Rendering Hypercomplex Fractals

by Anthony Atella

An Honors Project Submitted in Partial Fulfillment

of the Requirements for Honors in

The Department of Mathematics

and Computer Science

The School of Arts and Sciences

Rhode Island College

2018

Abstract

Fractal mathematics and geometry are useful for applications in science, engineering, and art, but acquiring the tools to explore and graph fractals can be frustrating. Tools available online have limited fractals, rendering methods, and shaders. They often fail to abstract these concepts in a reusable way. This means that multiple programs and interfaces must be learned and used to fully explore the topic. Chaos is an abstract fractal geometry rendering program created to solve this problem. This application builds off previous work done by myself and others [1] to create an extensible, abstract solution to rendering fractals. This paper covers what fractals are, how they are rendered and colored, implementation, issues that were encountered, and finally planned future improvements. An attached appendix contains UML diagrams.

Documentation, repositories, a gallery, and executables can be found at
`https://anchorwatchstudios.com/chaos/`

Contents

1 Introduction **1**
1.1 Overview 1

2 Fractal Mathematics **2**
2.1 Weierstrass 2
2.2 Cantor 3
2.3 Koch 4
2.4 Sierpiński 5
2.5 Hausdorff 5
2.6 Fatou & Julia 6
2.7 Mandelbrot 6
2.8 Norton 7
2.9 White & Nylander 8
2.10 Applications 8

3 Fractal Algorithms **10**
3.1 Cantor Set 10
3.2 Tree 10
3.3 Julia & Juliabulb 11
3.4 Mandelbrot & Mandelbulb 11
3.5 Newton-Basin 12
3.6 Mandelbox 12

4 Fractal Rendering **15**
4.1 Methods 15
4.2 Parallelism 18
4.3 Video 18

5 Fractal Shading **19**
5.1 Methods 19
5.2 Lighting 20

6 Implementation **21**
6.1 Design Patterns 21
6.2 Core 22
6.3 Swing 24
6.4 Android 26

7 Issues Encountered **28**
7.1 True Resolution 28
7.2 Observer Feedback 28
7.3 Document Serialization 28
7.4 Arbitrary Method Signatures 28
7.5 Complex Plot Rotation 29
7.6 Image Scaling 29
7.7 Timeline Visibility 29

8 Future Updates **30**
8.1 Microkernel Architecture . . . 30
8.2 Fractals . . . 30
8.3 Shaders . . . 30

9 Conclusion **31**
9.1 Special Thanks . . . 31

10 Bibliography **32**

11 Appendix: UML Diagrams **34**

1 Introduction

Chaos is capable of rendering fractals with different rendering methods and shaders applied dynamically. The program is extensible so new fractals, shaders, and rendering methods can be added. These fractals can be saved to and loaded from files. Each file has keyframes and video settings. Chaos uses these keyframes and settings to produce .mp4 video using linear interpolation between keyframes. Chaos also supports exporting .png images. Fractals currently implemented include the Cantor set, Julia, Juliabulb, Mandelbox, Mandelbrot, Mandelbulb, Newton Basin, and trees. Chaos can render fractals using either Java2D or OpenGL.

The program is implemented in Java for PC and is currently being implemented in Android. The interfaces rely on the AWT, Swing, and Android libraries. Geometry is rendered with OpenGL using the JOGL library, OpenGLES using the Android libraries, and standard Java2D. The portability of Java allows the program to run on four out of five of the largest platforms; Windows, Macintosh, Linux, and Android. Writing a version for iOS in Objective C would be trivial due to the similarity of the languages and the OpenGL bindings.

The PC version is geared more towards technical users and artists that want to produce high quality images and video or explore the mathematical extremes and behaviors of fractals. The mobile version would be more appropriate for non-technical users and is designed to reach a wider audience by trading functionality for ease of use. The mobile version will have a virtual reality mode. This mode will use stereoscopic cube maps that are pre-rendered (due to power consumption and the time cost of rendering). This version will also export images. Sharing to social media will be easier with built in sharing features that mobile platforms offer. Sharing features will streamline deployment of images and increase usership through word-of-mouth.

1.1 Overview

The first chapter will cover the background of fractal mathematics. It's difficult to describe what a fractal is. Learning the steps that people took to come to our current knowledge on the subject does a better job of explaining what a fractal is than a rambling, heuristic description.

The second, third, and fourth chapters contain pseudocode algorithms for each fractal, render method, and shader, respectively. Implementation specific code can be found in the appendix.

Chapter five covers implementation specific details. Design patterns and paradigms are covered in the first section. The second section contains details about the portable core system structures, such as the Document and Fractal. The third section covers interface implementations in PC and Android.

Chapters six and seven discuss the issues encountered when writing Chaos, and the bugs that still exist. Chapter eight is a plan for future development cycles and bug fixes.

Finally, chapter seven is the bibliography and chapter eight is the appendix which contains UML diagrams that illustrate the architecture of the system.

2 Fractal Mathematics

A fractal is a curve or geometric structure that is statistically self-similar at all scales [2]. These visually striking structures have interesting mathematical properties, like having a finite area but an infinite surface area. Another peculiar property of fractals is that they are continuous

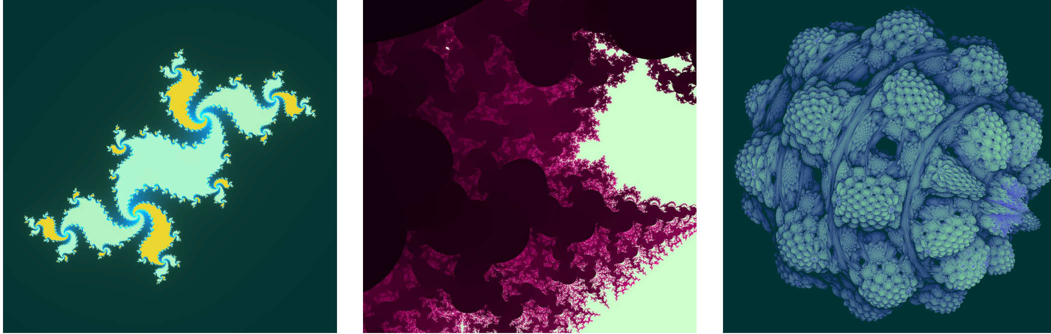

Figure 1: Examples of fractal geometry

everywhere, but differentiable nowhere. Fractals are found in nature in surprising abundance. The healthy human heartbeat [3, 4], the dendrites of a river, plants, lightning, galaxies, smoke swirling through the air, the tempo of a rapidly dripping faucet, and even the way humans build roads on a global scale are all examples of natural fractals. Figure 1 shows some examples of fractals rendered by Chaos.

2.1 Weierstrass

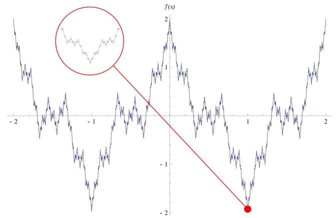

Figure 2: The Weierstrass curve [5]

The term fractal wasn't coined until 1975 by Benoit Mandelbrot [6]. Until then they were called "mathematical monsters". Mathematicians avoided these "monsters" because they had no tools like computers available to analyze them. Karl Weierstrass was the first mathematician to discover the fractal behavior of a curve in 1872. Weierstrass argued that a curve doesn't have to be differentiable just because it is continuous. He presented this argument as a sum of a Fourier series:

$$f(x) = \sum_{n=0}^{\infty} a^n cos(b^n \pi x)$$

Where $0 < a < 1$, b is a positive odd integer, and $ab > 1 + \frac{3}{2}\pi$. To put it simply, the Weierstrass curve contains smaller and smaller copies of itself. Any segment of the curve one observes will contain the whole. The curve isn't differentiable anywhere because there are critical points on it, and those critical points are contained within every point. In other words, every point is a critical point at some scale. This notion challenged the conventional wisdom of the time, which was that all continuous curves are differentiable, except at their critical points. Figure 2 illustrates how a point on the Weierstrass curve contains a smaller version of the curve.

2.2 Cantor

Georg Cantor was a student of Weierstrass at the University of Berlin. In 1883, Sixteen years after receiving his doctorate degree, Cantor came to a similar conclusion. He introduced the Cantor function, also known popularly as "The Devil's Staircase". This function is comparable to the Weierstrass curve in that it is self-similar at all scales and is produced by altering smaller graph segments on an existing graph with each iteration. The Cantor function can be expressed as a piecewise recursive function:

$$f_0(x) = x$$
$$0 \leq x \leq 1$$
$$f_{n+1}(x) = \begin{cases} \frac{1}{2} f_n(3x) & 0 \leq x \leq \frac{1}{3} \\ \frac{1}{2} & \frac{1}{3} \leq x \leq \frac{2}{3} \\ \frac{1}{2} + \frac{1}{2} f_n(3x-2) & \frac{2}{3} \leq x \leq 1 \end{cases}$$

The Cantor function is often expressed as ternary, and therefore calculated in base three for simplicity. On the 0th iteration the function is a straight line from (0, 0) to (1, 1). If the digit in the nth decimal place is a 1 on iteration n, that point is excluded from the set. In other words, the middle third is always excluded from the set each step. Figure 3 shows how a staircase is revealed as n approaches infinity.

Figure 3: The Cantor function after one, two, and three iterations [7]

The Cantor set is related to the Cantor function. The Cantor set is the set of points that have a derivative that is non-zero after iterating. The visible lines of the Cantor set at level n can be thought of as representing the diagonal lines of the Cantor function at iteration n, as illustrated in Figure 4. Each pair of smaller thirds are drawn beneath the next, like a binary tree. The Cantor ternary set is defined as:

$$C_n = \{x|x \in [0,1] \quad \text{and} \quad f_n'(x) \neq 0\}$$

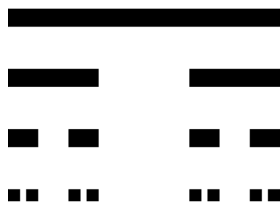

Figure 4: The Cantor set after three iterations

$$C = \cap_{n=0}^{\infty} A_n$$

$$A_0 = [0,1]$$

$$A_1 = [0, \frac{1}{3}] \cup [\frac{2}{3}, 1]$$

$$A_2 = [0, \frac{1}{9}] \cup [\frac{2}{9}, \frac{1}{3}] \cup [\frac{2}{3}, \frac{7}{9}] \cup [\frac{8}{9}, 1]$$

$$A_3 = ...$$

The Cantor function and set have strange and interesting properties that offer more insight into the "monsters". Cantor's function at infinity iterations has an overall increase of 1 on the interval [0,1], and yet each point has a derivative of 0. Another strange feature of the Cantor function is that it has a length of 2 at infinity iterations. At zero iterations it has a length of $\sqrt{2}$. A completely flat line would have a length of 1. This implies the line traveled straight across the x-axis, then straight up the y-axis! The Cantor set is also a paradox. The number of elements in the set at infinity iterations is infinite and uncountable. The set is extremely large, and yet the length of the set is 0 at infinity iterations, making it also extremely small [8].

