UNIVERSITAT DE BARCELONA

Treball final de grau

GRAUS DE MATEMÀTIQUES i ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

ITERATION OF HOLOMORPHIC FUNCTIONS AND VISUALISATION OF FRACTALS

Autor: Adrià Torralba Agell

Directores:	Dra. Núria Fagella Rabionet and
	Dra. Anna Puig Puig
Realitzat a:	Departament de Matemàtiques i Informàtica
Barcelona,	June 21, 2020

Abstract

When a holomorphic function is iterated, it generates a dynamical system on the complex plane. In this project we describe both the local and global theory of the different orbits of holomorphic functions, focusing on the polynomial families. We present the necessary results leading to two algorithms to draw both Julia sets and Mandelbrot (and Multibrot) set: the *Escape Algorithm* and *Henriksen Algortihm.* In addition, we present the development of an interactive application, made with Unity, that allows us to visualise fractals on the complex plane -rendered using the aforementioned algorithms- and a generalisation of them over a 3-dimensional space, directly on a website.

Resum

Quan una funció holomorfa és iterada, aquesta genera un sistema dinàmic al pla complex. En aquest projecte descrivim tant la teoria local com la global de les diferents òrbites de les funcions holomorfes, concentrant-nos especialment en les famílies polinòmiques al pla complex. A part, exposem els resultats necessaris per a justificar dos algorismes de renderització per a dibuixar tant conjunts de Julia com el conjunt de Mandelbrot (i Multibrot): l'*Algorisme d'Escapament* i l'*Algorisme de Henrisken.* A més a més, presentem el desenvolupament d'una aplicació interactiva, feta amb Unity, que permet tant la visualització de fractals al pla complex -renderitzats usant els algorismes esmentats- com una generalització d'aquests en un espai 3-dimensional, tot directament des d'una web.

Resumen

Cuando se itera una función holomorfa, ésta genera un sistema dinámico en el plano complejo. En este proyecto describimos tanto la teoría local como la global de las diferentes órbitas de las funciones holomorfas, concentrándonos especialmente en las familias de polinomios en el plano complejo. A parte, exponemos los resultados necesarios para justificar dos algoritmos de visualización para dibujar tanto conjuntos de Julia como el conjunto de Mandelbrot (y Multibrot): el *Algoritmo de Escape* y el *Algoritmo de Henriksen.* A de más, presentamos el desarrollo de una aplicación interactiva, hecha en Unity, que permite la visualización de fractales en el plano complejo -renderizados usando los algoritmos antes mencionados- así como una generalización de estos en un espacio 3-dimensional, todo directamente desde una web.

2020 Mathematics Subject Classification. 37F10, 97N80

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.

BENOÎT MANDELBROT

Acknowledgements

To my directors Anna Puig and Núria Fagella, for their absolute availability, patience, help and encouragement.

To my parents Montse and Francisco, my grandmother Gràcia, my aunt M^a Àngels and my brother Jordi because, without their support, none of this would have been possible.

To my friends Youwei, Àlex, Àngel, Ignasi and Jesús, for the funny moments studying and at class.

And to my life partner Sara, for her unconditional support.

Contents

Introduction

From the earliest times, all civilisations have had an interest in the superstition, the observation and the classification of the geometric figures that we can see in nature.

Most of the ancient civilisations were already fascinated by classical geometrical forms, such as triangles and circles, and they tried to provide a rational description of natural objects taking these classical forms as starting point.

Two of the most well known examples of this interest and knowledge about classical geometry are *The Elements*, a geometry treatise written by Euclid in the IV century b.C., or the famous welcoming sentence written in Plato's academy: "*Let no one ignorant of geometry enter*".

Nonetheless, nature often lacks these most wanted classical geometry properties, such as symmetry, proportion and harmony, making it impossible to describe the world only using these elementary shapes. In many cases, simplifications were tried and often they lead to error, like the assumption that celestial orbits were perfectly circular.

It was not until the XIX century when mathematicians like Weierstrass, Koch or Cantor explored objects whose properties were not explained by traditional geometry. These objects are known nowadays as *fractals*. However, due to a lack in computer visualisation, their works were very abstract.

In the XX century, Poincaré, Schröder, Fatou and Julia did their first incursions in the world of iteration in both the euclidean and the complex plane, giving birth to what we call *dynamical systems* nowadays. They realised that some of them behaved in a *chaotic* way, i.e., similar initial conditions produced very different behaviours when iterated.

In the complex setting, we can highlight the work of Fatou and Julia who used the recently developed theory of *normal families* in order to split the complex plane in two different regions with different dynamical behaviour. These sets are known today as the *Julia set* and *Fatou set*, also known as the *chaotic set* and the *stable set*, respectively.

Both Julia and Fatou were very fascinated by the *surprising* properties of these objects. However, both got rapidly stuck when they tried to visualise them, because of the lack of computing power to do so. However, we can find a first "draft" of the Julia set for the particular polynomial $P(z) = \frac{1}{2}(3z - z^3)$ that Julia draw in order to try to observe its *self-similarity* (See Figure 1, left) . Even with this lack of means, Julia managed to visualise a very early drawing of the Julia set for this polynomial, which we can compare with the right picture in Figure 1, the actual Julia.

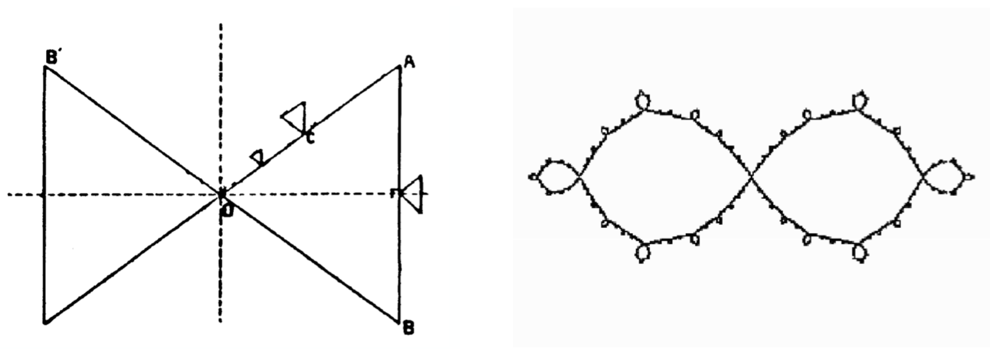


Figure 1: On the left, the hand drawn sketch of the Julia set for the polynomial $P(z) = \frac{1}{2}(3z - z^3)$, drawn by Julia. On the right the actual Julia set.

It was not until 1982 that D. Sullivan applied the theory of quasiconformal mappings in order to finalise the classification of *Fatou components* or *connected components of the Fatou set*. This fact, in combination with the advances in technology regarding processors, both *central processor unit (CPU)* and *graphical processor unit (GPU)*, provided the world with the capacity of doing millions of numerical operations per second, promoted the renaissance of this topic during the 80's. Thanks to this boost in computer performance, one was enabled to draw these sets in seconds, which allowed mathematicians to make conjectures and open new problems.

Nonetheless, these objects -far from being only a mathematical curiosity- kept appearing when studying natural phenomena and the universe that surrounds us, such as snow flakes or thunderbolts. These observations opened the eyes to a brand new world of possibilities.

Some years later, the studies from Brooks and Matelski about Kleinian groups[1] provided the first computer image of a well known fractal: the *Mandelbrot set* (See Figure 2 [39], left). Which we can compare with the right picture in Figure 2, a modern image of the Mandelbrot set.

Regarding this computer explosion, the summit has not been reached even today. Every day new visualisation techniques keep appearing that enhance the previous existing ones.

[1]See their work on [1].

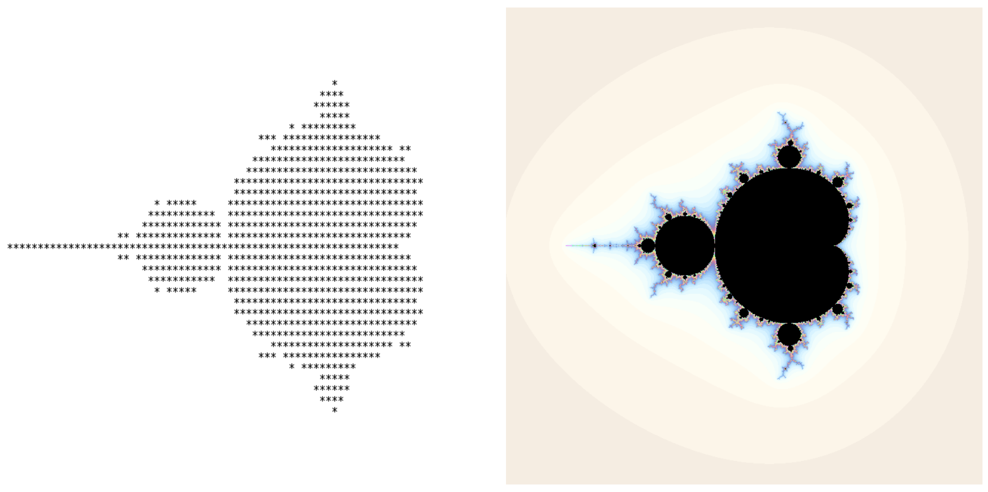


Figure 2: On the left we can see the first picture of the Manelbrot set made by Brooks and Matelski. On the right we can see a modern image of the Mandelbrot set.

Main goal

The goal of this project is twofold.

On the one hand, we aim to present an introduction to iteration of complex polynomials. Our guiding thread is to justify with mathematical rigour, the *rendering algorithms* that we aim to implement in the second part of this document to draw images of the complex fractals both in the dynamical and the parameter planes.

On the other hand, equipped with the results on the first part, we design and develop an application to visualise fractal objects that are the result of polynomial iteration. A generalisation of these sets on a 3-dimensional space is also presented as a rendering visualisation exercise.

The application we present here aims to be a practical tool with plenty of features to explore the world of complex polynomial dynamics. In particular, it has been designed and thought to be used by the students of *Models Matemàtics i Sistemes Dinàmics* (Mathematical Models and Dynamical Systems), a second year course in the Mathematics Bachelor's degree at Universitat de Barcelona.

Specific goals

In order to accomplish the above mentioned goal, we have subdivided it into the following specific achievements.

1. Give general concepts about dynamical systems and present the most important results about local theory of iteration of holomorphic functions.
2. Give the definitions of Fatou and Julia sets, and a compendium of their properties, for an arbitrary holomorphic function. In addition, we aim to show the basic results about classification of Fatou components.
3. Make a brief study of the dynamical plane of the polynomial family.
4. Present the parameter space with its basic dichotomy and main properties.
5. Analysis of the software requirements of the web application.
6. Analysis of the related work in 2D and 3D fractals. Optimisations of 3D visualisation using GPUs.
7. Design and development of a web application to render fractals.
8. Comparison of performance and accuracy of CPU and GPU based implementations.

Structure of the Document

This project is divided in two parts: the mathematical formalisation of the iteration of holomorphic functions and the development of a web application to render fractals.

Thus, in Part I, we present a background on iteration of holomorphic functions focusing on polynomials and with the main goal to state and prove the results that support the algorithms we implement on the software project in Part II. This first part is structured as follows:

In the 1st chapter, we make a brief introduction to general concepts about dynamical systems, we explain the most important results about local theory as well as others that will help us in order to obtain global results.

The 2nd chapter presents the definition of Julia and Fatou sets for a holomorphic function in general. It also contains a compendium of important properties of those sets that will help us to prove the algorithms on the nexts chapters. In this chapter we find the first algorithm to draw Julia sets called *Henriksen Algorithm*. In addition, we study the components of Fatou set, concluding with the statement of the theorem of classification of Fatou components, an important result from the 80's.

The 3th chapter is focused on the polynomial family. Here we present the particular definition of Julia and Fatou sets for polynomials as well as the escape criteria in which relies the *Escape Algorithm* for Julia sets.

Chapter 4 contains the basic dichotomy for the Mandelbrot and Multibrot sets, providing also the Escape criteria that leads to the *Escape Algorithm* for Mandelbrot and Multibrot sets, as well as another approach to these sets in which we obtain the *Henriksen Algorithm* for Mandelbrot and Multibrot sets.

Part II of this project consists on the development process of a software that draws fractals resulting from iteration of unicritical polynomials in $\mathbb{C}$, as well as a generalisation to 3-dimensional space.

In chapter 5, we introduce the basic notions about the visualisation pipeline when rendering images through a GPU.

In chapter 6, we expose the definition of the problem. This includes an analysis of requirements, both functional and non-functional, as well as a brief state-of-the-art about fractal visualisation and finally, a comparison between existing software that render fractals and shows its basic properties.

Chapter 7 contains our Proposal to solve the Problem definition defined on the previous chapter.

In chapter 8, we show the functionality of our developed application as well as a detailed benchmark in terms of rendering time and level of detail.

In addition to this, we provide the detailed Class Diagrams of the software project on Appendix A, many additional screenshots of the application on Appendix B and the minimum requirements in order to run this application on Appendix C.

Part I

Iteration of Holomorphic Functions

Chapter 1

General Concepts and Local Theory

The reader can find the references for this chapter in [2, Chapter 8], [3, Chapter 1], and [4, Part III].

Although dynamical systems can be studied in a very general setting (continuous maps on metric spaces), in this project we choose to restrict to those systems generated by the iteration of holomorphic maps on the complex plane.

We present the basic definitions and results that we will use in the following chapters. We start by recalling the definition of a holomorphic function. If $V \subset \mathbb{C}$ is an open set of complex numbers, a function $f\colon V \to \mathbb{C}$ is called *holomorphic (or "complex analytic")* if the first derivative

$$z \mapsto f'(z) = \lim_{h\to 0} \frac{f(z+h) - f(z)}{h}$$

is defined as a function from V to $\mathbb{C}$ or, equivalently if f has a power series expansion about any point $z_0 \in V$, which converges to f in some neighbourhood of z_0.

We recall that holomorphic functions are open maps.

Occasionally, we will be interested in studying a whole family of holomorphic functions. So, given an open set $\mathcal{U} \subset \mathbb{C}$, we denote the *family of holomorphic functions* from $\mathcal{U}$ to $\mathbb{C}$, this is $f\colon \mathcal{U} \to \mathbb{C}$ as

$$\mathcal{H}(\mathcal{U}) = \{f\colon \mathcal{U} \to \mathbb{C} \ \mid \ f \text{ holomorphic}\}.$$

In this text, we use the topology given by the uniform convergence on compact sets as follows.

Definition 1.1 (**Uniform convergence on compact sets**). A sequence of holomorphic functions $\{f_n\}_{n\in\mathbb{N}}$ of maps defined on $\mathbb{C}$ *converges uniformly on compact sets* to f if

$$\forall \text{ compact } K \subset \mathbb{C} \ \ \forall \epsilon > 0, \ \ \exists N = N(K,\epsilon) : |f_n(z) - f(z)| < \epsilon, \ \ \forall z \in K, \ \text{ if } n \geq N.$$

1.1 Basic concepts in dynamics

In this section, we run through the most basic definitions, observations and results about dynamical systems.

Definition 1.2 (Trajectory or Orbit. Seed). If $V \subset \mathbb{C}$ is an open set, given $f\colon V \to \mathbb{C}$ a holomorphic function and $z_0 \in V$, the sequence defined by the iterates (of f at z_0) is known as the *trajectory* or the *orbit* of z_0. This is usually noted as

$$\{z_n := f^n(z_0)\}_{n\geq 0}.$$

The point z_0 is called the *seed* or *initial condition.*

It is important to remember that, formally, this last definition defines the *positive orbit* of z_0, since when you "iterate backwards" you usually obtain more than one pre-image that defines what is called the *negative orbit of* z_0.

Henceforth, when we will use *orbit of* f *at* z_0 to refer to the *positive orbit of* z_0.

Definition 1.3 (Periodic Point. Fixed Point). We say that z_0 is a *periodic point* of *period* $p \geq 1$ if $f^p(z_0) = z_0$ and $f^n(z_0) \neq z_0$ $\forall n < p$. If $p = 1$, we call z_0 a *fixed point.*

Definition 1.4 (Pre-periodic Point). We say that z_0 is *pre-periodic* if z_0 is not periodic, but $f^k(z_0)$ is periodic for some $k \geq 1$.

As a first observation, if we have a fixed point of f^p, this will be either a fixed point of f or a point of period d of f for some divisor d of p.

Definition 1.5 (Periodic Orbit or Cycle). If z_0 is a periodic point, the sequence $\{z_0, z_1, \ldots, z_{p-1}, z_0, z_1, \ldots\}$ is called *periodic orbit* or *cycle.* We usually denote it as $\mathcal{O} = \{\overline{z_0 z_1 \ldots z_{p-1}}\}$.

Usually we are interested on how a certain set evolves with time. In particular, we are interested on those sets that remain *invariant* in time.

Definition 1.6 (Positive Invariant Set. Negative Invariant Set. Invariant Set). A set D is *positive* (resp. *negative*) *invariant* of $\{f_n\}_{n\in\mathbb{N}}$ if the set contains the positive (resp. negative) orbit of all its points.

A set D is called *invariant* (or *totally invariant*) if D is both positive and negative invariant.

For instance, a periodic orbit is a positive invariant set.

Observe that if $\{\overline{z_0 z_1 \ldots z_{p-1}}\}$ is a periodic orbit of period p then, by the chain rule, for any $i = 0, \ldots, p-1$

$$(f^p)'(z_i) = \prod_{s=0}^{p-1} f'(z_s) = f'(z_0) \cdot \cdots \cdot f'(z_{p-1}).$$

This quantity is known as the *multiplier* of the periodic orbit.

Most of the time, we will be interested in the *stability* of periodic points and periodic orbits.

Definition 1.7 (Stability of Periodic Points). Given $f\colon V \to \mathbb{C}$ a holomorphic function over $V \subset \mathbb{C}$ an open set. Let us suppose that z_0 is a fixed point of f. We say that z_0 is an *attracting point* if it is stable and all points close enough to z_0 have orbits that tend towards z_0, i.e.

$$\exists\epsilon > 0 : |z - z_0| < \epsilon \Rightarrow |f^n(z_0) - z_0| < \delta \text{ for some } \delta$$

and if $\mathcal{U}$ is a neighbourhood of z_0, then for all $z \in \mathcal{U}$, $f^n(z) \in \mathcal{U}$, $\forall n \geq 1$.

We say that a periodic point of period p of f is an *attracting periodic point* if it is attractor as a fixed point of f^p.

Moreover, we say that z_0 is a *repelling point* if it is attractor by f^{-1}. Likewise, a periodic point of period p of f is a *repelling periodic point* if it is repeller as a fixed point of f^p.

When talking about repelling points, we can only guarantee that the orbit will leave the neighbourhood at some point. However, we can not ensure that the orbit does not return to the neighbourhood later on.

1.2 Local Stability

In this section we study the local stability around periodic points.

Once we have introduced some general notions about dynamical systems, we are now ready to present a result about the local theory of iteration of holomorphic functions.

Theorem 1.8 (Local Stability of periodic points). *Let $f\colon V \subset \mathbb{C} \to \mathbb{C}$ holomorphic, $V \subset \mathbb{C}$ an open set and $z_0 = f^p(z_0)$ a periodic point of f of period $p \geq 1$. Then, z_0 is*

$$\begin{array}{lll} (a) & \textit{attracting} & \textit{if } |(f^p)'(z_0)| < 1, \\ (b) & \textit{repelling} & \textit{if } |(f^p)'(z_0)| > 1. \end{array}$$

If $|(f^p)'(z_0)| = 1$ we call z_0 *neutral*, and if $(f^p)'(z_0) = 0$, we call it *super-attracting.*

Proof. Without loss of generality and to simplify this proof, we consider $p = 1$.

(a) Let us suppose that z_0 is a fixed point of f and $|f'(z_0)| < 1$. So, if z is close enough to z_0, there will be $\mu < 1$ such that

$$\frac{|f(z) - f(z_0)|}{|z - z_0|} < \mu < 1.$$

Since $f(z_0) = z_0$, we can read the last inequality like $|f(z) - z_0| < \mu|z - z_0|$. i.e. $f(z)$ is closer to point z_0 rather than z (with a factor of $\mu < 1$). This lets us to apply the same argument n times to obtain $|f^n(z) - z_0| < \mu^n |z - z_0|$. Since $\mu < 1$, we can conclude that

$$\lim_{n \to +\infty} f^n(z) = z_0$$

for all z in a neighbourhood of z_0.

(b) Let us suppose now that z_0 is a fixed point of f and $|f'(z_0)| > 1$. Then, in this case, we have this inequality

$$\frac{|f(z) - f(z_0)|}{|z - z_0|} > 1.$$

Again, since $f(z_0) = z_0$, $|f(z) - z_0| > |z - z_0|$. So, the points close to z_0 go away, at least during the firsts iterations, from it. This does not guarantee that these points could return.

However, the fact is that we can build a neighbourhood around the fixed point z_0 in which the orbit of the points leaves this neighbourhood after a few iterates.

□

We just say a few words about neutral points. This case is equivalent to $f'(z_0) = e^{i\alpha}, \ \ \alpha \in [0, 2\pi)$.

As a first approach we consider $\alpha = 0$, this is $f'(z_0) = 1$. The canonical example for this case is $f(z) = z + z^2$ which has $z_0 = 0$ as fixed point and $f'(0) = 1$. One can easily check that the real axis is a positive invariant set. Furthermore, if we consider a point over the negative real axis, this is $x \lesssim 0$, its orbit will be closer and closer to origin over the negative real axis, while if we consider a point over the positive real axis, this is $x \gtrsim 0$, its orbit will tend to infinity over the same positive real axis. These directions are called *attractive and repulsive directions*, respectively.

Keeping the same example, to understand the orbit of points which are not over the real axis, it is needed to use special coordinates (called *Fatou Coordinates*) to proof that any complete neighbourhood around the origin behave like the Figure 1.1a [3, Page 34].

In general, there are three types of neutral points.

- Let us take $\alpha = \frac{p}{q} \in \mathbb{Q}$, that is $f(z) = e^{2\pi i \alpha} z + az^2 + \dots$. Then $f^q(z) = z + a_m z^m + \dots, a_m \neq 0$, for some m, in which case we return to case $\alpha = 0$.

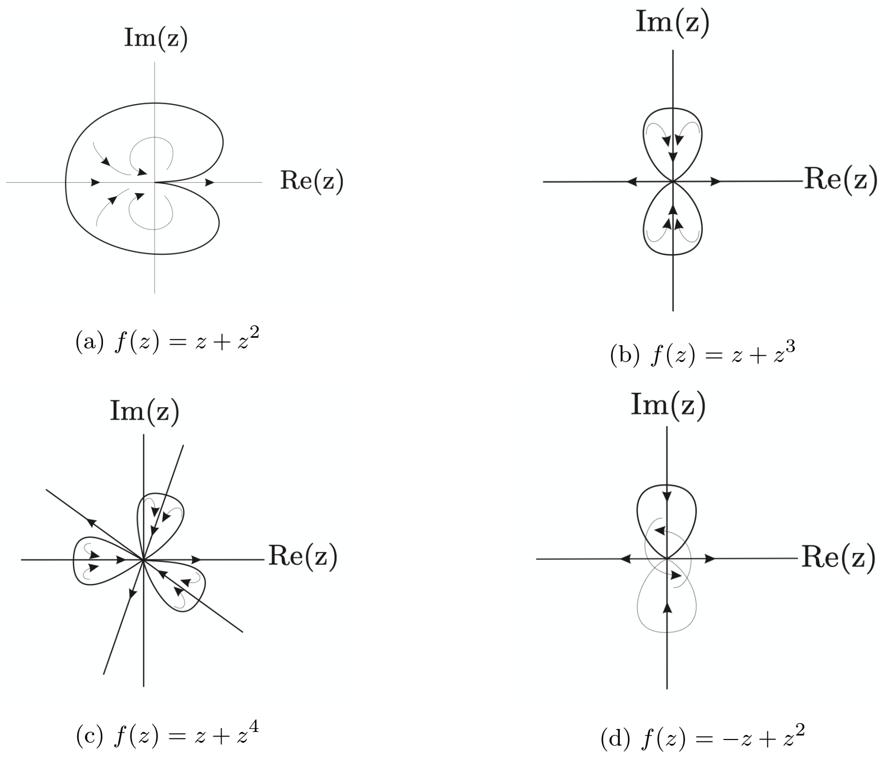
  When a fixed (or periodic) point has a multiplier of module 1 and its exponent α is rational, the fixed (or periodic) point is called *parabolic* or *rationally indifferent.*

- Let us take $\alpha \in \mathbb{R} \setminus \mathbb{Q}$. We should consider different cases.

  – If f is linear, then $f(z) = e^{2\pi i \alpha} z$. Let us consider the dynamics described by the iterates of any seed $z_0 \neq 0$. If $z_0 = r_0 e^{2\pi i \theta_0}$, then $|f^n(z_0)| = r_0$ for all $n \in \mathbb{Z}$. Then, the orbit for every seed $z_0 \in \mathbb{C}$ is

  $$A = \{z \in \mathbb{C} : z = r_0 e^{2\pi i (\theta_0 + k\alpha)}, \ \ k \in \mathbb{Z}\},$$

  inside the circumference of radius r_0. Jacobi's Theorem stands that A is dense over the circumference of radius r_0. (See Jacobi's Theorem on [5, Page 21]).



(a) $f(z) = z + z^2$

(b) $f(z) = z + z^3$

(c) $f(z) = z + z^4$

(d) $f(z) = -z + z^2$

Figure 1.1: The dynamic in a neighbourhood around the origin for different functions

- If f is non linear, i.e. $f(z) = e^{2\pi i\alpha}z + az^2 + \dots$, the main discussion is to decide which values of α, if any, give rise to the same behaviour as the linear case. To answer this, we need to talk about the degree of "irrationality" of α. In some cases, one can prove that the dynamics of a neighbourhood around the origin is *similar* to the dynamic on the linear case $z \mapsto e^{2\pi i\alpha}z$. This is, there are invariant curves where the dynamic of the orbits are dense. When the dynamic of the iterates has this geometry, we say that the point is *linearisable* or is a *Siegel point* and the neighbourhood around this fixed point where the dynamic is preserved is called *Siegel disc.*

  The remaining cases are called *Cremer points* and the dynamics are more complex. For more details we refer to [2] and [4].

It is important to observe that neutral points in complex plane are never attracting nor repelling. However, when we see this topic on real numbers we can have attracting neutral points, repelling neutral points or neither of this behaviour. For instance, on Figure 1.1b, if we restrict the study to the real line we can see that the origin is a neutral point, which is repelling.

In the previous explanation we talked about *similar dynamics.* In many situations we will require to compare dynamical planes for different functions, so we need some mathematical tool that lets us to decide if two dynamic planes are (essentially) different, or otherwise, we can see both of them as one, i.e., the first one is a deformation of the second.

Definition 1.9 (**Conjugacy**)**.** Given $f, g\colon \mathbb{C} \to \mathbb{C}$ two dynamical systems, we say that f is *conjugate* to g if there exists a homeomorphism φ such that

$$\varphi(f(z)) = g(\varphi(z)).$$

The condition of conjugacy forces the function φ to send the fixed points of the first dynamical plane to fixed points on the second dynamical plane. In the same way, it sends periodic points to periodic points, pre-periodic points to pre-periodic points, a certain asymptotic behaviour to the same asymptotic behaviour, etc. In general, φ sends orbits to orbits.

Alternatively, we say that φ *conjugates* or is a *conjugacy* between f and g, and we conclude that both dynamical planes from f and g are the same from a dynamical perspective.

For instance, a quadratic polynomial $P(z) = \gamma z^2 + \alpha z + \beta$ can be conjugated by $z' = az$ to a monic polynomial $z^2 + \alpha z + \beta$. Furthermore, this can be conjugated by a translation $z' = z + t$ to move any given point to 0. If we move one of the fixed points to 0 we have conjugated P to the form $\rho z + z^2$, where ρ is the multiplier of the fixed point. However, this does not determine the conjugacy class uniquely, as we can place the second fixed point at 0. But if we move the critical point to 0, we have conjugated P to the form

$$P_c(z) = z^2 + c.$$

It is important to note that, the more regular φ is, the stronger will be the conjugation between f and g. The strongest type of conjugacy is the conformal map.

Definition 1.10 (**Conformal Map**)**.** A map φ is called *conformal* if, and only if, φ is holomorphic and bijective. A conformal map is a function that locally preserves angles.

Conformal conjugacies preserve the multipliers of periodic orbits.

