Rendering 3D Fractals

Zachary Shore
Clemson University, zaiyugi@gmail.com

Follow this and additional works at: https://tigerprints.clemson.edu/all_theses

Recommended Citation
Shore, Zachary, "Rendering 3D Fractals" (2018). All Theses. 2879.
https://tigerprints.clemson.edu/all_theses/2879

This Thesis is brought to you for free and open access by the Theses at TigerPrints. It has been accepted for inclusion in All Theses by an authorized administrator of TigerPrints. For more information, please contact kokeefe@clemson.edu.
Rendering 3D Fractals

A Thesis
Presented to
the Graduate School of
Clemson University

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

by
Zachary Shore
May 2018

Accepted by:
Dr. Jerry Tessendorf, Committee Chair
Tony Penna
Dr. Brian Malloy
Abstract

This thesis presents Cackle, a GPU accelerated path tracer for rendering 3D fractals. It is comprised of a GLSL viewer for real-time visualization and a series of CUDA kernels for offline rendering. Cackle supports both point lights and environment lights for lighting. A Cook-Torrance shading model was implemented for materials. The results are demonstrated with a series of renders and a discussion on artistic scene selection and lighting.
Acknowledgments

I would like to express my sincere gratitude to my advisor, Dr. Jerry Tessendorf, for his support and guidance throughout my time at Clemson. His efforts and humour helped make this project possible.

I would like to thank Dr. Brian Malloy for his programming expertise and excitement during the course of the project. I would also like to thank Tony Penna for his help with lighting design and for letting me work in the theatre while at Clemson. It was a good distraction from the long hours at my desk.

Many Digital Production Arts students offered their support during the project and I am grateful to them all. In particular, I would like to thank Thaddaeus Swasynger for his feedback on the fractal formulae and his help when coding the shaders, Cassidy Lamm for her emotional support, and Phillip Hatfield for his feedback on scene composition and lighting for the path-traced scenes.

Lastly, I would like to thank my parents, Roger and Stephanie Shore and my siblings, Isaac and Molly. Without their emotional support, guidance and creativity, none of this would have been possible.
Dedication

Dedicated to my parents and siblings, to my friends Cassidy and Thaddaeus, and to Tom, the spider who lives under my doorsill.
# Table of Contents

Title Page ................................................................. i
Abstract ................................................................. ii
Acknowledgments ......................................................... iii
Dedication ................................................................. iv
List of Figures ........................................................... vi

1 Introduction ............................................................. 1

2 Background ............................................................. 9
  2.1 Complex Dynamics .................................................. 9
  2.2 Escape Time Algorithms .......................................... 12
  2.3 3D Fractals ......................................................... 14
  2.4 Distance Estimation ................................................ 16

3 Cackle ................................................................. 20
  3.1 Sphere Tracing ..................................................... 21
  3.2 Path Tracing ....................................................... 24
  3.3 Scene Selection .................................................... 26
  3.4 Lighting ............................................................ 27

4 Final Thoughts and Discussion ....................................... 36

Appendices ............................................................... 39
  A Algorithms .......................................................... 40
  B Color ................................................................. 44
  C Additional Images .................................................. 47

References .............................................................. 52
List of Figures

1.1 Romanesco broccoli ................................................. 2
1.2 Discharge from a Tesla coil ........................................... 3
1.3 Frost on glass .......................................................... 4
1.4 Circle Limit I – M.C. Escher (1958) .................................. 5
1.5 Great Wave off Kanagawa – Hokusai (1830s) ..................... 5
1.6 Autumn Rhythm (Number 30) – Jackson Pollock (1950) .... 6
1.7 Trip through the multiverse – Doctor Strange (2016) ...... 6
1.8 Fractal city – Doctor Strange (2016) .............................. 7
1.9 Planet Interior – Guardians of the Galaxy Vol. 2 (2017) ... 7
1.10 Palace Interior – Guardians of the Galaxy Vol. 2 (2017) ... 8

2.1 Examples of Julia sets .................................................. 11
2.2 The Mandelbrot set ..................................................... 12

3.1 Sphere tracing two rays ............................................... 21
3.2 Sphere Tracing vs. Ray marching .................................... 22
3.3 The Mandelbrot set with orbit traps ............................ 24
3.4 Closeup of the effects of dither on a Mandelbox ............ 25
3.5 Coral Precipice .......................................................... 27
3.6 Harlequin Hall ............................................................ 28
3.7 The city Alpha – Valerian and the City of a Thousand Planets (2017) ........................................... 29
3.8 Refinery concept – Jupiter Ascending (2015) ............... 30
3.9 Blade Runner 2049: Black Out 2022 (2017) .................. 30
3.12 European Dead Zone – Destiny 2 (2017) .......................... 32
3.14 Hive Temple, concept – Destiny (2014) .......................... 33
3.15 World Caliber – winner of the Science as Art 2018 festival .............. 33
3.16 Blade Runner ........................................................... 34
3.17 Buried Pagodas ........................................................... 35

B.1 Color palette: Desert .................................................... 44
B.2 The Mandelbrot set with smooth iteration coloring ........ 45

C.1 Cactus Wood Caves .................................................... 47
C.2 The Lapis Roots .......................................................... 48
C.3 On Wings of Ice .......................................................... 48
C.4 Mountainous Depths .................................................... 49
C.5 Dwarfen Halls ............................................................. 49
C.6 Viral Foundations ........................................................ 50
C.7 Alles Lösen ............................................................... 50
Chapter 1

Introduction

The term \textit{fractal} was first coined by Benoît Mandelbrot in 1975. Widely considered to be the father of fractals, Mandelbrot used the term to describe shapes arising from his “theory of roughness”. He was interested in shapes that appeared universally “rough”: shapes that could be observed at any scale, but would never resolve into simpler shapes. This “roughness” can be seen in a number of natural phenomena, such as plants (Figure 1.1), electrical discharges (Figure 1.2), frost patterns (Figure 1.3), and the shape of coastlines [Man67]. Because of how prevalent this “roughness” appeared in nature, Mandelbrot sought to find a way to describe this property mathematically. He believed that fractals could be used as a more intuitive way of describing objects compared to the unnatural smoothness of Euclidean geometry, as described in his book, \textit{The Fractal Geometry of Nature}:

Why is geometry often described as “cold” and “dry”? One reason lies in its inability to describe the shape of a cloud, a mountain, a coastline, or a tree. Clouds are not spheres, mountains are not cones, coastlines are not circles, and bark is not smooth, nor does lightning travel in a straight line. [Man83]

Fractals can be used to describe or simulate naturally occurring objects. Part of the beauty of them is in how simply they can be defined. A geometric fractal like the Mandelbrot set is defined as the result of iterating upon the mathematical formula \(z = z^2 + c\). And yet, while it uses a fairly simple formula, the set contains truly infinite complexity. It is this balance of simplicity and complexity that makes the study and exploration of fractals so compelling.
In addition to their prevalence in nature, fractals can be found in art as well. Artists like M.C. Escher (Figure 1.4), Hokusai (Figure 1.5) and Pollock (Figure 1.6) have all been influenced by the structure and complexity of fractals. There has even been work done to verify the authenticity of Pollock’s works by comparing the fractal nature of his known works to those recently discovered [Sha15]. Fractal patterns have also been found in music. By analyzing the patterns of notes in Bach’s Cello Suite No. 3, a pattern similar to Cantor dust can be found [Orn14].