2.3 Koch

Niels Fabian Helge von Koch, a Swedish mathematician, wanted to show that this phenomenon could be demonstrated with only simple geometric primitives. In a paper he wrote in 1904 [9] he describes a new curve that exhibits the same behavior as the Cantor function and the Weierstrass curve. The length of the Koch curve increases with each iteration by adding smaller triangles to the curve. Three of these curves together form the famous Koch snowflake, as seen in Figure 5. Koch's idea of combining three curves to form a closed polygon was an important transition from fractal algebra to fractal geometry.

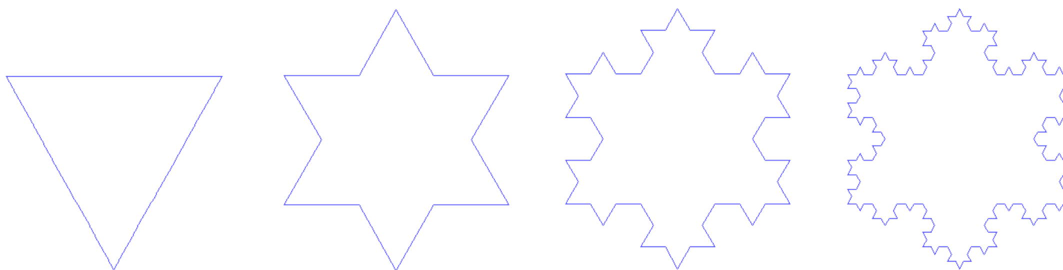

Figure 5: The Koch snowflake at zero, one, two, and three iterations [10]

2.4 Sierpiński

Wacław Sierpiński was a Polish mathematician who also extended the ideas of Cantor. He introduced the Sierpiński carpet and Sierpiński triangle, both of which are like the Koch snowflake in that they build off Cantor's idea of thirds in a geometric way. The carpet is constructed by making a Cantor set in two dimensions at the same time. The triangle is constructed by breaking up the larger triangle into three smaller similar ones with a hole in the middle, recursively. The Menger sponge is a three-dimensional variant of the Sierpiński carpet. The discovery of the Apollonian gasket by Gottfried Wilhelm Leibniz, a circular version, predates Sierpiński's findings. Figure 6 shows the similarities between these three fractals.

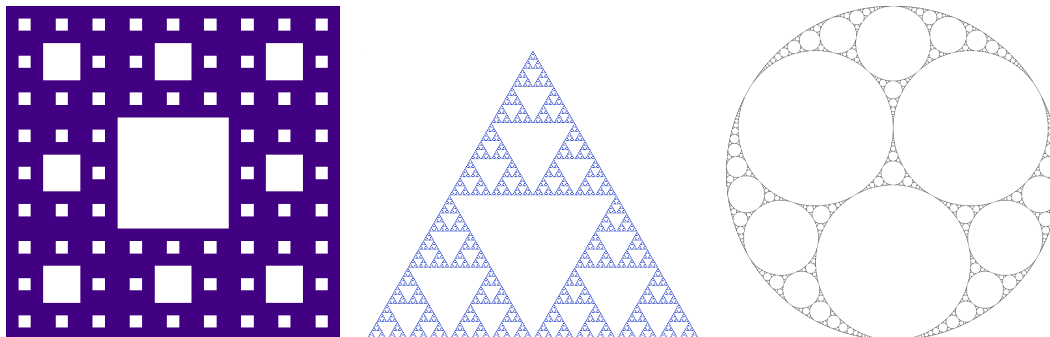

Figure 6: The Sierpinksi carpet, Sierpinski triangle, and Apollonian gasket [11, 12, 13]

2.5 Hausdorff

A German mathematician named Felix Hausdorff developed a formula for determining how rough a surface is in 1918. The Hausdorff dimension describes this property with a ratio of how similar the figure is to itself and the scale at which the similarity occurs. The Hausdorff dimension can be defined as:

$$D = \frac{log(n)}{log(s)}$$

Where n is the number of self-similar figures within the figure, and s is the scalar multiple that would make the self-similar figures the original size. Normal geometry always has an integer Hausdorff dimension, although non-integer dimensions are allowed. For example, if a line is divided into two equal parts it has a Hausdorff dimension of one because there are two parts that need to be increased by a multiple of two to become the original size, giving $\frac{log(2)}{log(2)} = 1$. Similarly, if a plane is divided into four equal parts it will have a Hausdorff dimension of two because it has four parts that need to be increased by a multiple of two to become the original size, giving $\frac{log(4)}{log(2)} = 2$. The same concept can be applied to higher dimensions. These results are integers because the self-similar figures fill the entire original figure. When the self-similar figures create space within the whole their Hausdorff dimension becomes a decimal number less than their original dimension. The Sierpinski triangle is a self-similar figure that has a decimal Hausdorff dimension. On the first iteration, a Sierpinski triangle is broken into three equal parts that are half the size of the original, leaving a space in the middle. This gives a Hausdorff dimension of $\frac{log(3)}{log(2)} \approx 1.58$. The triangle occupies a two-dimensional space, yet it is only using a

little more than three quarters of that space. Mandelbrot later used the Hausdorff dimension in his definition of a fractal.

2.6 Fatou & Julia

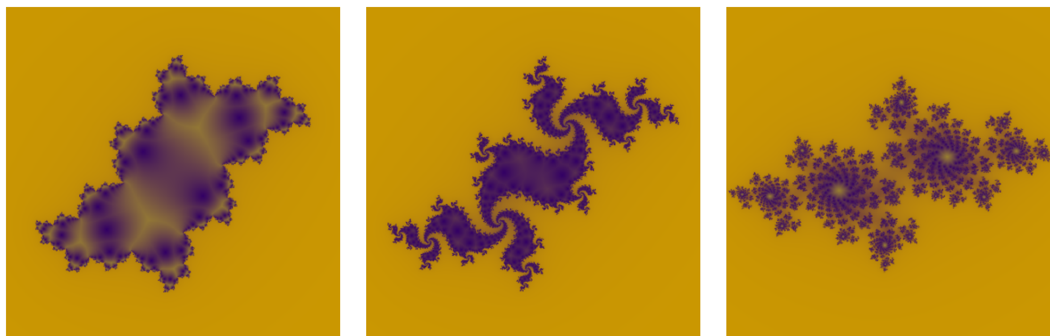

Figure 7: The Julia set with various C values

Pierre Fatou introduced the field of holomorphic dynamics to mathematics in 1920, or what we would now call complex dynamics. This field focuses on numbers in the complex plane and functions that involve iteration. Gaston Julia was simultaneously arriving at the same conclusions as Fatou. Unfortunately for Fatou, Julia was more persistent in patenting his work, and we now refer to the set they both discovered as the Julia set. The complement of the Julia set is sometimes referred to as the Fatou set as a conciliation to Fatou. The Julia set can be defined as:

$$J_n = \{z | \lim_{n\to\infty} f_n(z) \neq \infty\}$$

Where,

$$f_n(z) = \begin{cases} z & n = 0 \\ f_{n-1}(z)^2 + C & n > 0 \end{cases}$$

Where z is a complex number, and C is a complex constant. As the value of C changes the geometry changes drastically. Figure 7 shows a Julia set with various C values. The graph can be generated by calculating whether each pixel is in the set, passing the pixel as the complex number z. These sets were discovered relatively early, but they still couldn't be rendered in a practical way. Computers still weren't available commercially. It was only later that Mandelbrot would use computer science to peer deeper into the "monsters". This set would later serve as the backbone for Mandelbrot's work.

2.7 Mandelbrot

Fractal mathematics and academia in general lost momentum because of World War I and II, and it wasn't until the 1970s when Benoit Mandelbrot, the most recognized person in fractal geometry, discovered a way to unify all Julia sets into one incredibly similar equation. Mandelbrot's equation is the reverse of the Julia set. Instead of declaring some constant C and starting at the point z, Mandelbrot set z to $0 + 0i$ and passed the point as C. The Mandelbrot set relates to the Julia set because it contains all Julia sets. Each point of the Mandelbrot set represents

some value C that can be used in the Julia function. Julia sets with C values inside the Mandelbrot set are contiguous, where values outside the set are "fractured" and non-contiguous. The Mandelbrot set is self-similar, as illustrated in figure 8, and it describes an infinite number of Julia sets that are also self-similar. The Mandelbrot set is defined as:

$$M_n = \{C| \lim_{n \to \infty} f_n(z) \neq \infty\}$$

Where,

$$f_n(z) = \begin{cases} 0 + 0i & n = 0 \\ f_{n-1}(z)^2 + C & n > 0 \end{cases}$$

Where C is a complex constant. Mandelbrot was able to render the Julia and Mandelbrot sets for the first time with the help of computers, and the fractal community was reinvigorated once they saw the awesome beauty of a complex fractal set rendered for the first time.