Now let us present an important result about local linearisation. Let us take a function $f(z)$ and express it by the power series expansion, which can be chosen so that the fixed point corresponds to $z = 0$,

$$f(z) = \rho z + a_2 z^2 + a_3 z^3 + \cdots ,$$

which converges for $|z|$ sufficiently small. Just recall that the initial coefficient $\rho = f'(0)$ is called the multiplier of the fixed point.

Theorem 1.11 (**Kœnigs Linearisation**)**.** *Let $V \subset \mathbb{C}$ an open set and $f\colon V \to \mathbb{C}$ a holomorphic function. Let z_0 be a fixed point of f. If the multiplier ρ satisfies $|\rho| \neq 0, 1$, then there exists a local holomorphic change of coordinates $\omega = \phi(z)$ with $\phi(z_0) = 0$, so that $\phi \circ f \circ \phi^{-1}$ is the linear map $\omega \mapsto \rho\omega$ for all ω in some neighbourhood of the origin. Furthermore, ϕ is unique up to multiplication by a nonzero constant.*

In other words, the following diagram is commutative,

$$\begin{tikzcd} V \arrow[r, "f"] \arrow[d, "\phi"] & f(V) \arrow[d, "\phi"] \\ \mathbb{C} \arrow[r, "\rho\cdot\omega"] & \mathbb{C}, \end{tikzcd}$$

where ϕ is univalent (or injective) on the neighbourhood $V \cup f(V)$ of z_0.

It is important to note that in this linearisation theorem we need to avoid super-attracting and neutral points. We recall that this is $|\rho| \neq 0, 1$.

Proof.

- **Proof of Uniqueness.** If there were two such maps ϕ and ψ, then the composition $\psi \circ \phi^{-1}$ would commute with the map $w \mapsto \rho w$. Expanding it as a power series,

$$\psi \circ \phi^{-1}(w) = b_1 w + b_2 w^2 + b_3 w^3 + \cdots,$$

  and then composing on the left or right with multiplication by ρ, we see by comparing coefficients that $\rho b_n = b_n \rho^n$ for all n. Since ρ is neither zero nor a root of unity, this implies that $b_2 = b_3 = \cdots = 0$. Hence $\psi \circ \phi^{-1} = b_1 w$, or in other words, $\psi(z) = b_1 \phi(z)$.

- **Proof of Existence when $|\rho| < 1$.** Choose a constant $d < 1$ so that $d^2 < |\rho| < d$. As in the proof of Theorem 1.8, we can choose a neighbourhood $\mathbb{D}_r$ for some $r > 0$, in this case around the origin, so that $|f(z)| < d|z|$ for $z \in \mathbb{D}_r$. Thus for any starting point $z_0 \in \mathbb{D}_r$, the orbit $\{z_0, z_1, \cdots\}$ under f converges geometrically towards the origin, with $|z_n| < rd^n$. By Taylor's Theorem, we have $|f(z) - \rho z| \leq D|z^2|$ for $D < 1$ and $z \in \mathbb{D}_r$, and hence

$$|z_{n+1} - \rho z_n| \leq D|z_n|^2 \leq Dr^2 d^{2n}.$$

  Let us consider $k = \frac{Dr^2}{|\rho|}$ and then we have another sequence $w_n = \frac{z_n}{\rho^n}$ that satisfies

$$|w_{n+1} - w_n| \leq k \left(\frac{d^2}{|\rho|}\right)^n.$$

  Now these differences converge uniformly and geometrically to zero. Since the holomorphic functions $z_0 \mapsto w(z_0)$ converge, uniformly throughout $\mathbb{D}_r$, to a holomorphic limit $\phi(z_0) = \lim\limits_{n \to +\infty} \dfrac{z_n}{\rho^n}$.

  The required identity $\phi(f(z)) = \rho\phi(z)$ follows immediately. Furthermore, since each correspondence $z_0 \mapsto w_n = \frac{z_n}{\rho^n}$ has derivative 1 at the origin, it follows that the limit function ϕ has derivative $\phi'(0) = 1$ and hence is a local conformal isomorphism.

- **Proof of Existence when $|\rho| > 1$.** The statement in this case follows immediately by applying the argument above to the map f^{-1}, due to $0 < |\rho^{-1}| < 1$.

□

1.3 Towards Global Theory

Let us start by placing some basic definitions that lay the foundation about global theory.

Definition 1.12 (Basin of Attraction). Given $z_0 \in \mathbb{C}$ an attracting fixed point, we can define the *basin of attraction* (of z_0) as the open set of points of the complex plane whose orbits tend to z_0. We note this as $A(z_0)$, in other words,

$$A = A(z_0) = \{z \in \mathbb{C} : f^n(z) \xrightarrow[n \to +\infty]{} z_0\}.$$

It is important to note that, in many cases, the basin of attraction is not a connected set.

Definition 1.13 (Immediate Basin of Attraction). We call the *immediate basin of attraction* of an attracting fixed point z_0, denoted by $A^* = A^*(z_0)$, the connected component of $A(z_0)$ which contains z_0.

Finally, the points of f where its first derivative does not exists or is zero are called *critical points* of f.

Now, recalling the result of theorem 1.11, if we focus only on the attracting case $0 < |\rho| < 1$, we can go ahead and rewrite this last theorem in a more global form as follows. Suppose that $f\colon \mathbb{C} \to \mathbb{C}$ is a holomorphic function with an attracting fixed point $p = f(p)$ of multiplier $\rho \neq 0$.

Corollary 1.14. *With $p = f(p)$ as stated above and being A the basin of attraction of p, there is a holomorphic map ϕ from A to $\mathbb{C}$ with $\phi(p) = 0$, so that the diagram*

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[d, "\phi"] & A \arrow[d, "\phi"] \\ \mathbb{C} \arrow[r, "\rho\cdot"] & \mathbb{C}, \end{tikzcd}$$

is commutative, and so that ϕ takes a neighbourhood of p biholomorphically onto a neighbourhood of zero. Furthermore, ϕ is unique up to multiplication by a constant.

Proof. To compute $\phi(z_0)$ at an arbitrary point of A we must simply follow the orbit of p_0 until we reach some point p_k which is very close to p, then evaluate the Kœnigs coordinate $\phi(p_k)$ and multiply by ρ^{-k}. □

Finally we are able to present an important result to justify the rendering algorithms for the parameter space.

Theorem 1.15 (Finding Periodic Attractors). *If f is a polynomial map of degree $d \geq 2$, then the immediate basin of every attracting periodic orbit contains, at least, one critical point. Therefore, the number of attracting periodic orbits is finite, less than or equal to the number of critical points.*

This theorem is a consequence of Corollary 1.14. However, we skip the formal proof since it is out of the scope for this project. See [2, Lemma 8.5] for more details.

Furthermore, we present the following Corollary since this is a crucial result in order to justify the rendering algorithms of the parameter space that we aim to implement.

Corollary 1.16. *For a polynomial map of degree $d \geq 2$, there are at most $d-1$ finite critical points and, therefore, at most $d-1$ periodic attractors (without the fixed point at infinity).*

Chapter 2

Global Theory

The reader can find more details about the topics in this chapter on [2] and [3].

At the end of this chapter we will be able to define a first algorithm to draw Julia sets called *Henriksen Algorithm*.

In this chapter, we define the Julia and Fatou sets. Nonetheless, we need to study first one last topic in order to formalise them: normal families.

2.1 Normal Families

Definition 2.1 (Normal Family). The family of holomorphic functions $\mathcal{F} = \{f^n\}_{n\in\mathbb{N}}$ defined as the iterates of f on a domain D forms a *normal family* on D if every sequence of functions of $\mathcal{F}$ has a subsequence that converges uniformly over compacts of D. In that case, the limit is always a holomorphic map or infinity.

We can informally say that the family of iterates $\{f^n\}_{n\in\mathbb{N}}$ is a *normal family* in a neighbourhood of z if all the points in this neighbourhood behave, under iteration of f, in a manner similar to z.

Definition 2.2 (Normal or Stable Point). Let f be a holomorphic function on $\mathbb{C}$. We say that z_0 is a *normal* or *stable point* of f if there is any neighbourhood $\mathcal{U} \subset \mathbb{C}$ of z_0 such that the iterates family $\{f^n\}_{n\in\mathbb{N}}$ form a normal family on $\mathcal{U}$.

One can easily note that it is difficult to verify this definition of normal point in practise. Due to this, we present a classical result on analytical function theory that give us an easier criteria to check the normality of a normal family: Montel's Theorem.

Theorem 2.3 (Montel's Theorem). *Let $\mathcal{F} = \{f^n\}_{n\in\mathbb{N}}$ be the sequence of iterates of an analytical function f defined on a domain D. If $\cup_{n\in\mathbb{N}} f^n(D)$ omits at least three different points on $\widehat{\mathbb{C}}$, this is, if there are distinct points $a, b, c \in \widehat{\mathbb{C}}$ so that $f(D) \subset \widehat{\mathbb{C}} \setminus \{a, b, c\}$, then $\mathcal{F}$ is a normal family on D.*

We skip the proof for this theorem since it is out of the scope of this project.

2.2 Julia and Fatou Sets

Now we are able to place the definitions of Julia and Fatou sets for an arbitrary holomorphic function.

Definition 2.4 (Julia and Fatou sets). Let f be a holomorphic function on $\mathbb{C}$. We define the *Fatou set* of f, $\mathcal{F}(f)$ as the set of normal points of f.

We define the *Julia set* of f, $\mathcal{J}(f)$ as the complementary of Fatou set or, in other words, the set of non-normal points of f.

A first (and important) observation that one can easily extract using only the definition above is that the Julia set is a closed set and, hence, Fatou set is an open set.

As an early property for both Julia and Fatou sets we have the following Proposition.

Proposition 2.5 (Julia and Fatou sets are totally invariant). *Let f be a holomorphic function. If $z \in \mathcal{J}(f)$ (resp. $z \in \mathcal{F}(f)$), then the image and pre-image(s) of z also belongs to $\mathcal{J}(f)$ (resp. to $\mathcal{F}(f)$). In other words, Julia and Fatou sets are totally invariant.*

Proof. To prove this result, it is enough to prove that the Fatou set is totally invariant, since the Julia set is the complementary of the Fatou set.

Let us consider an open neighbourhood $\mathcal{U}$ of z such that $\mathcal{U} \subset \mathcal{F}(f)$. Since the iterates sequence $\{f^n\}_{n\in\mathbb{N}}$ in $\mathcal{U}$ forms a normal family, then we can get a subsequence $\{f^{n_j}\}_{j\in\mathbb{N}}$ convergent in $\mathcal{U}$.

Since $f(\mathcal{U})$ is an open set (for being f analytical), we can immediately see that the subsequence $\{f^{n_j-1}\}_{j\in\mathbb{N}}$ is convergent in $f(\mathcal{U})$, in a way that the image of $f(z)$ admits a neighbourhood $f(\mathcal{U})$ where the family of iterates is normal. If we repeat this process inductively for $f^n(\mathcal{U}),\ \ n = 2, 3, \dots$ we prove that Fatou set is positive invariant.

However, since $f^{-1}(\mathcal{U})$ is also an open set and if we consider the subsequence $\{f^{n_j+1}\}_{j\in\mathbb{N}}$ we can see, using induction once again, that Fatou set is also negative invariant. In consequence, Julia and Fatou sets are both fully invariant. □

We present the following theorem as a compendium of Julia and Fatou sets properties.

Theorem 2.6 (Julia and Fatou Sets Properties). *Let f be a holomorphic map, $\mathcal{F}(f)$ and $\mathcal{J}(f)$ are Fatou and Julia sets of f, respectively.*

(a) ***(Transitivity)*** *If $z \in \mathcal{J}(f)$ and $\mathcal{U}$ is a neighbourhood of z, the union of all iterates of $\mathcal{U}$, i.e. $\bigcup_{n\geq 0} f^n(\mathcal{U})$, covers the Riemann Sphere $\widehat{\mathbb{C}}$ with the exception of, at most, two points.*

(b) ***(Attracting points and basins are on Fatou set)*** *The attracting periodic points and their basins of attraction belong to the Fatou set.*

(c) ***(Iterates of f)*** *For any $k > 0$, the Julia set of f^k, $\mathcal{J}(f^k)$, coincides with the Julia set of f, $\mathcal{J}(f)$.*

(d) ***(Parabolic points)*** *All neutral periodic points belong to the Julia set except for Siegel points.*

(e) ***($\mathcal{J}(f)$ is not Empty)*** *If f is a polynomial map of degree $d \geq 2$, then the Julia set $\mathcal{J}(f)$ is non-empty.*

(f) ***(Iterated Pre-images are dense)*** *If z_0 belongs to $\mathcal{J}(f)$, the set of all pre-images of z_0 is dense on $\mathcal{J}(f)$.*

(g) ***($\mathcal{J}(f)$ is a perfect set)*** *The Julia set $\mathcal{J}(f)$ contains no isolated points, that is, $\mathcal{J}(f)$ is a perfect set.*

(h) ***($\mathcal{J}(f)$ is the closure of the repelling periodic points)*** *The repelling points belong to Julia set. Furthermore, they form a dense set on $\mathcal{J}(f)$. In other words,*

$$\mathcal{J}(f) = \overline{\{z \in \widehat{\mathbb{C}} : z \textit{ is a repelling periodic point}\}}.$$

(i) ***(Julia set with Interior)*** *If the Julia set contains an interior point, then it must be equal to the entire Riemann Sphere $\widehat{\mathbb{C}}$.*

(j) ***(Basin boundary = Julia set)*** *If $A \subset \widehat{\mathbb{C}}$ is the basin of attraction of some attracting periodic orbit, then the topological boundary $\partial A = \overline{A} \setminus A$ is equal to the entire Julia set. Moreover, every connected component of the Fatou set $\widehat{\mathbb{C}} \setminus \mathcal{J}(f)$ either coincides with some connected component of this basin A or else is disjoint from A.*

(k) ***(Julia set and union of components)*** *If D is a union of components of $\mathcal{F}$ that is completely invariant, then $\mathcal{J} = \partial D$.*

Proof. (a) **(Transitivity)** This fact is an immediate consequence of Montel's Theorem 2.3. If the iterates family were to avoid 3 or more points, then z would be a normal point. Hence, this is a contradiction with $z \in \mathcal{J}(f)$.

(b) **(Attracting points and basins are on Fatou Set)** We can observe first that the basin of attraction of any attracting periodic point of period $p \geq 1$ is an open set. Therefore, if z belongs to the basin of attraction of a fixed point z_0, for any neighbourhood $\mathcal{U}$ sufficiently small (inside the basin of attraction), the iterates of f converge on $\mathcal{U}$ towards the constant function $g(z) \equiv z_0$. Hence, z is normal and belongs to $\mathcal{F}(f)$.

On the other hand, if z belongs to the basin of attraction of a periodic point z_0 with period $p > 1$, we can use the same argument over the function $h = f^p$.

(c) **(Iterates of f)** We provide an outline for this proof. One first observation is that we can also work for Fatou set since $\mathcal{F}(f) = \mathbb{C} \setminus \mathcal{J}(f)$. Suppose, for instance, that z belongs to the Fatou set of $f \circ f$. This means that, for some neighbourhood $\mathcal{U}$

of z, the collection of all even iterates $f^{2n}{}_{|\mathcal{U}}$ is contained in a compact subset $K \subset \mathcal{H}(\mathcal{U})$. It follows that every iterate of f, restricted to $\mathcal{U}$, belongs to the compact set of $K \cup (f \circ K) \subset \mathcal{H}(\mathcal{U})$. Therefore, z belongs to the Fatou set of f.

(d) **(Parabolic points)** Let w be a local uniformizing parameter, with $w = 0$ corresponding to the periodic point. Then some iterate f^n corresponds to a local mapping of the w-plane with power series expansion of the form $w \mapsto w + a_p w^q + a_{q+1} w^{q+1} + \cdots$, where $q \geq 2$ and $a_q \neq 0$. It follows that f^{nk} corresponds to a power series $w \mapsto w + k a_q w^q + \cdots$. Thus the q^{th} derivative of f^{nk} at 0 is equal to $q! k a_q$, which diverges to infinity as $k \to \infty$. It follows from Weierstrass Uniform Convergence Theorem[1] that no subsequence $\{f^{nk_j}\}$ can converge locally uniformly as $k_j \to \infty$.

(e) **($\mathcal{J}(f)$ is not empty)** Let us assume that f is a polynomial map of degree $d \geq 2$. Suppose that $\mathcal{J}(f) = \emptyset$. Then $\{f^n\}_{n\in\mathbb{N}}$ is a normal family on all $\widehat{\mathbb{C}}$, and so there is a subsequence such that $\{f^{n_j}\}_{j\in\mathbb{N}} \xrightarrow[j\to+\infty]{} g(z)$ for some analytical function g from $\widehat{\mathbb{C}}$ to $\widehat{\mathbb{C}}$. Since g is holomorphic on all of $\widehat{\mathbb{C}}$ it is a polynomial function. If g is constant then the image of f^{n_j} is eventually contained in a small neighbourhood of the constant value, which is impossible since f^n covers $\widehat{\mathbb{C}}$. If f is not constant, eventually f^{n_j} has the same number of zeros as f (applying the Argument Principle) which is also impossible since f^n has degree d^n.

(f) **(Iterated Prei-mages are dense)** This fact is a consequence of (a). Let be $w \in \mathcal{J}(f)$ and $\mathcal{U}$ a neighbourhood of w. We shall proof that $\mathcal{U}$ contains any pre-image of z_0. Given that the images of $\mathcal{U}$ should cover all $\widehat{\mathbb{C}}$ (except, at most, two points), it must exist $N > 0$ such that $z_0 \in f^N(\mathcal{U})$. This yields that $\mathcal{U}$ has some point that, under N iterates, is sent to z_0; which is the definition of pre-image of z_0.

(g) **($\mathcal{J}(f)$ is a perfect set)** Take $z_0 \in \mathcal{J}(f)$ and $\mathcal{U}$ an open neighbourhood of z_0. First assume z_0 is not periodic and choose z_1 with $f(z_1) = z_0$. then $f^n(z_0) \neq z_1$ for all n. Since $z_1 \in \mathcal{J}(f)$, backward iterates of z_1 are dense in $\mathcal{J}(f)$ (we have proved this in (f)), so there is $\xi \in \mathcal{U}$ with $f^m(\xi) = z_1$. Hence $\xi \in \mathcal{J}(f) \cap \mathcal{U}$ and $\xi \neq z_0$.

Next suppose $f^n(z_0) = z_0$ for some minimal n. If z_0 were the only solution of $f^n(z_0) = z_0$ then z_0 would be a super-attracting fixed point for f^n, contradicting $z_0 \in \mathcal{J}(f)$. Hence, there is $z_1 \neq z_0$ with $f^n(z_1) = z_0$. Furthermore, $f^j(z_0) \neq z_1$ for all j since otherwise it would hold for some $0 \leq j < n$ (by periodicity) and hence $f^j(z_0) = f^{n+j}(z_0) = f^n(z_1) = z_0$, contradicting the minimality of n. As before, z_1 must have a pre-image in $\mathcal{U} \cap \mathcal{J}(f)$ which cannot be z_0.

(h) **($\mathcal{J}(f)$ is the closure of repelling periodic points)** Suppose there is an open disc $\mathcal{U}$ that meets $\mathcal{J}(f)$ and that contains no fixed points of any f^n. We may assume $\mathcal{U}$ contains no poles of f nor critical values of f. If f_1, f_2 are two different branches of f^{-1} on $\mathcal{U}$, then since there are no solutions of $f^n(z) = z$ in $\mathcal{U}$,

$$g_n = \frac{f^n - f_1}{f^n - f_2} \cdot \frac{z - f_2}{z - f_1}$$

[1]For more information see [2, Theorem 1.4].

omits the values 0, 1 and ∞ in $\mathcal{U}$. By Montel's Theorem 2.3, $\{g_n\}_{n\in\mathbb{N}}$ is normal and hence so is $\{f^n\}_{n\in\mathbb{N}}$ which is a contradiction. Therefore, periodic points are dense in $\mathcal{J}(f)$. Since there are only a finite number of attracting and neutral cycles (a consequence from Theorem 1.15) and, since $\mathcal{J}(f)$ is perfect , the repelling orbits are dense in $\mathcal{J}(f)$.

Observation 2.7. This provides us an with algorithm in order to draw Julia sets. Details can be found in Section 2.3.

(i) **(Julia set with interior)** If $\mathcal{J}(f)$ has an interior point z_1, then choosing a neighbourhood $N \subset \mathcal{J}(f)$ of z_1, the union $\mathcal{U} \subset \mathcal{J}(f)$ of forward images of N is everywhere dense, $\overline{\mathcal{U}} \subset \widehat{\mathbb{C}}$. Since $\mathcal{J}(f)$ is a closed set, it follows that $\mathcal{J}(f) = \widehat{\mathbb{C}}$.

(j) **(Basin boundary = Julia set)** If N is any neighbourhood of a point of the Julia set, then by (a) implies that some $f^n(N)$ intersects A, hence N itself intersects A. This proves that $\mathcal{J}(f) \subset \overline{A}$. But $\mathcal{J}(f)$ is disjoint from A, so it follows that $\mathcal{J}(f) \subset \partial A$. On the other hand, if N is a neighbourhood of a point of ∂A, then any limit of iterates $f^n_{|N}$ must have a jump discontinuity between A and ∂A, therefore $\partial A \subset \mathcal{J}(f)$. Finally, one can see that any connected Fatou component which intersects A must coincide with some component of A, since it cannot intersect the boundary of A.

(k) This fact is directly a consequence of being ∂D a subset of $\mathcal{J}(f)$, by (c).

□

2.3 Henriksen Algorithm for Julia set

With the results obtained from Theorem 2.6 (h), we now explain the Henriksen Algorithm for Julia set.

Given a point over the complex plane (a pixel on an image), we want to iterate it in order to check if this point is periodic, i.e., we want to check if $f^n(z) = z$ for some $n \in \mathbb{N}$, $n \geq 1$.

However, we need to tolerate an error for this calculation. So, we want to check if

$$f^n(z+\epsilon) \overset{?}{=} z+\epsilon, \quad |\epsilon| << 1.$$

Then, approximating the above by its Taylor expansion series of order 1, we want to check if:

$$f^n(z+\epsilon) \approx f^n(z) + \epsilon (f^n)'(z) \overset{?}{=} z+\epsilon \iff \frac{z - f^n(z)}{(f^n)'(z) - 1} = \epsilon$$

Hence we will check if $|\epsilon|$ is sufficiently small.

In order to compute $(f^n)'(z)$ we do the following. Since $f^n(z) = f(f^{n-1}(z))$, by the chain rule,

$$(f^n)'(z) = f'(f^{n-1}(z)) \cdot (f^{n-1})(z).$$

So, when coding, we can implement this by initialising an accumulator $dz = 1$ (since $z' = 1$) and we apply the recurrence $dz \leftarrow f'(z) \cdot dz$. In our case, since $f(z) = z^d + c$, we have that $dz \leftarrow d \cdot z^{d-1} \cdot dz$.

With this, we can place the following algorithm to draw Julia sets. See Algorithm 1.

Algorithm 1: Henriksen Algorithm for Julia set

```
Result: Returns the Julia set on a given image.
Input: A maximum number of iterations $N$, a degree $d \geq 2$, a $Tol$ value and a seed
       $(c_x, c_y)$.
Output: An image of the Julia set.
foreach pixel (x, y) on image do
    $c \leftarrow c_x + ic_y$
    $z \leftarrow x + iy$
    $dz \leftarrow 1$
    $w \leftarrow x + iy$
    $orbitFound \leftarrow false$
    $i \leftarrow 0$
    while $orbitFound = false$ and $i < N$ do
        $dz \leftarrow d \cdot z^{d-1} \cdot dz$
        $z \leftarrow z^d + c$
        if $|z| > 500$ then
            Set colour white on (x, y)
        end
        if $|dz - 1| > 1e^{-12}$ then
            $\epsilon \leftarrow \frac{w-z}{dz-1}$
        else
            continue
        end
        if $|\epsilon| < Tol$ then
            $orbitFound \leftarrow true$
            break
        end
        $i \leftarrow i + 1$
    end
    /* We check if i > 5 to avoid draw as black fixed points.        */
    if $orbitFound = true$ and $i > 5$ then
        Set colour black on (x, y)
    else
        Set colour red on (x, y)
    end
end
```

2.4 Classification of Periodic Components

The references for this section are [2, Chapter 5], and [4, Part IV].

In this chapter we focus on the behaviour of a polynomial function $f(z)$ on the Fatou set $\mathcal{F}$. Our main goal is to define a classification of the components of the Fatou set because this will help us when defining the rendering algorithms for the parameter space.

2.4.1 Wandering Domains

We assume that the degree of f is $d \geq 2$ and that the image of any component of the Fatou set $\mathcal{F}$ under f is a component of $\mathcal{F}$. Moreover, the inverse image of a component of $\mathcal{F}$ is the disjoint union of, at most, d component of $\mathcal{F}$.

Let us define the different types of component that we can obtain under the influence of f.

Definition 2.8 (**Types of components of $\mathcal{F}$**)**.** Consider a component $\mathcal{U}$ of $\mathcal{F}$. Then,

1. If $f(\mathcal{U}) = \mathcal{U}$, we call $\mathcal{U}$ a *fixed component* of $\mathcal{F}$.
2. If $f^n(\mathcal{U}) = \mathcal{U}$ for some $n \in \mathbb{N}, n \geq 1$, we call $\mathcal{U}$ a *periodic component* of $\mathcal{F}$. Furthermore, the minimal n is the period component. In particular, if $n = 1$, we have a fixed component.
3. If $f^m(\mathcal{U})$ is periodic for some $m \geq 1$, we call $\mathcal{U}$ a *pre-periodic component* of $\mathcal{F}$.
4. Otherwise, if all $\{f^n(\mathcal{U})\}$ are distinct, we call $\mathcal{U}$ a *wandering domain*.

One can proof that some entire functions have wandering domains[2]. However, this is not possible for polynomial functions.

Theorem 2.9 (**Sullivan**)**.** *A polynomial map has no wandering domains.*

Since this proof is out of the scope for this project, we skip the formal proof of this theorem.

Proposition 2.10. *If $\mathcal{U}$ is a completely invariant component of $\mathcal{F}$, then $\partial\mathcal{U} = \mathcal{J}$, and every other component of $\mathcal{F}$ is simply connected.*

Proof. If $\mathcal{U}$ is completely invariant then $\partial\mathcal{U} = \mathcal{J}$, by Theorem 2.6 (k). Furthermore, the sequence $\{f^n\}_{n\in\mathbb{N}}$ omits the open set $\mathcal{U}$ on $\mathbb{C} \setminus \overline{\mathcal{U}}$, so $\{f^n\}_{n\in\mathbb{N}}$ is normal there and $\mathbb{C} \setminus \overline{\mathcal{U}} \subset \mathcal{F}$. Since $\mathcal{U}$ is connected, each component of $\mathbb{C} \setminus \overline{\mathcal{U}}$ is simply connected. □

[2]See work from I.N. Baker on [6].

2.4.2 Classification Theorem of Fatou's Components

Having ruled out the possibility of wandering domains, we can now be sure that every component of the Fatou set is periodic or pre-periodic.

Apart from attracting basins, there are other types of Fatou components, some of which have been mentioned before.

Definition 2.11 (**Parabolic Component. Parabolic cycle**). A periodic component $\mathcal{U}$ of period n of the Fatou set $\mathcal{F}$ is called *parabolic* if there is a neutral fixed point ξ for f^n on its boundary, such that all points in $\mathcal{U}$ converge to ξ under iteration by f^n. The domains $\mathcal{U}, f(\mathcal{U}), \ldots, f^{n-1}(\mathcal{U})$ form a *parabolic cycle* of Fatou components.

Furthermore, we can have another behaviour.

Definition 2.12 (**Siegel Disc**). A Fatou component on which f is conformally conjugate to an irrational rigid rotation of the unit disc is called a *Siegel disc*, with the fixed point z_0 as centre.

You can see an example of a Siegel disc on Figure 2.1.