For VFX, fractals present somewhat of a challenge. While their formulae may be mathematically simple, evaluating them may be very computationally intensive. In addition, the shapes they define may not exist in a way that is easily represented elsewhere in the pipeline, like as a mesh or a grid of data. While they possess incredible complexity, this complexity is not very art-directable, further making them ill suited for production. However, there have been two recent films that both heavily featured examples of 3D fractals: Doctor Strange (2016) and Guardians of the Galaxy Vol. 2 (2017).

In Doctor Strange (2016), two examples stand out as being heavily fractal based. The first is during Stephen’s trip through the multiverse, where he falls into an evolving Mandelbulb fractal
Curiously, the actual study of fractals has only been around for the last 40 years or so, even considering their presence in art and nature. Their inclusion in films is even younger, with relatively few films utilizing fractal elements in their productions in the past 10 years. Much of the recent study in rendering fractals has been done by hobbyists building off of work from 20 years ago. The goal for this thesis is to explore two different techniques for rendering 3D fractals and how they can be applied in an artistic fashion.

Chapter 2 will discuss the foundations of the Julia and Mandelbrot set and how they can be extended to 3 dimensions. Chapter 3 will introduce Cackle, a path-tracer for generating ultra high resolution images using methods discussed in Chapter 2. Chapter 4 will wrap up with a discussion
of some problems encountered during the project and potential future directions.
Figure 1.4: Circle Limit I – M.C. Escher (1958)

Figure 1.5: Great Wave off Kanagawa – Hokusai (1830s)
Figure 1.6: Autumn Rhythm (Number 30) – Jackson Pollock (1950)

Figure 1.7: Trip through the multiverse – *Doctor Strange* (2016)
Figure 1.8: Fractal city – *Doctor Strange* (2016)

Figure 1.9: Planet Interior – *Guardians of the Galaxy Vol. 2* (2017)
Figure 1.10: Palace Interior – *Guardians of the Galaxy Vol. 2* (2017)
Chapter 2

Background

2.1 Complex Dynamics

Two of the most famous 2D fractals, the Julia set and the Mandelbrot set, both find their origins in the study of complex dynamics. This study first began with Pierre Fatou and Gaston Julia in the early twentieth century. However, after a brief period of about 15 years, the field quietly disappeared. The mathematicians Robert W. Brooks and Peter Matelski revitalized the study in 1978 when they first defined and drew what would come to be known as the Mandelbrot set. Adrien Douady and John H. Hubbard were the first to begin a serious mathematical study of the Mandelbrot set, and they named the set in honour of Benoît Mandelbrot, due to his contributions to the field.

Complex dynamics is the study of dynamical systems defined by the iterations of functions on complex number spaces. These function iterations take the form of iterated functions. A formal definition for an iterated function is as follows. Let $X$ be a set and $f : X \rightarrow X$ be a function. The $n$-th iterate of the function $f$, where $n$ is a non-negative integer, is defined as such:

\begin{align}
  f^0 &:= \text{id}_X \\
  f^{n+1} &:= f \circ f^n
\end{align}

where $\text{id}_X$ is the identity function on $X$ and $f \circ g$ denotes function composition. For complex dynamics, $X$ will be the set of complex numbers $\mathbb{C}$. An important idea related to iterated functions is that of an orbit. Given a $x_0 \in X$ and a function $f : X \rightarrow X$, the orbit of $x_0$ under $f$ is the
sequence of points: \(x_0, x_1 = f^1(x_0), x_2 = f^2(x_0), x_3 = f^3(x_0), \ldots, x_n = f^n(x_0)\). Orbits can be classified a number of different ways, but the two most important to this paper are bounded and unbounded. The orbit of \(x_0\) under \(f\) is bounded if there exists some \(K\) such that \(f^n(x_0) < K\) for all \(n\). Otherwise, the orbit is unbounded [Dev92].

For complex numbers, the definition of bounded orbits is changed to be the following: the orbit of \(z\) under \(f\) is bounded if there exists some \(K\) such that \(|f^n(z)| < K\) for all \(n\). Otherwise, the orbit is unbounded. Here, \(z\) is a complex number and \(|z|\) denotes its absolute value:

\[
\begin{align*}
z &= (x + iy) \\
|z| &= \sqrt{x^2 + y^2}
\end{align*}
\]  

Both the Julia set and Mandelbrot set are defined by the dynamical system of the class of quadratic polynomials [Dev92]:

\[
Q_c(z) = z^2 + c
\]  

where \(z\) and \(c\) are complex numbers. The filled Julia set for this system is defined by the following relation:

\[
J(Q_c) = \{z \in \mathbb{C}, |Q^n_c(z)| \not\to \infty\}
\]  

The filled Julia set is the set of all complex numbers whose orbits under \(Q_c\) remain bounded. The actual Julia set, also known as the chaotic set, is the boundary of the filled Julia set. In Figure 2.1a, the solid black region of the image is the filled Julia set. The border of this region is the Julia set.

Filled Julia sets can be classified as connected or disconnected. Visually, a connected Julia set is one solid, connected region (Figure 2.1a). A disconnected Julia set can be seen as infinitely fine dust (Figure 2.1b). A proof for connectedness is beyond the scope of this paper, but can be found in [Dev92]. In general, a filled Julia set \(J(Q_c)\) is connected if the orbit of 0 under \(Q_c\) is bounded.
With these definitions in place, the Mandelbrot set can now be defined as such:

\[ \mathcal{M} = \{ c \in \mathbb{C}, |Q^n_c(0)| \not\to \infty \} \]  

(2.7)
Thus, the Mandelbrot set is the set of all values for \( c \) where the filled Julia set of \( Q_c \) is connected. In this way, the Mandelbrot set can be thought of as an atlas of filled Julia sets (Figure 2.2) [Dev92].

![Figure 2.2: The Mandelbrot set](image)

### 2.2 Escape Time Algorithms

Both the Mandelbrot set and the Julia set can be rendered through the use of escape time algorithms. These algorithms track the orbit of a given number for a set number of iterations. If the value of the orbit exceeds a certain limit, then the value is considered to have “escaped” and the orbit is unbounded. If the value never escapes, then the orbit is considered bounded and the original number is treated as part of the set.

The escape limit is called the \textit{bailout}. The bailout radius can be used to control the level of detail present in the resulting image. Typically, the bailout radius will be set to 2, but increasing the bailout radius can allow for a higher amount of detail, because a larger number of orbits are treated as bounded.