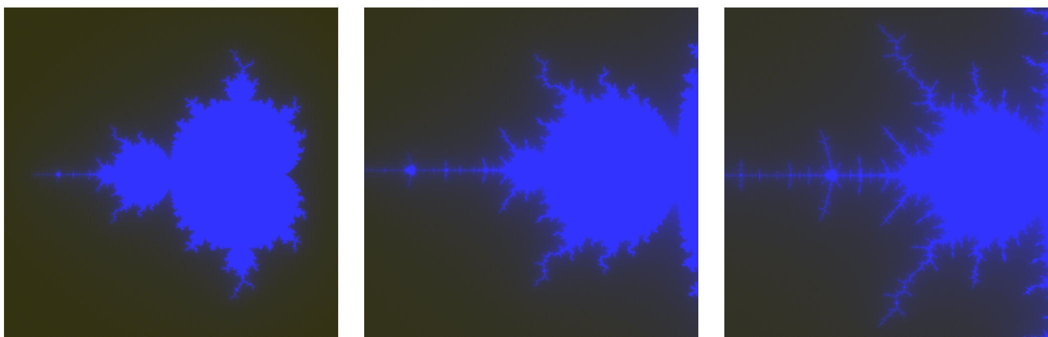

Figure 8: The Mandelbrot set at various zoom depth

2.8 Norton

Alan Norton was the first person to take these concepts to a higher dimension in 1982 using quaternions, a four-dimensional complex number [14, 15]. Complex dimensions must be bicomplex, or in pairs of two, to be valid. Each real number must be paired with a multiple of $\sqrt{-1}$, so it makes sense to skip the third dimension and increase the number of dimensions to four. Norton described his images as two-dimensional and three-dimensional "slices" of four-dimensional quaternions, as shown in Figure 9.

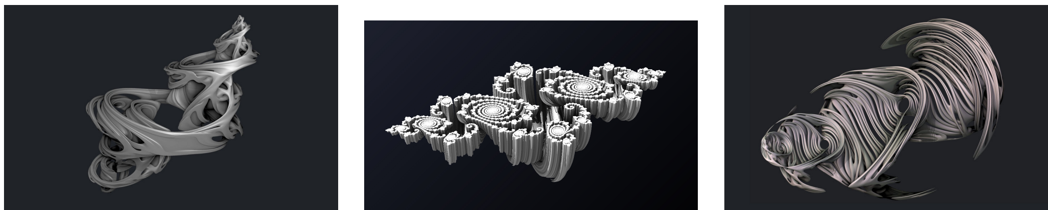

Figure 9: Quaternion (4D) Julia sets rendered in 3D [16, 17]

The methods used to generate these images are the same as the previously discussed complex plotting methods, except that quaternions are multiplied differently. Bisection of the three-dimensional geometry reveals the two-dimensional geometry within. Other complex fractals like the Mandelbrot have been rendered like this as well.

2.9 White & Nylander

Daniel White and Paul Nylander rendered a three-dimensional triplex version of the Mandelbrot set in 2009—the Mandelbulb. They created a formula to square triplex numbers. Sir William Rowan Hamilton struggled with this problem in the 1840s, which eventually led him to the discovery of quaternions. White and Nylander circumvented this issue using spherical coordinates. The mapping of triplex coordinates to Cartesian coordinates is as follows:

$$x + yi + zj \in \mathbb{R}^3 : \begin{cases} x = \rho cos(\theta)cos(\phi) \\ y = \rho cos(\theta)sin(\phi) \\ z = \rho sin(\theta) \end{cases}$$

Where $i^2 = j^2 = ij = -1$. The Juliabulb and other complex fractals can also be rendered using Julia's formula in conjunction with White and Nylander's formula. Bisections of these three-dimensional geometries also reveal the underlying two-dimensional geometry. White and Nylander inspired others to create many variations of triplex fractals, such as the Mandelbox, Menger sponge, Sierpiński systems, and n-gons. Figure 10 shows a Mandelbulb rendered with Chaos using White and Nylander's formula.

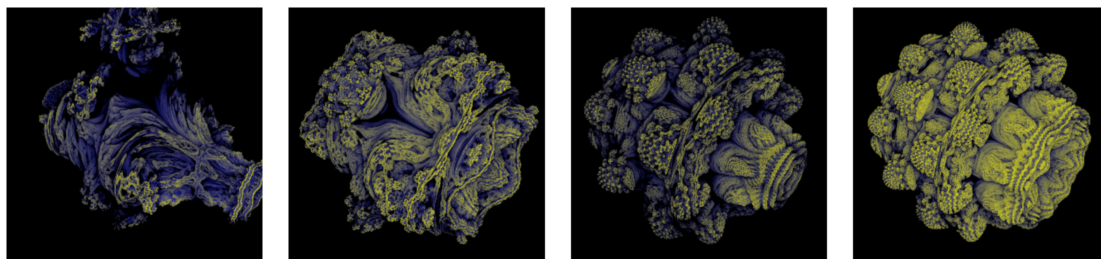

Figure 10: The Mandelbulb with increasing n values

2.10 Applications

Fractals aren't just pure mathematical constructs. They have surprisingly useful applications in real world situations. Fractal modeling software is useful for generating artwork, but it is also useful for visually identifying if a fractal model is appropriate for describing a system, as well as predicting how that system might change over time. Fractal geometry is used as a tool in various industries to improve product quality, speed, and size as well.

Fractals can be used to generate three-dimensional mountains and landscapes that appear to be more realistic than manually created meshes. This is done by breaking up a plane or prism into similar parts and raising or lowering the common vertices by a small amount, as shown in Figure 11. This process is repeated many times, creating a noise pattern that looks natural. Adjusting the way height perturbations are calculated changes the overall variation and height of the surface created. Some popular video games like Minecraft and No Man's Sky employ this strategy to generate their entire landscapes. Using fractal topology decreases the

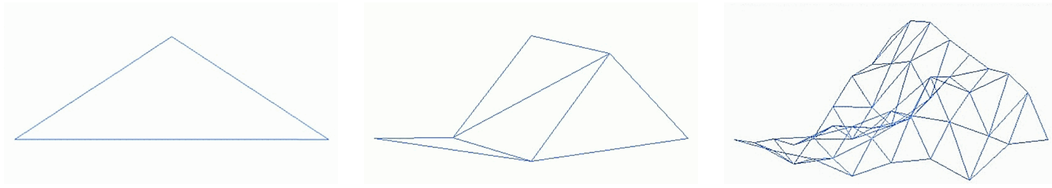

Figure 11: Topology can be generated using a fractal algorithm [18]

amount of space needed to store game world data to almost nothing. Only the altered parts of the landscape need to be saved, meaning that a save file can start at zero bytes, and will grow very slowly as the game progresses. Network load is also decreased because players are only sent packets containing the altered landscape. Games with maps of this size would not be possible without fractal topology.

Antennas are metal objects that collect electromagnetic waves in the air. A straight antenna occupies a lot of space but bending the antenna into a triangle or square occupies significantly less space while still offering the same amount of functionality. A key feature of fractals is that they have an infinite surface area, but a finite area. Fractal antennae exploit this phenomenon by maximizing the perimeter of the shape created when the antenna is bent while minimizing its area. Figure 12 shows some examples of fractal antennas. Due to the self-similar nature of the antennae they can also receive a wider band of frequencies. Cell phones use fractal antennas to increase power and decrease size.

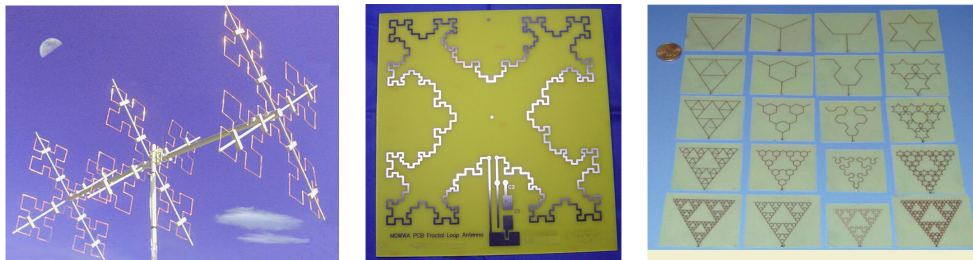

Figure 12: Antennas can capture a wider range of frequencies when shaped like fractals [19, 20, 21]

Ecologists studying trees and populations can take samples of data from individual trees and understand the way the entire forest works because the forest grows in a self-similar way to the trees it contains, and all tree branches are self-similar to the whole tree. They have extrapolated the overall resource consumption of the forest as well as its CO_2 absorption and oxygen output using this method.

The previous examples demonstrate the power of fractals and their ability to describe natural phenomena with very few tools. With a small sample of data scientists can extrapolate information about vastly larger self-similar systems. In addition to describing self-similar systems, their properties can also be exploited to improve the quality of technologies.

Fractal rendering software aids in developing these solutions and subsequently creates beautiful images that people seem to enjoy looking at. Chaos renders fractals abstractly, so it can be used to render any fractal. It can even render non-fractal geometry. Future iterations of the application will adopt a microkernel architecture, so that users can extend the platform themselves. This flexibility will increase its usefulness to professionals of any discipline that are trying to utilize the power of fractal mathematics.

3 Fractal Algorithms

Each algorithm implemented in Chaos will be described here in pseudo-code. For implementation specific code, please refer to the repository at `https://anchorwatchstudios.com/chaos/`.

3.1 Cantor Set

The Cantor set can be implemented recursively. A threshold minimum length is required as a base case to stop the recursion. This example illustrates the ternary set, but the Cantor set can be implemented in many ways by altering the segment length denominator and or number of Cantor function calls in the recursive case.

```
cantor(x, y, length, threshold) {

    drawLine(x, y, x + length, y);

    if(length > threshold) {

        segment = length / 3;
        cantor(x, y + 1, segment, threshold);
        cantor(x + segment * 2, y + 1, segment, threshold);

    }

}
```

3.2 Tree

A tree fractal can be implemented recursively. A threshold minimum length is required as a base case to stop the recursion. This example illustrates a two-dimensional, bi-directional tree, but a three-dimensional tree works the same way, and trees can branch more than twice in the recursive case. A common implementation is a tri-directional, two-dimensional tree. This variation can form a Sierpiński triangle with certain conditions. The step variable is a percentage greater than zero and less than one. The tree should start at an angle of $-angle$.

```
tree(length, angle, threshold, step) {

    rotate(angle);
    drawLine(0, -length);
    translate(0, -length);

    if(length > threshold) {

        tree(length * step, angle, threshold, step);
        tree(length * step, -angle, threshold, step);

    }

}
```

3.3 Julia & Juliabulb

The Julia set is traditionally implemented non-recursively. Each pixel in the image is passed into the Julia function as the complex number z. Each iteration either alters z to be smaller or larger and it is compared to the threshold. This algorithm works for higher dimensions as well.

```
julia(z, C, n, iterations, threshold) {

    result = 0;

    for(i=0;i<iterations;i++) {

        z = pow(z, n) + C;
        result++;

        if(length(z) > pow(threshold, 2))        {

            break;

        }

    }

    return result;

}
```