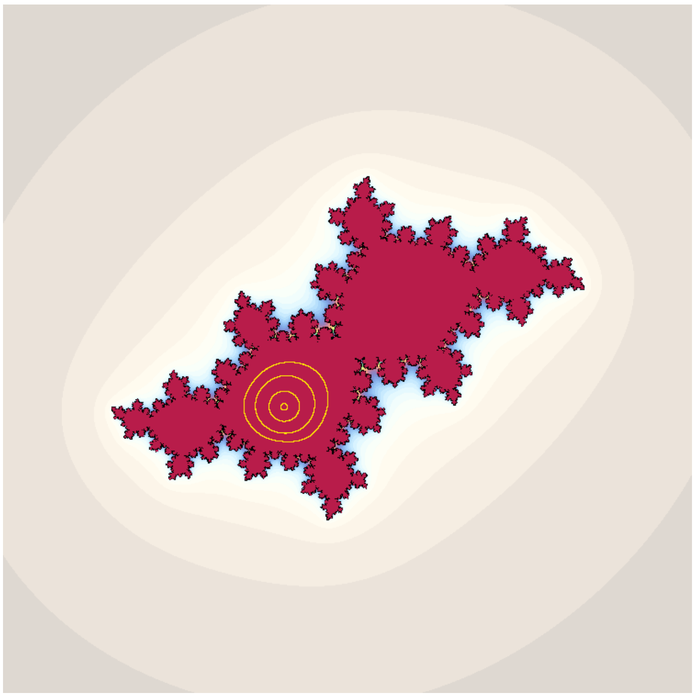


Figure 2.1: The Julia set for $c \approx -0.390541 - 0.586788i$ is a Siegel disc. You can see the invariant curves in yellow

Theorem 2.13 (**Classification of Fatou's Components for polynomials**). *Suppose $\mathcal{U}$ is a periodic component of period p of the Fatou set $\mathcal{F}$. Then exactly one of the following holds.*

1. *$\mathcal{U}$ is an immediate basin of attraction of an attracting fixed pint of f^p.*
2. *$\mathcal{U}$ an immediate basin of attraction of a parabolic fixed point of f^p.*
3. *$\mathcal{U}$ is a Siegel disc of f^p.*

We skip the proof for this theorem since it is out of the scope for this project.

Chapter 3

The Dynamical Plane of Polynomials

The references for this chapter are [3, Chapter 2] and [4, Part III].

In this chapter we aim to justify an algorithm to draw Julia sets called the *Escape Algorithm*. So, once again, we consider a polynomial of degree $d \geq 2$. We present here a first basic property about complex polynomials.

Theorem 3.1 (**∞ is a super-attracting fixed point**)**.** *Any polynomial P of degree $d \geq 2$ has ∞ as super-attracting fixed point.*

Proof. We first see that ∞ is a fixed point for any complex polynomial of degree $d \geq 1$. Let us take a polynomial $P(z)$ of degree $d \geq 1$, this is

$$P(z) = a_0 + a_1 z + \cdots + a_d z^d, \quad a_i \in \mathbb{C}, \quad a_d \neq 0.$$

Since its dominant term is $a_d \neq 0$, we can easily prove that

$$\lim_{|z| \to \infty} P(z) = \infty$$

which implies that infinity is a fixed point of $P(z)$.

Now let us proof that this fixed point is super-attracting, i.e. that this is a point with multiplier $|\rho| = 0$. In order to do this we consider the conjugacy $\phi(z) = \frac{1}{z}$. We observe that $\infty \mapsto 0$ by ϕ. We apply the conjugacy,

$$f(z) = (\phi \circ P \circ \phi^{-1})(z) = \phi(P(\phi^{-1}(z)))$$

where $\phi^{-1}(z) = \frac{1}{z}$. Then we obtain

$$P(\phi^{-1}(z)) = \frac{a_0 z^s + a_1 z^{s-1} + \cdots + a_d z^{s-d}}{z^s},$$

where $s = \sum_{i=0}^{d} i = \frac{d(d+1)}{2}$. Now,

$$f(z) = \phi(P(\phi^{-1}(z))) = \frac{z^d}{a_d + a_{d-1}z + \cdots + a_0 z^d}.$$

Where $f(0) = 0$ and

$$f'(z) = \frac{dz^{d-1}(a_d + a_{d-1}z + \cdots a_0 z^d) - z^d(a_{d-1} + a_{d-2}z + \cdots da_0 z^{d-1})}{(a_d + a_{d-1}z + \cdots a_0 z^d)^2}.$$

We can easily check as well that $f'(0) = 0$, so the multiplier $\rho = f'(0) = 0$. Since ϕ is a conjugacy, we can conclude that ∞ is a super-attracting fixed point for P. □

In summary, when considering polynomials we always have a super-attracting fixed point at ∞, hence, the *basin of attraction of infinity* is always well-defined and we denote it by $A(\infty)$.

Moreover, by Theorem 2.6 (k), the Julia set coincides with $\partial A(\infty)$.

3.1 Julia and Fatou Sets for Polynomials

Let us define the Julia and Fatou sets for polynomials.

Definition 3.2 (Filled Julia set. Julia set. Fatou set). We define the *filled Julia set of a polynomial* $P(z)$ as the set of points whose orbit is bounded, this is

$$\mathcal{K}(P) = \{z \in \mathbb{C} : P^n(z) \not\to \infty\}.$$

We define the *Julia set of a polynomial* $P(z)$, that we write as $\mathcal{J}(P)$, as the boundary of $\mathcal{K}(P)$. We define the *Fatou set* as the complementary of the Julia set, this is $\mathcal{F}(P) = \mathbb{C} \setminus \mathcal{J}(P)$.

Let us show that the general definition for Fatou and Julia sets coincides with this definition in the polynomial case.

Proposition 3.3. *Let $P(z)$ be a polynomial of degree $d \geq 2$. A point $z \in \mathbb{C}$ is non-normal if, and only if, $z \in \partial\mathcal{K}(P)$. Equivalently, $\mathcal{J}(P) = \partial\mathcal{K}(P)$ and $\mathcal{F}(P) = Int(\mathcal{K}(P) \cup A(\infty)$.*

Proof. If $z \in \partial\mathcal{K}(P)$ we consider a neighbourhood $\mathcal{U}$ of z. By definition, $\mathcal{U}$ should contain any points whose orbit is uniformly bounded and points whose orbit tends to infinity. Then, the limit function of the sequence of iterates of $\{P^n\}_{|\mathcal{U}}$ would not be even continuous, so we conclude that z is not normal.

If $z \notin \partial\mathcal{K}(P)$, we consider a neighbourhood $\mathcal{U}$ of z that does not intersect with $\partial\mathcal{K}(P)$. We consider two cases.

- If $\mathcal{U} \subset A(\infty)$, the family of iterates $\{f^n\}_{n\in\mathbb{N}}$ defined on $\mathcal{U}$ must avoid all the periodic points (since those do not belong to $A(\infty)$). Since there are infinitely many periodic points (a finite number for each period), Montel's theorem 2.3 ensures us that this family is normal in z.
- If $\mathcal{U} \subset \operatorname{int}(\mathcal{K}(P))$, the family of iterates $\{f^n\}_{n\in\mathbb{N}}$ defined on $\mathcal{U}$ must avoid the points of the basin of attraction of infinity so, applying Montel's theorem 2.3 once again, we can conclude that this family is normal in z.

□

Let us study now the topology of the Julia sets for the particular case of polynomials. We present two results that summarise the possibilities in the polynomial case.

Theorem 3.4. *The Julia set $\mathcal{J}$ is connected if, and only if, there is no finite critical point of P in $A(\infty)$, that is, if and only if, the positive orbit of each finite critical point is bounded.*

Furthermore, at the other extreme we have the following.

Theorem 3.5. *If $P^n(q) \xrightarrow[n\to\infty]{} \infty$ for each critical point q, then the Julia set $\mathcal{J}$ is totally disconnected.*

Proof. Let D be a large open disc containing $\mathcal{J}$ such that $P(\mathbb{C}\setminus D) \subset \mathbb{C}\setminus \overline{D}$. We choose N sufficiently big that P^N maps the critical points of P to $\mathbb{C}\setminus D$. Hence, for $n \geq N$ there are no critical values of P^n in $\overline{D}$, so that all inverse branches P^{-n} are defined and map $\overline{D}$ conformally into D. Let $z_0 \in \mathcal{J}$, then $P^n(z_0) \in \mathcal{J}$ (by Theorem 2.6 (c)), and we define f_n as the inverse branch of P^n, which maps $P^n(z_0)$ to z_0.

These $\{f_n\}_{n\in\mathbb{N}}$ are uniformly bounded on a neighbourhood of $\overline{D}$, hence they form a normal family there. Since $f_n(z)$ accumulates on $\mathcal{J}$ for $z \in D\cap A(\infty)$, any limit function f maps $D \cap A(\infty)$ into $\mathcal{J}$. Since $\mathcal{J}$ contains no open sets (by Theorem 2.6 (i)), f must be constant. Hence, $f_n(\overline{D})$ has diameter tending to zero. Since $f_n(\partial D)$ is disjoint from $\mathcal{J}$, $\{z_0\}$ must be a connected component of $\mathcal{J}$, and $\mathcal{J}$ is totally disconnected. □

We want to remark that Theorems 3.4 and 3.5 cover all possible cases when considering polynomial maps with one single critical orbit, like those of the form $z^d + c$. So in this case, the corresponding Julia set for each c is either a connected or totally disconnected set. This dichotomy allows us to draw the parameter plane as we explain later on.

3.2 Escape Criteria for Julia set

In this section we present the fundamental result for justifying the *Escape Algorithm for Julia set.*

We place an important result in order to draw Julia sets.

Proposition 3.6. *Let be $d \geq 2$ and $P_c(z) = z^d + c$, $R = \max\{2, |c|\}$. Set $\lambda = R^{d-1} - 1 > 1$. If z is such that $|z| > R$, then $|z^d + c| > \lambda |z|$ and hence*

$$\lim_{n\to\infty} P_c^n(z) = \infty.$$

Proof. We can first observe that if $|z| > R$, then $|z| > 2$ and $|z| > |c|$, so

$$\frac{|z^d + c|}{|z|} \geq |z|^{d-1} - \frac{|c|}{|z|} \geq R^{d-1} - 1 = \lambda > 1,$$

where, we can obtain the first inequality once we apply the $|a + b| \geq ||a| - |b||$ property. Hence, $|P_c(z)| \geq \lambda |z|$ and, therefore, $|P_c^n(z)| \geq \lambda^n |z| \xrightarrow[n\to\infty]{} \infty$, i.e. $\lim_{n\to\infty} P_c^n(z) = \infty$.

$\square$

So, we can conclude with the following localisation criteria.

Corollary 3.7 (Localisation Criteria). *The filled Julia set $\mathcal{K}(P)$ of the polynomial P is entirely contained inside the disc of centre 0 and radius R, where $R = \max\{2, |c|\}$.*

Proof. This Corollary is an immediate consequence of Proposition 3.6. $\square$

3.3 Escape Algorithm for Julia set

Proposition 3.6 and Corollary 3.7 give us another algorithm to draw Julia sets. You can find this algorithm on Algorithm 2.

We simply iterate through the pixels of an image. If $|P_c^n(z)| < 2$ for $1 \leq n \leq N$, where N is the maximum number of iterations allowed, we colour the pixel as black, otherwise, the colour of this pixel is white.

Algorithm 2: Escape Algorithm for Julia Set

Result: Returns the Julia Set on a given image.
Input: A maximum number of iterations N, a degree $d \geq 2$ and a seed (c_x, c_y).
Output: An image of the Julia Set.

```
foreach pixel (x, y) on image do
    $c \leftarrow c_x + ic_y$
    $z \leftarrow x + iy$
    $i \leftarrow 0$
    while $|z| < 2$ and $i < N$ do
        $z \leftarrow z^d + c$
        $i \leftarrow i + 1$
    end

    if $i = N$ then
        Set colour black on (x, y)
    else
        Set colour white on (x, y)
    end
end
```

Chapter 4

Parameter space: The Mandelbrot and Multibrot sets

The reader can find the reference for this chapter on [4, Part VIII].

Let us consider $P(z)$ a unicritical polynomial of degree $d \geq 2$. By translating the critical point to the origin, we can assume that the polynomial can be written as $P_c(z) = z^d + c$ for a complex parameter $c \in \mathbb{C}$.

In this chapter we aim to provide the needed results to justify two different algorithms to draw the Mandelbrot set.

4.1 The Basic Dichotomy

We are interested in *how* the dynamical behaviour of P_c depends on the parameter c. So, we place the following Theorem that is known as the *Basic Dichotomy.*

Theorem 4.1 (**Basic Dichotomy**). *If $P_c^n(0) \xrightarrow[n\to\infty]{} \infty$ then, the Julia set $\mathcal{J}$ is totally disconnected. Otherwise, $P_c^n(0)$ is bounded, and the Julia set is connected.*

Proof. This is an immediate consequence of Theorems 3.4 and 3.5. □

The set of parameter values c such that $P_c^n(0)$ is bounded is called the *Multibrot set* and denoted by $\mathcal{M}_d$. In the case $d = 2$, this is the well known *Mandelbrot set* (see Figure 4.1). Hence $c \in \mathcal{M}_d$ if and only if, 0 does not belong to the basin of attraction of the super-attracting fixed point at ∞ (stated on Theorem 3.1). You can see the Multibrot sets for $z^3 + c$ and $z^4 + c$ on Figures 4.2a and 4.2b, respectively.

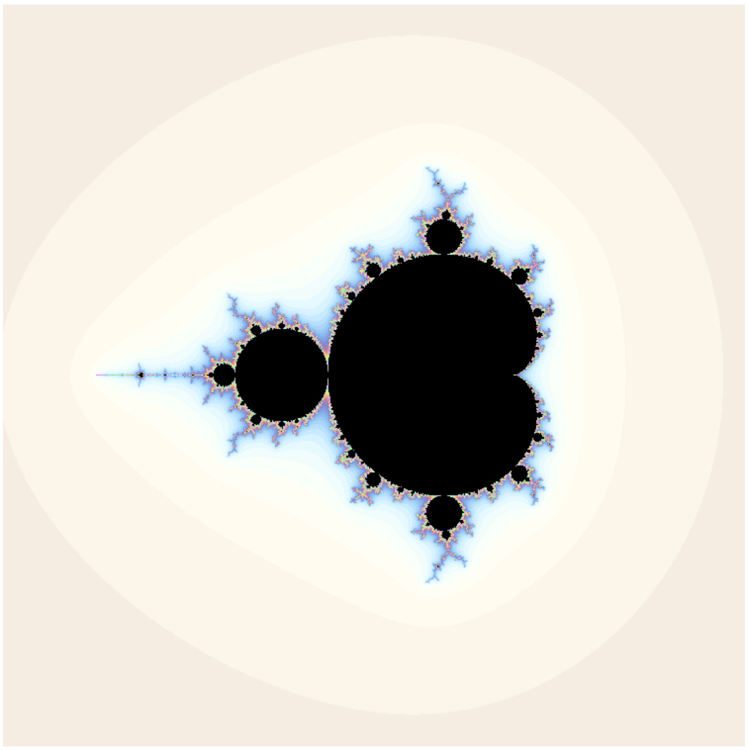
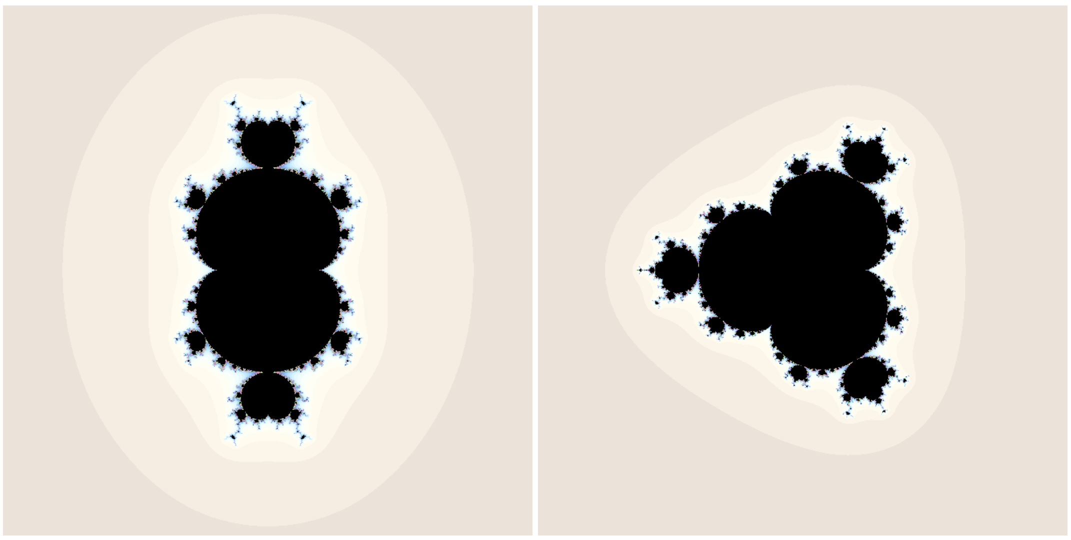


Figure 4.1: The Mandelbrot set

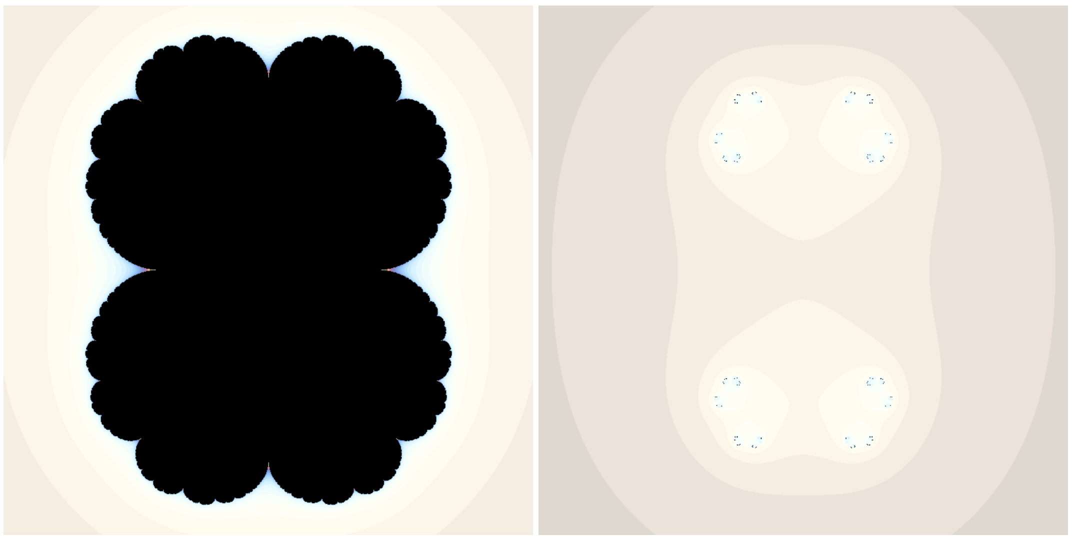


(a) Multibrot set for $z^3 + c$

(b) Multibrot set for $z^4 + c$

Figure 4.2: Multibrot sets

For instance, the value $c = 0.25 \in \partial\mathcal{M}_d$, while $c = 1 \notin \partial\mathcal{M}_d$. See Figure 4.3 to see a comparison between a connected Julia set for $c = 0.25$ on the left, and a totally disconnected Julia set for $c = 1$ on the right.



(a) $c = 0.25$ is connected (b) $c = 1$ is totally disconnected

Figure 4.3: In black we represent the filled Julia set. We can see on (a) that this set is connected, while in (b) $\mathcal{J}(P)$ is a totally disconnected set (a Cantor set)

An important observation about theorem 4.1 is that this result is given thanks to Corollary 1.16. We recall that this says that a polynomial can have as many attractors as critical points. So, since $z^d + c$ has 0 as the unique critical point, we can ensure that we are not missing any behaviour.

4.2 Escape Criteria

In this section we present the *escaping algorithm* to draw Multibrot sets. This algorithm is based on the following result.

Theorem 4.2 (**Structure of Multibrot set**). *The Multibrot set $\mathcal{M}_d$ is a closed subset of the disc $D_2 = \{c \in \mathbb{C} : |c| \leq 2\}$, which meets the real axis in the interval $[-2, \frac{1}{4}]$. Moreover, $\mathbb{C} \setminus \mathcal{M}_d$ is connected $\mathcal{M}_d$ consists of precisely those c such that $|P_c^n(0)| \leq 2$ for all $n \geq 1$.*

Proof. We apply Proposition 3.6. In the notation there, if $|c| > 2$ then $R = |c|$. Observe also that

$$|P_c^2(0)| = |c^d + c| \geq |c|(|c|^{d-1} - 1) > |c|.$$

Hence, by Proposition 3.6,

$$P_c^n(c^d + c) = P_c^{n+2}(0) \xrightarrow[n\to\infty]{} \infty$$

and $c \notin \mathcal{M}_d$. Hence $|c| \leq 2$ for $c \in \mathcal{M}_d$.

Suppose that $|P_c^m(0)| = 2 + \delta > 2$ for some $m \geq 1$, by the same reasoning, $c \notin \mathcal{M}_d$.

This proves the final statement of the theorem, from which it follows that $\mathcal{M}_d$ is closed.

Henceforth, by the maximum principle, $\mathbb{C} \setminus \mathcal{M}_d$ has no bounded components, so $\mathbb{C} \setminus \mathcal{M}_d$ is connected.

If c is real, then $P_c(x) - x$ has no real roots if $c > \frac{1}{4}$. It has one root at $\frac{1}{2}$ and if $c = \frac{1}{4}$ and it has two real roots if $c < \frac{1}{4}$. So, if $c > \frac{1}{4}$, $P_c^n(0)$ is increasing and must go to infinity, since any finite limit point would satisfy $P_c(x) = x$. On the other hand, if $c \leq \frac{1}{4}$, let $a = \frac{(a+\sqrt{1-4c})}{2}$ be the larger root of $P_c(x) = x$. If additionally, $c \geq -2$, one can check that $a \geq |c| = |P_c(0)|$. Then $|P_c^n(0)| \geq a$ implies $|P_c^{n+1}(0)| = |P_c^n(0)^2 + c| \leq a^2 + c = a$, and the sequence is bounded. Finally, $\mathcal{M}_d \cap \mathbb{R} = [-2, \frac{1}{4}]$. $\square$

In addition, we review the various possibilities for the Julia set $\mathcal{J}$ of P. Again by Corollary 1.16, there is at most one periodic cycle of bounded components of $\mathcal{F}$. Now, by Sullivan's Theorem 2.9, every bounded component of $\mathcal{F}$ is eventually iterated into the cycle since this theorem ensures that it can not end on wandering domains. Finally, the classification Theorem 2.13 gives us the following four possibilities for $c \in \mathcal{M}_d$.

1. There is an attracting cycle for P. Either there is an attracting fixed point, in which case there is only one bounded component of $\mathcal{F}$. Or the cycle has length two or more, in which case there are infinitely many bounded components of $\mathcal{F}$.

2. There is a parabolic cycle for P. Either there is a parabolic fixed point with multiplier 1, in which case there is only one bounded component of $\mathcal{F}$. Or the cycle of parabolic components of $\mathcal{F}$ has length two or more, in which case there are infinitely many bounded components of $\mathcal{F}$. This case only occurs for one value of c, in particular $c = \frac{1}{4}$.

3. There is a cycle of Siegel discs.

4. There are no bounded components of $\mathcal{F}$, i.e. $\mathcal{F}(P) = A(\infty)$.

4.2.1 Escape Algorithm for Mandelbrot and Multibrot sets

Theorem 4.2, give us a simple algorithm to draw Mandelbrot and Multibrot sets. We define this algorithm on Algorithm 3.

In this procedure, we simply iterate across all pixels on an image and iterate each pixel. If $|P_c^n(0)| \leq 2$ for $1 \leq n \leq N$, where N is the maximum number of iterations allowed, we colour the pixel as black, otherwise, the colour of this pixel is white.

Once again, this algorithm relies strongly on Corollary 1.16 and the fact that a polynomial $z^d + c$ has a unique critical point at 0 with multiplicity $d - 1$.

Algorithm 3: Escape Algorithm for Mandelbrot set

Result: Returns the Mandelbrot set on a given image.
Input: A maximum number of iterations N and a degree $d \geq 2$.
Output: An image of the Mandelbrot set.

```
foreach pixel (x, y) on image do
    $c \leftarrow x + iy$
    $i \leftarrow 0$
    while $|z| < 2$ and $i < N$ do
        $z \leftarrow z^d + c$
        $i \leftarrow i + 1$
    end

    if $i = N$ then
        Set colour black on (x, y)
    else
        Set colour white on (x, y)
    end
end
```

4.3 Another approach to rendering the Mandelbrot and Multibrot set

In this section we aim to justify another rendering algorithm for the Mandelbrot set and the Multibrot sets called *Henriksen Algorithm.* We start by defining the *centres* of $\mathcal{M}_d$.

Definition 4.3 (Centre of $\mathcal{M}_d$). A point $c \in \mathcal{M}_d$ is called *centre of* $\mathcal{M}_d$ if 0 is periodic under $P_c(z) = z^d + c$. This is, $P_c^n(0) = 0$ for some $n > 0$.

The centres of $\mathcal{M}_d$ are located in the interior of $\mathcal{M}_d$. Indeed, if c_0 is a centre, then P_c has a cycle with multiplier 0 since $P_c'(0) = 0$. If $|c - c_0|$ is small enough, this cycle persists and it is still attracting, so $c \in \mathcal{M}_d$.

We define $\mathcal{C}_d = \{c \in \mathbb{C} : c \text{ is a centre of } \mathcal{M}_d\}$.

This algorithm is based on the following theorem.

Theorem 4.4. *Let $\mathcal{M}_d$ be the Multibrot set for the polynomial $P_c(z) = z^d + c$, $d \geq 2$ and $\mathcal{C}_d$ the set of centres of $\mathcal{M}_d$. Then*

$$Acc(\mathcal{C}_d) \subset \partial \mathcal{M}_d,$$

where $Acc(\mathcal{C}_d)$ denotes the set of limit points of $\mathcal{C}_d$. In other words, the set $\mathcal{C}_d$ is dense at the boundary of the Multibrot set.

Proof. Let D be a disc that meets $\partial\mathcal{M}_d$ but $0 \notin D$. Let us suppose that $\mathcal{U}$ does not contains c values for which 0 is periodic. Then, consider a branch of $\sqrt[d]{-c}$ defined on D.

We have that $P_c^n(0) \neq \sqrt[d]{-c}$ because otherwise $P_c^{n+1}(0) = 0$ and 0 is periodic. If we define the sequence $\{f_n(c)\}_{n\in\mathbb{N}} = \frac{P_c^n(0)}{\sqrt{-c}}$, this sequence omits values 0, 1 and ∞ on D, hence, by Montel's Theorem 2.3, $\{f_n(c)\}_{n\in\mathbb{N}}$ is a normal sequence on D. However, since D meets $\partial\mathcal{M}_d$, it contains points c with $f_n(c)$ bounded and with $f_n(c) \xrightarrow[n\to\infty]{} \infty$. This is a contradiction. □

4.3.1 Henriksen Algorithm for Mandelbrot and Multibrot sets

Theorem 4.4 gives a way to draw the Multibrot sets. You can see the Henrisken Algorithm for Mandelbrot and Multibrot sets on Algorithm 4.

Given the polynomial $P_c(z) = z^d + c$, in this algorithm we iterate through an image and for each pixel we fix c and we check:

- If $|P_c^n(0)| > 500$, then $c \notin \mathcal{M}_d$ so we colour this pixel as white.
- If $P_c^n(0) = 0$ for some n, then $c \in \partial\mathcal{M}_d$ so we colour this pixel as black.

  To be precise, we need to tolerate an error in order to check this condition. That is, fixed a complex number c, we want to check if

  $$P_c^n(0+\epsilon) \overset{?}{=} 0.$$

  We approximate by the Taylor expansion series of first order,

  $$P_c^n(0+\epsilon) \approx P_c^n(0) + \frac{\partial}{\partial c}(P_c^n(0)) \cdot \epsilon \overset{?}{=} 0 \iff \epsilon = \frac{P^n(0)}{\frac{\partial}{\partial c}(P^n(0))}$$

  Hence we will check if $|\epsilon|$ is sufficiently small.

