaved float32 storage would quadruple the effect of any collisions — but reducing the size of the buffers helps further.

The accumulation buffers therefore represent accumulation values as a packed 64-bit integer, storing density and the color tristimulus values in a clamped format. The storage format assigns uneven bit ranges to the different values; one configuration currently under test uses 11 bits to represent the density, 19 bits to represent the first color tristimulus value, and 17 bits each for the remaining two values. At each accumulation, density is incrmented by one; when the density reaches, in this case, 2047, the next accumulation will cause this accumulation cell to wrap. Without recording the fact that the cell wrapped, this is harmful, as it leads to loss of the accumulations and color artifacts as the other counters also roll over. As a result, after each accumulation, each cell examines the density counter it just updated. If it is exactly equal to the largest representable density value less the number of threads in this thread group — 1023, in this example — that thread triggers a barrier, re-reads the value, and writes the cell to the global accumulation buffer, zeroing it afterward. By checking for exact equality, we implicitly ensure that only one thread handles the writeback, and by doing it well before the value wraps, we can perform writeback and barrier syncing lazily, increasing efficiency.

To ensure that the color cells don't wrap, the values added to those cells are scaled such that the integer corresponding to the maximum tristimulus value, 1.0, times the maximum number of iterations before the density counter wraps, is representable. In practice, that means each primary tristimulus value to be added to the accumulator is represented in 8 bits, and the secondary ones with 6 bits. Dithering is once again required for accurate color representation in low-intensity areas.

The uneven bit distribution between primary and secondary colors implies a difference in importance, and indeed there is such a difference — as long as we use YUV. Human vision is substantially more sensitive to luminance information than to chrominance information, both in terms of spatial frequency and intensity; video encoding schemes take advantage of this to encode much less chrominance information than luminance information with little noticeable impact on the final image quality. To do so, most coding schemes convert RGB values via an invertible linear transform into a YUV color space,[3] where the Y channel contains luminance information and UV represent chrominance via an opponent-color encoding. Both U and V are then spatially subsampled, often by a factor of two in each dimension, and aggressively quantized. We avoid the complexities of spatial subsampling in cuburn, since it is transaction count and not buffer size that limits performance, but we can increase performance by reducing the precision of the UV channels during accumulation, which allows us to perform more iterations before forcing a flush to global memory. Naturally, this requires conversion of RGB samples YUV.

The long chain of conversions, from log entry to color and time values to texture coordinates to bilerp texel four-vector to YUV-encoded four-vector to dithered four-vector to scaled integers to single packed integer, is quite costly, compounded by reliance by nearly all of these operations on the lower-throughput special function units. It is this chain of conversions, rather than the bilinear interpolation alone, which the preformatted texture structure is designed to accelerate, as it simplifies this chain to a simple lookup.

The actual process of adding the value from the look-up to the shared accumulation buffer is performed using a call to `atomicAdd`, providing the shared memory lock-and-load logic described previously.

[3] There are many YUV transformation matrices, and technically none of these represent color spaces themselves, but are simply encoding schemes for RGB values which may have an associated color space. Proper conversion to and from YUV requires careful use and signaling of specified color matrices. However, since cuburn is not a physically-based renderer and relies on user-specified colorimetry information, its only responsibility is to be consistent.

However, this requires interleaving of the high and low bits of the 64-bit word, which doubles the risk of bank collisions (although it does not strictly double their cost, as it only requires one lock operation). We are currently prototyping a tool that will allow us to perform post-compilation manipulation of opcodes within the compiled binaray, or "monkey-patching", to efficiently planarize this lookup and perform in-loop buffer zeroing to eliminate the need for a separate barrier operation when performing early flushing. Monkey-patching is necessary because the PTX ISA does not expose the underlying lock and unlock operations used to implement shared memory atomics.

CHAPTER 13

CUBURN SORT

Sorting is a common, intuitive operation, found as a core part of many optimized algorithm implementations. It's also surprisingly difficult to do quickly on GPU architectures. Because of the theoretical and practical importance of sorting implementations, the peak sorting performance is often used as a critical benchmark of an architecture's true performance and flexibility: simple enough to be portable and reproduceable, while cross-functional enough to be a holistic test. Sorting implementations for GPGPU systems remain a vigorously active area of research for both professional and academic individuals in the HPC community.

In cuburn, sorting is used in the deferred writeback mechanism to split the iteration sample log into tile groups for efficient processing by an accumulation kernel (Chapter 12). To be useful in cuburn, a sort function must be easily callable within the complex asynchronous dispatch code used to schedule operations on the GPU. It also must be fast; if the entire deferred writeback sequence, including the sort, cannot be made to be as fast as direct accumulation, development efforts would be better spent optimizing that process and developing statistical workarounds for the possibility of sample loss. The sort must also support operating on a partial bit range. While every notable implementation does so internally as a consequence of using multi-stage radix sorting to fully sort 32-bit keys, not every implementation exposes this at the API level.

Of the most commonly used sorts available for Fermi GPUs, only MGPU sort meets our performance requirements. It also exposes options for sorting partial bit ranges efficiently. Its C++ API is convenient to use from that language, but less so from Python; still, a C wrapper that could be called from Python code would not be overly difficult to create.

Instead of using MGPU sort, however, we elected to build our own. This was not an instance of Not Invented Here syndrome, but simply a consequence of having begun our implementation before MGPU sort was made public. Promising theoretical results (and a fair amount of stubbornness) encouraged us to produce a working implementation, which we have now done.

We stress here that cuburn sort is *not* a suitable replacement for MGPU sort, or other general-purpose sorting implementations. It is a key-only sort, and while it is usually a stable sort, it is absolutely not guaranteed to be stable, which means that multi-pass sorts risk increasingly large misorderings in lower bits with each pass. As a result, it's not useful for much outside of a preprocessing pass for global memory reductions.

At that one thing, however, it's *great*. Cuburn sort is optimized to sort between 7 and 9 bits, and does so faster than any other GPU sort implementation with public performance figures (see Figure 13.1). Additionally, an optional early-discard flag allows an additional workload reduction on top of the figures seen here; for typical flames, this amounts to a further 40% increase in performance.

We believe that the sort can be optimized further, and have concrete optimizations in mind which we will begin implementing immediately after finishing this documentation.

	Cuburn	MGPU	B40C	CUDPP
7 bits	821	740	551	221
8 bits	943	813	611	251
9 bits	966	877	475	191
10 bits	862	910	528	211

Figure 13.1: Performance of different sort implementations, as measured on a GTX 560 Ti 900MHz. Values are in millions of keys per second, normalized to 32-bit key length.

The sort is accomplished in four major steps. The first pass divides the buffer to be sorted into blocks of 8,192 values, and performs an independent scan operation on each. Unlike the other sorting techniques benchmarked here, cuburn's scan uses shared memory atomics to perform this accumulation, by performing an atomic increment on a shared memory index derived from the point under analysis. This process is much slower than traditional prefix scans, but because it is coordinated across all threads in a thread block, it allows the derivation of a radix-specific offset for each entry in the current key block. This offset is stored into an auxiliary buffer for every key processed, and the final radix counts are stored to a separate buffer.