3.4 Mandelbrot & Mandelbulb

The Mandelbrot set is the same as the Julia except that each pixel in the image is passed into the function as the complex number C, and z is usually set to 0+0i. This algorithm works for higher dimensions as well.

```
mandelbrot(z, C, n, iterations, threshold) {

    result = 0;

    for(i=0;i<iterations;i++) {

        z = pow(z, n) + C;
        result++;

        if(length(z) > pow(threshold, 2))        {

            break;

        }

    }

```

```
    return result;

}
```

3.5 Newton-Basin

The Newton-Basin converges on the roots of a function by comparing the function to its derivative. The equation used in this example is: $z_{n+1} = \alpha \frac{z_n^p + C}{p z_n^{p-1}}$ where z is a complex point, α, and C are complex constants, and p is an integer.

```
newton(z, a, n, C, iterations, epsilon) {

    result = 0;

    for(i=0;i<iterations;i++) {

        numerator = a * (pow(z, n) + C);
        denominator = n * pow(z, n - 1);
        oldZ = z;
        z = numerator / denominator;
        result++;

        if(length(z - oldZ) < epsilon) {

            break;

        }

    }

    return result;

}
```

3.6 Mandelbox

The Mandelbox is a system that uses two functions. One function folds the point in space in a rectangular way, and is called a “box fold”, and the other folds the point in a spherical way, and it is called a “sphere fold”. A box fold is first performed and then a sphere fold at each iteration. If the point does not escape a threshold distance it is in the set. This algorithm works like the Juliabulb and Mandelbulb, except the distance estimation is calculated differently. The point being tested for is C.

```
mandelbox(z, C, fixedRadius, minRadius, foldLimit, scale,
                                        iterations, threshold) {

    result = 0;

    for(i=0;i<iterations;i++) {
```

```
        z = scale * sphereFold(boxFold(z, foldLimit),
                              fixedRadius, minRadius) + C;

        if(length(z) > pow(threshold, 2)) {

            break;

        }

        result++;

    }

    return result;

}

boxFold(z, foldLimit) {

    if(z > foldLimit) {

        z *= (2 * foldLimit) - z;

    }
    else if(z < -foldLimit) {

        z *= -(2 * foldLimit) - z;

    }

    return z;

}

sphereFold(z, fixedRadius, minRadius) {

    r2 = dot(z, z);

    if(r2 < minRadius) {

        z *= (fixedRadius / minRadius);

    }
    else if(r2 < fixedRadius) {

        Z *= (fixedRadius / r2);
```

```
    }

    return z;

}
```

4 Fractal Rendering

It is important to separate the process of rendering fractals from calculating them and shading them. Fractals are abstract concepts and there are many ways of rendering them. Abstraction of the rendering process is the single most important paradigm of Chaos. This abstraction allows for maximum flexibility in both creating new methods of rendering as well as adapting to newly discovered ones. The methods of rendering implemented in Chaos are described here.

4.1 Methods

Drawing The most basic rendering method and the default for Chaos is simple drawing. Fractals that use this method include the Cantor set and tree fractals. This method usually doesn't utilize the GPU, so it is one of the least efficient methods of rendering, as well as the most primitive. The program may translate, rotate, and scale the graphics context and draws lines, points, and polygons with a chosen color.

```
draw(context) {

    context.setColor(#000000);
    context.fillRect(0, 0, width, height);
    context.translate(140, 283);
    context.rotate(16);
    context.scale(4);
    context.setColor(#ff0000);
    context.drawLine(90, 45, 450, 900);
    ...

}
```

Complex Plot To plot a graph of the complex plane each pixel must represent a point on the plane. Each point is evaluated by the complex function to be graphed. The result of the function at that point is later used in shading. Each pixel must represent a point at some scale, and there are an infinite number of continuous points, so the scale must be defined. In other words, before the pixel is evaluated it must first be scaled from screen coordinates to graph coordinates. Then, the point is rotated. Finally, the point is translated. Doing these operations in this order is designed to cause the observer to feel like the camera is moving, not the graph. The rotations and translations happen relative to the graph or camera's current position, not the origin. Switching the order of rotation and translation will cause the observer to feel as though everything is moving and rotating relative to the origin.

The graph can be represented by two complex numbers and one decimal number; the minimum and maximum possible values, and the rotation. The same graph can be rendered at different resolutions, so it is necessary to supply one as a parameter as well.

```
complexPlot(pixel, resolution, min, max, rotation) {

    point = scale(pixel, resolution, min, max);
    point = rotate(point, rotation);
    point = translate(point, min, max);
    return f(point);
```

```
}

scale(p, resolution, min, max) {

    range = max - min;
    ratio = p / resolution;
    aspect = resolution.x / resolution.y;
    x = min.x + range.x * ratio.x * aspect;
    y = min.y + range.y * ratio.y;
    return vec2(x, y);

}

rotate(p, rotation) {

    rc = cos(rotation);
    rs = sin(rotation);
    x = p.x * rc - p.y * rs;
    y = p.x * rs + p.y * rc;
    return vec2(x, y);

}

translate(p, min, max) {

    range = max - min;
    origin = min + range / 2;
    x = origin.x + p.x;
    y = origin.y + p.y;
    return vec2(x, y);

}
```

Ray March Ray marching is a volumetric rendering method. A camera in three-dimensional space looks in a direction and a plane in front of the camera represents the view, as seen in figure 13. When a frame is drawn each pixel on the view plane has a ray sent through it from the camera origin. Each ray starts off very small and is advanced by an increment at each step. A signed distance function is used to detect collisions with geometry in the space [22, 16]. If the ray collides with a geometry the function returns the number of steps it took to reach the geometry, as well as various information about the marching process. The background color is drawn if the ray does not collide with any geometry after the maximum number of iterations.

```
rayMarch(position, resolution, camOrigin, camLookAt, camUp,
                                          iterations, threshold) {

    result = 0;
    position = scale(position, resolution);
```

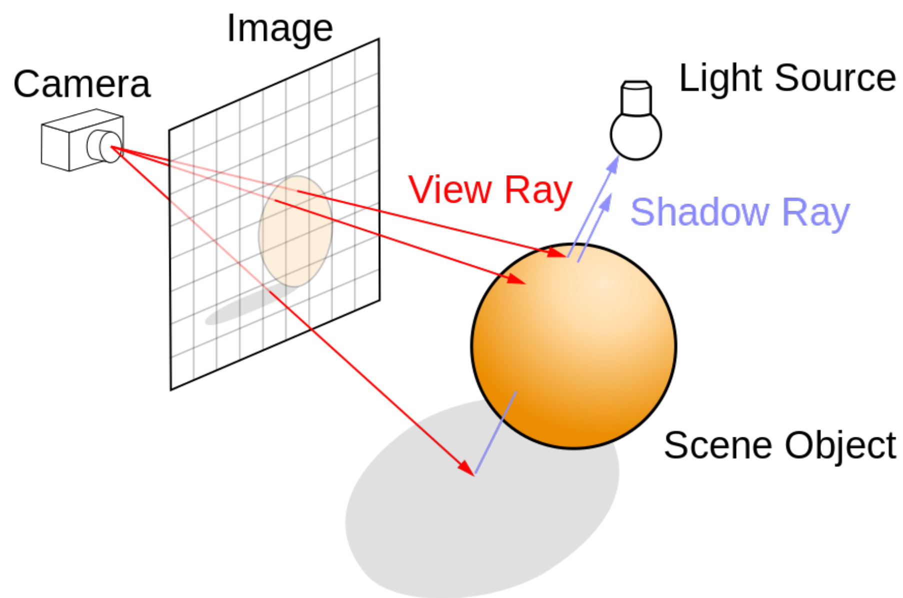

Figure 13: Rays are shot from the camera through the image plane [23]

```
    direction = getDirection(position, camOrigin, camLookAt, camUp);

    for(i=0;i<iterations;i++) {

        distance = f(position);

        if(distance < threshold) {

            break;

        }

        position += distance * direction;
        result++;

    }

    return result / iterations;

}

scale(position, resolution) {

    ratio = position / resolution;
    aspect = resolution.x / resolution.y;
    return vec2(ratio.x * aspect, ratio.y);
```

```
}

getDirection(position, camOrigin, camLookAt, camUp) {

    right = normalize(cross(lookAt, Up));
    lookAt = normalize(camLookAt);
    up = normalize(camUp);
    return normalize(right * position.x + up * position.y + lookAt);

}
```

4.2 Parallelism

Fractal algorithms are complex and take time to run due to their iterative nature, so parallelism should be used wherever possible. Exploring fractals without parallelism is almost impossible and always frustrating. All the escape time fractals are calculated on the GPU using SIMD parallel processing with OpenGL.

4.3 Video

In the absence of audio, a video is just a series of images or frames. Keyframes can be used to describe many frames of a video through interpolation. Chaos uses linear interpolation to export video from the keyframes contained in an open file. Each keyframe describes an entire fractal, shader, and render method. Each primitive variable within each construct is interpolated with a percentage based on the frames per second and frames per keyframe.

```
linearInterpolate(old, new, percent) {

    difference = new - old;
    return old + difference * percent;

}
```

5 Fractal Shading

Shading fractals is an arbitrary process. There are many variables that can be used to describe a given pixel's color. Choosing which ones and why is an art in and of itself. Shading is important because different coloring methods can offer different insight into the intrinsic properties of fractals. One method may highlight what happens when a certain value is changed, while another may highlight distance from the origin for example.

Shading should be abstracted away from the calculating and rendering processes, although variables that are a product of those processes can be useful, such as a ray's magnitude or a point's minimum distance from the origin. This complicates the shader's relationship to the renderer because there is a disparity between what values various renderers can return and what values various shaders expect to receive. The obvious solution would be to abstract what is returned from the renderer and use polymorphism to deal with the problem. However, GLSL doesn't support polymorphism, so somewhat arbitrary values must be passed in a standard order until low level languages like GLSL incorporate polymorphism. This is unlikely to ever change though due to the nature of the language.

5.1 Methods

Iterative The most common and intuitive way to shade a fractal is by counting the number of iterations it takes to break the threshold for each point and then dividing by the total number of iterations. This number is a percentage and is therefore quite convenient for multiplication purposes. When a color's components are multiplied with this number, lower values produce duller colors, and higher values produce brighter colors. The monochromatic, dichromatic, and minimum distance shaders included with Chaos all use a variation of this method.