  Likewise we stated on Section 2.3, in order to compute $\frac{\partial}{\partial c}(P^n(0))$ we do the following. Since $P^n(z) = P(P^{n-1}(z))$, by the chain rule,

  $$\frac{\partial}{\partial c}(P^n(0)) = \frac{\partial}{\partial c}(P(P^{n-1}(0))) = \frac{\partial}{\partial c}((P^{n-1}(0))^d + c) = d \cdot (P_c^{n-1}(0))^{d-1} \cdot \frac{\partial}{\partial c}(P_c^{n-1}(0)) + 1.$$

  So, when coding, we can implement this by initialising an accumulator $dc = 0$ (since $c' = 1$) and we apply the recurrence $dc \leftarrow d \cdot z^{d-1} \cdot dc + 1$.
- Otherwise, the point $c \in \operatorname{int}(\mathcal{M}_d)$, so we colour this pixel red.

Again, this algorithm relies strongly on the result from Corollary 1.16. We check all these conditions iterating at 0 because this result ensures us that there is at most one $d-1$ periodic attractor since 0 is the only critical point of $P_c(z) = z^d + c$ with multiplicity $d-1$.

Algorithm 4: Henriksen Algorithm for Mandelbrot Set

Result: Returns the Mandelbrot Set on a given image.
Input: A maximum number of iterations N, a degree $d \geq 2$ and a Tol value.
Output: An image of the Mandelbrot Set.

```
foreach pixel (x, y) on image do
    $c \leftarrow x + iy$
    $z \leftarrow 0 + 0i$
    $dz \leftarrow 1$
    $orbitFound \leftarrow false$
    $i \leftarrow 0$
    while orbitFound = false and $i < N$ do
        $dz \leftarrow d \cdot z^{d-1} \cdot dz + 1$
        $z \leftarrow z^d + c$
        if $|z| > 500$ then
            Set colour white on (x, y)
            return
        end

        $\epsilon \leftarrow \frac{z}{dz}$
        if $|\epsilon| < Tol$ then
            $orbitFound \leftarrow true$
            break
        end
        $i \leftarrow i + 1$
    end

    if orbitFound = true then
        Set colour black on (x, y)
    else
        Set colour red on (x, y)
    end
end
```

Part II

Visualisation of Fractals

Chapter 5

Basic Notions on Computer Graphics

In this chapter we explain the process of visualisation with special emphasis on the elements that define the projections, the different systems of coordinates, and the visualisation strategies used to obtain interactive renderings. The scene to render includes one fractal that is calculated in time of visualisation. Thus, we start this chapter with the definition of a camera and all its required parameters to render a scene, going deeper into two different types of projection. Then, we will detail the coordinates system used in all the different visualisation stages. Finally, we will present the different stages of this visualisation pipeline.

5.1 The camera object

The Camera is the main object of any scene, since without it one cannot render any scene. Since Camera is an actual object, you can centre it wherever you want. So, this centre point is called *viewpoint* or *lookFrom*. The point where the camera is looking at any given time is called *lookAt (VRP)*. In addition to this, we can consider an important parameter in form of vector called *vertically vector (VUP)* that its purpose is to define the verticality of the camera object respect to the scene. For instance, a $VUP = (0, 1, 0)$ keeps the same verticality as the scene. At this point, we can define a system of coordinates by picking an orthogonal basis of vectors with VRP as the origin point. This set of vectors is pointing to: *lookAt*, *VUP* and the third one is the vectorial product of the previous ones.

You can see in Figure 5.1 that there are other non-positional parameters that need to be defined in order to acquire a unique camera over the volume of vision or *frustrum* in the world space. These parameters are *clipping planes*, *type of projection*; which is detailed in the next section; and the *viewport*, where we display the content.

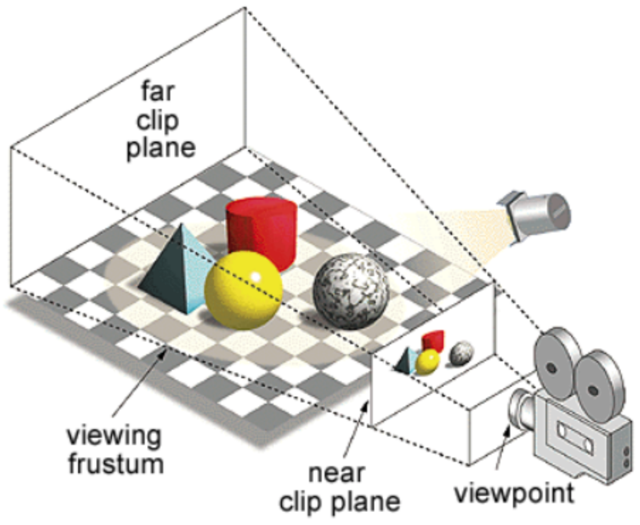


Figure 5.1: Non-Positional Camera

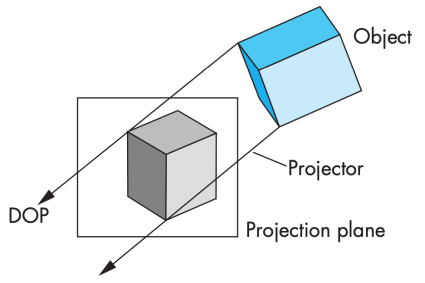
5.2 Types of Camera Projection

There are two basic types of projections when one want to project any 3D scene into a 2D plane.

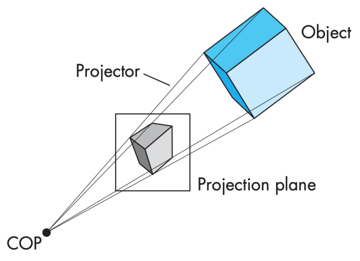
On the one hand there is the *Orthographic projection* where the projectors are perpendicular to the projection plane and we deal with a direction of projection (*DOP*) due to the centre of projection is located at infinity. So, the content visible of the scene is located inside a parallelepiped volume defined among the antero-posterior and the lateral planes (named *clipping planes*). One may find difficult to imagine this mode since we; as human beings; perceive the world through eyes in the following type of perspective. You can see a representation of this projection type on Figure 5.2a [7, Page 196].

On the other hand, the *Perspective Projection* projects the content of a scene in a more "natural" way. In this case, the projection is defined using the *COP* of projection. You can see a representation of this projection type on Figure 5.2b [7, Page 196]. This projection type renders the objects that are inside a cropped pyramid.

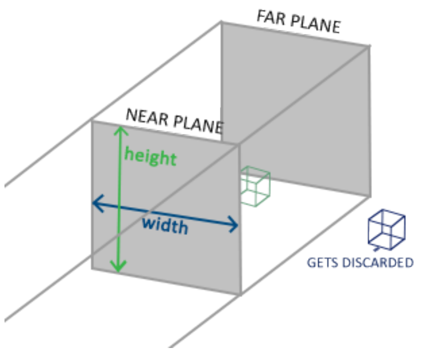
Figures 5.2c [40] and 5.2d [40] show the visibility of an object related to the clipping planes and the shape of visible volume. One can easily note that on both Figures, the green box is visible to the camera, while the blue one is hidden.



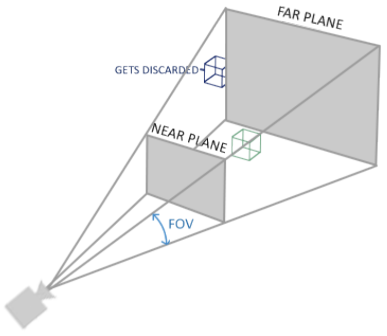
(a) Representation of orthographic projection



(b) Representation of perspective projection



(c) Orthographic Projection



(d) Perspective Projection

Figure 5.2: Projection types of a camera

5.3 Coordinate Systems

There are several coordinate systems involved in the process of rendering images from the visualisation pipeline. So, each object that will appear on the screen must go across the following.

1. Local Space mapped by the Local Coordinates.
2. World Space mapped by the World Coordinates.
3. View Space mapped by the Local Camera Coordinates.
4. Clip Space mapped by the Clipping Coordinates.
5. Screen Space mapped by the Window Coordinates.

The Figure 5.3 [41] summarises this process showing each phase with its transformation matrix.

Local Coordinates are used to represent the vertices of an object in a system where no other external element is involved. The point of reference in this case is called *centre of the object* or *centre of the model*. By applying the *model matrix* to all the vertices of an object,

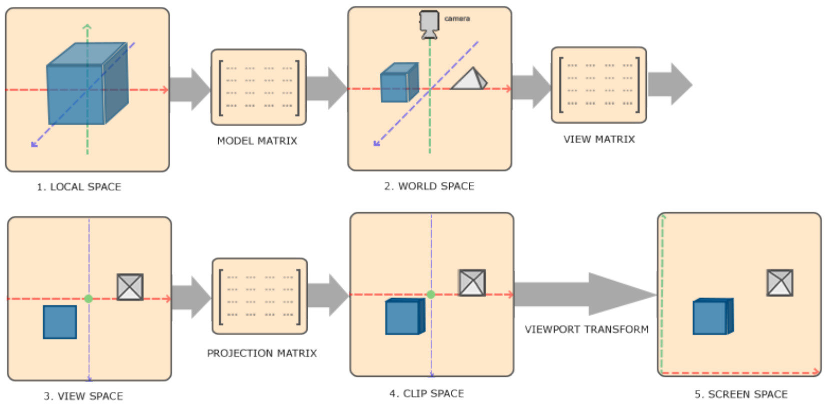


Figure 5.3: Coordinate Systems and its matrices involved.

local coordinates can be transformed to world coordinates. They are used to specify the position of a point in a global system where all the points are sharing the same origin called *centre of the scene*.

The *view matrix* is able to transform world coordinates to other coordinates that are linked to the position of a specific camera. The resulting coordinates are named *camera coordinates*.

Now, depending on the type of projection (that we have seen in the last section) the field of view might be clipped in a way or another. The field of view of a camera before clipped is called *frustrum of vision*. This transformation is carried on by the *projection matrix* that projects all the content of the scene into a squared defined by coordinates between 1 and -1. These are the *clipping coordinates*.

The last step is to fit the image into the screen of the camera. In order to do this we must apply the last transformation, where a scale and translations are applied. The coordinates resulted are called *window coordinates*.

5.4 Programable Visualisation Pipeline

In this section, we explain this the visualisation pipeline focusing on OpenGL. See [7].

The *visualisation* or *rendering pipeline* is the process, or set of stages, where the objects, lights and the camera (or cameras) of a scene are transformed and shaded in order to be displayed on the screen with some characteristics that do not appear in their definitions. To name a few, we can have illumination, shadows or glows among other many effects.

Here we define an object as a set of 3D points (by default) that are displayed in triangles called *faces*. Then, the *shaders* are the programs that handles these triangles by using the GPU. In addition to this, *shaders* can be classified into different categories depending on the coordinates they modify work out.

You can see on Figure 5.4 [42] the scripts available to program in blue. These parts will be explained on section 5.6.

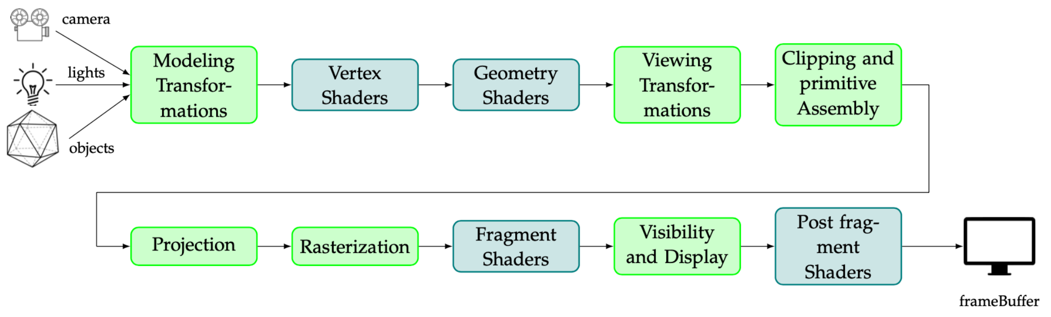


Figure 5.4: OpenGL Visualisation Pipeline

As we have stated before, the main advantage using shaders on a GPU over a CPU (for instance) is their capability of working in parallel. Since each *shader* is executed on a core of a GPU, all the vertexs of an object can be processed in parallel using the same program distributed among all the GPU cores available. This specific technique is known as *SIMD* or *Single Instruction Multiple Data.*

At this point, one can easily think that most of the modern CPU can easily have 4 or 8 cores and even usually each core can handle two threads at the same time, so it is easy to have, at the very end, 8 or 16 threads of execution on a "regular" CPU. So, what is the point about using a GPU in this case? Well, one low-tier GPU can easily have, at least, 400 cores available to do these kind of computations. This is the real improvement about using a GPU over CPU to do this particular computation.

5.5 Shader Structure

A *shader* is a special program written in high-level language that does not compile together with the project. So, this is usually compiled "on-the-fly" during the execution time. It handles the data that receives from the CPU to the GPU using special modifiers called `in`, `out` and `uniform`. Thanks to this, the programmer gain control over the GPU. Let us detail each of these modifiers.

`Uniform` These variables are used to pass the exact same information from CPU to all the *shaders*. For instance, if we want to colour one object as red, we shall pass the red colour through a `uniform` variable.

`In` The `in` modifier behaves in the opposite way compared to the `uniform`. For example, a `in` should be used for customising the content of each *shader*, like the normal vector associated to each of them.

`Out` This modifier assigns a variable to be passed to the next pipeline step.

5.6 Type of Shaders

There are several types of shaders, we usually will distinguish them by when they are applied on the visualisation pipeline. In general terms, there are *vertex shaders*, *geometric shaders*, *fragment shaders* and *post-fragment shaders*.

5.6.1 Vertex Shaders

At this stage in the visualisation pipeline, each vertex of the model is processed in a GPU core. The vertex coordinates, the normal vector to this vertex, and some other attributes are passed as input to this program. The expected output of this shader is also a vertex, but it is modified to have some special effect. The transformations applied at this stage are usually related to individual lightening of each specific vertex, modifying its position and its normal.

At this moment, the model matrix and view matrix (seen on Figure 5.3) are applied and with this, usually, we obtain the window coordinates at the end of the stage. In addition, some calculations can also be done using other properties of the scene. For example, we can modify some matrices that we will use on the nexts stages.

It is important to note that the *rasterisation* step will produce the interpolation effect that is useful when we want a smooth transition between two different colours.

5.6.2 Geometry Shaders

Typically, the Geometry Shader handles the vertices manipulation of the model. At this stage, the vertices can be displaced or even split to create new ones that the original model did not has. This type of shaders shines when the loaded object from memory has not enough vertices and, using Geometry Shaders, new vertices can be created in order to smooth surfaces or to have more accurate objects.

5.6.3 Fragment Shaders

These kind of shaders are applied after the rasterisation process, as stated on Figure 5.4. At this stage, the assembly of the primitives (results of the flow from vertices, normals and buffers involved in the whole process) give place to smaller elements that are pixel-related. These are the *fragments* where this program *run* in parallel.

5.6.4 Post-Processing Shaders

Once all the visualisation process is completed, the final image is sent to the screen. At this moment, one can apply some special shaders, called post-processing shaders, but in this case they are applied over the output image directly (this means that any vertex or objects are involved at this point).

It is important to note that one can also apply a bunch of post-processing shaders in a certain order to obtain some effect, for instance, blurring effect or paint-style effect.

5.7 Some Shader Parameters

In this subsection we present some relevant input parameters of the shaders, since we will be using them when colouring the fractals. You can see these parameters summarised in Figure 5.5 [44]

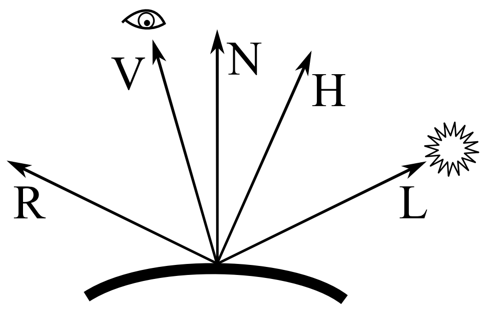


Figure 5.5: Vectors involved in illumination

Let us detail them,

Position, P This is a four component vector that holds the position of a vertex on world coordinates using *homogeneous coordinates*[1]. We usually use *position* or *vertex* to refer to the same concept.

Normal vector, $\overrightarrow{N}$ This vector refers to the normal vector of a vertex when stored in an object.

Light Vector, $\overrightarrow{L}$ Usually a shader is linked with a point of light that affects its properties. This vector has its origin at the vertex of the object and it is pointing to the source of light.

Viewing vector, $\overrightarrow{V}$ Since a camera is needed in order to visualise any scene, this vector can be defined as the vector with origin at the vertex pointing to the camera's coordinates (*LookFrom*).

Reflection Vector, $\overrightarrow{R}$ This stores the reflection from the light point. It depends on the surfaces where the shader is applied to.

Middle Vector, $\overrightarrow{H}$ It is defined as $\overrightarrow{H} = \dfrac{\overrightarrow{L} + \overrightarrow{V}}{||\overrightarrow{L} + \overrightarrow{V}||}$ and it is used sometimes in order to boost some calculations.

[1]See [8] for more details.

5.8 Blinn-Phong Illumination

Blinn-Phong Lightning [9, 10] is an empirical lightening model. It defines the light model as three different behaviours. See Figure 5.6 to see the three separated components.

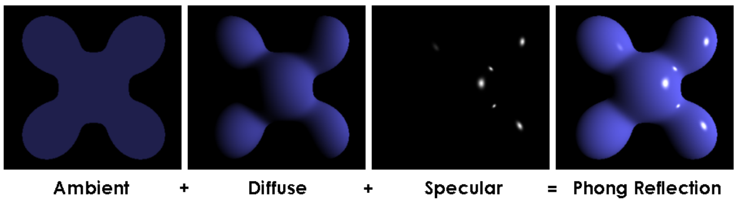


Figure 5.6: Blinn-Phong Illumination

Let us specify each of these components:

- **Ambient component:** This is obtained by the influence of the diffuse lights that do not act directly over the object.
- **Diffuse component:** This is the diffuse reaction of the object light. In many cases, this defines the main colour of the object.
- **Specular component:** This is the reflectance level of the material. Basically, it handles the size of the highlights.

The general formula of Blinn-Phong illumination is the following.

$$I_{\text{glob}} = I_{a_{\text{glob}}} k_a + \sum_{l \in \text{Lights}} \left(\frac{1}{a_l d_l^2 + b_l d_l + c_d} \underbrace{(I_{d_l} k_d \max(\overrightarrow{l_l} \cdot \overrightarrow{n}, 0)}_{\text{diffuse}} + \underbrace{I_{s_l} k_s \max(\overrightarrow{n} \cdot \overrightarrow{h_l}, 0)^{\beta}}_{\text{specular}} + I_{a_l} k_a \right) \tag{5.1}$$

Where k_a, k_d, k_s are properties of the emulated material in terms of ambient, diffuse and specular behaviour respectively. I_a, I_d, I_s are the light properties. $\overrightarrow{n}$ and $\overrightarrow{h}$ are the normal and middle vectors detailed in Section 5.7. d is the coefficient that handles the attenuation if the light is far from the object and, finally, β controls the level of reflectance of the material (its *shininess*).

It should be noted that in this project, we will not use this illumination approach directly, however, we will use some parts from formula (5.1) in order to illuminate the scenes in our solution.

Chapter 6

Problem Overview

In this chapter we define our problem, we specify the requirements for this problem, we review the state-of-the-art on this topic and, finally, we make a comparison between some existing available software that display fractals using computer graphics.

6.1 Analysis of Requirements

In this section we will analyse the requirements (both functional and non-functional) for our application.

However, first things first, the aim of this application is to provide a software that allows the User to visualise and manipulate some fractals interactively. In particular, we are interested on two types of fractals,

1. **2D Fractals:** These fractals are the result of iterating holomorphic functions on the complex plane. Specifically, we want to visualise what are called *Mandelbrot* and *Julia sets* over polynomial functions.

2. **3D Fractals:** These are a generalisation about the aforementioned 2D Fractals. In particular, we want to visualise a generalisation of fractals over polynomial functions called *Mandelbulb* [11] and we are also interested on visualise another type of fractals that are the result of apply an *IFS (Iterated Function System)* [12].

On both types, we also want to be able to tweak some parameters in order to see how they react at these changes.

Finally, we will be interested on using the GPU in order to boost the visualisation process for both on 2D Fractals and 3D Fractals as well.

6.1.1 Functional Requirements

In this subsection we will analyse the functional requirements for our problem. In order to do this, and as a helping tool, we provide four Use Cases Diagrams that will help us to understand this problem.

The software application should have four different parts. Let us specify each of them.

Main Page

The first one can be found on Figure 6.1. Let us specify the main use cases on this page.

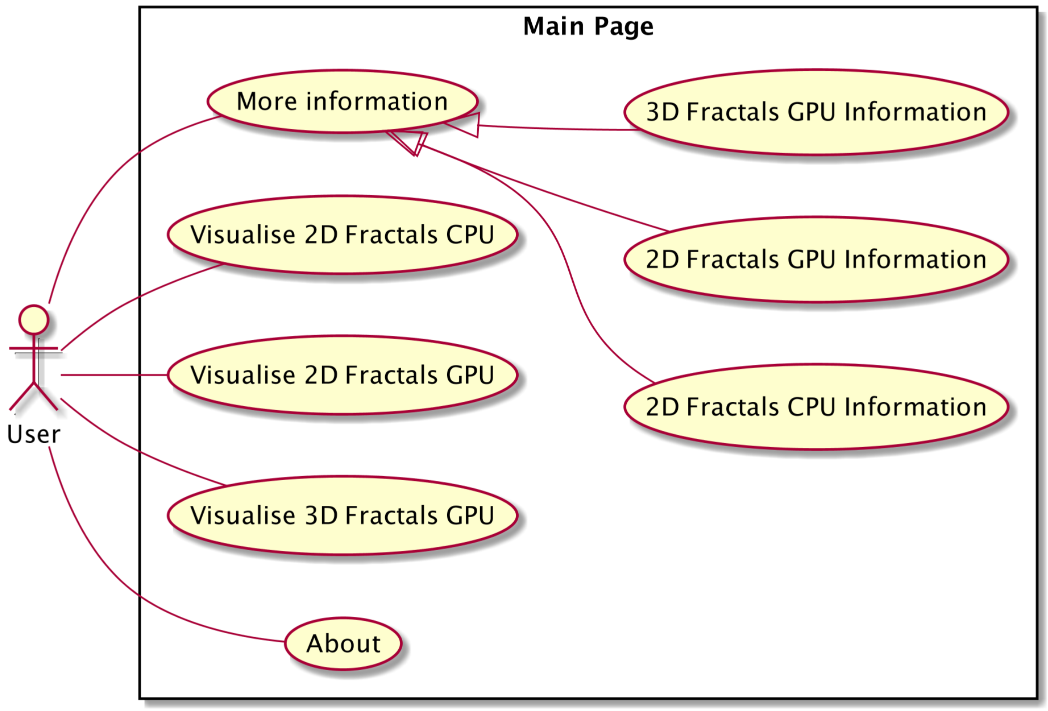

Main Page
More information
3D Fractals GPU Information
Visualise 2D Fractals CPU
2D Fractals GPU Information
Visualise 2D Fractals GPU
2D Fractals CPU Information
User
Visualise 3D Fractals GPU
About

Figure 6.1: Use Cases Diagram for the Main Page

- **Visualise 2D Fractals CPU:** This user story should drive the User to visualise 2D Fractals using the CPU.
- **Visualise 2D Fractals GPU:** This user story should drive the User to visualise 2D Fractals using the GPU.
- **Visualise 3D Fractals GPU:** This user story should drive the User to visualise 3D Fractals using the GPU.

- **More information:** This use case will provide information about each main user story about visualisation.
- **About:** This use case will show a small *about* prompt to the User.

It is important to note that, since the 3D requires much more calculations to be displayed, this problem only consider them to be rendered using the GPU due to; generally; it has more computation power than a CPU.

Nonetheless, we can consider both approaches regarding the 2D Fractals, this is CPU and GPU, since it usually takes less computational cost to render them.

2D Fractals on CPU

You can see the Use Cases Diagram for this on Figure 6.2. Let us specify the main use cases on this page.

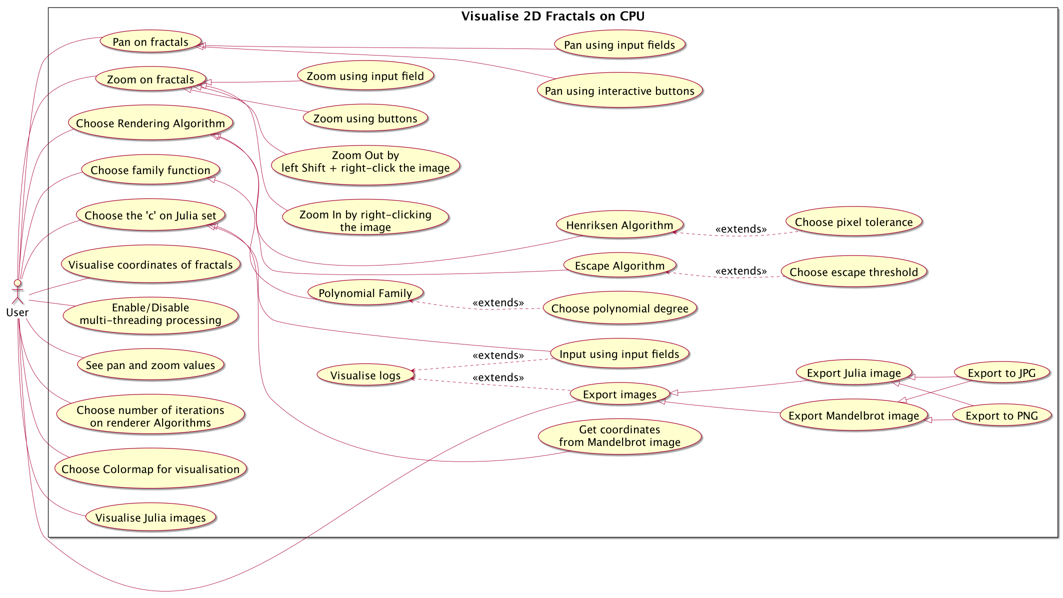


Figure 6.2: Use Cases Diagram for 2D Fractals on CPU Page

- **Pan on fractals:** This user story should allow the User to interact with the fractal (both Mandelbrot and Julia), either using interactive buttons or filling input fields, in order to *navigate* through the fractal vertically and/or horizontally.
- **Zoom on fractals:** This use case should allow the User to interact with the fractal (both Mandelbrot and Julia), either using interactive buttons, Right-Clicking with the mouse the image to Zoom In (resp. Left Shift + Right-Clicking the image to Zoom Out) or filling an input field, in order to apply *zoom in* or *out* into the fractal.

- **See pan and zoom values:** This user story allows the User to visualise all time the current values for pan and zoom on both fractal images.
- **Choose Rendering Algorithm:** In this user story, the User will chose the algorithm that will render each of these sets. The rendering algorithms considered are the *Escape Algorithm* and *Henriksen algorithm*.
- **Choose number of iterations for renderer Algorithm:** This use case should allow the User to choose the maximum number of iterations for the current rendering algorithm.
- **Choose Family Functions:** This use case will allow the User to choose a *family of functions* that will be iterated in order to render fractals. Initially, only the polynomial family is considered.
- **Visualise coordinates of fractals:** This user story will allow the User to see the coordinates on the complex plane when the mouse is placed over a certain pixel of the resulting image.
- **Choose c on Julia Set:** Since you need a *seed* to draw Julia sets, this use case will provide the User a way to choose this parameter c, either clicking on the Mandelbrot image or filling input fields.
- **Visualise logs:** During the interaction between the User and the software, we should provide some logs in order to produce feedback to the User. These logs will be fed when
  - a drawing is started,
  - a drawing is finished (providing also the amount of time it took to complete it) or stopped,
  - the User fill an input field with an incorrect value (meaning "incorrect" an invalid value here, such as placing a negative number on a only positive field) or a not-a-number value,
  - exporting an image it should feedback with the status of this operation.
- **Enable/Disable multi-threading processing:** This user story should allow the User to enable or disable an option to render these sets using only a single thread or using all CPU power available with multi-thread.
- **Choose Colormap for visualisation:** This user story will allow the user to choose the Colormap that will fill the Fractal images.
- **Visualise Julia images:** This user story will allow the user to see the first n iterates over the Julia image given a certain point (that the user will choose with the cursor). This feature is interesting in order to find "empirically" periodic orbits. See Definition 1.5.
- **Export images:** This user story should allow the User to export the resulting image of the rendering to a static image, for both Mandelbrot and Julia Sets. The exporting formats considered are `PNG` and `JPG`.