The second step loads the final radix counts and converts them to local exclusive prefix sums, storing these in a separate buffer. This is performed quickly and with perfect work efficiency by loading the radix counts into shared memory using coalesced access patterns (see Figure ??), rotating along the diagonal of the shared memory buffer, and performing independent prefix sums in parallel horizontally across the buffer, updating values in place.

Figure 13.2: Shared-memory patterns used in cuburn sort's work-efficient radix scan reduction pass.

A third step operates on the final radix counts, transforming them in-place to per-key-block, per-radix offsets into the output buffer. This is accomplished by first reducing the buffers via addition in parallel to a very small set in a *downsweep* operation, performing the prefix scan on this extremely limited set in a manner that is not work-efficient but, due to the small number of buffers involved, completes in microseconds, and then broadcasting the alterations back out to the full set of buffers in an *upsweep* operation.

Sorting is accomplished by using the offsets and local prefixes to load each key in the block to a shared memory buffer, then using the local and global prefixes to write each key to the output buffer. Transaction lists are not employed, but large block sizes help to minimize the impact of transaction splitting.

One unusual characteristic of this architecture is that performance on data that does not display a good deal of block-local entropy in radix distribution is actually considerably slower than sorting truly random data. Sorting already-sorted data is a worst case, with a penalty of more than an order of magnitude arising from shared-memory atomic collisions during the initial scan. We avoid this in cuburn by ensuring that the radix chosen to sort never includes bits that are zero throughout the log. In some cases, this means that some bits of the accumulation kernel's shadow window are also sorted, since cuburn sort does not scale down to arbitrarily small radix sizes. This oversorting is theoretically less efficient, but is actually faster than including the zero bit in the sort.

A simple but powerful optimization enables discarding of any key equal to the flag value `0xffffffff`. This value is used internally by cuburn to indicate that a particular point log entry corresponded to an out-of-bounds point and should be ignored. Discarding these during the sort stage improves performance by around 40% on average for our test corpus of flames.

CHAPTER 14

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. 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 GPUs. These problems and their many solutions are discussed in more detail below.

14.1 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* [36]. See Figure 14.1 for an example.

Figure 14.1: 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 [37].

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 real-time graphics will have a quarter of the original frame-rate [38]). 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 [38]. See Figure 14.2 for visual depictions of these algorithms.

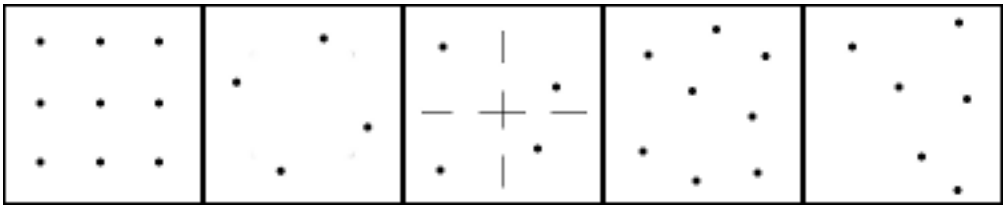


Figure 14.2: From left to right: Ordered Grid algorithm, Rotated Grid algorithm, Jitter algorithm, Poisson algorithm, Random algorithm.

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 [39].

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 [40].

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 [40].

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 [41]:

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 [41].

14.2 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 [42].

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 [43]. 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. [43]

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 [44].

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 [44].

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. [44]

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 [44].

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. [45]

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 [46]. 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 [46].

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 [46], 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 $O(d^2(n + l))$ [46].

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 $O(dn \log n)$ [47].

A kd-tree is a binary tree data structure used to store a finite number of points from a k-dimensional space [48]. Each leaf stores one point and each inner node represents a d-dimensional rectangular cell [47]. 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} [47]. 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 $O(dn \log n)$ [47].

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.

14.3 Filtering in cuburn

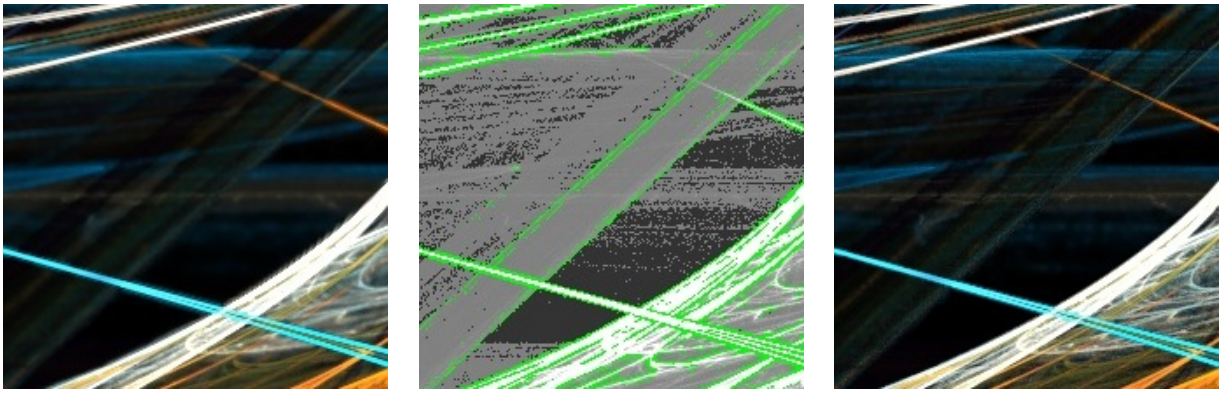
The chaos game is fundamentally related to Monte Carlo integration, and collects thousands of samples per pixel on average. This is a natural fit for multisample antialiasing.

During precalculation of the camera transform, which maps IFS coordinates to accumulation buffer indices, the sampling grid for each temporal sample is offset according to a 2D Gaussian distribution with a user-controlled standard deviation that defaults to one-third of a pixel for compatibility with flam3. With thousands of samples per image pixel taken on average, this technique gives very high quality antialiasing with essentially zero overhead.

After iteration and accumulation have finished, a density estimation kernel processes the output. The DE kernel again avails itself of copious amounts of shared memory to accelerate processing of image elements. Because variable-width Gaussian filters are nonseparable, the DE filter is applied in square 2D blocks of size $2R + 32$ pixels in each dimension, where $R = 10$ is a constant governing the maximum permissible filter radius and consequently the gutter size, and 32 is is both the horizontal and vertical size of the thread block. Each thread in the block loads a point from the accumulation buffer (with thread $[0, 0]$ loading the value corresponding to a local offset of $[R, R]$, and thread $[31, 31]$ loading $[R + 31, R + 31]$), calculates the filter radius, and proceeds to write the offsets in a spiral pattern designed to avoid bank conflicts and enable early termination without warp divergence. After the 1,024 pixels in the current block have been processed, the entire shared memory region is added to the output buffer, including gutters, and the column advances vertically by 32 pixels. This pattern treats every input pixel exactly once, but due to gutter overlap, output pixels may be handled by as many as four separate thread blocks. Nevertheless, this kernel operates efficiently, taking less than one percent of a typical frame's GPU time.