There are many subtle ways in which this solution can vary. A boolean option would be to only color values that are ones or zeros. This creates a Rorschach inkblot like image. Another option would be to multiply by the iteration percent as described above. More complex variants use other variables, like the magnitude of the point being tested, in conjunction with iteration percentage to produce more exotic color schemes.

```
getFragment(fragCoord, color) {

    percent = f(fragCoord);
    return color * percent;

}
```

Random Random colors can be chosen for each fragment, creating a gaussian noise effect that looks like television static. This is rather simple when rendering on the CPU due to the availability of random number generators (RNGs), but when using the GPU there are no RNGs available, so care must be taken when selecting how each random color will be chosen. There are many ways to approach this problem. Here is one solution that offers a high degree of randomicity at the cost of execution time [24].

```
getFragment(fragCoord) {

    float a = 12.9898;
    float b = 78.233;
```

```
    float c = 43758.5453;
    float dt = dot(fragCoord.xy, vec2(a, b));
    float sn = mod(dt, 3.14);
    return fract(sin(sn) * c);

}
```

5.2 Lighting

Lighting is a component of shading, and it too can complicate the relationship the shader has to the renderer. Lighting is optional, but it can expose intrinsic properties of the subject that weren't immediately apparent without it. Light quality can benefit from variables produced from the rendering process as well. Chaos doesn't implement lighting, but future iterations will allow multiple lights to be added to a scene.

6 Implementation

Chaos was implemented in Java for Windows, Macintosh, and Linux, and is currently being implemented in Android. The system can be divided into two parts; The core and the interfaces. The core system components are designed to be portable from standard Java to the Android flavor. The interfaces are implemented in a non-portable way but use the same core components for things like the document structure and managing tasks.

6.1 Design Patterns

Command Pattern The command pattern, as seen in figure 14, was used to decouple the user interfaces from the core system. Actions and tasks are kept separate from core constructs, and the running program manipulates the constructs through these actions and tasks. Actions and tasks are coupled with the interface and are therefore not portable, but all core structures subsequently are. The advantages of using the command pattern far outweigh the disadvantages.

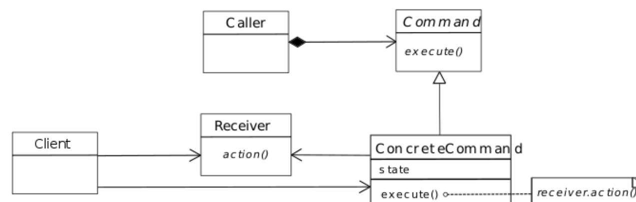

Figure 14: The Command Pattern separates commands from the classes that call them [25]

The command pattern decouples classes that invoke operations from the actual operations [26, 27]. This allows for executing the same command from separate places, like the toolbar and the menu. This also allows command queue systems, undo, and redo to be possible. Extending systems that implement the command pattern is easy because it can be done by simply writing a new command class.

The disadvantage of such a system is that there are a high number of classes and maintenance can be cumbersome[26, 27]. If the core system changes, actions that use any deprecated methods must be corrected.

Observer Pattern The observer pattern, as seen in Figure 15, was used to notify interface components of document changes. Observers register with subjects, asking to be notified if any changes are made. When changes are made to a document the document manager notifies all registered document manager observers, such as value sliders, status bars, menu bars, and the rendering canvas.

This pattern is useful because it streamlines event handling with a single registration for each observer. The observer pattern decouples the observer's actions from the subject and simplifies the very complicated process of keeping track of and notifying each observer.

The disadvantage of the observer pattern is that it can cause memory leaks if not implemented properly. If an observer is registered with a subject it cannot be collected for garbage. Also, feedback loops and race conditions are possible when observers call the subjects notify method. This is necessary when a value slider updates, for example. The slider must notify the manager that the document has changed, and that will in turn notify the slider to move to the correct position.

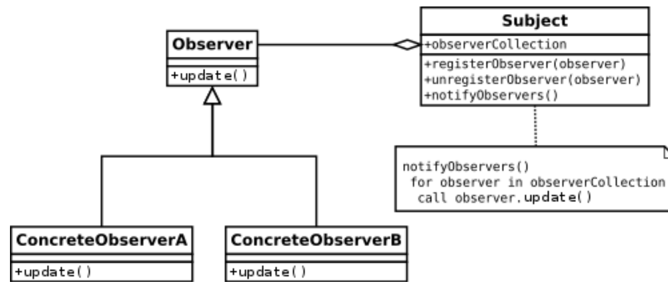

Figure 15: The Observer notifies the Subject when an event has occurred [28]

Chaos uses an observer-observer pattern to simplify handling multiple documents for multiple interface elements. The manager observes the open documents and keeps track of the selected document, and the interface elements observe the manager. This way multiple documents can be open and the interface elements only need to observe one document manager instead of observing all open documents.

Interface Pattern A simple interface pattern [29] was used to abstract and decouple the stages of the calculation, rendering, and shading processes on the GPU. The calculation, rendering, and shading interfaces are all well-defined, and shader program fragments can be concatenated to form complete shader programs. Abstract method definitions and method overloading were used to create an interface pattern. For example, a Julia set can be rendered by concatenating the complex plot, complex math, Julia, and color program fragments.

Using multiple files increases readability, reusability, and offers a kind of pseudo-encapsulation by keeping variables being passed to the shader at various stages in separate files. This approach is much easier to debug because processes are independent of each other and can be unit tested. The only drawback to using multiple files is having more files to keep track of.

6.2 Core

The core contains portable system elements that describe the document, document and task management, math, reflection, and common OpenGL interfaces. This section describes the details of the core system architecture.

Document The document model is abstract and allows for multiple file types to be defined. All documents contain a file and a boolean value that describes if the document has been edited. The document can register observers and notify them when a change has been made. A default file type of .frc2 is defined. The .frc2 file type contains global settings and a list of keyframes. Each keyframe contains a fractal, a render method, and a shader. Figure 18 in the appendix shows the relationships between the document model components.

A document manager contains documents, observes them, and keeps track of the current document index. Document manager observers such as interface elements then register with the document manager to vicariously observe changes to the current document. This helps to avoid memory leaks because each subject is responsible for registering and unregistering itself as an observer. The document manager, for example, registers itself as an observer to a document when it is opened, and unregisters itself when it is closed. Likewise, each interface element registers itself with the document manager when the program starts and is destroyed when the program completes.

The document parser is responsible for saving and loading documents to and from a file. This process is abstract and relies heavily on the file format. The default method implemented in Chaos uses Google's GSON library to convert Java data structures to JSON and vice versa.

Fractal Fractals are completely abstract at the lowest level. The only things all fractals have in common is that they can be interpolated, cloned, and have a description. An escape time interface extends the fractal interface by adding iterations, threshold, and a method to determine if a point falls in a set. This interface covers most popular fractals like the Mandelbrot, Julia, and Newton-Basin, as well as their higher dimensional counterparts. The escape time interface is parameterized with the type of point being tested, like complex or triplex. Simpler fractals like the tree and Cantor set implement the fractal interface directly. Each fractal contains variables that describe the fractal, getters and setters, and static defaults for those variables. Figure 19 in the appendix shows the relationships between fractals.

Render Method Render methods are also abstract, can be cloned and interpolated, and have a description. Render methods all have a method to return their available render implementations. Render implementation is defined as an enum, and each render method must supply a list of these enums with the get capabilities method. Each render method contains variables that describe the render method, getters and setters, and static defaults for those variables. Figure 20 in the appendix shows the relationships between the rendering model components.

Render Implementation A render implementation manager is a parameterized interface that keeps track of concrete rendering objects and pairs them with fractal classes. When the program needs to produce a rendering object it calls get implementation on the render implementation manager, supplying the fractal class as an argument. An object of the parameterized type capable of completing the rendering is returned if a match can be found, or null if none exists. A render implementation manager must exist for each render implementation enum.

Shader Shaders can be cloned, interpolated and have a description. Each shader must define a get fragment method which takes a four-dimensional vector, containing information about the fractal and rendering process, as a parameter and returns a color. Figure 21 in the appendix shows the relationships between the shader model components.

Color is represented as a class that holds four integers for red, green, blue, and alpha. The monochromatic, dichromatic, and trichromatic interfaces extend each other and define getters and setters for one, two, and three colors respectively.

Settings The settings class is cloneable and contains variables that apply to all keyframes. This structure contains the document resolution, current render implementation, and video settings like frames per second and frames per keyframe, as well as static defaults for these variables.

Math The math package contains structures for vectors and hypercomplex numbers, as well as a discrete number and interpolation interface. The interpolation interface provides static methods for interpolating primitives, as well as a parameterized interpolate method. The discrete number interface simply defines a getter and setter for a boolean value. Figure 22 in the appendix shows the relationships between the interfaces and classes in the math package.

OpenGL The GL package contains a common interface for OpenGL shaders. These shaders should not be confused with the shader contained within the document model. Shaders in OpenGL are programs that run on the GPU in parallel for each pixel. Implementations of this interface should provide a method to get and set an integer id, get a string array representing a program, get a string log, and compile the program.

Simple GL shader is an abstract class that implements the GL shader interface and provides convenience methods to load a program from an input stream or a string. All concrete implementations of GL shader in Chaos extend the simple GL shader class. Concrete implementations are not included in the core because all platforms do not recognize all OpenGL implementations.

Reflection The reflect package contains a single utility class for retrieving all classes in a package and getting their simple names. This class is useful for filling menus dynamically.

Task Tasks extend the runnable interface and represent a series of instructions to be run on a thread. Tasks must specify a pause and stop method, and they must provide a method to return the current progress. Tasks can register task observers and notify them when progress is made. Figure 23 in the appendix shows the relationships between components of the task model.

A task manager contains tasks, observes them, and keeps track of the current task. Task manager observers such as interface elements then register with the task manager to vicariously observe the progress of the current task. This helps to avoid memory leaks because each subject is responsible for registering and unregistering itself as an observer. The task manager, for example, registers itself as an observer to a task when it is added, and unregisters itself when it is complete. Likewise, each interface element registers itself with the task manager when the program starts and is destroyed when the program completes.

A priority task is a task which implements the priority interface. The priority interface extends comparable by providing a getter and setter for an integer priority. A priority queue task manager is an abstract class that uses a priority queue to select its next task. All concrete implementations of task manager except the single task manager and multi-task manager extend the priority queue task manager.