2D Fractals on GPU

This page is similar to the last one, however there are some differences. The Use Cases Diagram for this page is on Figure 6.3. Since many use cases in this page are identical to the previous one, we only specify here the different ones (they are the green ones on the Figure).

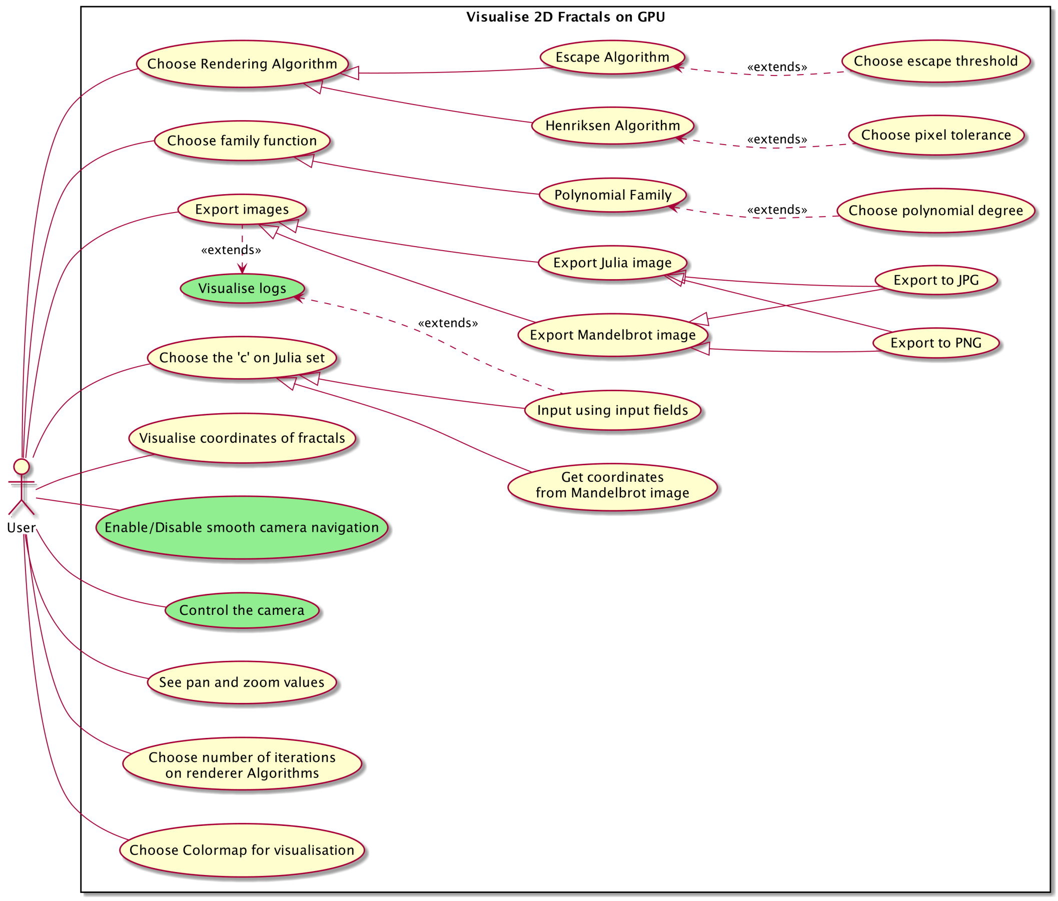


Figure 6.3: Use Cases Diagram for 2D Fractals on GPU Page

- **Control the camera:** This user story should allow the User to interact with the fractal (both Mandelbrot and Julia) in order to *navigate* through the fractal using the mouse and the keyboard. The scroll wheel from the mouse will handle the *zoom* of the fractal, while the arrows of the keyboard will handle the *pan* of the fractal.

- **Visualise logs:** During the interaction between the User and the software, we should provide some logs in order to produce feedback to the User. These logs will be fed when

- – the User fill an input field with an incorrect value (meaning "incorrect" an invalid value here, such as placing a negative number on a only positive field) or a not-a-number value,
  - – exporting an image it should feedback with the status of this operation.

- **Enable/Disable smooth-camera navigation:** This user story should allow the User to enable or disable an option that changes the behaviour of the camera when the User navigates through the fractal. If this option is enabled, the camera performs an interpolation at each step giving to the User the sensation of a smooth movement. If this option is disabled, the camera behaves without any interpolation at all.

3D Fractals on GPU

Finally, the Use Cases Diagram for this page is Figure 6.4. Let us detail the main use cases for this page.

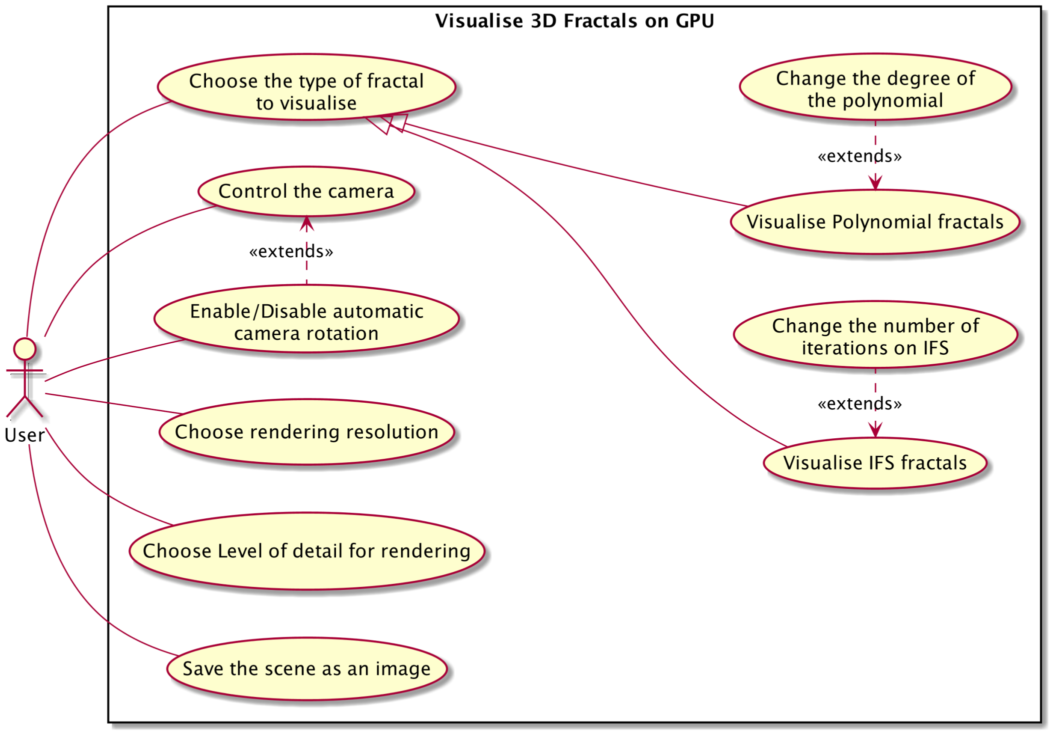

Visualise 3D Fractals on GPU
Choose the type of fractal to visualise
Change the degree of the polynomial
«extends»
Control the camera
Visualise Polynomial fractals
«extends»
Enable/Disable automatic camera rotation
Change the number of iterations on IFS
«extends»
User
Choose rendering resolution
Visualise IFS fractals
Choose Level of detail for rendering
Save the scene as an image

Figure 6.4: Use Cases Diagram for 3D Fractals on GPU Page

- **Control the camera:** In a similar way to 2D Fractals GPU, this use case should give to the User the ability of control the camera using the arrows the keyboard to move the camera around the fractal and, use the mouse to change the orientation of the camera in the scene.

- **Choose rendering resolution:** This user story will allow the User to change the resolution in which the resulting scene is displayed.
- **Choose Level of Detail for rendering:** This use case should allow the User to choose the amount of detail to be displayed on the fractal.
- **Choose the type of fractal to visualise:** This user story will allow the user to select which type of fractal is going to be displayed. Initially, two types of fractals are considered, that are polynomial-based fractals (in which case the User will be able to choose its degree) and an IFS fractal (in which case the User could choose the number of IFS iterations).
- **Enable/Disable automatic camera movement:** This feature will let the User to toggle between an automatic camera rotation and a static one.
- **Save the scene as an image:** This user story allows the user to save the renderer scene to a static image as `PNG` format.

6.1.2 Non-Functional Requirements

Once we have specified the Functional Requirements, we detail the Non-Functional Requirements. These are related to the platforms where this software will be deployed, its controls, the project maintenance, the extensibility of the code, its reusability, etc.

Since the aim of this project has academic and lecture finalities, it is important to present a software reachable by as many Users as possible. Due to this, the main Non-Functional Requirement for this software is its platform. For instance, a Web deployment would do the work for this purpose.

Therefore, the main interaction with it will be using a keyboard and a mouse (or trackpad) on a Desktop computer, for instance.

Since this exploration topic about fractals is very large, we need a software as extensible as possible, so we can easily add new requested features from User feedback, or rework an existing one.

Speaking about reusability, this is always a nice to have requirement since it makes the development more easily and robust, hence reusability provides stronger and more reliable code.

6.2 Antecedents

In this section, we will provide a review about the existing fractal visualisation software and techniques for both 2D Fractals and 3D Fractals. Then we will compare some of the available software that draws fractals and, finally, place conclusions about the motivation of this project.

6.2.1 State-of-the-Art: 2D Fractal Visualisation

Let us start with 2D Fractals. This subsection is based on the work from [13].

Formerly, we place here just a recall that we are only interested on visualising the fractals resulting from iterating complex polynomials on the complex plane. In particular, we are interested on drawing the Mandelbrot (and Multibrot) sets and Julia sets for the polynomial family $P_c(z) = z^d + c$, with $z, c \in \mathbb{C}$ and $d \geq 2$. Therefore, this state of art will be focused on the drawing of these particular sets.

There are plenty of algorithms that are capable about plotting Multibrot and Julia Sets. Some of them rely on a very simple mathematical formalisation and have exceptional results in terms of accuracy and performance.

However, if one dig deeper on these mathematical formalisation, one can find even better -in many cases in terms of accuracy rather than performance- algorithms capable to draw these sets.

Let us explain a couple of rendering algorithms.

1. **Escape Algorithm:** This is, probably, the best known rendering algorithm for plotting Mandelbrot and Julia Sets. This is one of the simplest algorithm as well. In the case of Mandelbrot (and Multibrot) set, it relies on the Theorem 4.2 in Section 4.2.

   In general terms, this algorithm iterates across each pixel of a given image and computes $z^d + c$, starting at $z = 0$ and where c is the point given by the double loop iteration through the whole image. These algorithms keeps iterating until a maximum number of iterations has reached or until the complex number z has reached a module greater than an arbitrary number (greater than 2).

   The aforementioned works only when drawing the Mandelbrot set. However, this same algorithm admits a variation in order to draw Julia sets. This modification rely on the same structure but in this case we start at z the point given by the double loop iteration and now the c is a given fixed seed.

2. **Inverse Iteration Method (IIM):** This algorithm only plots Julia Sets. The approach for this algorithm is different from the previous one, since this algorithm draws simple points. This means that this does not iterate across all the pixels of an image but this only paint a few pixels of an image. This algorithm is based on the results that the Julia set is $\mathcal{J}(P_c) = \overline{\{P_c^{-n}(z_0) : n \in \mathbb{N}\}}$. In other words and let us fix $d = 2$ (this is a complex quadratic polynomial for simplicity on the exposition), then the Julia set for this polynomial is the set of $z_n = \pm\sqrt{z_{n-1} - c}$. However, it is important to note that the number of iterated pre-images grows exponentially, so this is not feasible computationally. This is the reason why we usually at each step we choose, by random, one of the pre-images of P_c.

Although, there are many other algorithms. For instance, the *escape angle algorithm*, the *Curvature Estimation*, using *Statistics*, the *Orbit Traps Method* and *Gaussian Integers Algorithm* to name a few [14].

Finally, it is worth mentioning that one can enhance almost any of the aforementioned algorithms by applying a *distance estimation* [15]. This technique is based on estimate the distance of a current pixel to the boundary of the set.

6.2.2 State-of-the-Art: 3D Fractal Visualisation

Let us review the current state of art regarding 3D Fractal Visualisation. The references for this subsection are on [16].

Unlike 2D Fractal Visualisation, where each type of algorithm requires a certain rendering technique, when one try to render 3D Fractals in real time, especially using the GPU, you have no other choice but use the "traditional" rendering techniques. These are, mostly, *Z-Buffer, Ray-Tracing* and *Ray-Marching.*

This is driven by the fact that, when visualising scenes using the GPU, these chipsets are specialised computing using the *Single Instruction Multiple Data (SIMD)* mode. With this, most part of the rendering algorithm for 3D fractal visualisation are intrinsic to them.

Let us review each one of these techniques. Generally, the main difference between them resides on the degree of realism achieved and its performance capability (in terms of *frames per second* or *FPS*).

1. **Z-Buffer [17] :** This technique is also known as *Depth Buffering.* The main purpose for this approach is to decide which elements of a renderer scene are visible and which are hidden. When applying this technique to our problem, the main concern here is to determine which points are visible (part of the fractal) and which not. In addition to this, one need to sample the fractal to decide *which* points will be sent to the rendering pipeline.

2. **Ray-Tracing [18]:** This technique and the next one are the most common ones used for this type of problem. This is because its algorithm matches really well with the problem to solve. This is a rendering technique for generating an image by tracing the path of light through pixels from a plane image and tries to *simulate* the effects of the intersection of these paths with virtual objects. With this technique, one is capable to obtain high detailed images with a very high degree of realism. However, this carries a great computational cost with it.

3. **Ray-Marching [19]:** This algorithm is a modification from Ray-Tracing. The main difference between them is the way the ray is sampled. Generally, Ray-Marching tries to ease the manner the *rays* are sampled across the scene with the finality to obtain more performance in sacrifice of image quality and degree of realism.

In addition, when using any of these rendering technique, one needs to calculate the *normal* associated to each point of the fractal in some way, since they are an essential component when colouring surfaces and objects on 3D scenes (as we have seen on Section 5.7). However, usually the objects used to represent these fractals does not have the normal associated with them.

Regarding the fractal colouring, we recall the formula defined on formula (5.1), typically it is used the diffuse component and the normal vector from the Blinn-Phong formula.

Finally, in order to get a better and more realistic approaches when rendering 3D Fractals, *shadowing* techniques are usually applied to obtain shadows in our result. See [20] for details.

6.2.3 Existing Software Comparison

Once we have done a state of the art for both 2D and 3D fractal visualisation, in this subsection we compare some of the existing software available on the Internet that draw fractals using computer graphics.

We are going to review the most important software available right not. Let us detail each considered software.

- **Mandel from Wolf Jung [21]:** This is an interactive program for drawing the Mandelbrot set and Julia sets and for illustrating and researching their mathematical properties. It is available on Desktop (MacOS, Linux and Windows). The main features are the ability of drawing the Mandelbrot and Julia sets and tweak some parameters, export results as images and it contains many functionalities to explore the mathematical properties of these sets. You can see an image of this software on Figure 6.5.

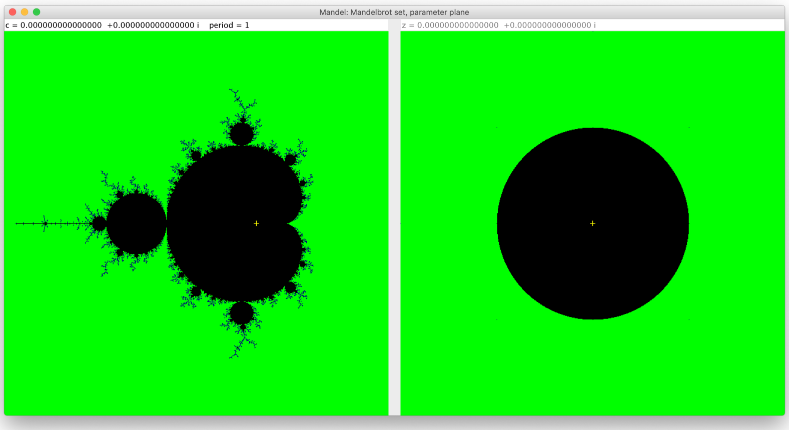
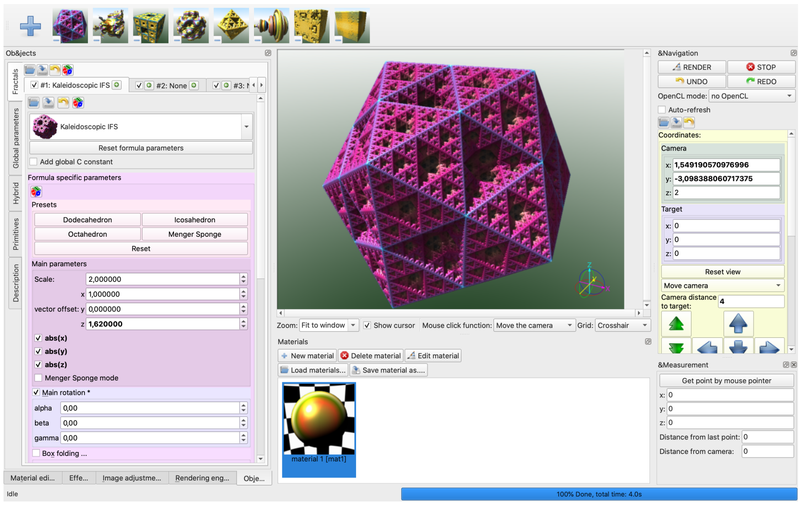

Mandel: Mandelbrot set, parameter plane
c = 0.000000000000000 +0.000000000000000 i period = 1
z = 0.000000000000000 +0.000000000000000 i

Figure 6.5: Mandel Software

- **Mandelbulber [22]:** This software renders 3D Fractals. In particular this software allows you to visualise trigonometric, hyper-complex, Mandelbox, IFS among other 3D fractals. It offers Desktop build (for MacOS, Linux and Windows) and its main features are high-performance computing with multiple graphics accelerators cards (multi-GPU support via OpenCL), it offers a rich GUI developed in Qt5 and plenty of options to decorate and enhance the final visualisation image. You can see an image of this software on Figure 6.6.



Fractal: Kaleidoscopic IFS
☐ Add global C constant

Main parameters
Scale: 2,000000
vector offset: x 1,000000
vector offset: y 0,000000
vector offset: z 1,620000
✓ abs(x)
✓ abs(y)
✓ abs(z)
☐ Menger Sponge mode
✓ Main rotation *
alpha 0,00
beta 0,00
gamma 0,00
☐ Box folding ...

Zoom: Fit to window ✓ Show cursor Mouse click function: Move the camera Grid: Crosshair

OpenCL mode: no OpenCL
☐ Auto-refresh
Coordinates:
Camera
x: 1,549190570976996
y: -3,098388060717375
z: 2
Target
x: 0
y: 0
z: 0
Move camera
Camera distance to target: 4

&Measurement
x: 0
y: 0
z: 0
Distance from last point: 0
Distance from camera: 0

Idle
100% Done, total time: 4.0s

Figure 6.6: Mandelbulber Software

- **FRAX [23]:** This is a software that renders 2D fractals. It offers pre-selected scenes where the user can zoom infinitely and change rendering values, such as texture and lightning, in order to enhance the final visualisation. This software offers only a mobile application on iOS. You can see an image of this software on Figure 6.7.

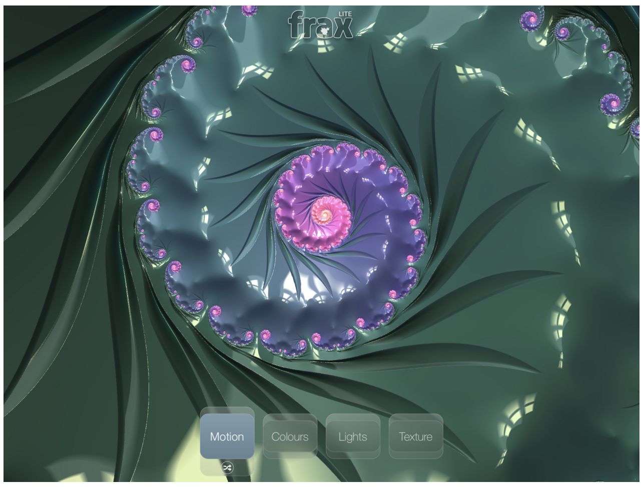


Figure 6.7: FRAX Software

- **XaoSjs [24]:** This is a version of the XaoS software based on Java-script and it offers an experimental support of visualising 2D Fracals, in particular the Mandelbrot set for the quadratic polynomial family, directly on the Web. You can see an image of this software on Figure 6.8.

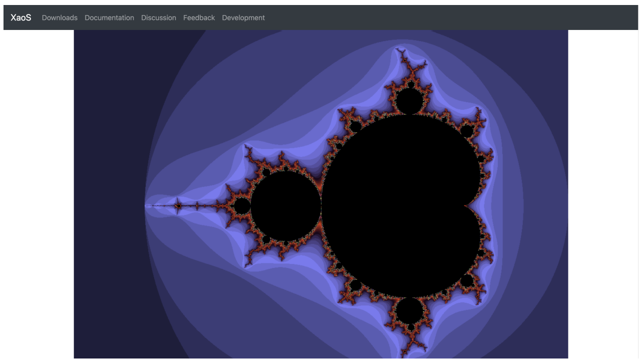


Figure 6.8: XaoSjs Software

6.3 Conclusions

In conclusion, we place a comparison Table as a summary of the existing software that tries to solve our problem on Table 6.1.

	Mandel	**Mandelbulb**	**FRAX**	**XaoSjs**
2D Fractal Visualisation	×		×	×
Mathematical properties	×			
3D Fractal Visualisation		×		
Availability on Desktop Platforms	×	×		
Availability on Mobile Platform			×	
Availability on Website				×

Table 6.1: Comparison Table among existing Fractal Visualisation Software

It is important to note that only one of them offers a Website version, however, it only renders 2D Fractals and it is on a very early stage of development.

So, we can conclude that there is an open line to explore a new development that allows to visualise fractals both 2D and 3D directly on the web giving the possibility of tweak parameters and that offers features to explore its mathematical properties.

Chapter 7

Proposal

In this chapter we place our proposal to solve the problem stated on the previous chapter.

We have developed this software using Unity3D [25]. On the one hand, it allows a perfect combination among visualising 2D and 3D graphics all on the same graphical user interface. On the other hand, Unity3D has a built-in engine that provides many tools to the developer to easily build software for computer graphics.

Figure 7.1: Unity3D Logo

Since stated on Section 6.1 an important feature for this software is to enhance it with tools that allows you to visualise its dynamical and mathematical properties behind them. This has been accomplished by letting the user click over the Julia set (or press 'P') in order to visualise the function image of the selected point. In addition, there is a slider that allows you to choose how many images will be calculated. With it, we can visualise periodic orbits.

In an aim to make this software more *engaging* to the User, it provides interactive buttons, input fields, sliders as well as a logs system. With this last, the User can immediately receive feedback at any change on the application. In addition to this, we bring many keyboard-bindings to build up the interaction with the system.

The exact user interaction is specified on Section 8.1.

As we had just analysed on subsection 6.2.3, there are several (and wide-ranged) applications available all over the Internet that allows you to visualise fractals, both 2D and 3D.

Nonetheless, almost all of them require you to download some software or an application in order to use them. Since, as we have aforementioned, one of the main requirement for this project is to present a software solution that lets you to draw fractals (both 2D and 3D) and interact with them (from simply changing the zoom to change the colormap of them), we have implemented an application with all packed that runs on a Website. So, one can simply browse to this site and start use this application from scratch. This website version has been deployed using WebGL technology [26].

However, since Unity allows easy deployment into many platforms, we could offer an enhanced version for Desktop (MacOS, Linux and Windows) and an application for mobile platforms (iOS and Android). The main benefit using these versions is a boost on the rendering speed for fractals drawn over CPU by implementing the rendering algorithms using multi-thread. Unfortunately, WebGL software does not allow the developer to use more than one main Thread.[1]

In the following sections we detail the strategy taken for both visualise 2D and 3D fractals.

7.1 Strategy of 2D visualisation

In this section we present the strategy of visualisation we take when visualising 2D Fractals. For this particular visualisation, two approaches were considered: 2D fractal visualisation using the CPU and its optimisations using the GPU.

7.1.1 Rendering 2D Fractals using the CPU

As we have aforementioned on Section 6.1, two different rendering algorithms were considered when visualising 2D Fractals: the *Escape Algorithm* and the *Henriksen Algorithm*.

In general, since we are rendering images, we need to define the *viewport* of the image. This is done by defining $x_{\min}, x_{\max}, y_{\min}, y_{\max}$ values that defines the size of the *window* to be rendered in world coordinates.

However, since we are drawing Mandelbrot and Julia sets, each of them needs a modification of these mentioned algorithms. Let us explain each of them.

Escape algorithm for Mandelbrot set

You can find the Escape Algorithm for Mandelbrot set on Algorithm 3 in Subsection 4.2.1.

We have modified this algorithm so we allow the user to choose the "escape" condition on Line 6. This parameter is called *threshold* and allow us to visualise what are called *equipotential curves*.

In addition, we have modified the final `if-else` condition, starting on line 11, in a way that we colour the pixel black if we have reached the maximum amount of iterations allowed. However, when otherwise, we have implemented a colour gradient to colorise the image in

[1]See [27] for details.

function of how many iterations we delayed to escape. To be precise, we have implemented five different colour maps.

Henriksen algorithm for Mandelbrot set

Here we have simply let the user to choose the value of *Tol*, representing the size of a pixel in a way that, the smaller this value is, the more defined will be the Mandelbrot set.

In addition, and similar to the Escape Algorithm for Mandelbrot set, we also have modified this algorithm when we colour the pixel white (on Line 11), allowing to the user to choose the colormap based on how many iterations did it last until it escaped (if it escaped).

You can find this algorithm on Subsection 4.3.1, Algorithm 4.

Escape algorithm for Julia Set

Since this algorithm is quite similar to the Mandelbrot Escape Algorithm, we have applied the same modifications to enhance its visualisation.

You can see the Escape Algorithm for Julia set on Algorithm 2 in Section 3.3.

Henriksen algorithm for Julia Set

You can find this last algorithm on Algorithm 1 in Section 2.3.

In a similar way as the Henriksen algorithm for Mandelbrot set, we have modified this algorithm to allow the user to input a custom *Tol* value representing the size of a pixel. In addition, we have also implemented here a colormap gradient based on how many iterates it took to escape (on Line 12).

7.1.2 Optimisations on GPU

Since we are drawing pixels over an image, there are many optimisations about the stated above using a GPU as the processor. As we previously detailed on Chapter 5, we use programs on the GPU in order to boost the drawing performance.

To be precise, we have implemented two different shaders to visualise 2D Fractals on the GPU. One for Mandelbrot Set and one for Julia Set.

When we tried to port the Henriksen Algorithm from CPU to a shader GPU we realised that, due its nature, it will be not feasible this conversion. Since this algorithm requires deeper precision than the escape algorithm implemented. So we dismissed it.

7.2 Strategy of 3D visualisation

In this section we present the strategy of visualisation we took when visualising 3D Fractals. In this particular case, we only considered rendering them using the GPU since it usually takes a great computational cost, as we aforementioned before.

As we explained on subsection 6.2.2, there are several ways to implement software that renders 3D Fractals.

However, the approach taken in this project is to use *ray-marching* as the rendering technique to visualise them. So, we go deeper on this topic on the next subsection.

7.2.1 *Ray-Marching*

This subsection is based on the work from [28].

To explain this algorithm, we need to define first a special kind of functions called *signed distance functions* or *distance estimators*.

Definition 7.1 (**Signed Distance Function (SDF) or Distance Estimator (DE)**). A *signed distance function* (SDF) or *distance estimator* (DE), is a function that returns the shortest distance between a point in space; given by its coordinates x, y, z; and some surface. The sign of the returned value indicates whether the point is inside or outside.

For example, let us consider a sphere centred at the origin. So, in this case, points inside the sphere will have a distance from the origin less than the radius, points on the sphere will have distance equal to the radius, and point outside the sphere will have distances greater than the radius.

For a sphere centred at the origin with radius 1, the Distance Estimator function look like this:

$$DE(x, y, z) = \sqrt{x^2 + y^2 + z^2} - 1.$$

Once we have defined the key function for this rendering technique, we are now ready to explain the basics of the algorithm.