In cuburn, the grid jittering is applied entirely before DE, which can in some instances create image artifacts. Edges that have been smoothed by JGAA whose tangents lie close to but not exactly on the image grid will have a strong density gradient adjacent to the edge, including one point at every pixel step which will receive a small fraction of the edge's full density. When this occurs on a strong edge adjacent to a very low density area, DE will blur this lowest-intensity point very strongly, leading to "edge bloom" (see Figure 14.3).

Cuburn adds an edge-detection filter to the DE process. When a weak pixel adjacent to a strong edge in a low-density area is detected, the filter's radius is clamped to prevent edge bloom. Different edge detection methods were tested, including the common Sobel and Roberts cross image kernels, but these methods resulted in incorrect detection, with higher sensitivities resulting in excessive false positives and lower ones allowing some bloom to escape detection. In keeping with the authors' penchant for reinventing the wheel, cuburn uses the L^2 norm of a pair of three-pixel-wide convolution kernels given in (14.1), which are essentially stretched versions of the Roberts cross. Desipte their simplicity and ad-hoc origins, these kernels have been found to perform admirably at edge detection while rejecting noise.



(a) (b) (c)

Figure 14.3: (a) An example of edge bloom caused by improper handling of antialiased edges during density estimation. (b) A view of the same image highlighting detected strong edges. (c) The results of applying edge detection and filter clamping to density estimation.

$$A = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad B = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix} \tag{14.1}$$

Currently, some parameters involved in cuburn's approach to filtering are derived empirically via trial and error. While these constants produce satisfying images for most flames, they fail to account for certain exceptional conditions, such as a combination of extremely high brightness, positive highlight power, and localized singularities along a density curve. In these cases, small artifacts such as noticeable aliasing may remain in the output. Deriving these parameters analytically in response to image parameters is challenging, due to the presence of the variable-width density estimation step, but we look forward to examining possibilities for such an analytical framework.

CHAPTER 15

BENCHMARKING

A major objective of this project is to benchmark the performance of cuburn while it is executing on the GPU. A secondary objective is for cuburn to accurately render the entire catalogue of variations available in flam3, our reference implementation. In order to verify that both the completeness and performance objectives have been reached it is necessary to benchmark and each flame variation. The benchmark's framework, setup, results, and analysis will be described in the following sections.

A very important note is on the matter of comparing the flam3, the reference implementation, and cuburn is that it is a complicated matter which could be done many ways that do not lend any meaningful insight. Because performance can vary so greatly depending on the flam3 used we instead present an alternative, meaningful discussion on benchmarking the interesting performance statistics of cuburn.

15.1 Framework

NVIDIA has released a profiler with CUDA which aids developers in gathering information about a wide array of information such as kernel execution and memory transfer times. The profile can be enabled and configured by the use of a handful of environment variables that need to be set on the user's operating system.

By exporting the environment variable `CUDA_PROFILE` and setting it equal to `1`, the profiler exports the program's GPU statistics into a log file. By default, the profile will write the data to `./cuda_profile.log`. If a different location is desired, the user can set the `CUDA_PROFILE_LOG` environment variable accordingly. Additionally by exporting the environment variable `CUDA_PROFILE_CSV` and setting it equal to `1`, the log file's format can be changed to a comma-separated value (CSV) format. This format eases the processing of parsing this file for analysis because of it's more rigidly defined format specification.

Performance counters can be used in addition to the default timing information (timestamp, method, gputime, cputime, and occupancy) by creating a profiler configuration file specifying additional performance counters using the `CUDA_PROFILE_CONFIG` environment variable. The performance counters use on-chip hardware counters to gather statistics and hence their use change the algorithm's performance and should be used with caution. These performance counters were explored but in the end did not provide useful statistics or yield any insight that could make for an interesting discussion.

15.2 Benchmark Machine

In order to accurately talk about the benchmarking results as well as provide a frame of reference for the user, the benchmarking machine is described. The machine used for the benchmarking was an

`Alienware M17xR3` laptop model with the following hardware and software specifics:

- Intel Core i7-2630QM CPU @ 2.00Ghz
- 6 GB Ram
- Windows 7 x64
- GeForce GTX 460M with:
  - 192 CUDA Cores
  - 675 Mhz Graphics Clock
  - 1350 Mhz Processor Clock
  - 1250 Mhz Memory Clock
  - 1.5 GB GDDR5 Memory

Note: All tests were performed with the AC/DC adapter plugged in and system performance set to the maximum allowable levels. This is an important note because GPU performance can be severely throttled, for power concerns, if the adapter is not plugged in or performance is being limited by the user.

15.3 Benchmark Setup and Design

Premise

After several empirical tests and observations it was noted that GPU time was one of the main performance statistics that we wanted to explore and discuss. With our goal of accurately comparing all of the catalogued variations, a design procedure was needed that attempted to isolate all other factors that could influence execution time and exclusively focus on how the variation code injected in the device.

Auto generation of flames

The procedure is as follows. First, all of the variations needed to be auto-generated into respective `flam3` files which could be used as input for cuburn. Cuburn can run these `flam3` files individually and the CUDA profiler can produce output statistics (using methods described above). Immediately, using this approach a complication arises. A valid iterated function system must be contractive or contractive on average. Using a single transform that is solely our selected variation is not a valid iterated function system and cuburn may prematurely abort rendering the frame. Because of this, we chose a baseline iterated function system that each variation would be added to. This baseline iterated function consisted of two linear transforms. The transform code is seen in Figure 15.1:

$< flame >$
$< xform\ weight = $"0.33" ... $linear = $"1" $coefs = $" $-0.26\ 0.0\ 0.0\ -0.26\ 0.0\ 0.020$" $/ >$
$< xformweight = $"1.00" ... $linear = $"1" $coefs = $"$0.70\ 0.70\ -0.70\ 0.70\ 0.51\ -1.18$" $/ >$
...
$< /flame >$

Figure 15.1: Baseline Iterated Function System that each variation was applied to.

When rendering a single frame with standard coloring parameters, a size of 640 $\times$ 480, quality of 50, and the predefined autumn-themed color palette 10 the resulting flame looks like that of Figure 15.2:

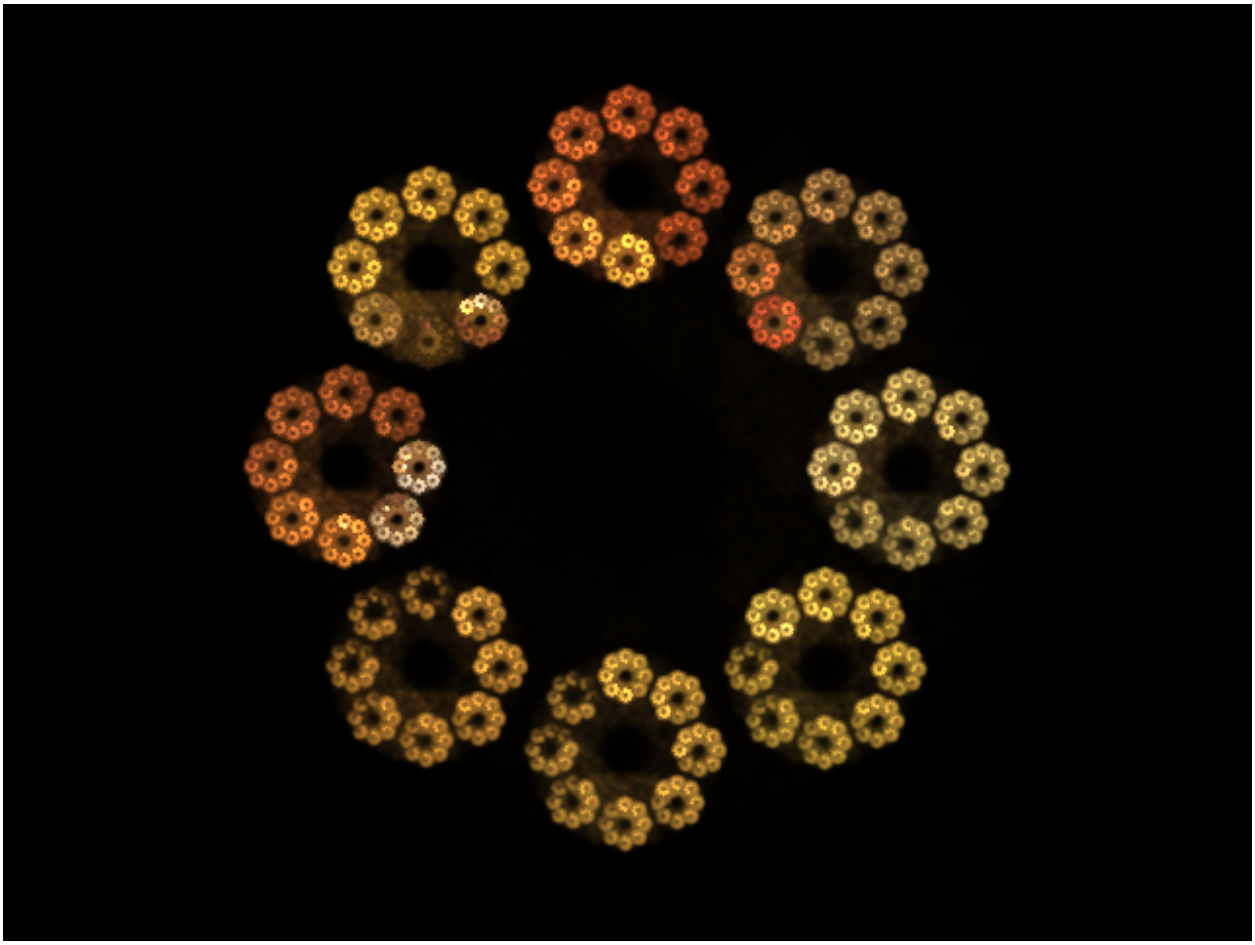


Figure 15.2: Visual of Baseline Iterated Function System

Now that an acceptable iterated function has been chosen we can append the appropriate variation to one of the xforms. We have chosen to append it to the second xform but have to note that this decision was does not have any significant effects besides that the chance of the variation being applied is $Second\ Xform_{Weight} \times Variation_{Weight}$ versus $First\ Xform_{Weight} \times Variation_{Weight}$ if it were applied to the first variation.

The new transform code for the second xform would look like Figure 15.3 if the `julia` variation was applied:

$< xformweight = \text{"}1.00\text{"} \ldots linear = \text{"}1\text{"}\ coefs = \text{"}0.70\ 0.70\ -0.70\ 0.70\ 0.51\ -1.18\text{"}\ julia = \text{"}1e-07\text{"}\ /\ >$

Figure 15.3: Julia variation being applied to Baseline Iterated Function System.

By keeping the baseline IFS fixed and just appending an additional variation for each flame we can effectively isolate the runtime differences caused by having to load the additional code onto our device and compute the additional variation. This will only hold if our initial assumptions that the runtime is consistent that for each generation of a flame with minimal deviation.

The extremely low weighting value (chance of variation being applied) of $1e-7$ was chosen because it will still be applied given the tremendous amount of points being computed however it will only influence it in a minor fashion. This small influence is what we hope to capture. The weight could have been dramatically increased but the entire benchmarking process would have taken an greatly increased amount of time and the graphs would more than likely need data transformations such as a log transformation in order to be useful.

Accumulating Kernel Execution Times

After using CUDA's `Compute Visual Profiler` developer tool to compare several pairs of the numerous variation profile results it was found that the only interesting kernel which changed was the main `iter` kernel. We have decided to focus on comparisons between this kernel when presenting our results. A drawback of the `Compute Visual Profiler` is that it is limited to comparing 2 profile results and lacks more sophisticated tools such as averaging multiple runs, computing standard deviations, and comparing numerous profile results. A hand crafted solution was in order if the data was needed to be properly visualized and analyzed.

The CUDA profile log file we are using displays each instance of the kernel running regardless if it is a kernel that has run previously. This results in numerous `iter` kernel entries for our log file. In order of this information to be of any use the individual `iter` GPU times needed to be accumulated to produce a Total GPU time. Once this was done all of the variations were compared using a bar chart sorted in ascending order. The graph confirmed our predictions that variations that applied more expensive operations such as modulus performed worse than variations that applied simple arithmetic such as linear. However, this was not enough to convince ourselves and conclude that this was the way the system operated.

Multiple Runs and Standard Deviation Analysis: Convincing ourselves, and you

Comparing `iter` kernel execution times between all catalogued variations is not merely enough. In order to account for the difference of execution times between runs, multiple runs were executed and the total execution time was averaged. Additionally, the standard deviation was computed in order to verify that it was not a statistically significant deviation which needed further analysis. The standard deviation proved to be of minimal concern and the conclusive results are presented in the next section and then visualized afterwards.

GPU Execution Time Table

Below in Table 15.1 are the average GPU execution times of 20 runs and their respective standard deviations of the `iter` kernel sorted in ascending order.

VARIATION APPLIED:	GPU EXECUTION TIME (μSEC):	STANDARD DEVIATION (μSEC):
linear	41,215.01	30.41
oscope	41,220.54	25.60
sinusoidal	41,482.11	33.98
spherical	41,541.97	36.79
bent	41,591.70	36.05
exp	41,603.94	34.31
bubble	41,609.78	37.94
exponential	41,670.39	36.64
horseshoe	41,684.67	33.05
square	41,687.31	30.63
waves	41,688.71	29.72
swirl	41,714.31	23.33
cross	41,726.91	34.17
cylinder	41,768.29	42.57
loonie	41,779.30	40.05
arch	41,815.04	37.59