There are four concrete implementations of the task manager; the single task manager, the synchronous task manager, the asynchronous task manager, and the multi-task manager. The single task manager stops any running task to begin new tasks immediately. Only one task may run at a time. The synchronous task manager runs one task at a time but has a priority queue of waiting tasks that execute when the current task finishes or is cancelled. The asynchronous task manager runs a fixed number of tasks simultaneously and has a queue for waiting tasks. The multi-task manager is a convenience class containing the previous three implementations with delegate methods and constants to access them.

6.3 Swing

The desktop interface for Chaos was written using the Java Swing API in conjunction with the Java for OpenGL bindings (JOGL) and the Java AWT library. The application window contains several components used to manipulate the document, as shown in figure 16. All interface elements implement the document manager observer or task manager observer interfaces to facilitate updates.

Interface The menu and tool bars are implemented using a JMenuBar and JToolBar, respectively. The status bar shows progress and a message on a JPanel using a JProgressBar and a JLabel. The timeline displays JButtons on a JPanel that represent the keyframes in the current

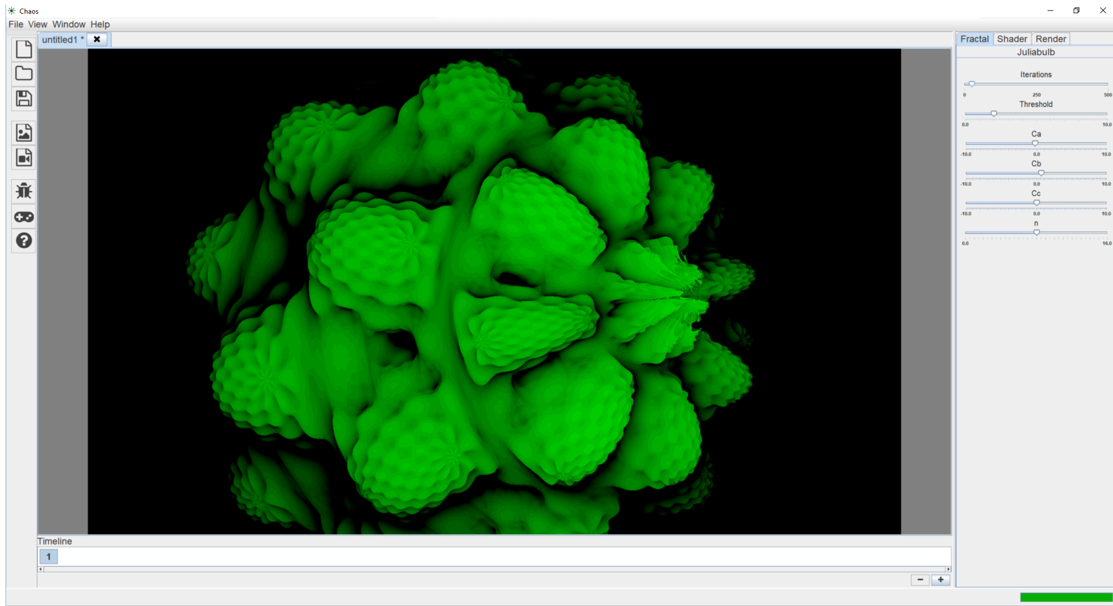

Figure 16: The PC interface, written in Java Swing

document. A JOptionPane is used to deliver popup messages to the user, and a JFileChooser is used to select files to save or open. FileFilters were used to prevent the selection of files with the wrong extension.

The workspace extends JTabbedPane and contains canvases that the fractals are rendered on. Clicking a tab causes the document manager to change the current document index to the selected tab index. Canvases are implementation specific, and a multi-canvas contains all other implementations in a CardLayout. When the document is updated the appropriate canvas is shown for the current render implementation.

The rollout also extends JTabbedPane. The rollout contains three tabs; fractal, shader, and render. Each tab contains a JPanel with a CardLayout. The CardLayout contains all editors for the tab category. When the document index is updated the CardLayout switches to the appropriate editor. Editors contain JSliders, JColorChoosers, and other various Swing elements that manipulate the document. A debug area that displays a description of the current document using a JTextArea is attached to the bottom of the rollout.

Event Chaos uses AWT actions to execute commands. The action class in Chaos contains one static sub-class that extends Action for each command. This makes it easier to remember what the names of commands are when using an IDE with code completion, and it reduces the number of files. It does however increase the size of the action class by a significant amount.

AWT key and mouse listeners collect input from the user. Key listeners and mouse listeners manipulate the render method to move the camera. Each render method has its own key and mouse listeners that are added to the multi-canvas.

Java 2D Rendering with the Java 2D Graphics API was implemented using tasks from the Chaos core. The Cantor, complex plot, and tree have corresponding tasks that extends a base

class. These tasks describe the steps required to render the fractal and display it using the Graphics API and a Java 2D canvas. Figure 24 in the appendix shows the relationships between components of the Java2D rendering model.

OpenGL 4.0 Chaos utilizes OpenGL 4.0 through the Java OpenGL Bindings library (JOGL). The base interfaces for abstract OpenGL shaders from the Chaos core are implemented to produce concrete programs and shaders that are capable of rendering fractals using an OpenGL 4.0 canvas. Figure 25 in the appendix shows the relationships between components of the OpenGL rendering model.

An image program renders the image using another program with the desired resolution to a frame buffer and then samples a resized image to the back buffer to be displayed. When a request to save an image is received, the canvas returns the image on the frame buffer. This additional step is needed so the image resolution is preserved when saving data.

Related rendering programs have intermediary ancestors that contain common variables and methods. Vertex buffer and array objects are represented in an object-oriented way to increase readability. Each program ancestor passes its own uniform variables to the shader at runtime, and classes that extend these ancestors override the method used to pass uniforms and add their own instructions to the process. For example, the render program specifies the resolution, the complex plot program specifies graph data, and the fractal programs specify information about fractals to the shader. This encapsulation of uniform shader variables increases readability and reusability. It also corresponds one-to-one with the encapsulation used in the GLSL files, making it even easier to understand. Figure 26 in the appendix shows the relationships between components of the OpenGL 4.0 shading model.

The OpenGL shaders used by the previously mentioned programs all extend the SimpleGLShader class from the Chaos core. Each shader loads a series of GLSL fragment files and concatenates them, then returns the resulting program as a string. Loop variables are replaced dynamically when the shaders are loaded because GLSL requires the use of hard-coded loop variables.

6.4 Android

Chaos is currently being implemented for Android. Creating a mobile version will increase usership and brand awareness because of the increased accessibility to a wider audience of people. The Chaos core system and OpenGL ES programs have been written, and an interface has been marked up using Balsamiq. The limited screen size for mobile applications means less space to place controls and descriptions, so the interface is currently undergoing testing to determine the correct ratio of functionality and ease-of-use.

Interface The desktop version of Chaos is geared more towards technical users and artists that want to produce high quality images and video or explore the mathematical extremes and behaviors of fractals. The mobile version will be more appropriate for non-technical users who are just discovering fractals for the first time. It will have responsive layouts for multiple screen sizes, so a tablet or phone can be used comfortably. Touch gestures will be used to modify values and move the camera instead of sliders and desktop controls. Mock-ups have been created, as shown in figure 17, and prototyping has begun.

The mobile version will have a virtual reality mode. VR mode will use stereoscopic cube maps that are pre-rendered (due to power consumption and the time cost of rendering). Sharing to social media will be easier with built in sharing features that mobile platforms offer. Google Chromecast will allow users to explore fractals on a larger screen and more easily share their experience with others.

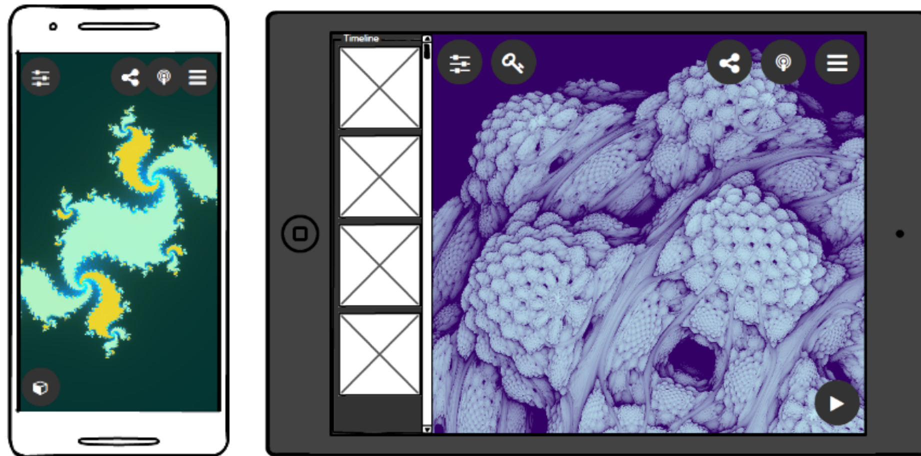

Figure 17: Mock-ups of the Android interface

OpenGL ES 2.0 Chaos utilizes OpenGL ES 2.0 through the Android bindings. The architecture of the OpenGL shader and program system is very similar to the OpenGL 4.0 implementation. Each level of program abstraction is responsible for passing its own uniform variables to the shader at runtime. Shaders extend the SimpleGLShader class from the Chaos core. 27 and 28 in the appendix shows the relationships between components of the OpenGLES 2.0 rendering and shading models, respectively.

7 Issues Encountered

There were several unexpected issues encountered that slowed the implementation process significantly. Some of these issues were solved after many hours of research and debugging, while others were not solved and still exist as bugs.

7.1 True Resolution

Originally there was no image program and programs drew directly to the back buffer. Each program was responsible for resizing the image it produced. This led to the realization that the issue of resizing could be abstracted away from the drawing process.

Including a frame buffer solves this problem. Drawing to the frame buffer and then sampling it allows the program to keep a full resolution image in memory while drawing to the screen at the desired resolution.

Learning about how to use a frame buffer was daunting and time-consuming because updated documentation and examples are scarce. Documentation and forums often refer to the fixed-function pipeline (FFP) used in OpenGL 2.0. This method of rendering has essentially been deprecated but its documentation somehow dominates the results of online searches.