In general, we select a point for the camera, put a grid in front of it, send rays from the camera through each point in the grid and we match each point to each pixel of the output image. In Figure 7.2 [46] you can see a diagram with the stated above.

Until this point, the algorithm is identical to *Ray-Tracing*, however, the difference with this other technique comes in *how* the scene is defined which, consequently, changes our approach for finding the *intersection* between the ray view and the scene.

In *ray-marching*, the entire scene is defined in terms of a distance estimator. Hence, in order to find the intersection between the view ray and the scene, we start at the camera, and move a point along the view ray, bit by bit. Then, at each step we ask "*Is this point inside the scene surface?*" or, in other words, "*Does the DE evaluate a negative number*

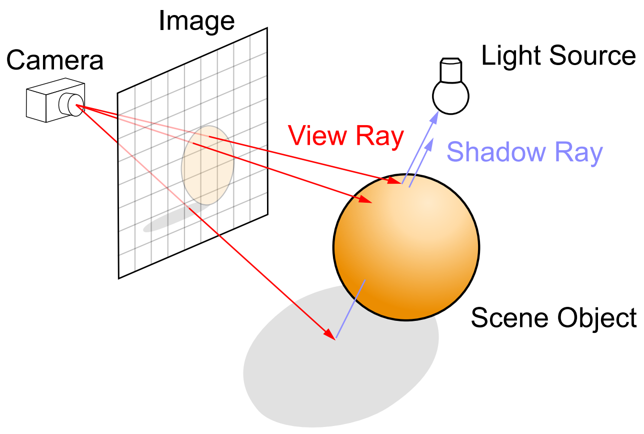


Figure 7.2: Ray-Tracing

at this point?" When this occurs, we hit something. If not, we keep going up until some maximum number of iterations has been reached along the ray.

One can easily note that it is important to define *how* to take these steps at each iteration. However, we can handle this using the *sphere tracing*, instead of taking a tiny step. This is, we take the maximum step we know is safe without going through the surface. So, we step by the distance to the surface, which the DE provided us. You can see an example on Figure 7.3 [47].

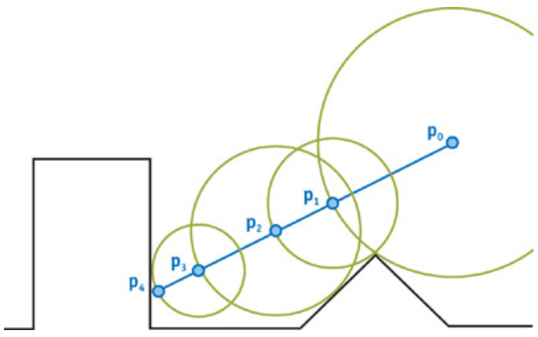


Figure 7.3: Sphere Tracing for Ray-Marching

In Figure 7.3, p_0 corresponds to the camera position and the blue line lies along the ray direction cast from the camera through the view plane. So, we keep going the algorithm until the current point is close to the surface.

7.2.2 Our implementation

Once we have detailed the main *ray-marching* algorithm, we are ready to place the algorithm implemented on this software project in order to render 3D Fractals. You can find this algorithm on Algorithm 5.

Algorithm 5: Ray-Marching Algorithm

```
Result: Returns the minimum distance to a surface.
Input: The lookFrom and lookAt from the camera, an integer MAX_STEPS and
       DE as a Distance Estimator function.
Output: The minimum distance to a surface.
1  $t \leftarrow 0$
2  $d \leftarrow 0$
3  $totalDistance \leftarrow 0$
4  $minimumDistance \leftarrow \epsilon$, $|\epsilon| << 1$
5  foreach $step = 0$ to $MAX\_STEPS$ do
       /* Sample $t$                                         */
6      $t \leftarrow lookFrom + totalDistance \cdot lookAt$
7      $d \leftarrow DE(t)$
8      $totalDistance \leftarrow totalDistance + d$
       /* If $t$ is close enough to the surface              */
9      if $d < minimumDistance$ then
10         return $d$
11     end
12 end
```

So, this simple algorithm gives us the *ray-marching* technique to render 3D fractals.

However, we need to define explicitly what $DE(t)$, or Distance Estimator, exactly is. It depends on the type of fractal we aim to visualise. As we have aforementioned on Section 6.1, we recall that we are interested on drawing two types of fractals here: the Mandelbulb, and *IFS* Fractals. Let us define the DE of each of them, we start with the Mandelbulb Distance Estimator on Algorithm 6.

Mandelbulb Distance Estimator

Here we present the Distance Estimator for the Mandelbulb set.

In this algorithm, we iterate from 0 to a maximum number of steps (defined by `MAX_STEPS`). We consider a 3-Dimensional point p and at each iteration we check if the length of r is greater than 5, in that case we can ensure that the point iterated does not belong to Mandelbulb set. If otherwise, we convert the coordinates of the point p to polar coordinates, so we can generalise the behaviour done with the Multibrot algorithm on 3-Dimensional space. This is, we multiply the polar coordinates by the degree d, the accumulator r powered by $d-1$ and the derivative plus 1 (this is a simply an application of the chain rule over a conjugacy of functions). Then we scale the point and apply the correspondent rotation and finally convert back to cartesian coordinates. At the end of this algorithm, we return the distance accumulated though this loop tweaked in order to represent the minimum distance from a given point to the Mandelbulb set. See [29] for details.

Algorithm 6: Mandelbulb Distance Estimator

```
Result: Returns the distance to the Mandelbulb of degree $d$.
Input: A point $p$, a degree $d \geq 2$ and an integer MAX_STEPS.
Output: The distance from $p$ to the Mandelbulb surface.

$z \leftarrow p$
$dr \leftarrow 1$
$r \leftarrow 0$
$\theta \leftarrow 0$
$\phi \leftarrow 0$
$s \leftarrow 0$
foreach $i = 0$ to MAX_STEPS do
    $r \leftarrow ||z||$
    if $r > 5$ then
        break
    end
    /* Convert the point to polar coordinates */
    $\theta \leftarrow \arccos(\frac{z.z}{r})$
    $\phi \leftarrow \arctan(\frac{z.y}{z.x})$
    $dr = r^{d-1} \cdot d \cdot dr + 1$
    /* Scale and rotate the point */
    $s \leftarrow r^d$
    $\theta \leftarrow \theta \cdot d$
    $\phi \leftarrow \phi \cdot d$
    /* Convert back to cartesian coordinates */
    $z \leftarrow s \cdot (\sin(\theta) \cdot \cos(\phi),\ \sin(\phi) \cdot \sin(\theta),\ \cos(\theta))$
    $z \leftarrow z + p$
end
return $0.5 \cdot \log(r) \cdot \frac{r}{dr}$
```

Let us specify the other distance estimator about IFS. We want to visualise two particular instances of these types of fractals: the *Sierpiński Tetrahedron* [30] and the *Menger Sponge* [31].

Sierpiński Tetrahedron Distance Estimator

Let us start with Sierpiński Distance Estimator Tetrahedron on Algorithm 7.

In this algorithm, we first define the four vertex of our initial tetrahedron and we iterate from 0 to `MAX_IFS_STEPS` calculating the distance between the given point p and each vertex.

At each iteration, we update the position of p and we move it towards the closer vertex to it. The more we iterate, the closer we get to one of these vertices. See [32] for more details.

Algorithm 7: Sierpiński Tetrahedron Distance Estimator

```
Result: Returns the distance to the Sierpiński Tetrahedron.
Input: A point $p$ and an integer MAX_IFS_STEPS.
Output: The distance from $p$ to the Sierpiński Tetrahedron.
$a_1 \leftarrow (1,1,1)$
$a_2 \leftarrow (-1,-1,1)$
$a_3 \leftarrow (1,-1,-1)$
$a_4 \leftarrow (-1,1,-1)$
$c \leftarrow 0$
$dist \leftarrow 0$
$d \leftarrow 0$
foreach $i$ = 0 to MAX_IFS_STEPS do
    $c \leftarrow a_1$
    $dist \leftarrow \text{length}(z - a_1)$
    $d \leftarrow \text{length}(p - a_2)$
    if $d < dist$ then
        $c \leftarrow a_2$
        $dist \leftarrow d$
    end
    $d \leftarrow \text{length}(p - a_3)$
    if $d < dist$ then
        $c \leftarrow a_3$
        $dist \leftarrow d$
    end
    $d \leftarrow \text{length}(p - a_4)$
    if $d < dist$ then
        $c \leftarrow a_4$
        $dist \leftarrow d$
    end
    $p \leftarrow 2 \cdot p - c$
end
return $\text{lenght}(p) \cdot 2^{-MAX\_IFS\_STEPS}$
```

Menger Sponge Distance Estimator

Finally, let us present the last algorithm for the Menger Sponge Distance Estimator on Algorithm 8.

In a similar way to Sierpiński Tetrahedron Distance Estimator, we define first a cube of length 10 centred at point p. Then we iterate from 0 to `MAX_IFS_STEPS`. At each step, we consider the cube created before and we create the a "cross" made by 3 infinite boxes. Finally, we substract this cross from our initial cube. Iterating this process we get the effect desired. See [33] for more details.

Algorithm 8: Menger Sponge Distance Estimator

```
Result: Returns the distance to the Menger Sponge.
Input: A point $p$ and an integer MAX_IFS_STEPS.
Output: The distance from $p$ to the Menger Sponge.
$da \leftarrow 0$
$db \leftarrow 0$
$dc \leftarrow 0$
$c \leftarrow 0$
$s \leftarrow 0.05$
$a \leftarrow 0$
$d \leftarrow sdBox(p, (10,\ 10,\ 10))$
foreach $i$ = 0 to MAX_IFS_STEPS do
    $a \leftarrow \mathrm{frac}(p \cdot s) - 0.5$
    $s \leftarrow s \cdot 3$
    $r \leftarrow \mathrm{abs}(1. - 6 \cdot \mathrm{abs}(a))$
    $da \leftarrow \max(r.x, r.y)$
    $db \leftarrow \max(r.y, r.z)$
    $dc \leftarrow \max(r.z, r.x)$
    $c \leftarrow \frac{\min(da, \max(db, dc)) - 1}{2 \cdot s}$
    if $c < d$ then
        $d \leftarrow c$
    end
    $p \leftarrow 2 \cdot p - c$
end
return $d$
Function sdBox($p$, $b$):
    $di \leftarrow \mathrm{abs}(p) - b$
    $mc \leftarrow \max(di.x, \max(di.y,\ di.z))$
    return $\min(mc, \mathrm{length}(\max(di,\ 0)))$
```

After all, it is important to note the power about this rendering technique. Since, with one simple algorithm of visualisation showed on Algorithm 5, we are able to visualise a wide variety of different fractals just changing the Distance Estimator.

7.3 Architecture of the System

In this section we detail the architecture of the system implemented in this software project. We provide the Class Diagram that will help us to understand the software engineering behind this solution. Since the whole Class Diagram is huge, we have separated it in different parts related by its behaviour or main goal.

As a general note and since we are working on Unity, most of the classes implemented inherit from `MonoBehaviour` class.

You can find the global simplified class diagram on Figure 7.4.

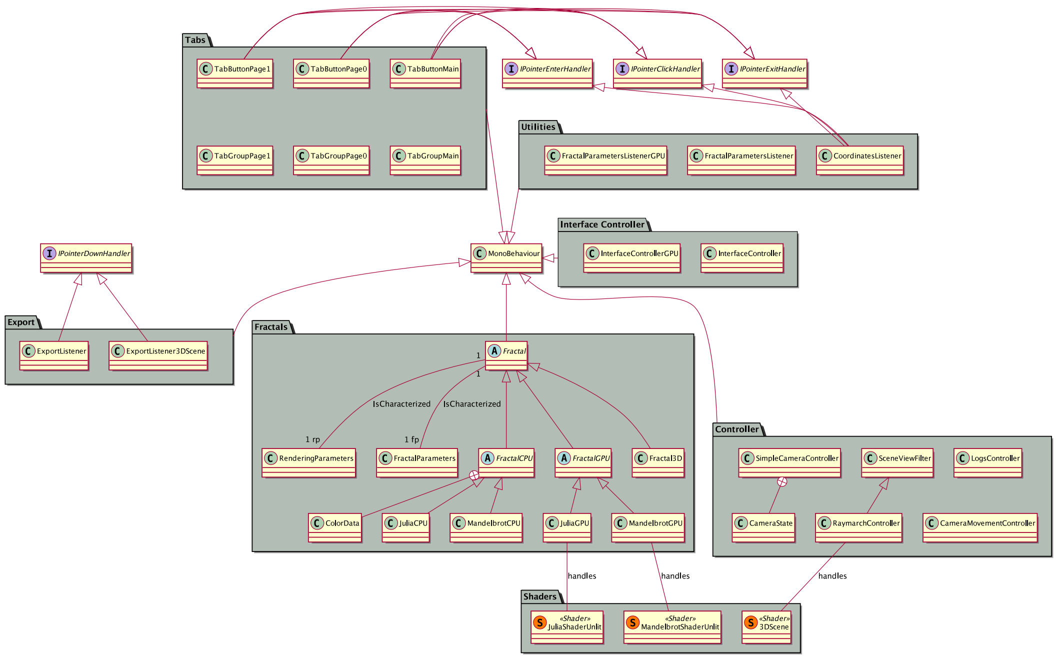


Figure 7.4: Class diagram showing the relation between the different packages

7.3.1 Fractals Package

Let us start with the Fractals Package. You can find a simplified class diagram for this on Figure 7.5. However, we provide the complete the class diagram on Figure A.1.

Let us detail a bit the most important classes in this diagram.

- `Fractal`: This is an abstract class that represents any fractal in our application. I.e., any fractal drawn with this application; either 2D or 3D fractal; inherits from this class. It also holds the main and common properties to represent a fractal. Hence, it contains an instance of `RenderingParameters` class, which contains all necessary data

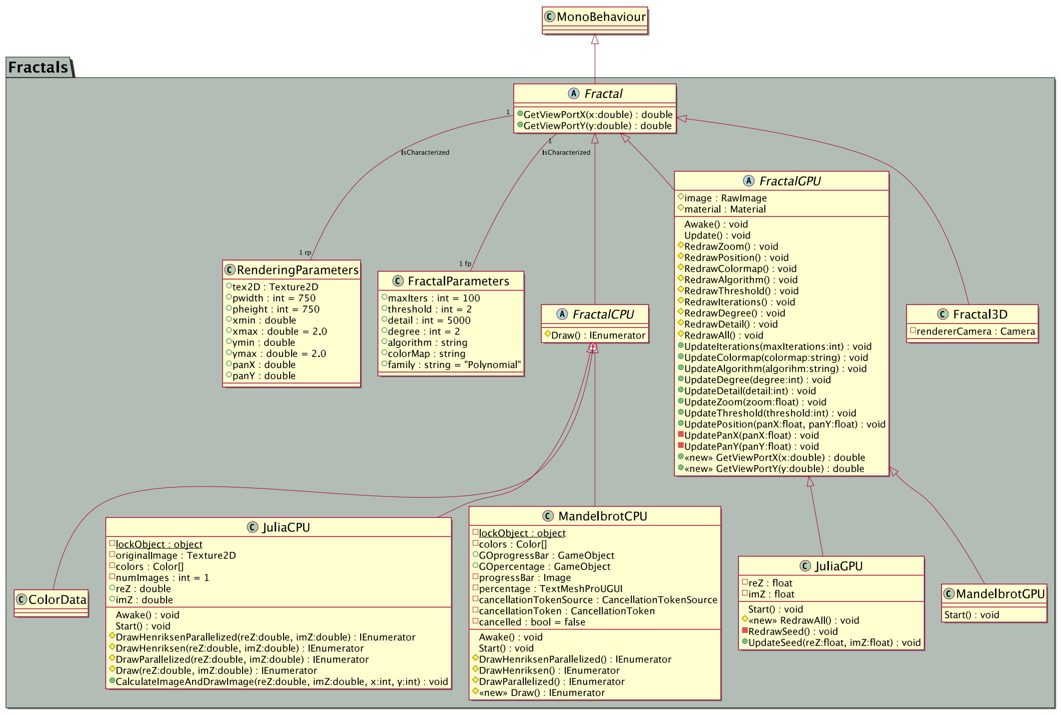

MonoBehaviour
Fractals
Fractal
GetViewPortX(x:double) : double
GetViewPortY(y:double) : double
isCharacterized
isCharacterized
1 rp
1 fp
RenderingParameters
tex2D : Texture2D
pwidth : int = 750
pheight : int = 750
xmin : double
xmax : double = 2.0
ymin : double
ymax : double = 2.0
panX : double
panY : double
FractalParameters
maxIters : int = 100
threshold : int = 2
detail : int = 5000
degree : int = 2
algorithm : string
colorMap : string
family : string = "Polynomial"
FractalCPU
Draw() : IEnumerator
FractalGPU
image : RawImage
material : Material
Awake() : void
Update() : void
RedrawZoom() : void
RedrawPosition() : void
RedrawColormap() : void
RedrawAlgorithm() : void
RedrawThreshold() : void
RedrawIterations() : void
RedrawDegree() : void
RedrawDetail() : void
RedrawAll() : void
UpdateIterations(maxIterations:int) : void
UpdateColormap(colormap:string) : void
UpdateAlgorithm(algorihm:string) : void
UpdateDegree(degree:int) : void
UpdateDetail(detail:int) : void
UpdateZoom(zoom:float) : void
UpdateThreshold(threshold:int) : void
UpdatePosition(panX:float, panY:float) : void
UpdatePanX(panX:float) : void
UpdatePanY(panY:float) : void
«new» GetViewPortX(x:double) : double
«new» GetViewPortY(y:double) : double
Fractal3D
rendererCamera : Camera
ColorData
JuliaCPU
lockObject : object
originalImage : Texture2D
colors : Color[]
numImages : int = 1
reZ : double
imZ : double
Awake() : void
Start() : void
DrawHenriksenParallelized(reZ:double, imZ:double) : IEnumerator
DrawHenriksen(reZ:double, imZ:double) : IEnumerator
DrawParallelized(reZ:double, imZ:double) : IEnumerator
Draw(reZ:double, imZ:double) : IEnumerator
CalculateImageAndDrawImage(reZ:double, imZ:double, x:int, y:int) : void
MandelbrotCPU
lockObject : object
colors : Color[]
GOprogressBar : GameObject
GOpercentage : GameObject
progressBar : Image
percentage : TextMeshProUGUI
cancellationTokenSource : CancellationTokenSource
cancellationToken : CancellationToken
cancelled : bool = false
Awake() : void
Start() : void
DrawHenriksenParallelized() : IEnumerator
DrawHenriksen() : IEnumerator
DrawParallelized() : IEnumerator
«new» Draw() : IEnumerator
JuliaGPU
reZ : float
imZ : float
Start() : void
«new» RedrawAll() : void
RedrawSeed() : void
UpdateSeed(reZ:float, imZ:float) : void
MandelbrotGPU
Start() : void

Figure 7.5: Class Diagram for Fractals Package

to determine the drawing parameter of a 2D fractal; such as the *panX* and *panY*, the *zoom* given by *xmin, xmax, ymin, ymax* as well as the *Texture2D* that holds the image itself, among others. It also contains a `FractalParameters` instance that stores all the data related to the fractal itself (that are not rendering parameters), for instance, the *maxIters* for the algorithms, the *algorithm* chosen or the *degree* of the polynomial are stored here. Finally, this class provides a couple of methods in order to obtain the real and imaginary part over the complex plane given two coordinates x and y from the image.

- `FractalCPU`: This is an abstract class that represents a 2D Fractal drawn by CPU. However, since we want to draw Mandelbrot and Julia sets, this class provides a common point for the classes that actually implements the drawing algorithms. In order to provide code extensibility, it implements an abstract method called `Draw()` that any child class must implement. With it, we ensure that any class that inherits from this will be a drawing class. In addition, this class also stores the `ColorData` class as a way to provide gradient colouring when drawing.

- `MandelbrotCPU`: This class inherits from `FractalCPU` and implements the most important method: `Draw()`. With this method a Mandelbrot set is drawn using the escape algorithm. Furthermore, there are other *Draw* methods implemented in this class. Concretely, there are four methods in total. One for each algorithm and one for each rendering mode (this is using multi-thread and single-thread).

- `JuliaCPU`: This class inherits from `FractalCPU` as well and, hence, implements the `Draw()` method. However, in this case; as we have seen on Algorithm 1 and 2; we need to use the *seed* in order to draw the Julia set. In addition, this class also implements four *Draw* methods in total, one for each algorithm and one for each rendering mode. Finally, this class also implements the `CalculateImageAndDrawImage()` method that calculates the image of a given point and draws a line over the Julia set image.

- `FractalGPU`: On the other hand, this class represents a 2D fractal drawn using the GPU. Since the fractals drawn by a GPU are rendered using *shaders*, it implements the basic methods to update each needed rendering parameter.

- `MandelbrotGPU`: This is a simple instance of the `FractalGPU` class. Since the superclass implements all its needed methods, this class only initialises its values when the scene is started.

- `JuliaGPU`: This class inherits from `FractalGPU`. As long as this class is used to handle the data to draw the Julia set on GPU, using the methods from the parent class is almost enough. Nonetheless, it needs to implement one additional method to handle the *seed* of each Julia set.

- `Fractal3D`: Finally, this class represents a 3D Fractal. It also controls the minimal interface on the scene where this fractal is rendered. The most important part on this class is that it holds an auxiliary camera which renders the fractal into a Texture. This Texture is the object that we visualise on the scene.

As a general commentary, it is important to note that the implementation of all drawing methods are implemented using Unity *coroutines*. This provides us a way to send the workload to the background while maintaining the GUI responsive. Moreover, on the cases of parallel processing, we have used Unity *Tasks* as well as *coroutines*.

7.3.2 Interface Controller Package

A simplified Class Diagram for Interface Controller classes is on Figure 7.6. You can find the complete class diagram on Figure A.2.

Let us detail a bit the most important classes on this package.

- `InterfaceController`: This is the main controller class on the 2D CPU scene. It handles all the input data from the User as well as it updates all the data manipulation through text labels, sliders, dropdown menus and input fields into the screen.

- `InterfaceControllerGPU`: In a similar way as the previous item, this handles all the input data and its text labels, sliders, dropdown menus and input fields into the screen on the 2D GPU scene. We needed to create a separate script since the 2D CPU scene and 2D GPU do not require the same methods and class attributes.

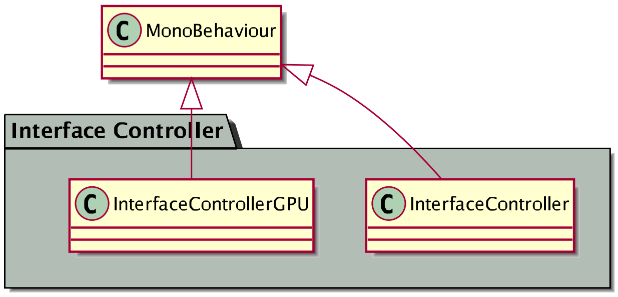


Figure 7.6: Class Diagram for Interface Controller Package

7.3.3 Controller Package

A simplified Class Diagram for Controller classes is on Figure 7.7. However, you can find the complete class diagram on Figure A.3.

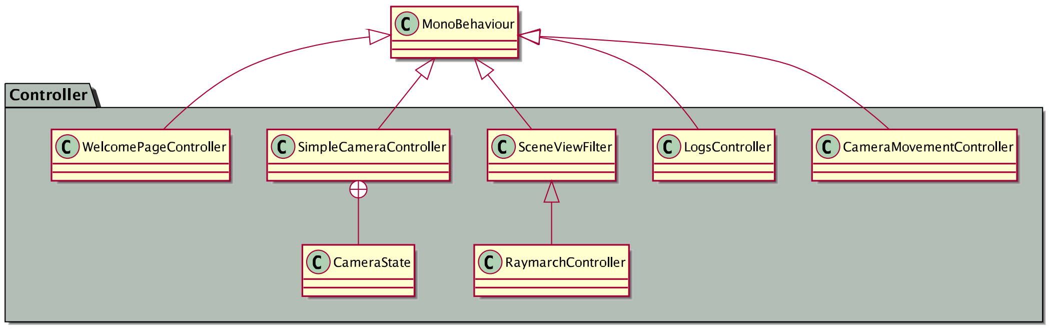


Figure 7.7: Class Diagram for Controller Package

Let us detail a bit the most important classes on this package.

- `CameraMovementController`: This class handless all the input motion data from the user when the mouse is over the Mandelbrot or Julia image on the 2D GPU scene. In order to do this, it modifies the required attributes on `MandelbrotGPU` and `JuliaGPU` classes.

- `LogsController`: This class provides static methods to access the Logs system of this application from anywhere.

- `SimpleCameraController`: This class allows to the User to move the transform of Camera object using WASD or the arrows on a keyboard. It also rotates the camera if the User hold the right click pressed and drags around the screen. Finally, the scroll allows the user to choose the movement speed of the camera. This class uses the `CameraState` class as the class that actually modifies the Camera transform.

- `RaymarchController:` This class inherits from `ScreenViewFilter` class. With it, here is where parameters for the 3D fractal shader are updated.
- `WelcomePageController:` This class handles all the logic about the Welcome Page of the application.

7.3.4 Utilities Package

Let us continue to a simplified Class Diagram about Utilities classes on Figure 7.8. You can find the complete class diagram on Figure A.4.

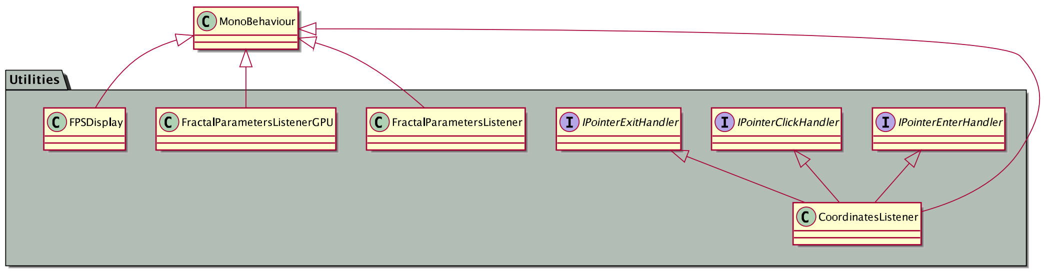


Figure 7.8: Class Diagram for Utilities Package

Let us detail a bit the most important classes on this package.

- `Coordinates Listener:` This class implements the interfaces `IPointerEnterHandler`, `IPointerClickHandler` and `IPointerExitHandler` and it is attached to an image. Thanks to this, this class allows the user to visualise the coordinates of the Mandelbrot and Julia sets on both 2D CPU and 2D GPU scenes by mouse hover over the image.
- `FractalParametersListener:` This class manages all the dropdown menus, input fields and sliders regarding the fractal parameters. For instance, the dropdown menu for choosing the algorithm, the slider to choose the maximum iterations or the input field to specify the degree.
- `FractalParametersListenerGPU:` Since the 2D GPU scene is slightly different to the 2D CPU scene, we needed to create another script in order to manage its fractal parameters in this case.
- `FPSDisplay:` This class calculates the FPS achieved on both 2D GPU and 3D GPU scenes and display this result into a label on the scene.

7.3.5 Tabs Package

A simplified Class Diagram for this package is on Figure 7.9. You can find the complete class diagram on Figure A.5.

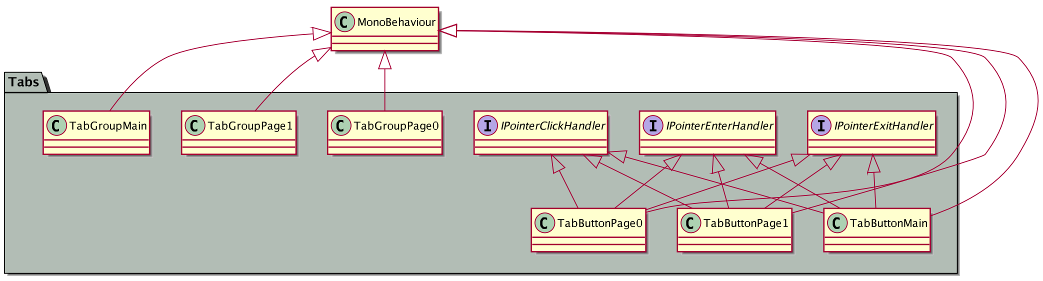


Figure 7.9: Class Diagram for Tabs Package

Since Unity does not provide a native tab system solution, we needed to implement one. This implementation of tab system is very extensible since it does not constrain you on the maximum number of tabs. Furthermore, due the implementation of the methods in `IPointerEnterHandler`, `IPointerClickHandler` and `IPointerExitHandler`, it provides a way to give feedback to the user when a tab is selected, hovered or deselected.