For the Julia set, \( c \) is fixed to some user-defined 2D point. Each pixel of the image is mapped to the 2D complex plane and treated as the value \( z \). The algorithm then tracks the orbit of that particular pixel. The pseudocode for an algorithm like this is described in Algorithm 2.1.
Algorithm 2.1 Julia set escape time algorithm

Input: Two 2D points $z, c$ where $z, c \in \mathbb{R}^2$; An integer $n$, the maximum number of iterations

Output: The integer $i$ on which bailout occurred

1: function $\text{JULIA}(z, c, n)$
2:     $i \leftarrow 0$
3:     while $i < n$ do
4:         if $|z| \geq 2$ then
5:             break
6:         end if
7:         $w_x \leftarrow z_x \cdot z_x - z_y \cdot z_y$
8:         $w_y \leftarrow 2 \cdot z_x \cdot z_y$
9:         $z \leftarrow w + c$
10:        $i \leftarrow i + 1$
11:     end while
12:     return $i$
13: end function

The escape time algorithm for the Mandelbrot set takes a slightly different approach from that of the Julia set. Each pixel is still mapped to a value in the 2D complex plane. However, by the definition of the Mandelbrot set, for a complex number $c$ to be in the set, the filled Julia set of $Q_c$ must be connected. By definition, this means the orbit of 0 under $Q_c$ must be bounded. Thus, each pixel is treated as the value $c$, with $z$ now fixed to 0. The pseudocode for such algorithm can be found in algorithm 2.2.

Algorithm 2.2 Mandelbrot set escape time algorithm

Input: A 2D point $c$ where $c \in \mathbb{R}^2$; An integer $n$, the maximum number of iterations

Output: The integer $i$ on which bailout occurred

1: function $\text{MANDELBROT}(c, n)$
2:     $i \leftarrow 0$
3:     $z \leftarrow (0, 0)$
4:     while $i < n$ do
5:         if $|z| \geq 2$ then
6:             break
7:         end if
8:         $w_x \leftarrow z_x \cdot z_x - z_y \cdot z_y$
9:         $w_y \leftarrow 2 \cdot z_x \cdot z_y$
10:        $z \leftarrow w + c$
11:        $i \leftarrow i + 1$
12:     end while
13:     return $i$
14: end function
2.3 3D Fractals

Creating a canonical 3D analogue of the Mandelbrot set requires defining a 3D complex algebra. Due to Hurwitz’s Theorem [Wes09] on composition algebras, defining such an algebra is impossible. The only possible choices are the real numbers (1D), the complex numbers (2D), the quaternions (4D) or the octonions (8D). This does not mean that fractals cannot be defined in 3D, only that evaluation of the quadratic polynomial in Equation 2.5 in a canonical 3D complex space is impossible [Chr11b]. In spite of this limitation, a wide range of creative solutions have been discovered. This thesis will focus primarily on solutions using conformal transformations.

A conformal transformation, also known as a conformal mapping, is a function that locally preserves angles. Such a transformation will preserve the angle and shape of a tiny object, but not necessarily its size or curvature. According to Liouville’s Theorem [AEM98] on conformal mappings, any smooth conformal mapping in \( \mathbb{R}^n \) with \( n > 2 \) is either a translation, similarity, orthogonal transformation, or inversion, or is some composition of the previously listed transforms. This greatly restricts the transformations possible in \( \mathbb{R}^3 \). Within the context of fractals, non-conformal transformations are still possible. However, they tend to have the effect of stretching apart detail and reducing the zoom depth [Chr11a]. For this reason, only conformal transformations will be considered.

These conformal transformations can be used in a number of different ways. They can be used to construct an iterated function system (IFS) for generating fractals. If one of the transformations in the IFS is a fold or inversion transform, then the IFS is known as a Kaleidoscopic IFS (KIFS). A KIFS will display the self-similarity commonly associated with an IFS, but depending on the fold transform, it will also contain numerous reflections of itself at recursively deeper levels. Conformal transformations can also be added to an existing system to modify the resulting fractal in some way. Introducing even a small rotation or fold can cause radical changes in the system’s iterates. The primary fractal used in this thesis is a hybrid of the system defined by Equation 2.5 and a KIFS: the Mandelbox.
2.3.1 Mandelbox

The Mandelbox was first discovered by Tom Lowe in 2010 [Low10]. The Mandelbox formula replaces the equation \( Q_c(z) = z^2 + c \) with the following:

\[
v = s \cdot \text{sphereFold}(r_m, r_f, \text{boxFold}(f, v)) + c
\]  

(2.8)

The parameter \( s \) is the scale. The scale can be any nonzero real number, but as the scale exceeds \( \pm 3 \), the resulting Mandelbox tends towards infinitely fine dust. For scales \(-1 < s < 1\), the size of the Mandelbox scales off towards \( \infty \) as \( s \) approaches 0.

A boxFold(\( f, v \)) is a fold transform that takes elements outside of a box and reflects them back inside the box. The value \( f \) is a vector containing the dimensions of the box. The subscript \( i \) denotes the \( i \)-th component of a vector.

\begin{algorithm}
1: function boxFold(f, v)
2: [if \( v_i > f_i \) then]
3: \( v_i \leftarrow (2f_i - v_i) \)
4: [else if \( v_i < -f_i \) then]
5: \( v_i \leftarrow (-2f_i - v_i) \)
6: end if
7: end function
\end{algorithm}

Algorithm 2.3 can also be condensed to a single line using a component-wise \( \text{clamp}() \) function:

\[
v = 2 \cdot \text{clamp}(v, -f, f) - v
\]  

(2.9)

A sphereFold(\( r_m, r_f, v \)) is a sphere inversion transform. Points inside a sphere of radius \( r_f \) are reflected across the boundary of the sphere. Commonly, a second radius \( r_m \) is also defined. Points inside the sphere of radius \( r_m \) are scaled linearly away from the center.
Algorithm 2.4 Sphere Fold

1: function sphereFold($r_m, r_f, v$)
2:   $d \leftarrow |v|$  
3:   $t \leftarrow 1$
4:   if $d < r_m$ then
5:      $t \leftarrow \frac{r_f}{r_m}$
6:   else if $d < r_f$ then
7:      $t \leftarrow \frac{r_f}{d}$
8:   end if
9: $v \leftarrow t \cdot v$
10: end function

2.4 Distance Estimation

Rendering the Mandelbox or other 3D fractals using a traditional escape time algorithm like that used on the Mandelbrot (Algorithm 2.2) is not technically feasible. A significant part of the appeal of these objects is their infinite complexity and detail. To capture the Mandelbox at a level of detail similar to that of the 2D Mandelbrot set would require a very high resolution 3D grid. Storing such a grid would consume a prohibitively large amount of resources, in addition to the computation time required to calculate the orbit of every single grid point. For this reason, an alternative method is used to render 3D fractals.

A distance estimator, or simply DE, is a function $d(x)$ that given a point $x$ returns the distance from that point to the closest point on the surface of a shape. A DE is very similar to a signed distance field (SDF), but there are a few key differences. Unlike a SDF, a DE may or may not be signed. Furthermore, the structure of a DE for points exterior to a surface may differ from the structure of the DE for points interior to the same surface. This is especially so for 3D fractals. For fractals, their DEs will also tend to be an upper bound on the actual distance.