tangent	41,832.91	34.37
rays	41,870.36	41.76
blur	41,871.57	28.48
bent2	41,871.63	35.45
scry	41,875.42	36.75
foci	41,893.15	36.83
stripes	41,911.58	39.83
blade	41,921.20	30.32
pre_blur	41,930.98	33.02
noise	41,941.71	40.02
fisheye	41,968.95	36.98
eyefish	41,977.16	30.31
split	42,008.76	33.99
butterfly	42,018.22	36.21
secant2	42,030.67	33.96
rectangles	42,035.00	29.72
curl	42,051.49	24.40
splits	42,069.96	34.11
perspective	42,181.27	31.24
waves2	42,257.88	25.12
popcorn	42,259.05	33.54
pdj	42,292.23	26.90
parabola	42,333.18	31.25
popcorn2	42,401.33	37.46
gaussian_blur	42,409.56	28.35
cell	42,562.43	44.09
curve	42,617.94	26.56
conic	42,636.51	43.74
lazysusan	42,642.68	31.45
separation	42,698.13	25.62
log	42,725.48	54.63
cosine	42,809.25	44.79
cos	42,823.28	38.41
sin	42,835.36	42.60
polar2	42,845.89	43.50
polar	42,853.39	51.13
cosh	42,861.36	44.98
pie	42,868.96	23.46
hyperbolic	42,884.68	52.65
handkerchief	42,909.86	62.95
heart	42,965.43	66.75
cot	42,966.10	47.70
tan	42,975.99	40.86
diamond	42,993.51	55.80
julia	43,026.11	79.77
power	43,058.50	60.50
disc	43,075.76	76.50
sinh	43,148.75	54.12

ex	43,213.52	46.36
tanh	43,221.40	47.42
coth	43,268.99	38.96
sec	43,316.82	46.80
spiral	43,317.85	59.94
csc	43,318.06	48.53
sech	43,451.97	50.67
fan2	43,466.66	62.67
flower	43,516.26	42.37
rings2	43,519.23	50.51
mobius	43,537.89	25.99
julian	43,575.00	51.50
juliascope	43,640.76	48.41
blob	43,698.43	56.94
boarders	43,741.34	110.07
escher	43,763.02	55.41
disc2	43,845.57	62.24
bipolar	43,847.78	77.48
csch	43,994.95	51.53
wedge	44,104.10	55.02
radial_blur	44,131.67	76.65
cpow	44,185.85	61.63
ngon	44,333.52	74.90
elliptic	44,484.01	76.08
super_shape	44,871.59	77.20
edisc	45,076.11	134.23
flux	45,527.62	217.42
rings	47,397.50	355.91
fan	47,726.17	393.94
modulus	49,125.81	225.78

Table 15.1: 'Iter' kernel performance results on each variation.

GPU Execution Time Bar Graph

This data can easily be visualized on a bar graph and as an additional feature error bars representing the standard deviation have been added. This is shown in Figure 15.4:

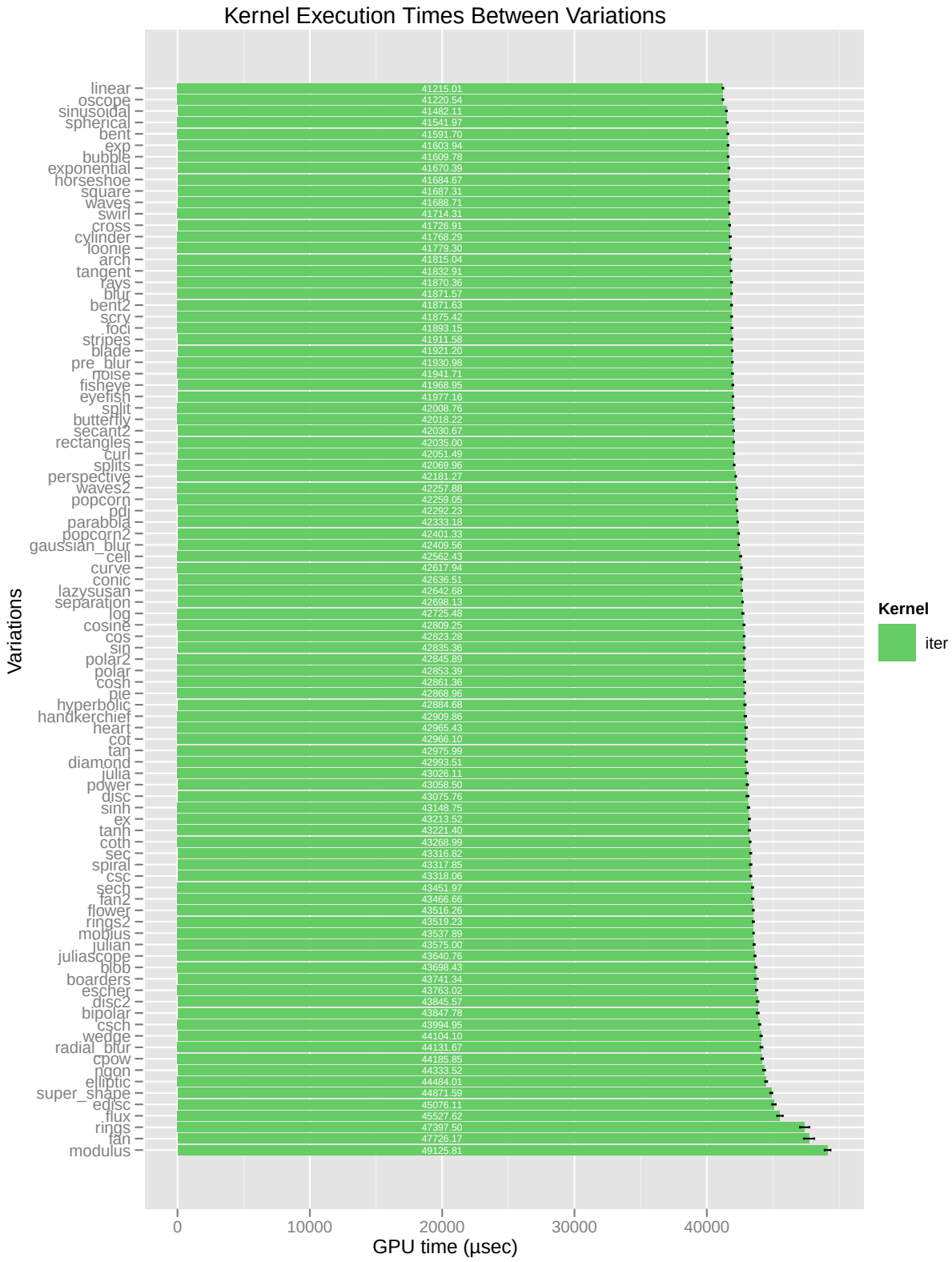


Figure 15.4: Bar graph of ‘iter‘ kernel performance results on each variation.

15.4 Discussion and Analysis

Visualizing the execution times easily makes for an interesting discussion concerning why certain variations performed the way they did as well as the explanations that explain the differences in execution times. The first observation is that it is rather difficult to cluster variations into performance groups as the execution times increase linearly (with a few outliers such as the `rings`, `modulus`, and the `fan` variation). The performance difference between the best and worst performing variations is on the order of 20% which is shows significance but for single frame renders is negligible to casual human observation. Of course if this was reframed with our goals being real time rendering then that casual 20% is now whopping difference which would prompt further optimizations. Furthermore, the standard deviations of lower performing variations were dramatically higher than more well behaved higher performing variations such as linear.