7.3.6 Export Package

Let us move on the next Class Diagram about exporting scenes into images. You can see a simplified version on Figure 7.10 and a complete version on Figure A.6.

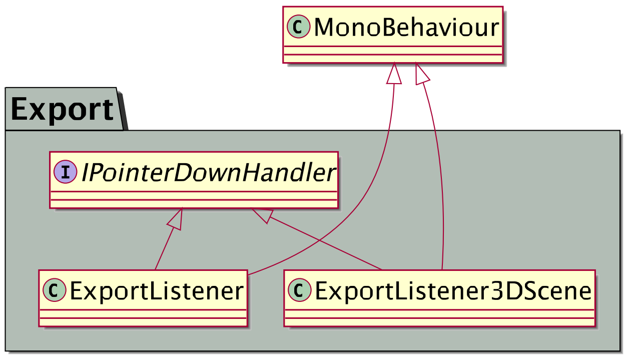


Figure 7.10: Class Diagram for Export Package

- `ExportListener:` This class is attached to a button on the screen and it provides the necessary methods to store the result from a Texture2D into an image and store it into the hard drive of the current platform. This class is used on both 2D CPU and 2D GPU scenes.
- `ExportListener3DScene:` Due to technical reasons, we had to use another separated class to port the exact same functionality as above to the 3D scene.

It is important to note that we have used 3^{rd} party Assets in order to store images from WebGL, Desktop and mobile platforms iOS and Android.

We have used *UnityStandaloneFileBrowser* asset [48] to store images from WebGL and Desktop, and we have used *unity-native-toolkit* asset [49] to store images from mobile platforms into the camera roll.

7.3.7 Shaders Package

In this package we store all the shaders implemented in this project. See Figure 7.11 to see a Diagram with the shaders implemented.

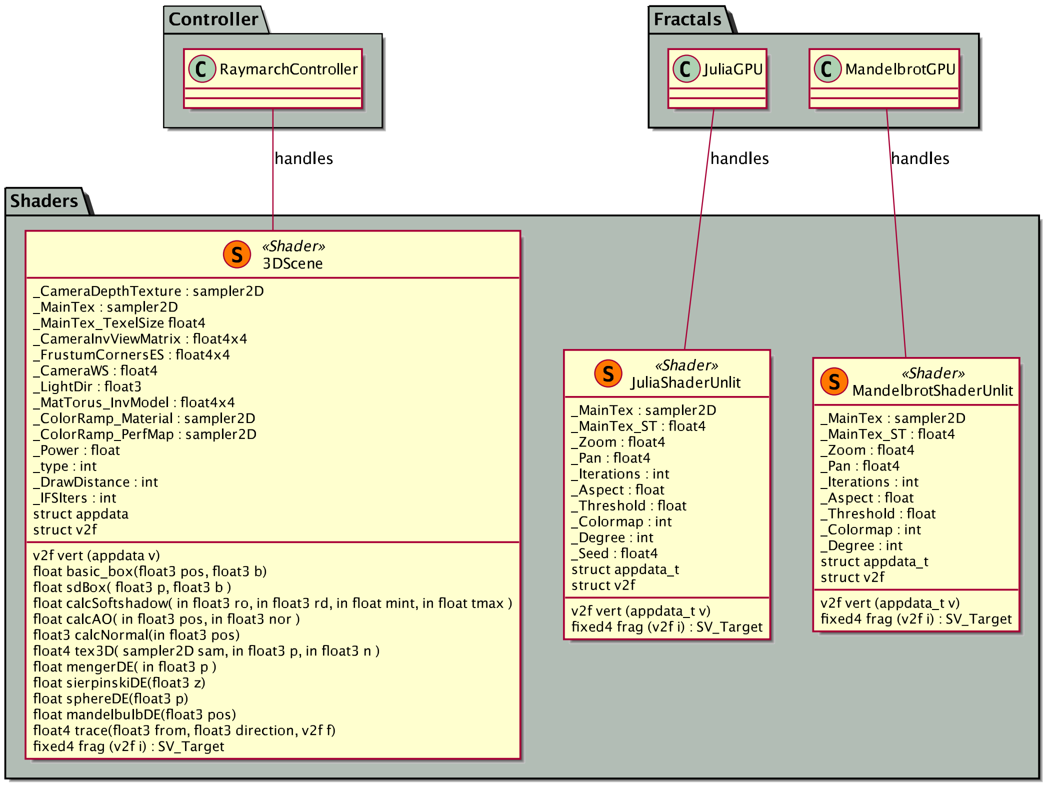

Controller
C RaymarchController
Fractals
C JuliaGPU
C MandelbrotGPU
handles
handles
handles
Shaders
S «Shader» 3DScene
_CameraDepthTexture : sampler2D
_MainTex : sampler2D
_MainTex_TexelSize float4
_CameraInvViewMatrix : float4x4
_FrustumCornersES : float4x4
_CameraWS : float4
_LightDir : float3
_MatTorus_InvModel : float4x4
_ColorRamp_Material : sampler2D
_ColorRamp_PerfMap : sampler2D
_Power : float
_type : int
_DrawDistance : int
_IFSIters : int
struct appdata
struct v2f
v2f vert (appdata v)
float basic_box(float3 pos, float3 b)
float sdBox(float3 p, float3 b)
float calcSoftshadow(in float3 ro, in float3 rd, in float mint, in float tmax)
float calcAO(in float3 pos, in float3 nor)
float3 calcNormal(in float3 pos)
float4 tex3D(sampler2D sam, in float3 p, in float3 n)
float mengerDE(in float3 p)
float sierpinskiDE(float3 z)
float sphereDE(float3 p)
float mandelbulbDE(float3 pos)
float4 trace(float3 from, float3 direction, v2f f)
fixed4 frag (v2f i) : SV_Target
S «Shader» JuliaShaderUnlit
_MainTex : sampler2D
_MainTex_ST : float4
_Zoom : float4
_Pan : float4
_Iterations : int
_Aspect : float
_Threshold : float
_Colormap : int
_Degree : int
_Seed : float4
struct appdata_t
struct v2f
v2f vert (appdata_t v)
fixed4 frag (v2f i) : SV_Target
S «Shader» MandelbrotShaderUnlit
_MainTex : sampler2D
_MainTex_ST : float4
_Zoom : float4
_Pan : float4
_Iterations : int
_Aspect : float
_Threshold : float
_Colormap : int
_Degree : int
struct appdata_t
struct v2f
v2f vert (appdata_t v)
fixed4 frag (v2f i) : SV_Target

Figure 7.11: Diagram for Shaders Package

Let us specify each shader.

- `MandelbrotShaderUnlit`: This shader implements the optimisation for drawing Mandelbrot sets over the GPU and it is handled by the `MandelbrotGPU` class. It features a simple vertex shader and a fragment shader (that implements the actual algorithm) all together on a single file. It is worth mentioning that this shader is based on a special type of shaders on Unity called *Unlit shaders*. The main particularity about them is that when colouring, they ignore the source (or sources) of light when computing the final colour. So we can freely choose the colour without light interferences. It stores values (that the `MandelbrotGPU` class updates) for the Zoom and Pan of the fractal, the maximum number of iterations allowed on the *Escape Algorithm*, the Degree of the polynomial, among others.

- `JuliaShaderUnlit:` This shader is similar to the previous one and it is handled by the `JuliaGPU` class. It is also based on the same *Unlit shader* family. However, the fragment shader implements the modified version of the *Escape Algorithm* that makes use of the Seed value to render it.

- `3DScene:` This shader packs all the implemented 3D fractal shader together on one file. It is handled by the `RaymarchController` class. The most important method is `trace` that is called by the fragment shader at each ray of the *ray-marching* algorithm. Then it features the four different Distance Estimators explained at Subsection 7.2.2, the `mandelbulbDE` for the Mandelbulb Set, `sierpinskiDE` for Sierpiński Tetrahedron, `mengerDE` for the Menger Sponge and `sphereDE` for the distance estimator of a sphere centred at origin with radius 1. In addition, it also implements different auxiliary methods in order to compute shadows and ambient occlusion for the Menger Sponge. This shader also stores all its needed rendering parameters for each of these fractals. For instance, the degree of the polynomial on `_Power` or the maximum amount of IFS iterations on `_IFSIters`.

Chapter 8

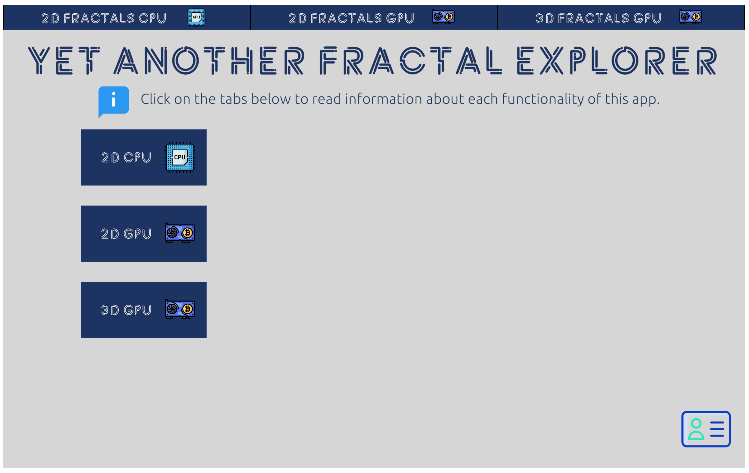
Results and Benchmarks

In this chapter we place many screenshots of the application and we explain what the user can do on each screen. Then, we do benchmarks testing the time it takes to render in the case of CPU rendering and the Frames Per Second (FPS) we can obtain in the case of GPU rendering.

8.1 Application

In this section we present some screenshots about the result of this software implementation.

You can see the welcome screen of the application on Figure 8.1.

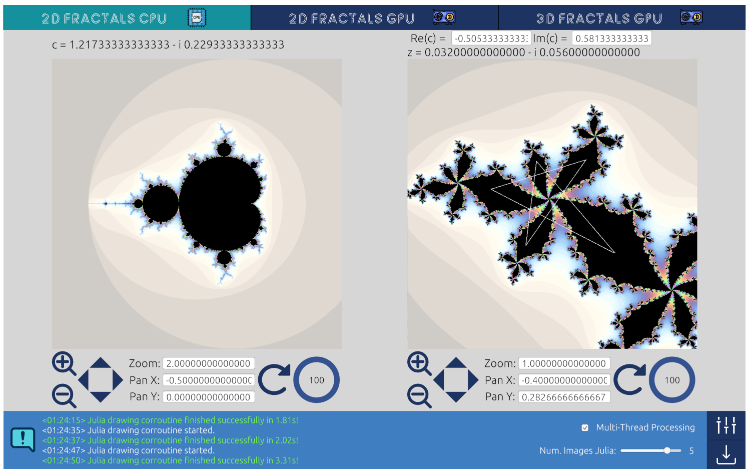


Click on the tabs below to read information about each functionality of this app.

Figure 8.1: Main Page of Yet Another Fractal Explorer

In this page you can click on the tabs at the centre of the screen in order to read some text explaining how to use the different scenes. On the bottom right you can click the button to toggle an about dialog. See Figure B.1.

We present on Figure 8.2 an example of finding empirically a periodic orbit of period 5 by clicking over the Julia set image. In addition you can hold 'P' while hovering over the image to do this.



c = 1.21733333333333 - i 0.229333333333333

Re(c) = -0.50533333333333 Im(c) = 0.581333333333
z = 0.03200000000000 - i 0.05600000000000

Zoom: 2.00000000000000
Pan X: -0.50000000000000
Pan Y: 0.00000000000000

Zoom: 1.00000000000000
Pan X: -0.40000000000000
Pan Y: 0.28266666666667

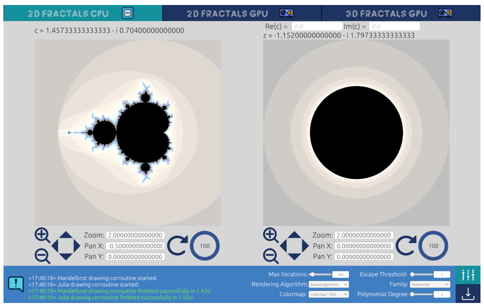
Num. Images Julia: 5

```
<01:24:15> Julia drawing corroutine finished successfully in 1.81s!
<01:24:35> Julia drawing corroutine started.
<01:24:37> Julia drawing corroutine finished successfully in 2.02s!
<01:24:47> Julia drawing corroutine started.
<01:24:50> Julia drawing corroutine finished successfully in 3.31s!
```

Figure 8.2: Finding periodic orbits

On both 2D CPU and 2D GPU scenes, you can press 'R' while the pointer is over an image to reset its rendering values.

Use WASD or movement arrows to pan through the fractal and use the mouse wheel to zoom the fractal on 2D GPU scene. Alternatively, you can press '-' to Zoom In and '-' + Right Shift to Zoom Out.



c = 1.45733333333333 - i 0.70400000000000

Re(c) = 0.0 Im(c) = 0.0
z = -1.15200000000000 - i 1.79733333333333

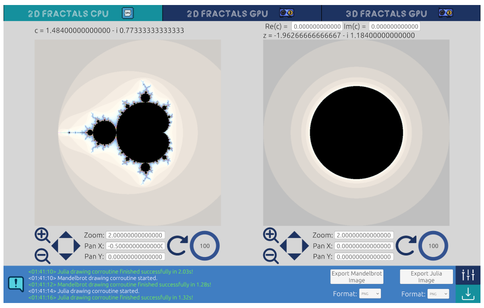
Zoom: 2.00000000000000
Pan X: -0.50000000000000
Pan Y: 0.00000000000000

Zoom: 2.00000000000000
Pan X: 0.00000000000000
Pan Y: 0.00000000000000

```
<17:40:18> Mandelbrot drawing corroutine started.
<17:40:18> Julia drawing corroutine started.
<17:40:19> Mandelbrot drawing corroutine finished successfully in 1.43s!
<17:40:19> Julia drawing corroutine finished successfully in 1.52s!
```

Max Iterations: 100
Rendering Algorithm: Escape Algorithm
Colormap: Colormap 1 (Sin)
Escape Threshold: 2
Family: Polynomial
Polynomial Degree: 2

(a) Fractal Parameters



c = 1.48400000000000 - i 0.77333333333333

Re(c) = 0.000000000000 Im(c) = 0.000000000000
z = -1.96266666666667 - i 1.18400000000000

Zoom: 2.00000000000000
Pan X: -0.50000000000000
Pan Y: 0.00000000000000

Zoom: 2.00000000000000
Pan X: 0.00000000000000
Pan Y: 0.00000000000000

```
<01:41:10> Julia drawing corroutine finished successfully in 2.03s!
<01:41:10> Mandelbrot drawing corroutine started.
<01:41:12> Mandelbrot drawing corroutine finished successfully in 1.28s!
<01:41:14> Julia drawing corroutine started.
<01:41:16> Julia drawing corroutine finished successfully in 1.32s!
```

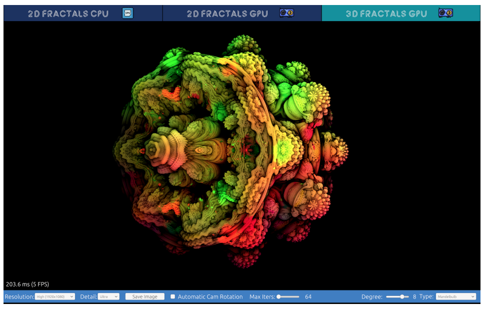
Format: PNG
Format: PNG

(b) Export into images buttons

Figure 8.3: Fractal Parameters and Export Options

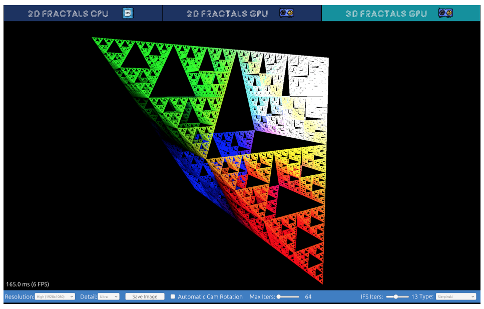
On Figure 8.3a, we can see the options that manipulates the fractals. We can see sliders and input fields to enter the maximum number of iterations; the escape threshold (in case we choose the Escape Algorithm) or the detail (in case we choose the Henriksen Algorithm); and the degree of the polynomial. In addition, we provide dropdown menus to choose the rendering algorithm; the colormap; and the family of functions.

Moreover, on Figure 8.3b, we can see the export buttons on bottom left side. This is implemented both on 2D CPU and 2D GPU scenes. You can save the images into PNG and JPG format. On 3D GPU scene you can find this button on the bottom centre of the screen.



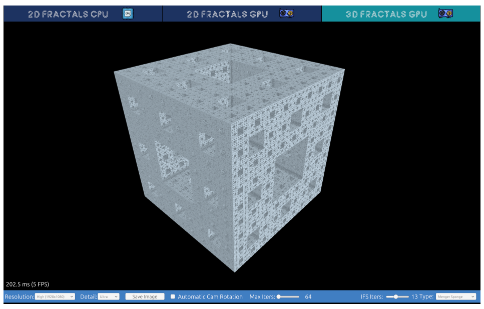
203.6 ms (5 FPS)
Resolution: High (1920x1080) Detail: Ultra Automatic Cam Rotation Max Iters: 64 Degree: 8 Type: Mandelbulb

(a) Mandelbulb for $z^8 + c$ rendering



165.0 ms (6 FPS)
Resolution: High (1920x1080) Detail: Ultra Automatic Cam Rotation Max Iters: 64 IFS Iters: 13 Type: Sierpinski

(b) Sierpiński Tethreaedon rendering



202.5 ms (5 FPS)
Resolution: High (1920x1080) Detail: Ultra Automatic Cam Rotation Max Iters: 64 IFS Iters: 13 Type: Menger Sponge

(c) Menger Sponge rendering

Figure 8.4: 3D Fractal Results

You can see on Figure B.4 a sample of different GPU rendering of Mandelbrot and Julia sets for $z^d + c$ for $d = 2, 3, 4, 5$.

Regarding the 3D GPU scene, we can see on Figure 8.4 the three different fractal implemented on this page. It is important to note that this scene has a minimal interface placed on the bottom of the screen. Containing; from left to right; the dropdown menus for choosing the Rendering Resolution and Level of detail, the button to save the result as image, a toggle that switch between static camera and automatic rotation, a slider that handles the maximum iterations of the ray-marching algorithm and finally, the type of fractal to render with its slider, allowing to manipulate the degree of the polynomial in case of Mandelbulb and the IFS Iterations in case of IFS fractals.

The controls in this scene are binded to WASD or movement arrows to move around the scene and you can hold the right-click of the mouse while dragging around to change the orientation of the camera. Finally, you can press 'R' anytime to reset the camera to its initial position.

Finally, let us do a brief comparison between the results between Escape Algorithm and Henriksen Algorithm when applying the same colormap. See differences between Figure B.2 and Figure B.3.

You can find on Annex C the minimum requirements for all the aforementioned compatible platforms to run this software.

8.2 Benchmarks

The following benchmarks have been made using a Macbook Pro Retina 15 inch, late 2013 under MacOS Catalina 10.15.5, using an i7 4850HQ 2.3GHz Quad-Core CPU [34] and an Nvidia Geforce GT 750M with 2GB of GDDR5 [35] and 16GB of RAM DDR3 at 1600MHz [36]. The software version tested has been compiled to MacOS using Unity 2019.3.15f.

In the following, the software has been tested running the application on fullscreen on a native resolution of 2880x1800 and being this application the only user process opened at the moment of rendering.

Now we present some tables with benchmarks tested on both 2D and 3D Fractals.

8.2.1 2D Fractals

We divide this subsection in two parts. One for each 2D Fractal rendered. This is, the Mandelbrot Set and the Julia Set.

All renders in this subsection are plotted over a 750x750 image.

Mandelbrot Set

We have tested our rendering pipeline for different values of the maximum iterations N of the rendering algorithm. Tables 8.1, 8.2 and 8.3 summarise the results for the rendering on the Mandelbrot Set for values $N = 100, 200$ and 300, respectively. In order to do this comparison, we have fixed `Zoom` = 0.000122, `Pan X` = -1.253600 and `Pan Y` = 0.384466. On Henriksen algorithm, the parameter that handles the size of a pixel is fixed to 10.000 on all cases. In addition to this, on all CPU benchmarks (both Single-Thread and Multi-Thread), the value displayed on the table is the best result in a Best of 5 runs on the same parameters.

We can observe an increase in rendering time as we increase the maximum iterations value. It is important to note that the *Shader GPU* column shows this data using other units because it makes no sense to measure this in terms of seconds.

Mandelbrot Set N=100	**Single-Thread CPU**	**Multi-Thread CPU**	**Shader GPU**
Escape Algorithm	10.16s	2.08s	40.8ms (25 FPS)
Henriksen Algorithm	22.64s	4.65s	-

Table 8.1: Rendering results for Mandelbrot set of $z^2 + c$. $N = 100$. s: seconds, ms: milliseconds, FPS: Frames Per Second

Mandelbrot Set $N = 200$	**Single-Thread CPU**	**Multi-Thread CPU**	**Shader GPU**
Escape Algorithm	11.80s	2.38s	47.5ms (21 FPS)
Henriksen Algorithm	26.16s	5.13s	-

Table 8.2: Rendering results for Mandelbrot set of $z^2 + c$. $N = 200$. s: seconds, ms: milliseconds, FPS: Frames Per Second

Mandelbrot Set $N = 300$	**Single-Thread CPU**	**Multi-Thread CPU**	**Shader GPU**
Escape Algorithm	13.00s	2.62s	47.6ms (21 FPS)
Henriksen Algorithm	28.66s	5.80s	-

Table 8.3: Rendering results for Mandelbrot set of $z^2 + c$. $N = 300$. s: seconds, ms: milliseconds, FPS: Frames Per Second

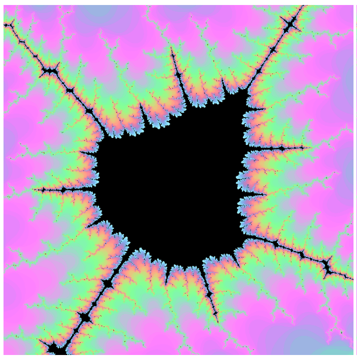
In concrete terms, the GPU boost the rendering pipeline by an average of $\approx \times 52$ if we compare it with Multi-Thread on CPU. Nonetheless, we can achieve a $\approx \times 273$ in rendering performance if we compare it with Single-Thread. For instance,

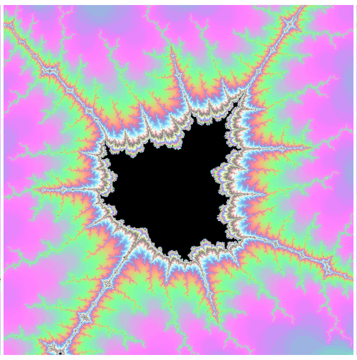
$$\text{Single-Thread: } 13.00\text{s} = 13.000\text{ms} \Longrightarrow \frac{13.000\text{ms}}{52,6\text{ms}} \approx 273.$$

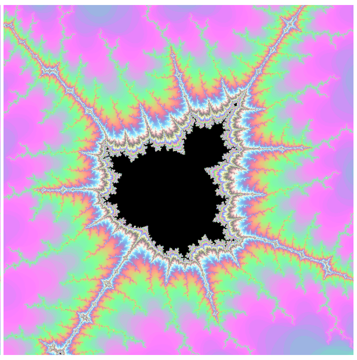
$$\text{Multi-Thread: } 2,08\text{s} = 2.080\text{ms} \Longrightarrow \frac{2.080\text{ms}}{44,5\text{ms}} \approx 50.$$

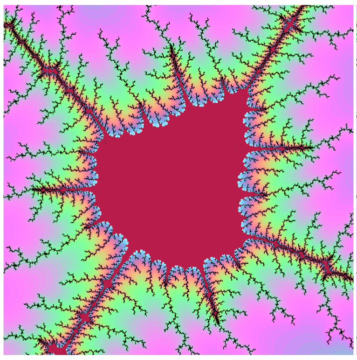
In addition, on Figure 8.5 you can see the results obtained during this rendering.

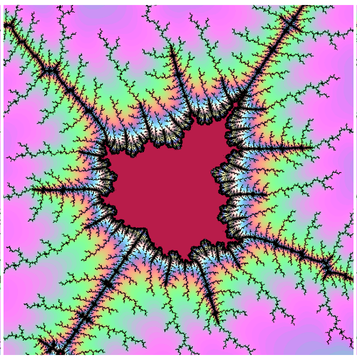
One can easily note that the results from the Henrisken Algorithm are more detailed and well defined. However, we can achieve more level of detail with this algorithm in exchange of larger rendering times as we can see on the previous tables.


(a) Escape Algorithm. $N = 100$


(b) Escape Algorithm. $N = 200$.


(c) Escape Algorithm. $N = 300$.


(d) Henriksen Algorithm. $N =$ 100.


(e) Henriksen Algorithm. $N =$ 200.

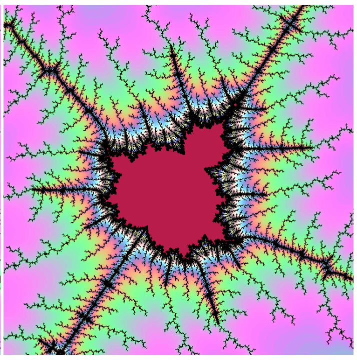

(f) Henriksen Algorithm. $N =$ 300.

Figure 8.5: Images obtained for Mandelbrot Set.

Julia Set

As before, we have tested our rendering pipeline for different values of the maximum iterations N of the rendering algorithm. Tables 8.4, 8.5 and 8.6 summarise the results for the rendering on the Julia set for values $N = 100, 200$ and 300, respectively. In order to do this comparison, we have fixed `Zoom` = 0.250000, `Pan X` = -0.498667 and `Pan Y` = 0.054667, as well as the `Seed` = -0.765 + 0.12i. On Henriksen algorithm, the parameter that handles the size of a pixel is fixed to 10.000 on all cases. In addition to this, on all CPU benchmarks (both Single-Thread and Multi-Thread), the value displayed on the table is the best results in a Best of 5 runs on the same parameters.

Just like before, we can observe an increase in rendering time as we increase the maximum iterations on the algorithms.

In concrete terms, this time the GPU boosts the rendering pipeline by an average of $\approx \times 19$. If we compare it with the Single-Thread rendering, we get a boost in rendering time by an average of $\approx \times 96$.

In addition, on Figure 8.6 you can see the results obtained during this rendering.