Given an implicit scalar field $f(x)$ with $f = 0$ on the surface, a distance estimation function $d(x)$ can be constructed using the following method. Let $x$ be a point close to the surface and $x + h$ be some point on the surface. So $f(x + h) = 0$ and $d(x) = |h|$. If $|h|$ were sufficiently small, $f(x + h)$ could be approximated in the following way:

$$0 \approx |f(x + h)| \approx |f(x) + \nabla f(x) \cdot h|$$

where $\nabla f(x)$ is the gradient of $f$ and $\cdot$ is the dot product. Using the triangle inequality and the
definition of a dot product, Equation 2.10 can be expanded to the form:

\[ 0 \geq |f(x)| - |\nabla f(x) \cdot h| \geq |f(x)| - |\nabla f(x)| \cdot |h| \]  
(2.11)

Thus, \(|h|\) can be approximated as the following:

\[ |h| \geq \frac{|f(x)|}{|\nabla f(x)|} \]  
(2.12)

Equation 2.12 gives an upper bound to the estimated distance from \(x\) to the 0-isosurface of \(f\). \(d(x)\) can now be written as such:

\[ d(x) = \frac{|f(x)|}{|\nabla f(x)|} \]  
(2.13)

This just leaves the question of how to create a scalar field for the Mandelbox. The only requirement for this scalar field is that \(f = 0\) at the surface of the Mandelbox. A simple solution is to use the expression:

\[ f(z) = (|z| - r_{min}) \]  
(2.14)

where \(z\) is the result of iteration of Equation 2.8. The term \(r_{min}\) is some value where the orbit is considered bounded if \(|z| < r_{min}\). This definition is similar to that of the bailout radius, but in practice, \(r_{min}\) will actually be the radius of the fractal’s bounding sphere. A DE for the Mandelbox can now be constructed as per Equation 2.13:

\[ d(z) = \frac{|(|z| - r_{min})|}{|\nabla f(z)|} \]  
(2.15)

In practice, constructing the DE this way is too costly, due to the \(|\nabla f(z)|\) term. Because of how the Mandelbox is defined, a central difference method is used to calculate \(\nabla f(z)\). This means that finding the value of \(d(z)\) requires 7 separate iterations of Equation 2.8. A simpler construction of the Mandelbox’s DE is as follows:

\[ d(z) = \frac{(|z| - r_{min})}{dr} \]  
(2.16)

where \(dr\) is a running scalar derivative that is added to the iteration of Equation 2.8. Pseudocode for this iteration can be seen in Algorithm 2.5.
Algorithm 2.5 Simple Mandelbox DE

Input: A point \( c \) where \( c \in \mathbb{R}^3 \); An integer \( n \), the maximum number of iterations

Output: The distance \( d \) to the fractal surface

1: function MANDELBOX\((c, n)\)
   2: \( a \leftarrow 0 \)
   3: \( z \leftarrow \langle 0, 0, 0 \rangle \)
   4: \( dr \leftarrow 1 \)
   5: \( \text{while } a < n \text{ and } |z| < 2 \text{ do} \)
   6: \( z \leftarrow 2 \cdot \text{clamp}(z, -f, f) - z \quad \triangleright \text{Box Fold} \)
   7: \( d \leftarrow |z| \)
   8: \( t \leftarrow 1 \)
   9: \( \text{if } d < r_m \text{ then} \)
   10: \( t \leftarrow \frac{r_f^2}{r_m^2} \quad \triangleright \text{Sphere Fold} \)
   11: \( \text{else if } d < r_f \text{ then} \)
   12: \( t \leftarrow \frac{r_f^2}{|z|^2} \)
   13: \( \text{end if} \)
   14: \( z \leftarrow z \cdot t \)
   15: \( dr \leftarrow dr \cdot t \)
   16: \( z \leftarrow z \cdot \text{Scale} + c \)
   17: \( dr \leftarrow dr \cdot |\text{Scale}| + 1 \)
   18: \( a \leftarrow a + 1 \)
   19: \( \text{end while} \)
   20: return \( \frac{(|z| - r_{min})}{dr} \)
   21: end function

Due to Equation 2.13, any DE constructed from a scalar field is only an upper bound on the distance. This can be exploited to introduce ways that the Mandelbox’s DE may be modified to refine the shape of the resulting surface. In this thesis, two major changes were made to the DE used by Cackle and Chortle.

First, in testing Algorithm 2.5, close zooms of the fractal would reveal sections of the surface that appeared bumpy, as if made of very tightly packed spheres. In Equation 2.16, the term \((|z| - r_{min})\) is very similar to the signed distance function for a sphere centered at the origin. In this way, this term can be thought of as a base shape that is seeded throughout the fractal. Based on this observation, this term was changed to use the signed distance function for a rounded box:

\[
\begin{align*}
    b_i &= \max(|x_i| - 2, 0) \\
    d(x) &= |b|
\end{align*}
\]  

(2.17)

The second change made to the DE was to take the absolute value of line 20 in Algorithm 2.5 before returning. Previously, the solid structures seen in test renders had no interior. If the camera
passed through them, renders would either be completely black or contain fuzzy blobs of color. With this change, all solid structures now have interiors, allowing for a greater depth of exploration.

The pseudocode for a complete Mandelbox DE with these changes along with DEs for other 3D fractals can be found in Appendix A.
Cackle consists of two main components: a real-time renderer called Chortle and an offline tiled renderer called Cackle. Cackle originally started off as just the real-time renderer. It was written in C++ using OpenGL with the actual renderer implemented entirely inside a GLSL fragment shader. However, while the GLSL version excelled at real-time performance, due to OpenGL’s inability to render without an active context, it was impossible to render anything offline. For this reason, an offline renderer was created using CUDA. The CUDA version can render scenes at a much higher resolution and visual fidelity than that of the GLSL version. The CUDA version uses Swig to generate Python bindings so renders can be easily scripted. Currently, Chortle is used to explore a fractal and discover new scenes. The parameters for a scene are output as a series of Python commands that Cackle can interpret to recreate it and then render offline.

Both Cackle and Chortle use a variation of the ray marching algorithm, a numerical approach to solving the rendering equation [TK11]:

\[
L(x_C, n_P) = \int_0^\infty C^T(x(s))\rho(x(s))\exp\left\{-\int_0^s \kappa \rho(x(s'))ds'\right\}ds
\]

(3.1)

\(L(x_C, n_P)\) is the amount of light received at position \(x_C\) along a direction \(n_P\) for some pixel \(P\). \(x(s)\) is some position on the ray path in the direction of \(n_P\) starting at the camera and moving out some distance \(s\):

\[
x(s) = x_C + sn_P
\]

(3.2)
where $s$ changes by some discrete step size $\Delta s$. $\rho$ is a scalar field representing the density per unit volume for any given point in space. The total color $C^T$ represents all light emitted at a given point in space, be it from external light sources or from the volume itself. $\kappa$ is the extinction coefficient [TK11]. Simply put, ray marching takes discrete steps along a ray, samples the density at each step and then accumulates the total transmissivity of light along that ray.

### 3.1 Sphere Tracing

Cackle and Chortle use a modified version of ray marching called sphere tracing. In sphere tracing, instead of using a discrete step size, $\Delta s$ is replaced with the value $r$. Given some point $x(s)$ along a ray, a sphere is centered at $x(s)$ and expanded until it touches a surface. The radius of this sphere corresponds to $r$ and acts as a bound on how far the tracer can step along the ray before $x(s)$ lands on a surface. The value $r$ can also be used as the tracer’s stopping criterion. When $|r|$ falls below some threshold $\epsilon$, the point $x(s)$ is “close enough” and the ray can be treated as having hit the surface [Har96]. Figure 3.1 shows the sphere tracing process for two rays, while Figure 3.2 shows the primary difference between sphere tracing and ray marching.