By observational sampling of high, mid-grade, and low performing variations, a conjecture that operations such as addition, subtraction, multiplication, division, generating a MWC random number, and basic trigonometric functions such as sine and cosine are less expensive in terms of GPU time than the operations that follows. These more expensive operations were found to be modulus, exponential math, square root values, and logarithms. By observing the actual code that will be dynamically generated on the device we can verify the conjecture above. Some of the highest performing variations in terms of performance are seen in Figure 15.5, Figure 15.6, and Figure 15.7. Lower performing variations are seen in Figure 15.8, Figure 15.9, and 15.10.

High Performing Variation

Three high performing variations are presented below:

```
ox += tx * w;
oy += ty * w;
```

Figure 15.5: High Performing Variation 1: Code for ‘linear‘ variation

```
float tpf = 2.0f * M_PI * {{pv.frequency}};
float amp = {{pv.amplitude}};
float sep = {{pv.separation}};
float dmp = {{pv.damping}};

float t = amp * expf(-fabsf(tx)*dmp) * cosf(tpf*tx) + sep;
ox += w*tx;
if (fabsf(ty) <= t)
        oy -= w*ty;
else
        oy += w*ty;
```

Figure 15.6: High Performing Variation 2: Code for ‘oscope‘ variation

```
ox += w * sinf(tx);
oy += w * sinf(ty);
```

Figure 15.7: High Performing Variation 3: Code for ‘sinusoidal‘ variation

Low Performing Variation

Three low performing variations are presented below:

```
float mx = {{pv.x}}, my = {{pv.y}};
float xr = 2.0f*mx;
float yr = 2.0f*my;

if (tx > mx)
        ox += w * (-mx + fmodf(tx + mx, xr));
else if (tx < -mx)
        ox += w * ( mx - fmodf(mx - tx, xr));
else
        ox += w * tx;

if (ty > my)
        oy += w * (-my + fmodf(ty + my, yr));
else if (ty < -my)
        oy += w * ( my - fmodf(my - ty, yr));
else
        oy += w * ty;
```

Figure 15.8: Low Performing Variation 1: Code for 'modulus' variation

```
float dx = M_PI * ({{px.affine.xo}} * {{px.affine.xo}});
float dx2 = 0.5f * dx;
float dy = {{px.affine.yo}};
float a = atan2f(tx, ty);
a += (fmodf(a+dy, dx) > dx2) ? -dx2 : dx2;
float r = w * sqrtf(tx*tx + ty*ty);
ox += r * cosf(a);
oy += r * sinf(a);
```

Figure 15.9: Low Performing Variation 2: Code for 'fan' variation

```
float dx = {{px.affine.xo}} * {{px.affine.xo}};
float r = sqrtf(tx*tx + ty*ty);
float a = atan2f(tx, ty);
r = w * (fmodf(r+dx, 2.0f*dx) - dx + r * (1.0f - dx));
ox += r * cosf(a);
oy += r * sinf(a);
```

Figure 15.10: Low Performing Variation 3: Code for 'rings' variation

In closing, these performance benchmarks allow us to observe how the device code runs reliably without having to crawl over ten thousand lines of assembler to find out which operations they use. These benchmarks show that we don't have to count opcodes in order to understand performance. It is evident from the charts that the performance estimates from assembly are reliable without things such as memory accesses getting in the way.

CHAPTER 16

USAGE AND HOST-SIDE API

Despite cuburn's internal complexity, its API is straightforward. Only two user-facing modules are required to render a flame: `cuburn.genome` and `cuburn.render`.

To load a JSON genome file, call `cuburn.genome.load_info` with the file's contents as a string argument. This loads all genomes in the file, as well as information about rendering parameters, if present. Pass this information to `cuburn.render.Renderer` to create a new instance of that object. Use the `compile` method to perform code generation, infer and store runtime parameters, and attach the compiled module to the current CUDA context. Call the `render` object with a list of `(name, start_time, end_time)` tuples, producing a generator, and read the resulting `RenderedImage` objects from that generator.

16.1 Behind the scenes

That one call to `render` does quite a bit.

Because it is a generator function, the `render` method is not necessarily blocking. After the initial call to create the generator loads two frames into the GPU's asynchronous task dispatch queue, each frame that gets read triggers the dispatch of another frame for later reading. In blocking mode, if a frame is requested when none is available, the thread will repeatedly sleep until a frame is ready to return, allowing other tasks to execute in different threads. When blocking is disabled, even this behavior is gone: the generator simply returns `None` immediately if no frames are available. This allows rendering to be used from, say, a GUI-driven application without the inconvenience of threading *or* the performance loss that comes from infrequent polling; since multiple frames are queued, polling only needs to happen once per frame to keep the GPU at full load.

In order for this method to work, care must be taken to avoid conflicting use of shared resources by asynchronously-scheduled kernels. On the other hand, strict serialization of kernels reduces performance, as it does not allow proper load-balancing — important in cuburn due to the use of unusually long-running kernel invocations. Asynchronous dispatch therefore requires the use of multiple, cross-synchronized CUDA streams, ensuring that asynchronous work which does not result in buffer conflicts is free to proceed until shared resources need once again be handled individually.

There are three contested resources which must be synchronized: the point log, used as the destination of iteration samples and the source for sorting; the accumulation buffer, used as the destination for the entire iteration process and the source for density estimation; and the final output buffer, destination of color filtering and source for the host-to-device copy. Each source has its own stream, and events are injected into the destination streams to act as barriers to prevent one stream from outpacing another in a manner that could cause inconsistent access.

To allow for asynchronous processing by the host, there are two independent device buffers that are preallocated for asynchronous copies. The generator's control logic only yields a buffer when it has been copied, and queues everything up to the next overwrite of that buffer on the device before yielding it again. When the next iteration begins, the previous buffer is reclaimed. This ensures that the current buffer is never overwritten by asynchronous DMA while in use by the application.

An instance of this asynchronous dispatch architecture is depicted in an abbreviated form in Figure 16.1.

A curiosity of this architecture is that the `render` method is a single monolithic function. This is not an example of poor programming practice or lazy design, but rather a cautious and proactive choice intended to make development easier and program operation more reliable. Inside the rendering function, dozens of resources on both the host and the device must be obtained, many of which depend on values calculated during the allocation of previous resources. Using a single, local namespace for these values ensures that the complex, interrelated calculations are all present in a single file for easy inspection, modification, and documentation. These parameters are never accessed outside of the render loop, so a more modular, object-oriented approach would have added overhead and almost certainly introduced additional bugs resulting from repeated calculations drifting out of sync with revisions to different files. This also enables more accurate tracking of resource lifetime, as Python's reference tracking occasionally frees device resources too aggressively as they pass out of scope, resulting in stalls due to synchronization barriers imposed by CUDA on deallocation functions.

16.2 Command-line use

Cuburn also comes with a command-line client. This client includes several options to make convertingn and rendering an animation from an XML genome easier, as well as a simple OpenGL interface which displays frames as they are rendered. The documentation for the command-line interface is repoduced in Figure 16.2.