7.2 Observer Feedback

When an observer is registered with a subject and it tells the subject to notify all observers it subsequently notifies itself. The user interface is designed to both update and be updated by changes made to a document. This causes observer feedback because when a slider value changes it must notify the document manager, which in turn notifies the observer. The values of the sliders are set, and the feedback loop continues.

To stop the feedback a boolean value was included in the observer that indicates whether the observer is currently listening for updates from the document manager. This value is set to false when a document manager must be notified and set to true when the notification is complete.

7.3 Document Serialization

Whenever a change is made to the document model all previously saved files will fail to load in the updated version. This is because the document and its components must be serializable, so GSON can determine what version of the data structures to use. Due to time restrictions serialization of the document model could not be included. Future iterations of the project will solve this problem.

7.4 Arbitrary Method Signatures

The GLSL shader language doesn't allow the use of polymorphic classes, so the renderer returns an arbitrary array of data to the shader. Not all shaders use all the information being passed. In the future a new shader might need more information than what is currently provided. There clearly needs to be a layer of abstraction here for the type returned from the renderer. Unfortunately, polymorphism isn't possible with GLSL, so the only solution was to be cautious about what was sent to the shader. This problem will be revisited in future iterations. GLSL allows structures and interface blocks, so perhaps they can be used to solve the problem.

7.5 Complex Plot Rotation

When the user clicks and drags the complex plot moves relative to the mouse. When the plot is rotated it must still move relative to the mouse, so rotation must be considered. When images are square the direction that the mouse moves can simply be rotated to translate the graph in the correct direction. When an image is not square the aspect ratio, $r = \frac{\text{width}}{\text{height}}$, must be considered.

A bug exists in the program that causes the plot to move asynchronously with the mouse if the image is not square and the plot is rotated. This is probably due to the order of operations when considering aspect ratio and rotation simultaneously.

7.6 Image Scaling

The image program renders another program to a frame buffer, then samples that image to draw a scaled version on the screen. The size of the drawing area determines the sampled image size and position. If the drawing areas aspect ratio is greater than the image aspect ratio then the height of the drawing area is used to calculate the sample image size and position. If the image aspect ratio is greater than the width of the drawing area is used. This behavior is the same as the background fit CSS style. The image is always centered and as large as it can be within the bounds of the drawing area.

The Java2D canvas successfully implements the background fit behavior, but it was never finished in the OpenGL implementation. Only the height is used to calculate the size of the sampled image with OpenGL.

The OpenGL background fit behavior was never finished because another problem arose during implementation that was never solved. It appears that the frame buffer overflows when the image is significantly larger than the drawing area, and as a result the image has missing parts. This is evidently due to the frame buffer because when the image is saved there are missing parts as well, proving that it isn't the sampling that is causing the problem. This shouldn't be a big issue because most people will be generating images roughly the size of their own displays or less. This means that a large display is needed to generate higher quality images though.

7.7 Timeline Visibility

Sometimes when a new fractal is created the timeline doesn't inflate properly and display the first frame button. When a new frame is added the timeline inflates. A user would never need to click the first frame button when there are no other frames, so this issue is very minor.

The timeline adds and removes buttons from a JPanel when updated, then repacks and revalidates itself, so this is almost certainly the problem. There are a few alias methods to pack and validate Swing interfaces. Future research will probably reveal that an inappropriate or deprecated method was used.

8 Future Updates

Several updates in future development cycles will fix existing bugs and introduce new features. A major microkernel architecture update is anticipated, as well as new fractals, shaders, and render methods.

8.1 Microkernel Architecture

In order to make a clean interface layer for extensions a kernel must be developed to load fractals, shaders, render methods, and render implementations dynamically and apply them when necessary. A base plugin class will be included that developers can extend to include their own fractals, shaders, render methods, and render implementations by extending and implementing the base classes and interfaces provided by the API. The kernel will load plugins or modules from a folder containing .jar files. A library of GUI components will come with the API to speed up the process of making controls. Moving to a microkernel architecture will decrease coupling, increase cohesion, and increase the overall system modularity and extensibility at the cost of increased complexity.

8.2 Fractals

Several new fractals will be added, and a few fractals will be updated. New fractals will include n-gons, apollonian gaskets, quaternion escape time variants, and multifractals.

The Cantor set will be made to look more interesting by coloring the space between lines, and a set of controls will be created to move around the set. The Cantor set will also be implemented in two and three dimensions, the Sierpiński carpet and the Menger sponge, respectively.

The tree fractal will be modified to have branch width, so the result will look more like a tree. A set of controls will also be created for the tree. The number of branches will be dynamic. This will allow a tree to form a Sierpiński triangle as a bonus. A three-dimensional variant will be implemented as well.

The updated Newton basin will feature a dynamic C value. Two-dimensional and three-dimensional variations of the Newton basin will also be explored.

8.3 Shaders

Shaders are arguably the most important part of Chaos and yet they were focused on the least due to time restrictions. There is a lot of room for improvement. The interface layer between the renderer and shader needs to be cleaner, new and interesting shaders must be implemented, and light needs to be incorporated to provide a sense of realism.

9 Conclusion

The document model should contain only portable abstract data types, so the document can be opened on multiple platforms and loaded with various implementations of the document parser. Changes to the document model should be serialized so different versions of the file type can be handled.

Fractals, renderers, and shaders should be abstracted from each other. Encapsulation should be used to hide the many variables used to describe these structures. Nothing should be assumed about any structure. Fractals shouldn't be expected to render a certain way, they are only a geometric construct. Likewise, a renderer should never choose the color of the pixels, it should only iterate through them. A shader should determine the color of the pixels.

GLSL shaders should be written in layers that complete one task each and abstractly reference each other through an interface pattern. Shader fragments increase readability and reusability through encapsulation.

Render method should be abstracted from the implementation because it allows the user to change the implementation at runtime. An interface should never be forced to use a certain render method.

Using a multi-level observer pattern allows multiple observations to occur vicariously. Multiple documents can be open by using this pattern, and many user interface elements can update automatically when the document is edited or changed.

The command pattern couples commands with implementation specific user interfaces and decouples commands from the system core. How the system is used is abstracted from what the system is intrinsically.

Chaos follows these principles to deliver maximum code portability, modularity, flexibility, and reusability when rendering hypercomplex fractals. It will become an open source project and hopefully a community of enthusiasts will embrace it. Chaos is a tool created to make beautiful artwork and accelerate education in fractal mathematics, calculation, rendering, and shading.

9.1 Special Thanks

I would like to thank my advisors and professors, Dr. Sally Hamouda, Dr. Namita Sarawagi, Dr. Robert Ravenscroft, and Dr. John Burke, for helping me develop the application and write the thesis. Their insights allowed me to re-frame and fix problems that I initially could not solve.

10 Bibliography

References

[1] T. Beddard, “Fractal lab,” http://sub.blue/fractal-lab, 2011.

[2] B. Mandelbrot, “How long is the coast of britain?” *Science*, vol. 156, pp. 636–638, 1967.

[3] B. A. Cipra, “A healthy heart is a fractal heart,” *SCIAM News*, vol. 36, no. 7, 2003.

[4] A. L. Goldberger, “Fractal electrodynamics of the heartbeat,” *Annals of the New York Academy of Sciences*, vol. 591, pp. 402–409, 1990.

[5] “Weierstrass curve,” https://en.wikipedia.org/wiki/Weierstrass_function, 2008.

[6] H. Trochet, “A history of fractal geometry,” http://www-groups.dcs.st-and.ac.uk/history/HistTopics/fractals.html, 2009, accessed on 1/25/18.

[7] “Plotting the cantor function,” https://tex.stackexchange.com/questions/241622/plotting-the-cantor-function, 2015.

[8] C. Shaver, “An exploration of the cantor set,” https://www.missouriwestern.edu/orgs/momaa/ChrisShaver-CantorSetPaper4.pdf, 2009, accessed on 1/27/18.

[9] H. V. Koch, “On a continuous curve without tangents, constructible from elementary geometry,” 1904.

[10] A. M. de Campos, “Koch snowflake,” https://en.wikipedia.org/wiki/Koch_snowflake, 2007.

[11] “Sierpiński carpet,” https://en.wikipedia.org/wiki/Sierpinski_carpet, 2014.

[12] “Sierpiński triangle,” https://en.wikipedia.org/wiki/Sierpinski_triangle, 2013.

[13] “Apollonian gasket,” https://en.wikipedia.org/wiki/Apollonian_gasket, 2008.

[14] A. Norton, “Generation and display of geometric fractals in 3d,” *ACM SIGGRAPH Computer Graphics*, vol. 16, no. 3, pp. 61–67, 1982.

[15] ——, “Julia sets in the quaternions,” *Computers & Graphics*, vol. 13, no. 2, pp. 267–278, 1989.

[16] T. Beddard, “4d quaternion julia set ray tracer,” http://2008.sub.blue/blog/2009/9/20/quaternion_julia.html, 2009, accessed on 2/9/18.

[17] “Quaternion julia,” https://commons.wikimedia.org/wiki/File:Quaternion_Julia_x%3D-0,75_y%3D-0,14.jpg, 2010.

[18] A. M. de Campos, “Fractal landscape,” https://en.wikipedia.org/wiki/Fractal_landscape, 2007.

[19] “Fractal antennas,” http://www.vk6fh.com/vk6fh/fractal.htm.

[20] J. J. de Oñate, “Fractal antenna experiment,” http://www.m0wwa.co.uk/page/M0WWA_fractal_antenna.html.

[21] "Sacred geometry: How cell phones work using fractals," http://www.theoracleslibrary.com/2015/05/19/sacred-geometry-how-cell-phones-work-using-fractals/, 2015.

[22] J. Wong, "Ray marching and signed distance functions," http://jamie-wong.com/2016/07/15/ray-marching-signed-distance-functions/, 2016, accessed on 2/25/18.

[23] "Ray tracing (graphics)," https://en.wikipedia.org/wiki/Ray_tracing_(graphics), 2008.

[24] Lumina, "Tutorials: Noise," http://web.archive.org/web/20080211204527/http://lumina.sourceforge.net/Tutorials/Noise.html, 2008, accessed on 3/9/18.

[25] "Command pattern," https://en.wikipedia.org/wiki/Command_pattern, 2017.

[26] C. Giridhar, *Learning Python Design Patterns: 2nd Edition*, 2016, ch. 7, sec. 5.