![Figure 3.1: Sphere tracing two rays](https://www.scratchapixel.com)
Figure 3.2: Sphere Tracing vs. Ray marching

The properties of distance fields make them a natural choice for \( r \). For a point \( x(s) \) and a distance field \( d \), \( d(x(s)) \) is the distance from that point to the closest surface. By sampling \( d \) along a ray, the tracer can take huge jumps each iteration and greatly augment the speed of rendering. Pseudocode for a sphere tracer with distance field \( d \) is given in the Algorithm 3.1.

**Algorithm 3.1 Simple Sphere Tracer**

<table>
<thead>
<tr>
<th>Input:</th>
<th>Point ( x_C ) and direction ( n_P ) where ( x_C, n_P \in \mathbb{R}^3 ); convergence threshold ( \epsilon ); max iterations ( n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>True/False indicating a hit</td>
</tr>
</tbody>
</table>

1: function \( \text{TRACE}(x_C, n_P, \epsilon, n) \) 
2: \( i \leftarrow 0 \) 
3: \( t \leftarrow 0 \) 
4: while \( i < n \) do 
5: \( p \leftarrow x_C + n_P \cdot t \) 
6: \( s \leftarrow d(p) \) 
7: \( t \leftarrow t + s \) 
8: if \( |s| < \epsilon \) then 
9: \( \text{return} \ True \) 
10: \( \text{end if} \) 
11: \( i \leftarrow i + 1 \) 
12: \( \text{end while} \) 
13: \( \text{return} \ False \) 
14: \( \text{end function} \)

Values \( x_C \) and \( n_P \) keep their definitions from Equation 3.2. \( \epsilon \) is the convergence threshold.
n indicates the maximum iterations before the ray is considered to have diverged and missed the surface.

Sphere tracing does come with a trade-off. Unlike traditional ray marching, sphere tracing is unable to accumulate density. It is primarily used for locating iso-surfaces inside distance fields. In return, sphere tracing can be much faster than traditional ray marching. Many rays are able to converge in relatively few iterations, particularly those who are closely aligned with the surface's normal. The worst case is for rays that either intersect or pass extremely close to edges. For these rays, they continue to take smaller and smaller steps as they approach the surface, with some never actually converging. There are ways to combat this downside. One possible solution is to increase $\epsilon$. This comes with the trade-off of decreasing the overall level of detail. Another approach is to increase the maximum iterations $n$, but this can cause render times to fluctuate.

Once the surface is located, surface properties can be inferred using the distance field $d$. The surface normal can be approximated by unitizing the gradient $\nabla d(x(s))$. For lighting, the visibility of a light from the shading point can be computed by using another sphere trace by starting at the shading point and heading towards the light. This can be repeated for each light in the scene to compute shadows.

For surface shading, standard shading techniques like Blinn-Phong or GGX can be used. Chortle uses an implementation of Blinn-Phong for shading and a single point light located directly above the camera for lighting. Cackle uses a Cook-Torrance shading model with a GGX distribution for specularity. This allows for the use of both dielectric and metallic materials. These materials can be specified by the user or controlled through the use of fractal statistics.

The actual color of the material can prove to be tricky. A uniform color can be used, but this tends to result in a rather Lambertian surface and is not very visually appealing. A more interesting approach is to use orbit traps. As a value is iterated upon, it moves throughout the space taking on a number of different values. This sequence of values is conceptualized as the orbit of that value. An orbit trap tries to analyze the behaviour of this orbit by looking at how the value at a particular iteration interacts with a primitive, like a point or line. By mapping the output of an orbit trap to a color map, various elements of a fractal's structure can be emphasized. Figure 3.3 is an example of a Mandelbrot set with two line traps and a point trap. A more in-depth discussion of orbit traps and color can be found in Appendix B.
3.2 Path Tracing

Sphere tracing can also be extended to work with path tracing. In path tracing, the goal is to use Monte Carlo methods to solve the rendering equation in [Kaj86]. It does this by integrating over all light received by a point on a surface and then using that surface’s reflectance function to determine how much light is passed back to the camera. So for every pixel, a path is generated by taking an outgoing ray and tracing it through the scene. When it intersects with a surface, the contribution from each light in the scene is accumulated, attenuated by the surface’s reflectance function and then added to the total light for that path. A reflected ray is then created by taking a random sample of the hemisphere around the normal of the hit point. This reflected ray is traced through the scene and the process is repeated for some set number of bounces. Multiple paths are generated for each pixel with the pixel’s final color determined by the average of each path’s total light.

This method results in very natural lighting. Unlike other rendering methods, path tracing will naturally include effects like ambient occlusion, soft shadows or depth of field. Unfortunately, an unmodified renderer can have significant visual artifacts if the number of paths per pixel is not sufficiently high. Increasing this number will cut down on the noise, but it will also incur a hefty
time penalty. Cackle makes two modifications to combat this.

First, when sampling the hemisphere to produce a reflected ray, the default method is to use a uniform distribution. However, by taking advantage of a surface’s reflectance function, this distribution can be altered so that it changes the convergence rate. For example, Lambertian materials use a cosine factor like the following as their reflectance function:

\[ r = \cos (\mathbf{n} \cdot \mathbf{w}_o) \]  

(3.3)

where \( \mathbf{n} \) is the surface normal and \( \mathbf{w}_o \) is the direction of the reflected ray. So rays close to the normal will have a larger effect on the resulting path than rays further from the normal. The sampling distribution can thus be biased so that samples closer to the normal are weighted more than samples further from the normal. This technique is called *importance sampling*. It changes the rate of convergence by biasing paths towards the important areas of surfaces’ reflectance functions.

The second modification is the addition of *dither*. Dither does not necessarily work to reduce artifacts, but rather tries to hide them by making them more visually appealing. For each ray, the ray’s direction is perturbed by some small amount based on the dither. Introducing noise here is actually useful, because it can diffuse otherwise very structured artifacts. Figures 3.4a and 3.4b show the difference between rendering with and without dither, respectively. Algorithm 3.2 shows a possible implementation of dither.

Figure 3.4: Closeup of the effects of dither on a Mandelbox

(a) 50% dither

(b) No dither
Algorithm 3.2 Basic Dither

1: function DITHER(\(w_0, d\))
2: \(w_0 \leftarrow w_0 \times (1 - d) + (\text{rand}(), \text{rand}()) \times d\)
3: return \(w_0\)
4: end function

3.3 Scene Selection

As discussed earlier, scene selection is first done using Chortle. Once a scene is found, the camera settings and fractal parameters are exported as a series of Python commands which Cackle can interpret to recreate the scene. The available parameters vary from fractal to fractal, but in general, the fractals used in this thesis use some combination of the following:

- **Fractal Iterations**: Number of iterations of the fractal formula
- **Scale**: Internal scaling of features. See Equation 2.8
- **FLimit**: Limits of the box fold
- **MR2**: Min. Radius (Squared). Used for sphere fold
- **FR2**: Fixed Radius (Squared). Used for sphere fold
- **Dither**: Amount of dither, as described above
- **Epsilon**: Surface intersection threshold; hit detected when \(|d(x(s))|\) is less than this value
- **DE Offset**: An offset to the final distance. Acts as a local feature scaling

When choosing scenes, the focus was on two particular aspects of fractals. First, because of how they recursively stack detail, fractals are very good at establishing a sense of grandness. When looking at a scene, some areas can be fairly concise. They may have high detail, but they will maintain a concise structure throughout. At the same time, other areas will take these same structures and repeatedly stack them on each other, twisting and packing them until they became impossible to make out. It is this range of scales that works to create a feeling of grandeur within a scene.