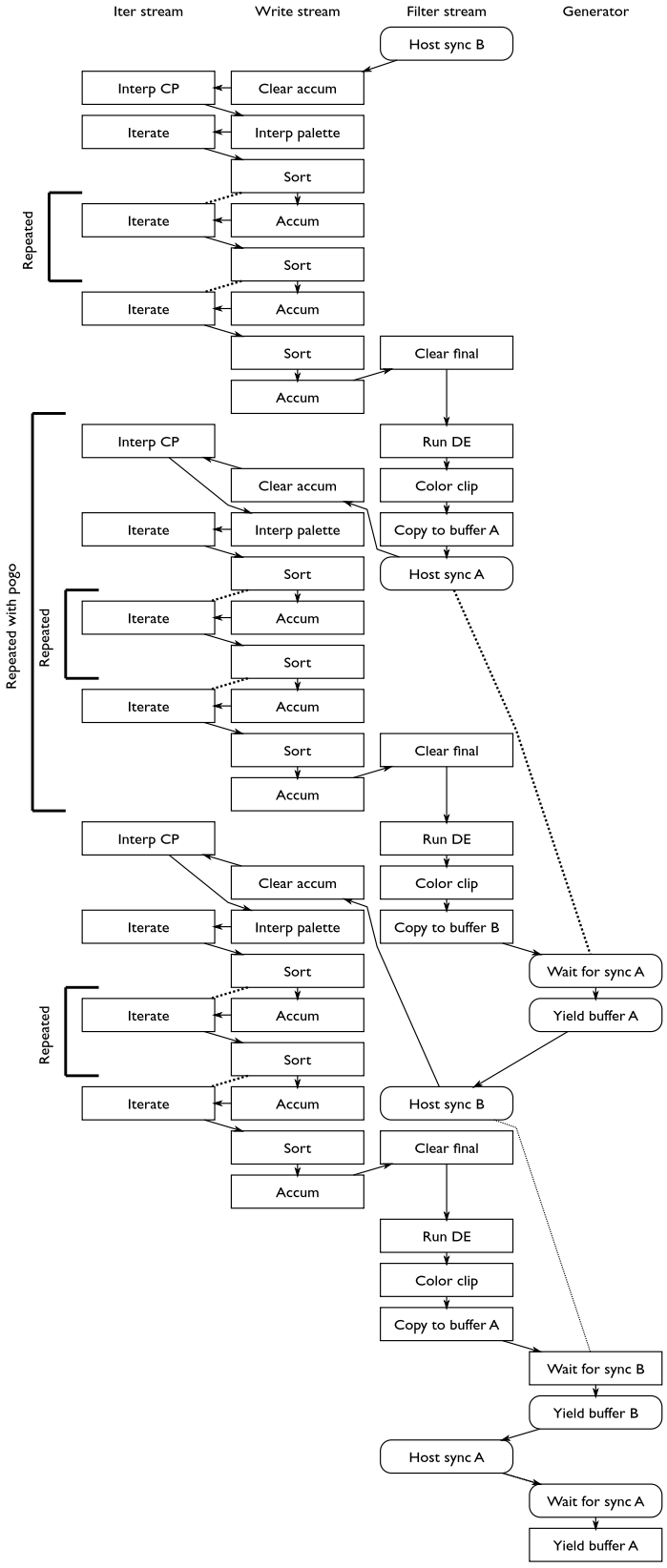


Figure 16.1: One example of the task dispatch pattern for a particular flame animation rendered using deferred writeback. Solid arrows indicate host dispatch order; dashed arrows indicate buffer contention and event barriers; vertical order within a stream indicates mandatory execution order; vertical order between streams indicates expected execution order. Tasks that may block are represented by boxes with rounded corners.

```
usage: main.py [-h] [-g] [-j [QUALITY]] [-n NAME] [-o DIR] [--resume] [--raw]
               [--nopause] [-s TIME] [-e TIME] [-k TIME] [--renumber [TIME]]
               [--qs SCALE] [--scale SCALE] [--tempscale SCALE]
               [--width PIXELS] [--height PIXELS] [--test] [--keep] [--debug]
               [--sync] [--sleep [MSEC]]
               FILE

Render fractal flames.

positional arguments:
  FILE                Path to genome file ('-' for stdin)

optional arguments:
  -h, --help          show this help message and exit
  -g                  Show output in OpenGL window
  -j [QUALITY]        Write .jpg in addition to .png (default quality 90)
  -n NAME             Prefix to use when saving files (default is basename of
                      input)
  -o DIR              Output directory
  --resume            Do not render any frame for which a .png already exists.
  --raw               Do not write files; instead, send raw RGBA data to
                      stdout.
  --nopause           Don't pause after rendering when preview is up

Sequence options:
  Control which frames are rendered from a genome sequence. If '-k' is not
  given, '-s' and '-e' act as limits, and any control point with a time in
  bounds is rendered at its central time. If '-k' is given, a list of times
  to render is given according to the semantics of Python's range operator,
  as in range(start, end, skip). If no options are given, all control points
  except the first and last are rendered. If only one or two control points
  are passed, everything gets rendered.

  -s TIME             Start time of image sequence (inclusive)
  -e TIME             End time of image sequence (exclusive)
  -k TIME             Skip time between frames in image sequence. Auto-sets
                      --tempscale, use '--tempscale 1' to override.
  --renumber [TIME]   Renumber frame times, counting up from the supplied start
                      time (default is 0).
```

Figure 16.2: Usage information for the command-line cuburn application.

```
Genome options:
  --qs SCALE          Scale quality and number of temporal samples
  --scale SCALE       Scale pixels per unit (camera zoom)
  --tempscale SCALE   Scale temporal filter width
  --width PIXELS      Use this width. Auto-sets scale, use '--scale 1' to
                      override.
  --height PIXELS     Use this height (does *not* auto-set scale)

Debug options:
  --test              Run some internal tests
  --keep              Keep compilation directory (disables kernel caching)
  --debug             Compile kernel with debugging enabled (implies --keep)
  --sync              Use synchronous launches whenever possible
  --sleep [MSEC]      Sleep between invocations. Keeps a single-card system
                      usable. Implies --sync.
```

Figure 16.2: Usage information for the command-line cuburn application (continued).

CHAPTER 17

DESIGN SUMMARY

Cuburn is a Python library for rendering fractal flames on the GPU. This chapter provides an overview of the operation of this library.

17.1 Device software

After host-side initialization, a frame rendering begins with an invocation of the interpolation function constructed for the current genome (Chapter 12). During interpolation, the splines representing an animation are loaded from global memory, interpolated against at the current control point time, and optionally used as the input of more complex functions to produce control point parameters. Many control points are evaluated simultaneously, and the results are stored in a global memory array.

An iteration kernel plays the chaos game, evaluating the trajectory of individual points as they pass aroudn the attractor. The iteration kernel is carefully tuned for high occupancy. When the kernel is generated to operate in deferred mode, it is ALU-bound, and so care is taken to avoid warp divergence without resulting in trajectory convergence (Chapter 8). Each iteration of the kernel produces an image sample, which is stored to a sample log on the device in a condensed format (Chapter 12).