[27] Miafish, "Command pattern pros and cons," https://miafish.wordpress.com/2015/01/16/command-pattern-pros-and-cons/, 2015, accessed on 3/9/18.

[28] G. Bleiker, "Observer pattern," https://en.wikipedia.org/wiki/Observer_pattern, 2018.

[29] "Interface pattern," http://best-practice-software-engineering.ifs.tuwien.ac.at/patterns/interface.html, 2013, accessed on 3/9/18.

11 Appendix: UML Diagrams

Figure 18: The document model

Figure 19: The fractal model

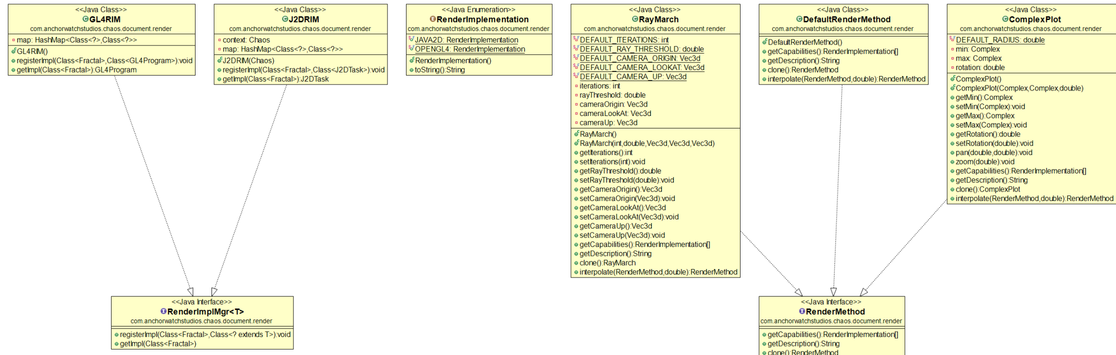

Figure 20: The rendering model

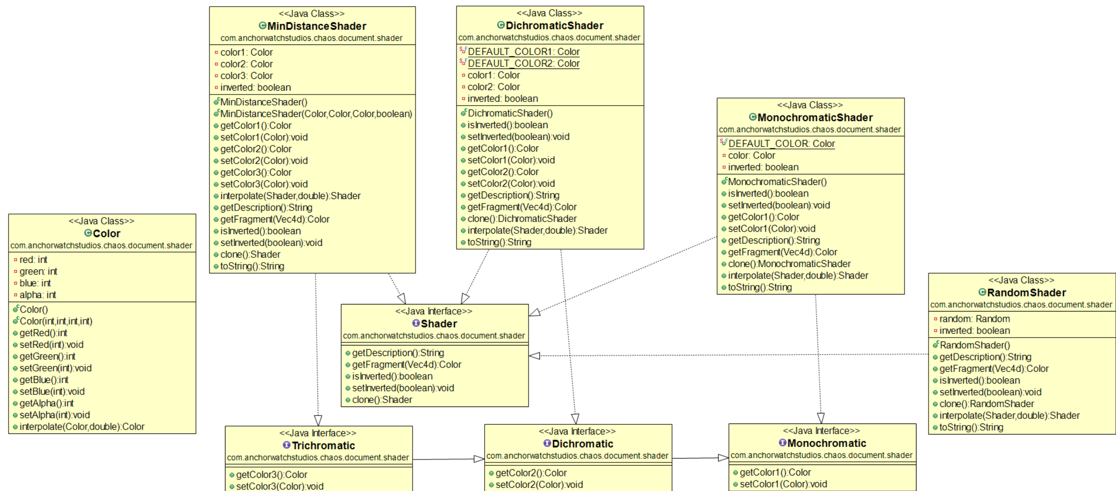

Figure 21: The shader model

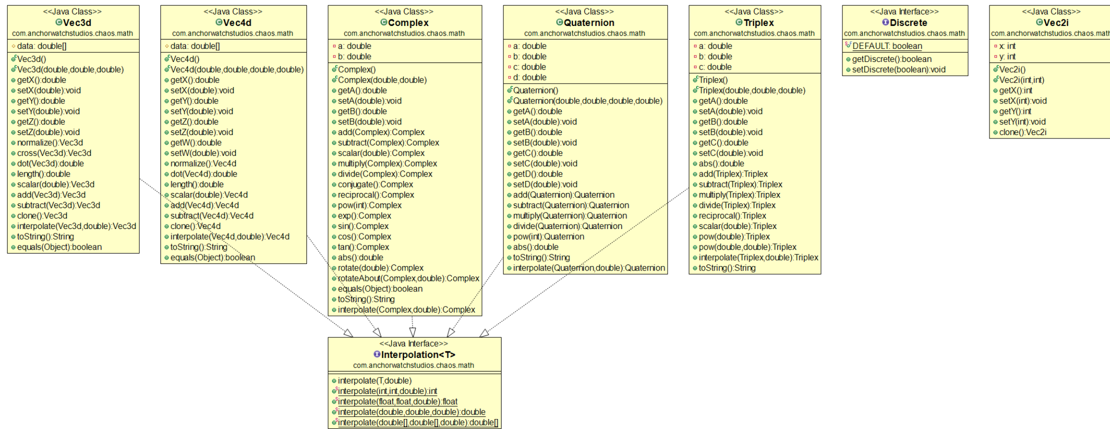

Figure 22: The math package

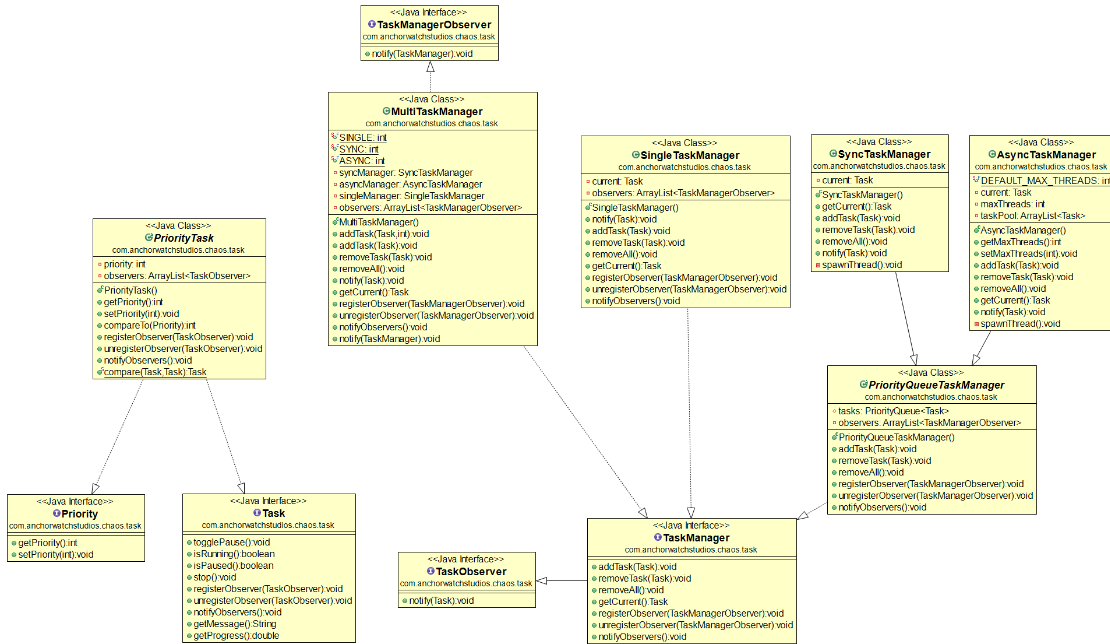

Figure 23: The task model

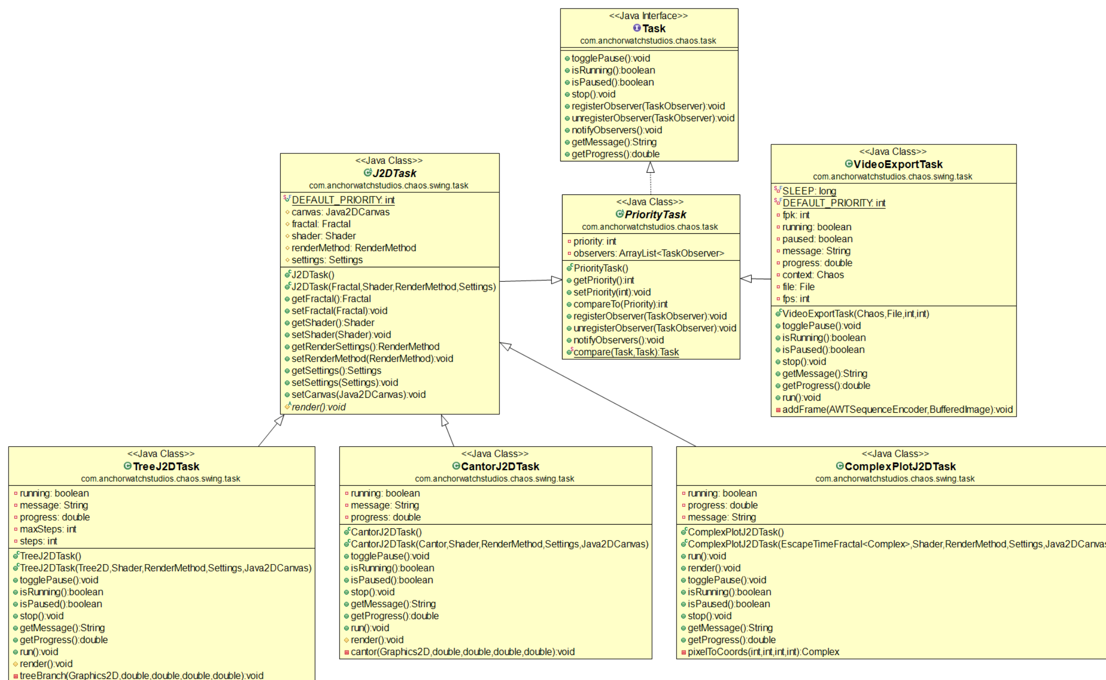

Figure 24: The Java2D rendering model

Figure 25: The OpenGL 4.0 rendering model

Figure 26: The OpenGL 4.0 shading model

Figure 27: The OpenGL ES rendering model

Figure 28: The OpenGL ES shading model