The second aspect relates to the forms found within a fractal. They can warp and fold shapes in ways that would be impossible to describe with regular Euclidean geometry. The chaotic structure of a fractal can lead to the rise of some very surprising shapes and greatly rewards exploration. Exploring a fractal is similar to watching clouds. Instead of looking for a particular shape,
exploration focuses more on how shapes can be interpreted. Some examples of such shapes would be the clockwork-like elements of *World Caliber* (Figure 3.15), the branching coral patterns of *Coral Precipice* (Figure 3.5) and the façades of *Harlequin Hall* (Figure 3.6). All of these scenes were found entirely through exploration.

![Figure 3.5: Coral Precipice](image)

### 3.4 Lighting

Two types of lighting are used in Cackle. The first type is a simple point light with variable falloff. The second is a skydome light driven by an HDR image. For skydome lights, after loading the image, the $n$ brightest pixels within the image are identified and a point light is created for each. $n$ may be set by the user; the default is 16. When accumulating light during rendering, these point lights are used instead of the skydome light. This was done to decrease noise and increase the rate of path convergence. If a reflected ray hits nothing, then the skydome light will be sampled as part of the background.

The colors for point lights were selected from a collection of Roscolux gels converted to RGB values. Scenes that did not make use of a skydome light used about two to three point lights, with one scene using five. This is not a limitation of the implementation. As stated above, using a
When lighting a scene, the goal was twofold: reinforce the scene’s sense of scale and establish a mood related to the scene’s shapes. Inspiration was drawn mainly from sci-fi and cyberpunk. In particular, urban landscapes, like those in Valerian (Figure 3.7), Jupiter Ascending (Figure 3.8), and Blade Runner 2049: Black Out 2022 (Figure 3.9), were referenced for establishing scale. A common thread in all three of these references is the glow. This glow pervades the setting; it filters up from below or from off in the distance. It establishes that not only is the city of grand scale, but that it is also alive and filled with activity.

The structure of several scenes lent itself well to underground or interior lighting, so a decision was made to focus mainly on that aspect. Two main sources of inspiration were the Remnant Vaults from the video game Mass Effect: Andromeda (Figures 3.10, 3.11) and environments from the video game franchise Destiny (Figures 3.12, 3.13, 3.14). Within the Remnant Vaults, the sickly green glow enhances the alien atmosphere, acting like the energy of the vault. The cool blues from above establish the feeling that this is cold, lost environment hidden deep below the surface. With Destiny, each figure represents a different feeling. In Figure 3.12, this feeling is coziness. The warm
glow of the fungi makes the scene feel close and sheltered, even with the stark white of the sunlight above. In Figure 3.13, the mood in the scene is one of caution or anxiety. The way that the light silhouettes the structures through the fog also creates this sense of something trying to escape. And in Figure 3.14, the lighting creates a feeling of secrecy. This can be seen in the shadows. All of the light is directed upward, casting long shadows up and along the walls. These shadows cast the upper reaches of the room into darkness and gives the scene its feeling of mystery.

To better describe the lighting process, three case studies were developed. Each discusses what fractal was used, initial impressions of the scene, lighting inspirations and the lights used to create the final look. The pieces chosen were World Caliber, Blade Runner, and Buried Pagodas.

Figure 3.7: The city Alpha – Valerian and the City of a Thousand Planets (2017)
Figure 3.8: Refinery concept – Jupiter Ascending (2015)

Figure 3.9: Blade Runner 2049: Black Out 2022 (2017)
Figure 3.10: Remnant Vault, Corridor – *Mass Effect: Andromeda* (2017)

Figure 3.11: Remnant Vault, Interior – *Mass Effect: Andromeda* (2017)
Figure 3.12: European Dead Zone – *Destiny 2* (2017)

Figure 3.13: The Iron Tomb – *Destiny: Rise of Iron* (2016)
3.4.1 Case Studies

Figure 3.15: World Caliber – winner of the Science as Art 2018 festival

Figure 3.15 is titled 'World Caliber'. The fractal used is that of the Pseudo-Kleinian Menger (PKM) fractal. The PKM is a hybrid fractal, in that it combines multiple different fractal formulae to achieve its final look. The PKM uses a Juliabox composed with a Menger sponge. On first glance, this scene looked like clockwork, as if it depicted an endless mass of gears and sprockets. This was the inspiration for the title: 'caliber' being another term for a watch’s movement or mechanism. This piece was the first to use the new Cook-Torrance shading model to create a metallic finish. To enhance the clockwork feel and to give the scene a bit of mystery, the center light was initially
placed much further back in the scene. This was intended to create long shadows and light the scene mostly with the glow of rim lighting. Unfortunately, placement proved a challenge and this idea was scrapped. When the light was brought forward, the interior surfaces started displaying a very beautiful iridescent effect. Two additional lights were placed to help the foreground pop out and to add a little contrast. The gels used are R17 Light Flame (Far left), R66 Cool Blue (Right), and R14 Medium Straw (Center).

Figure 3.16 is titled 'Blade Runner'. The fractal used is a negative scale Mandelbox with a high FLimit. This fractal is broken up into these deep canyons and shelves, one of which is where this scene is found. The inspiration for this piece was drawn from the megacities and nightlife of the cyberpunk genre, like those in *Ghost in the Shell* or *Blade Runner*. The lighting used is an skydome light with a sunset HDR image and a point light far below with the color R70 Nile Blue. The original plan was to use an HDR of an afternoon or morning sky, but it was changed to sunset to reflect a city transitioning to the nightlife setting. The blue glow is meant to represent the glow of vehicles and shops and to contrast with the reds of the setting sun.
Figure 3.17 is titled 'Buried Pagodas'. This particular scene was accidentally stumbled upon while testing a new variation on the Mandelbox formula. The scene uses a Scale 1.0 Mandelbox and is located above the central core of the fractal. The scene’s first impression was that the spheres and tower resembled Japanese temples or pagodas. The details warping up into the unknown made it feel like the structures were in a vast cavern. Keeping with that theme, a point light was placed below the scene with the color R18 Flame. This light creates the molten glow, like the structures on a shelf hanging over a lake of magma deep beneath. For general illumination, a point light with R09 Pale Amber Gold was placed just above the camera’s field of view. This color was chosen because of the natural incandescent color it adds to the scene.
Chapter 4

Final Thoughts and Discussion

In conclusion, this paper has introduced techniques for rendering 3D fractals with distance estimation. This work has also discussed ways in which these 3D fractals can be rendered as art pieces through scene and lighting selection. Algorithms for path tracing and rendering are included in Appendix A along with a number of additional distance estimators for different fractals. Appendix B covers a discussion on color and techniques for coloring fractals in visually complex ways.