A sort engine built for this project (Chapter 13) efficiently rearranges the sample log's contents for efficient histogram generation. The sort engine is a scan-based radix sort, although it makes extensive use of low-level hardware features in a manner that differs from most other GPU radix sorts to attain record-setting levels of performance.

After the functions comprising the sort engine have concluded, the log is processed by an accumulation function (Chapter 12). This function uses shared memory to generate a histogram of the logged trajectory data quickly and accurately. Since memory is limited, the point log is finite, and often too short to contain a full image's length of samples; therefore, the process of iterate, log, sort, and accumulate usually repeats several times before the next stage is reached.

Density estimation with antialiasing correction is applied to the generated full-scale image (Chapter 14). This step, along with colorspace conversion and out-of-bounds clamping, is used to convert the generated histogram buffer into a low-dynamic-range image. The image is then sent via a memory copy to the host, where it is compressed in the current output format.

17.2 Host software

The cuburn library uses NumPy extensively and returns buffers as NumPy arrays, meaning applications using cuburn as a library will also need to use NumPy for numerical manipulation. Internally, cuburn also depends on PyCUDA and Tempita. When using XML genome files, cuburn additionally depends on frostlib and libflam3 to perform conversion. A cuburn client application can be constructed in under ten lines of Python code, and an example command-line application which provides a rich set of options is available for non-programmers to use.

Despite a simple API, the control flow of the rendering process is complex. An example internal flow is provided in Figure 17.1.

Cuburn relies heavily on runtime code generation to attain both speed and flexibility (Chapter 7). Kernels are generated before rendering starts in response to analysis of both the genome to be rendered and the hardware it is to be rendered on. The code needed to render the genome is computed in multiple passes, each pass adding additional auxiliary information to prepare the next pass, and then sent to `nvcc` for compilation. A final pass performs "monkey-patching" of opcodes in place to work around the absence of certain hardware instructions from the PTX ISA. The module is then loaded in the current CUDA context, allowing it to be used for host computation.

Resource allocation is performed at the call to the `render` function (Chapter 16). This function masks complicated asynchronous dispatch behind the Python generator API; in most cases, a Python `for` loop or `imap` statement is all that is needed to read and process frames. When the function exits, Python's reference management system cleans up outstanding references, setting the CUDA context up for further rendering or a clean exit.

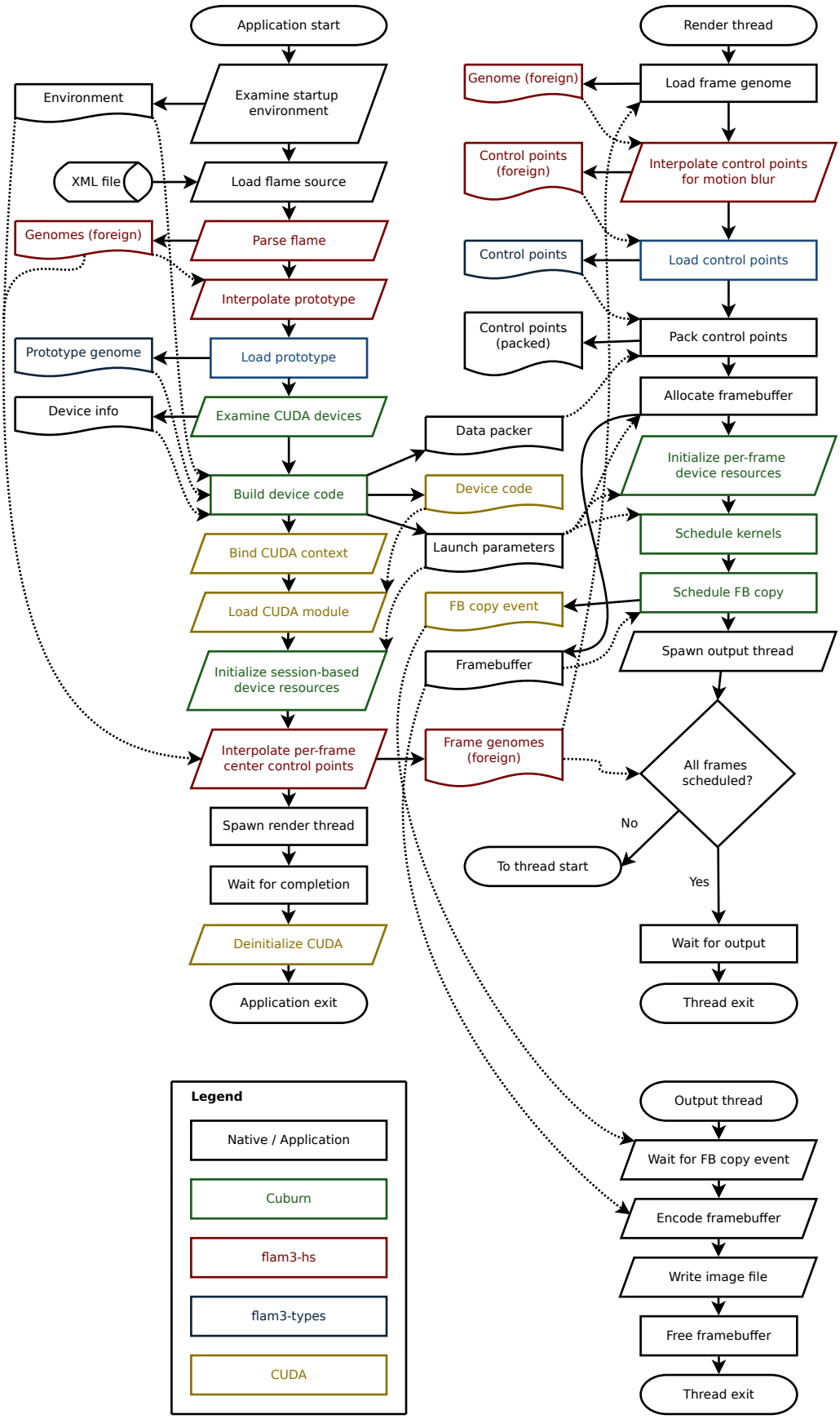


Figure 17.1: The host-side workflow of the example application. This diagram describes the workflow of the most recent Windows port. The most recent development version differs slightly, and no longer follows a fixed dispatch pattern.

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 11.1: 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 11.2: 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 11.3: 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 11.5, FIGURE 11.6, FIGURE 11.7, FIGURE 11.8, FIGURE 11.9, FIGURE 11.10, FIGURE 11.11, FIGURE 11.12: 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 10.1

- *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 14.1: 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 14.2: 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] S. Marlow, "The Haskell 2010 Language Report," 2010. http://www.haskell.org/onlinereport/haskell2010/

[24] S. Robertson, "PyPTX," 2010. http://bitbucket.org/srobertson/pyptx

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

[26] 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

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

[28] 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

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

[30] 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

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

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

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

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

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

[36] 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

[37] 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

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

[39] 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

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

[41] 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

[42] 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

[43] 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

[44] 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

[45] 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

[46] 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

[47] 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

[48] 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