Getting the resolutions needed for printing presented an interesting technical problem. During early development, Cackle would occasionally crash shortly after starting a new render. Upon further investigation, it was discovered that if a CUDA kernel causes the display driver to hang for more than a system defined amount of time, then the system would kill the offending kernel to free up resources. This is a setting defined at a kernel level by the system and thus incapable of being changed from a user level. To get around this, Cackle now breaks the image up into a series of smaller tiles. Each tile is rendered using CUDA and then written to an image buffer. Once all of the tiles are complete, the image buffer can be written out to a file. Tile size is required to be a multiple of the thread block size (typically 128) and care must be taken to ensure that the tiles are not so large that they cause the issue to arise again. This change allows for image resolutions up to the size of the available RAM on the machine, provided it is broken up into small enough tiles.

Once the image resolution exceeds around 8k×8k, it becomes prohibitively expensive to render it on a single machine. Images can take multiple days to render, and if the machine restarts during the process, all progress is lost. To address this, images are broken into a series of chunks, each of which can be rendered individually. Each chunk is then further broken into a series of
renderable tiles as described above. Once all the chunks have finished rendering, they are assembled into the final image using a Python script. This allows for an image to be distributed across a large number of machines, greatly decreasing render times and increasing fault tolerance. If a machine fails, only that chunk needs to be rerendered instead of the entire image. For this project, the DPA CheesyQ was leveraged for distribution.

There are a number of ways Cackle can be extended, both technically and artistically. A possible extension would be to add new types of lights. Currently, only point lights and environment lights are supported. However, the latter are approximated using point lights due to performance issues with directly sampling the environment map. Adding in new types of lights like spot or area lights would allow for greater customization of the scene. Lights could also be adjusted to use a color temperature instead of just pure white as the base light color. This could lead to a better approximation of gel color.

On a more technical note, currently Cackle and Chortle are wholly separate systems. After selecting a scene in Chortle, the Python commands to recreate that scene in Cackle are exported. After selection, these commands still have to be inserted into an appropriate Python script and then a lighting setup has to be established. This manual approach worked well enough during development, but was frustrating to use when selecting multiple scenes or multiple variants of the same scene. A better solution that was considered was to integrate Chortle’s export scene feature with the DPA CheesyQ system. With this method, scenes from Chortle could be directly exported to a Python script with a default lighting setup that could be run through the queue to generate a render through Cackle while the user continued to select scenes.

Another application of distance estimation that was explored was volume extraction. The idea is to use a bounding shape, such as a sphere, to extract shapes and features from a fractal by intersecting the bounding shape’s distance estimator with that of the fractal. The resulting intersection can be converted to a mesh using Marching Cubes and then remeshed either by hand or using some auto-remeshing tool. This method lets the user take some of a fractal’s complex shapes and use them elsewhere in a production setting.

This technique was implemented in Houdini using a Volume Wrangle and OpenVDB grids and while it worked reasonably well, it suffered from a few major drawbacks. First, as discussed in Section 2.4, because the fractal is now being evaluated across a volume, it is much more expensive to visualize than with sphere tracing. The level of detail in the fractal is tied to the grid resolution;
finer resolutions are sharper and capture more detail but require longer computational time. Second, the final mesh can lose these fine details if the remeshing operation does not use a high enough resolution. Third, auto-remeshing will typically only produce good results for volumes with large, distinct features. As the mesh becomes finer and details are sharper, the need for manual remeshing increases, which can be very time consuming on the user side. Overall, these problems compound to create a rather unfriendly user experience.

A potential direction going forward would be to return to this application and see what improvements could be made. There has been some success in using a particle based approach instead of a volume based one as discussed above. In particular, Animal Logic used such an approach for their work on *Guardians of the Galaxy Vol. 2*. Much of the alien flora, architecture, and planet interiors were done using fractals [ESHG17]. It would be interesting to compare the two approaches, as the particle based approach closely resembles the techniques discussed in this paper.
Appendices
Appendix A

Algorithms

**Algorithm A.1** Box Fold

1: function boxfold(v, f)
2:   if \( v_i > f_i \) then
3:     \( v_i \leftarrow (2f_i - v_i) \)
4:   else if \( v_i < -f_i \) then
5:     \( v_i \leftarrow (-2f_i - v_i) \)
6:   end if
7: end function

**Algorithm A.2** Sphere Fold

1: function spherefold(v, dr, \( r_m \), \( r_f \))
2:   \( d \leftarrow |v| \)
3:   \( t \leftarrow 1 \)
4:   if \( d < r_m \) then
5:     \( t \leftarrow \frac{r_m^2}{d^2} \)
6:   else if \( d < r_f \) then
7:     \( t \leftarrow \frac{r_f^2}{d^2} \)
8:   end if
9:   \( v \leftarrow t \cdot v \)
10:  \( dr \leftarrow dr \cdot t \)
11: end function
Algorithm A.3 Mandelbox DE

1: function MANDELBOX(c, f, r_m, r_f, Scale, T, n)
2:    a ← 0
3:    z ← ⟨0, 0, 0⟩
4:    dr ← 1
5:    T ← ⟨1 × 10^{20}, 1 × 10^{20}, 1 × 10^{20}⟩
6: while a < n and |z| < 100 do
7:    boxfold(z, f)
8:    spherefold(z, dr, r_m, r_f)
9:    z ← z · Scale + c
10:   dr ← dr · |Scale| + 1
11:   r^2 ← z · z
12:   T_i ← min(|z_i|, T_i), i ∈ {0, 1, 2} ▷ min across components
13:   T_w ← min(r^2, T_w)
14:   a ← a + 1
15: end while
16: b_i ← max(|z_i| − 2, 0), i ∈ {0, 1, 2} ▷ max across components
17: r ← |b|
18: return \[ \frac{|r − r_{\text{min}}|}{dr} \] 
19: end function

Algorithm A.3 has the following I/O:

- **Input**
  - \(c\): A point in \(\mathbb{R}^3\)
  - \(f\): A point in \(\mathbb{R}^3\); FLimit for box fold
  - \(r_m\): A value in \(\mathbb{R}\); Min. radius for sphere fold
  - \(r_f\): A value in \(\mathbb{R}\); Fixed radius for sphere fold
  - \(Scale\): A value in \(\mathbb{R}\)
  - \(T\): A point in \(\mathbb{R}^4\); stores the orbit trap results
  - \(n\): A nonzero integer, represents the maximum number of iterations

- **Output**
  - \(d\): The distance from the point \(c\) to the closest surface
Algorithm A.4 Menger DE

1: function Menger\( (z, H)\)
2: \( n \leftarrow 0 \)
3: \( s \leftarrow 3 \)
4: \( z \leftarrow (|z_0|, |z_1|, |z_2|) \)
5: if \( z_0 < z_1 \) then
6: \( \text{swap}(z_0, z_1) \)
7: end if
8: if \( z_0 < z_2 \) then
9: \( \text{swap}(z_0, z_2) \)
10: end if
11: if \( z_1 < z_2 \) then
12: \( \text{swap}(z_1, z_2) \)
13: end if
14: if \( z_2 < \frac{1}{3} \) then
15: \( z_2 \leftarrow z_2 - 2(z_2 - \frac{1}{3}) \)
16: end if
17: while \( n < 5 \) and \( (z \cdot z) < 100 \) do
18: \( z \leftarrow s \cdot (z - H) + H \)
19: \( z \leftarrow (|z_0|, |z_1|, |z_2|) \)
20: if \( z_0 < z_1 \) then
21: \( \text{swap}(z_0, z_1) \)
22: end if
23: if \( z_0 < z_2 \) then
24: \( \text{swap}(z_0, z_2) \)
25: end if
26: if \( z_1 < z_2 \) then
27: \( \text{swap}(z_1, z_2) \)
28: end if
29: if \( z_2 < \frac{1}{3} \) then
30: \( z_2 \leftarrow z_2 - 2(z_2 - \frac{H_2}{3}) \)
31: end if
32: \( n \leftarrow n + 1 \)
33: end while
34: return \( (z_0 - H_0) \cdot s^{-n} \)
35: end function

Algorithm A.4 has the following I/O:

- Input
  - \( z \): A point in \( \mathbb{R}^3 \)
  - \( H \): An offset to the position \( z \). Example: \( H \leftarrow (1.4898, 1.95918, 1.10202) \)

- Output
  - \( d \): The distance from the point \( z \) to the closest surface
Algorithm A.5 Pseudo-Kleinian Menger DE

1: function juliabox($z, c, f, Scale, T, n$)
2:   $a \leftarrow 0$
3:   $dr \leftarrow 1$
4:   $ds \leftarrow 0.00262$
5:   $H \leftarrow (0.55552, 0.48148, -0.1852)$
6:   $T \leftarrow (1 \times 10^{20}, 1 \times 10^{20}, 1 \times 10^{20})$
7:   $r_2 \leftarrow \max(|z_i|), i \in \{0, 1, 2\}$  
8:   while $a < n$ and $r_2 < 60$ do
9:       boxfold($z, f$)
10:      $r_2 \leftarrow z \cdot z$
11:      $k \leftarrow \max(Scale, r_2)$
12:      $z \leftarrow z \cdot k + c$
13:      $dr \leftarrow dr \cdot k$
14:      $r_2 \leftarrow \max(|z_i|), i \in \{0, 1, 2\}$  
15:      $T_i \leftarrow \min(|z_i|, T_i), i \in \{0, 1, 2\}$  
16:      $T_w \leftarrow \min(r_2, T_w)$
17:      $a \leftarrow a + 1$
18:   end while
19:   return $\frac{\text{menger}(z-H)}{dr} - ds$
20: end function

Algorithm A.5 has the following I/O:

- **Input**
  - $z$: A point in $\mathbb{R}^3$
  - $c$: A point in $\mathbb{R}^3$
  - $f$: A point in $\mathbb{R}^3$; FLimit for box fold
  - $Scale$: A value in $\mathbb{R}$
  - $T$: A point in $\mathbb{R}^4$; stores the orbit trap results
  - $n$: A nonzero integer, represents the maximum number of iterations

- **Output**
  - $d$: The distance from the point $z$ to the closest surface

Variables $H$ and $ds$ can also be user defined to further control the resulting shape of the fractal.

The values used here are for example purposes.
Appendix B

Color

Color can be used to enhance the structure or detail present within a fractal. Commonly, a mapping \( t \) is created using the data available from each iterate of a fractal and then used as input to a color palette or LUT. The examples in this thesis use the following cyclical color palette:

\[
\text{color}(t) = a + b \cdot \cos(2\pi[c \cdot t + d]) \tag{B.1}
\]

The values \( a, b, c, \) and \( d \) are vectors of three components \( \langle R, G, B \rangle \). To ensure that the palette cycles from \([0, 1]\), \( c \) is limited to an integer number of halves. The following is an example of one such palette, which has been christened “Desert”:

\[
\begin{align*}
\mathbf{a} &= \langle 0.5, 0.5, 0.5 \rangle \\
\mathbf{b} &= \langle 0.5, 0.5, 0.5 \rangle \\
\mathbf{c} &= \langle 1.0, 1.0, 1.0 \rangle \\
\mathbf{d} &= \langle 0.0, 0.1, 0.2 \rangle
\end{align*}
\tag{B.2}
\]

which results in the color palette seen in Figure B.1.

Figure B.1: Color palette: Desert
One of the simplest mappings is to color by iteration. The bailout iteration is mapped to a $[0, 1]$ range, like so:

$$t = \frac{i}{n}$$

where $n$ is the max number of iterations. Figure 2.2 uses this method. A more sophisticated method for coloring by iteration is to use smooth iteration coloring. For smooth iteration coloring, the goal is to define some fractional level between iterations to smooth away the boundaries between colors as seen in Figure 2.2. One way this can be defined is with the following:

$$t = \frac{\log(\log(|z_n|)) - \log(\log(B))}{\log(p)}$$

where $|z_n|$ is the magnitude of the final term of the fractal iteration, $B$ is the bailout radius, and $p$ is the power of the fractal formula (2 for the classic Mandelbrot and Julia set). Figures B.2, 2.1a and 2.1b all use this technique.

![Mandelbrot set with smooth iteration coloring](image)

Figure B.2: The Mandelbrot set with smooth iteration coloring

One of the most visually striking methods for coloring is to use orbit traps. In figure 3.3, the Mandelbrot set was rendered using two line orbit traps $D_x \leftarrow x = 0$ and $D_y \leftarrow y = 0$. Coloring
was done using the color map defined in equation (B.1), with $t$ defined by the equation:

$$t = (\max(D_x, D_y))^2$$  \hfill (B.5)

Algorithm B.1 shows a full pseudocode version of the Mandelbrot escape time algorithm with orbit traps necessary to generate an image similar to that of Figure 3.3.

**Algorithm B.1** Mandelbrot set escape time algorithm with orbit traps

**Input:** 2D point $c$ where $c \in \mathbb{R}^2$; An integer $n$, the maximum number of iterations

**Output:** The integer $i$ on which bailout occurred

1: function MANDELBROT($c, n$)
2:     $i \leftarrow 0$
3:     $z \leftarrow (0, 0)$
4:     $D \leftarrow (1 \times 10^{20}, 1 \times 10^{20}, 1 \times 10^{20})$
5:     while $i < n$ do
6:         if $|z| \geq 2$ then
7:             break
8:         end if
9:         $w_x \leftarrow z_x \cdot z_x - z_y \cdot z_y$
10:        $w_y \leftarrow 2 \cdot z_x \cdot z_y$
11:        $z \leftarrow w + c$
12:        $D_x \leftarrow \min(D_x, |z_x|)$
13:        $D_y \leftarrow \min(D_y, |z_y|)$
14:        $D_z \leftarrow \min(D_z, |z|)$
15:        $i \leftarrow i + 1$
16:     end while
17:     return $D$
18: end function
Appendix C

Additional Images

Figure C.1: Cactus Wood Caves
Figure C.2: The Lapis Roots

Figure C.3: On Wings of Ice
Figure C.6: Viral Foundations

Figure C.7: Alles Lösen
Figure C.8: Shoreline
Bibliography