Julia Set $N = 100$	**Single-Thread CPU**	**Multi-Thread CPU**	**Shader GPU**
Escape Algorithm	6.62s	1.33s	45.8ms (22 FPS)
Henriksen Algorithm	15.68s	3.22s	-

Table 8.4: Rendering results for Julia set of $z^2 + c$. $N = 100$. s: seconds, ms: milliseconds, FPS: Frames Per Second

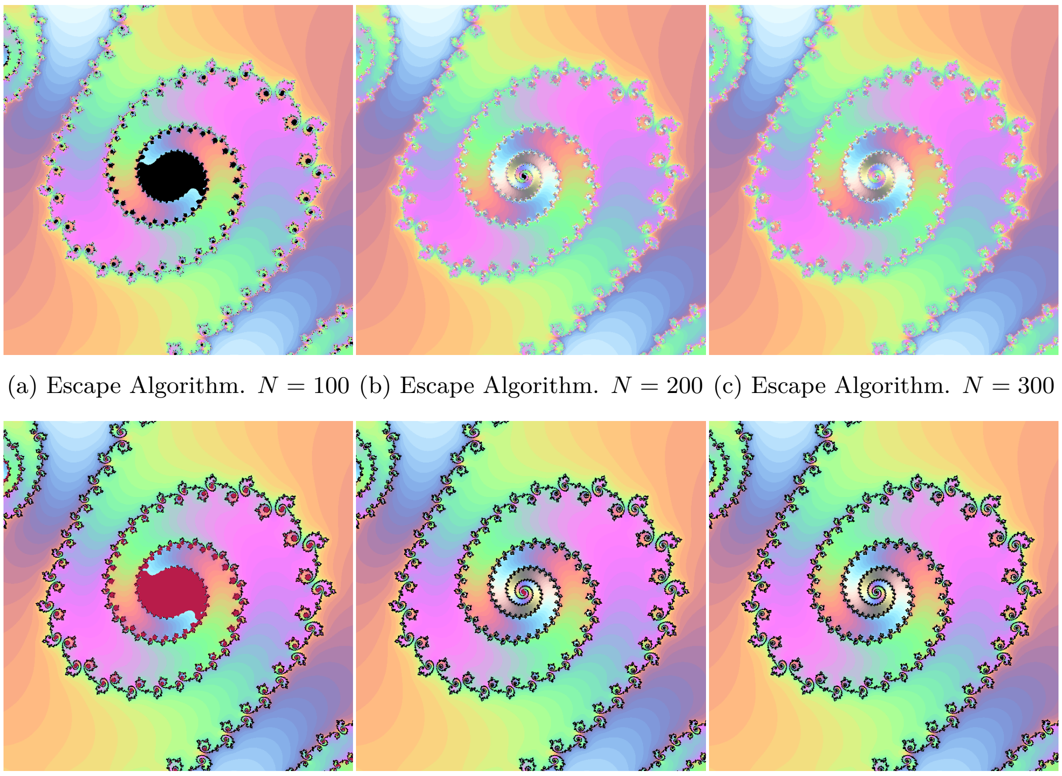
Julia Set $N = 200$	**Single-Thread CPU**	**Multi-Thread CPU**	**Shader GPU**
Escape Algorithm	6.77s	1.38s	79.1ms (13 FPS)
Henriksen Algorithm	15.93s	3.31s	-

Table 8.5: Rendering results for Julia set of $z^2 + c$. $N = 200$. s: seconds, ms: milliseconds, FPS: Frames Per Second

Julia Set $N = 300$	**Single-Thread CPU**	**Multi-Thread CPU**	**Shader GPU**
Escape Algorithm	6.78s	1.37s	115.1ms (9 FPS)
Henriksen Algorithm	15.94s	3.29s	-

Table 8.6: Rendering results for Julia set of $z^2 + c$. $N = 300$. s: seconds, ms: milliseconds, FPS: Frames Per Second

Again as before, one can easily note that Henriksen performs better in terms of detail, however, it is slower than the Escape Algorithm.



(a) Escape Algorithm. $N = 100$ (b) Escape Algorithm. $N = 200$ (c) Escape Algorithm. $N = 300$

(d) Henriksen Algorithm. $N = 100$ (e) Henriksen Algorithm. $N = 200$ (f) Henriksen Algorithm. $N = 300$

Figure 8.6: Images obtained for Julia Set.

8.2.2 3D Fractals

Let us move to the 3D Fractals benchmark. In this case we divide this subsection in three parts. One for each type of 3D Fractal rendered.

On all cases, we fixed `MAX_STEPS` to 64. This is the maximum number of iterations allowed to iterate the *Ray-Marching* algorithm. The automatic camera rotation was deactivated as well.

Mandelbulb

In this benchmark we fixed the `Power` parameter to 8 and fixed the camera position. The results of this test are on Table 8.7.

Resolution vs. Detail	**Very Low (640x360)**	**Low (960x540)**	**Medium (1280x720)**	**High (1920x1080)**
Low	32.3ms (31 FPS)	53.8ms (19 FPS)	84.3ms (12 FPS)	164.4ms (6 FPS)
Medium	36.5ms (27 FPS)	62.1ms (16 FPS)	95.8ms (10 FPS)	171.1ms (6 FPS)
High	41.8ms (24 FPS)	68.4ms (15 FPS)	109.7ms (9 FPS)	205.2ms (5 FPS)
Ultra	45.6ms (22 FPS)	79.1ms (13 FPS)	113.3ms (9 FPS)	237.5ms (4 FPS)

Table 8.7: Rendering results for Mandelbulb image of $z^8 + c$. ms: milliseconds, FPS: Frames Per Second

You can see some images of the results above on Figure 8.7.

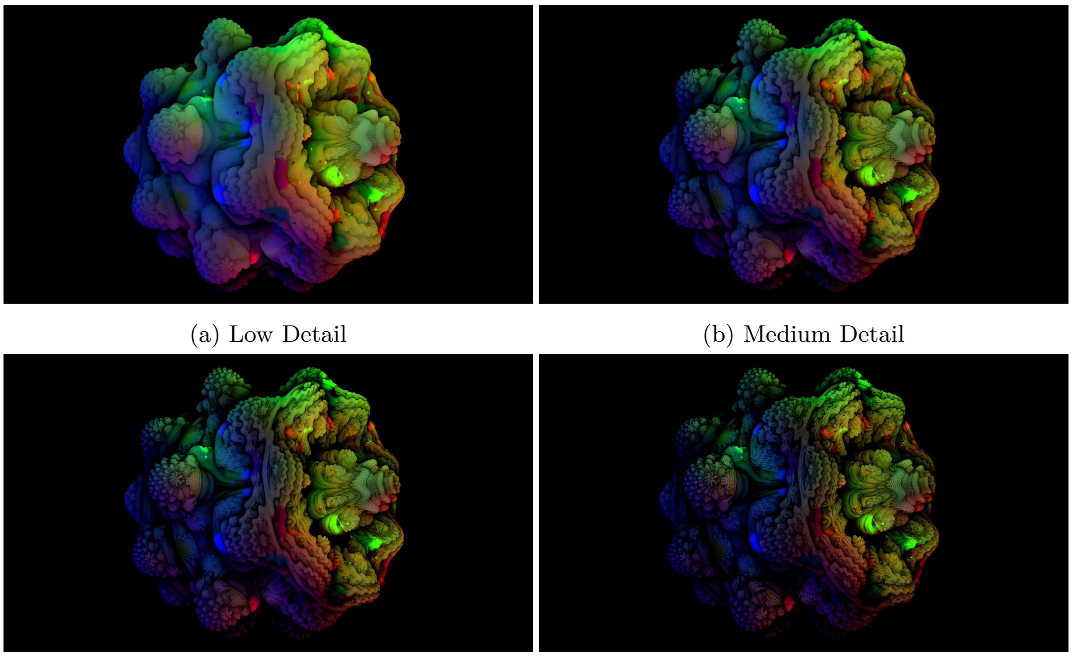

(a) Low Detail (b) Medium Detail
(c) High Detail (d) Ultra Detail

Figure 8.7: Images obtained on Mandelbulb shader for High Resolution (1920x1080)

It is important to note that, since we are estimating distances using a Distance Estimator function, the does not provide the normal vector used when colouring. However, we have estimated the normal of each point using the Finite Difference Method (FDM) [37].

Once we have obtained an estimation for the normal, we have applied the same Distance Estimator of the Mandelbulb to obtain the final colour.

Sierpiński Tetrahedron

In this benchmark we fixed the `IFS Iters` parameter to 12 and fixed the camera position. The results of this test are on Table 8.8.

Resolution vs. Detail	Very Low (640x360)	Low (960x540)	Medium (1280x720)	High (1920x1080)
Low	77.2ms (13 FPS)	145.2ms (7 FPS)	207.2ms (5 FPS)	347.5ms (3 FPS)
Medium	79.6ms (13 FPS)	163.6ms (6 FPS)	260.3ms (4 FPS)	549.1ms (2 FPS)
High	80.8ms (12 FPS)	157.2ms (6 FPS)	266.1ms (4 FPS)	580.2ms (2 FPS)
Ultra	81.5ms (12 FPS)	159.7ms (6 FPS)	274.5ms (4 FPS)	589.4ms (2 FPS)

Table 8.8: Rendering results for Sierpiński Tetrahedron. `IFS Iters` = 12. ms: milliseconds, FPS: Frames Per Second

You can see some images of the results above on Figure 8.8.

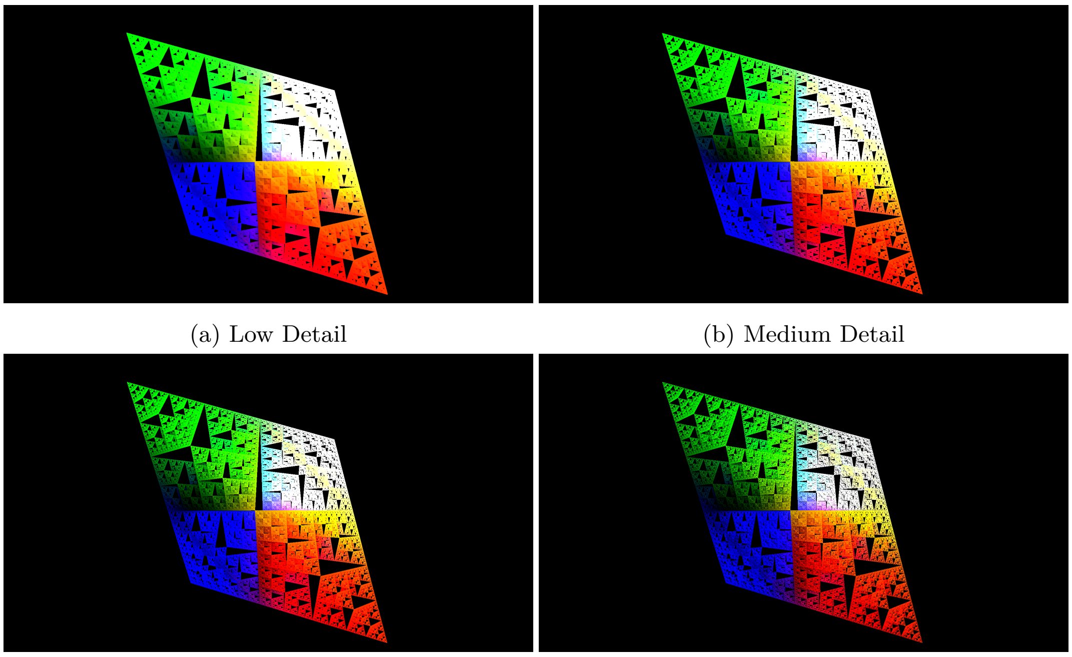

(a) Low Detail (b) Medium Detail
(c) High Detail (d) Ultra Detail

Figure 8.8: Images obtained on Sierpinski Tetrahedron shader for High Resolution (1920x1080)

In a similar way as the Mandelbulb, we have estimated the normals using the Finite Difference Method as well. However, we have coloured the Tetrahedron using the Distance Estimator from a Sphere or radius 1 centred at origin.

Menger Sponge

In this benchmark we fixed the `IFS Iters` parameter to 7 and fixed the camera position. The results of this test are on Table 8.9.

Resolution vs. Detail	**Very Low (640x360)**	**Low (960x540)**	**Medium (1280x720)**	**High (1920x1080)**
Low	46.0ms (22 FPS)	84.9ms (12 FPS)	125.7ms (8 FPS)	256.1ms (4 FPS)
Medium	48.6ms (21 FPS)	89.4ms (11 FPS)	149.3ms (7 FPS)	314.3ms (3 FPS)
High	49.3ms (20 FPS)	93.5ms (11 FPS)	154.4ms (6 FPS)	317.0ms (3 FPS)
Ultra	52.2ms (19 FPS)	94.3ms (11 FPS)	160.4ms (6 FPS)	332.8ms (3 FPS)

Table 8.9: Rendering results for Menger Sponge. `IFS Iters` = 7. ms: milliseconds, FPS: Frames Per Second

You can see some images of the results above on Figure 8.9.

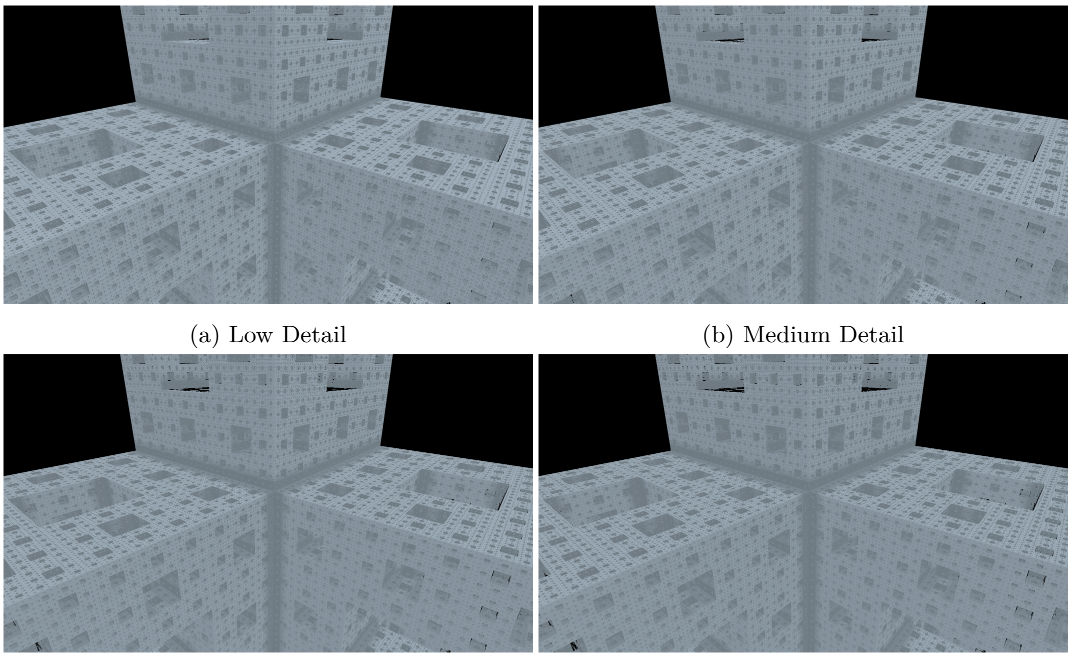

(a) Low Detail
(b) Medium Detail
(c) High Detail
(d) Ultra Detail

Figure 8.9: Images obtained on Menger Sponge shader for High Resolution (1920x1080)

In this case, we have not coloured the results. Nonetheless, we have calculated shadows in order to visualise better this set.

In general, we can see low FPS on some benchmarks (specially when we approach to High Resolution and/or Ultra level of detail). This is, mostly, because the GPU used for this benchmark is a mid-tier one [38].

However, in general, any GPU will have similar behaviour in terms of performance, it will perform worse as long as we increase the resolution and/or level of detail.

Chapter 9

Conclusions and Future work

9.1 Conclusions

Regarding the objectives that were set at the beginning of the project, we can state that they have been satisfactorily fulfilled.

Firstly, an extensive study has been carried out about iteration of holomorphic functions and its fractals resulting in Part I, specially focusing on polynomial mappings. However, we have done this study with the goal of obtain and proof the rendering algorithms that have been implemented on the software project.

Once this study has been done, we have analysed the existing software that accomplishes goals similar to our Problem, defined on Chapter 6. However, we have realised that there is no software that fulfil these requirements, so we get encouraged and hence we designed and developed a software project that allows the user to visualise 2D and 3D fractals directly on the web.

Furthermore, thanks to WebGL technology, we have *enhanced* this fractal visualisation by optimising these algorithms implementing them over *shaders* that run on the GPU. For this reason, we have obtained a boost in rendering time up to 273 times faster than using the regular CPU algorithms.

Finally, we wanted to compare the results obtained due to this optimisation on the GPU against the same implementation over the CPU, so we placed a detailed benchmark comparing them on Chapter 8.

In summary, we have obtained an application to explore and analyse 2D and 3D fractals, that could be used to as an educational tool to teach properties of fractals all provided by an easy interface.

9.2 Future work

The overall result is encouraging, however, one can always improve a project like this. It should not cost too much to implement new features since we have developed the software as extensible and reusable as possible.

Regarding the visualisation of 2D fractals, new rendering algorithms can be implemented in order to enrich the visualisation. For instance, one can consider to implement the *Inverse iteration method* explained on Subsection 6.2.1. In order to do this, our software design allows to add new types of fractals easily.

In order to build up the mathematical features that this project provides about fractals, one can study another family of functions like the logistic family.

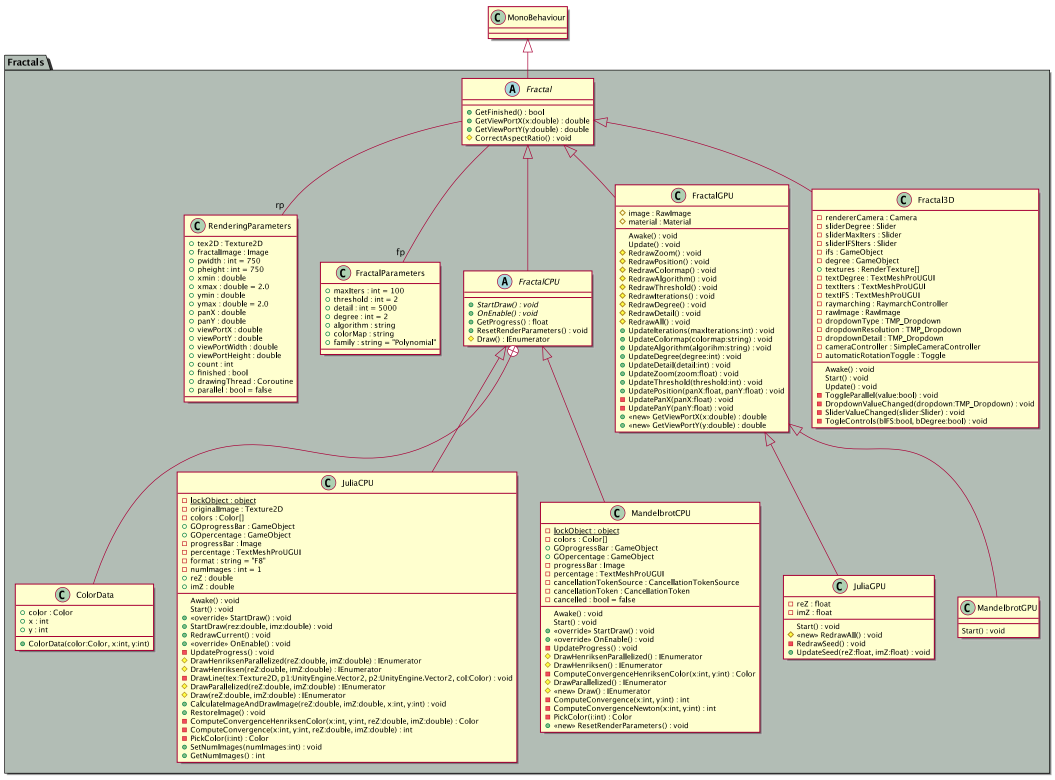
Another improvement in order to enhance the visualisation about 3D fractals, is the implementation of a new feature that automatically handles the level of detail displayed on the scene, based on the position of the camera. This is, the closer to the fractal we are, the more details it renders.

Finally, a clear next step of this project is to make a study of usability over real students.

Appendix A

Class Diagrams

In this appendix we present the complete version of the Class Diagrams showed on Chapter 7, Section 7.3.



MonoBehaviour
Fractals
Fractal
GetFinished() : bool
GetViewPortX(x:double) : double
GetViewPortY(y:double) : double
CorrectAspectRatio() : void
rp
RenderingParameters
tex2D : Texture2D
fractalImage : Image
pwidth : int = 750
pheight : int = 750
xmin : double
xmax : double = 2.0
ymin : double
ymax : double = 2.0
panX : double
panY : double
viewPortX : double
viewPortY : double
viewPortWidth : double
viewPortHeight : double
count : int
finished : bool
drawingThread : Coroutine
parallel : bool = false
fp
FractalParameters
maxIters : int = 100
threshold : int = 2
detail : int = 5000
degree : int = 2
algorithm : string
colorMap : string
family : string = "Polynomial"
FractalCPU
StartDraw() : void
OnEnable() : void
GetProgress() : float
ResetRenderParameters() : void
Draw() : IEnumerator
FractalGPU
image : RawImage
material : Material
Awake() : void
Update() : void
RedrawZoom() : void
RedrawPosition() : void
RedrawColormap() : void
RedrawAlgorithm() : void
RedrawThreshold() : void
RedrawIterations() : void
RedrawDegree() : void
RedrawDetail() : void
RedrawAll() : void
UpdateIterations(maxIterations:int) : void
UpdateColormap(colormap:string) : void
UpdateAlgorithm(algorihm:string) : void
UpdateDegree(degree:int) : void
UpdateDetail(detail:int) : void
UpdateZoom(zoom:float) : void
UpdateThreshold(threshold:int) : void
UpdatePosition(panX:float, panY:float) : void
UpdatePanX(panX:float) : void
UpdatePanY(panY:float) : void
«new» GetViewPortX(x:double) : double
«new» GetViewPortY(y:double) : double
Fractal3D
rendererCamera : Camera
sliderDegree : Slider
sliderMaxIters : Slider
sliderIFSIters : Slider
ifs : GameObject
degree : GameObject
textures : RenderTexture[]
textDegree : TextMeshProUGUI
textIters : TextMeshProUGUI
textIFS : TextMeshProUGUI
raymarching : RaymarchController
rawImage : RawImage
dropdownType : TMP_Dropdown
dropdownResolution : TMP_Dropdown
dropdownDetail : TMP_Dropdown
cameraController : SimpleCameraController
automaticRotationToggle : Toggle
Awake() : void
Start() : void
Update() : void
ToggleParallel(value:bool) : void
DropdownValueChanged(dropdown:TMP_Dropdown) : void
SliderValueChanged(slider:Slider) : void
TogleControls(bIFS:bool, bDegree:bool) : void
ColorData
color : Color
x : int
y : int
ColorData(color:Color, x:int, y:int)
JuliaCPU
lockObject : object
originalImage : Texture2D
colors : Color[]
GOprogressBar : GameObject
GOpercentage : GameObject
progressBar : Image
percentage : TextMeshProUGUI
format : string = "F8"
numImages : int = 1
reZ : double
imZ : double
Awake() : void
Start() : void
«override» StartDraw() : void
StartDraw(rez:double, imz:double) : void
RedrawCurrent() : void
«override» OnEnable() : void
UpdateProgress() : void
DrawHenriksenParallelized(reZ:double, imZ:double) : IEnumerator
DrawHenriksen(reZ:double, imZ:double) : IEnumerator
DrawLine(tex:Texture2D, p1:UnityEngine.Vector2, p2:UnityEngine.Vector2, col:Color) : void
DrawParallelized(reZ:double, imZ:double) : IEnumerator
Draw(reZ:double, imZ:double) : IEnumerator
CalculateImageAndDrawImage(reZ:double, imZ:double, x:int, y:int) : void
RestoreImage() : void
ComputeConvergenceHenriksenColor(x:int, y:int, reZ:double, imZ:double) : Color
ComputeConvergence(x:int, y:int, reZ:double, imZ:double) : int
PickColor(i:int) : Color
SetNumImages(numImages:int) : void
GetNumImages() : int
MandelbrotCPU
lockObject : object
colors : Color[]
GOprogressBar : GameObject
GOpercentage : GameObject
progressBar : Image
percentage : TextMeshProUGUI
cancellationTokenSource : CancellationTokenSource
cancellationToken : CancellationToken
cancelled : bool = false
Awake() : void
Start() : void
«override» StartDraw() : void
«override» OnEnable() : void
UpdateProgress() : void
DrawHenriksenParallelized() : IEnumerator
DrawHenriksen() : IEnumerator
ComputeConvergenceHenriksenColor(x:int, y:int) : Color
DrawParallelized() : IEnumerator
«new» Draw() : IEnumerator
ComputeConvergence(x:int, y:int) : int
ComputeConvergenceNewton(x:int, y:int) : int
PickColor(i:int) : Color
«new» ResetRenderParameters() : void
JuliaGPU
reZ : float
imZ : float
Start() : void
«new» RedrawAll() : void
RedrawSeed() : void
UpdateSeed(reZ:float, imZ:float) : void
MandelbrotGPU
Start() : void

Figure A.1: Class Diagram for Fractals Package

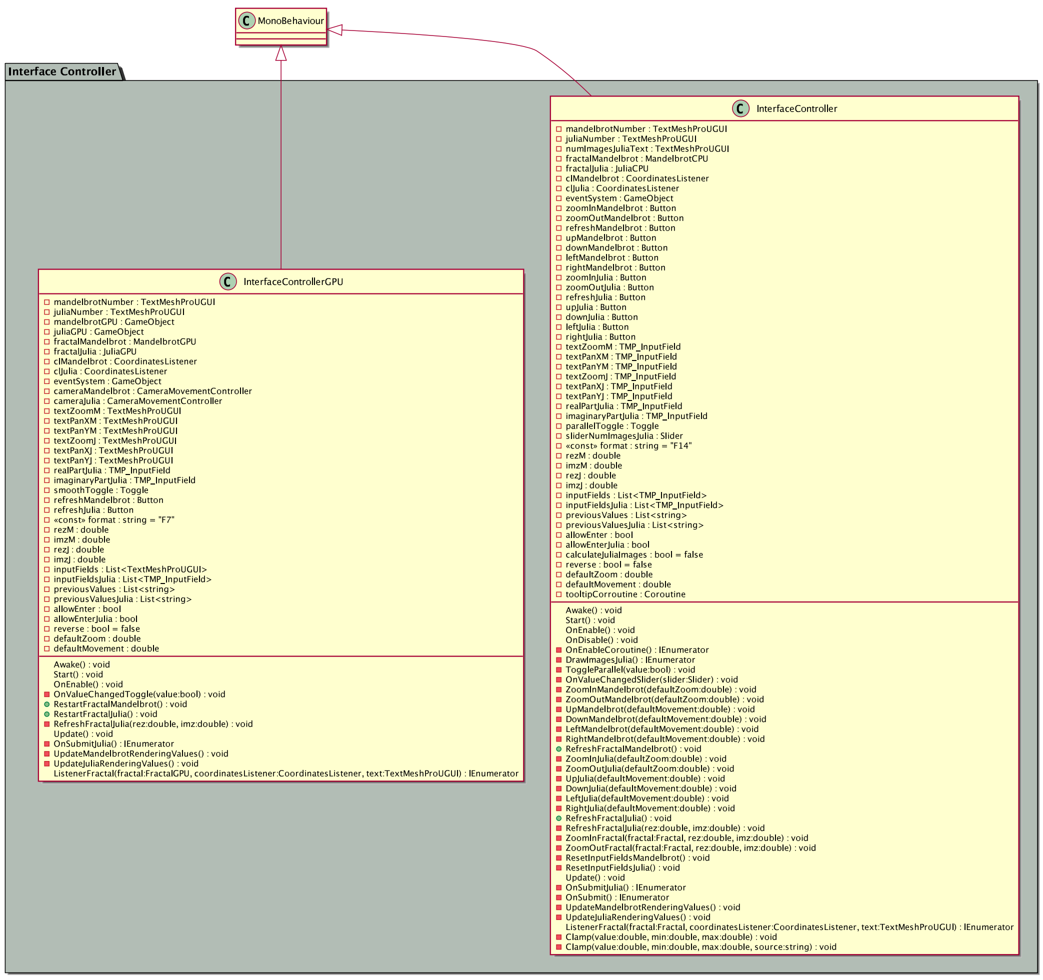

MonoBehaviour
Interface Controller
InterfaceControllerGPU
mandelbrotNumber : TextMeshProUGUI
juliaNumber : TextMeshProUGUI
mandelbrotGPU : GameObject
juliaGPU : GameObject
fractalMandelbrot : MandelbrotGPU
fractalJulia : JuliaGPU
clMandelbrot : CoordinatesListener
clJulia : CoordinatesListener
eventSystem : GameObject
cameraMandelbrot : CameraMovementController
cameraJulia : CameraMovementController
textZoomM : TextMeshProUGUI
textPanXM : TextMeshProUGUI
textPanYM : TextMeshProUGUI
textZoomJ : TextMeshProUGUI
textPanXJ : TextMeshProUGUI
textPanYJ : TextMeshProUGUI
realPartJulia : TMP_InputField
imaginaryPartJulia : TMP_InputField
smoothToggle : Toggle
refreshMandelbrot : Button
refreshJulia : Button
«const» format : string = "F7"
rezM : double
imzM : double
rezJ : double
imzJ : double
inputFields : List<TextMeshProUGUI>
inputFieldsJulia : List<TMP_InputField>
previousValues : List<string>
previousValuesJulia : List<string>
allowEnter : bool
allowEnterJulia : bool
reverse : bool = false
defaultZoom : double
defaultMovement : double
Awake() : void
Start() : void
OnEnable() : void
OnValueChangedToggle(value:bool) : void
RestartFractalMandelbrot() : void
RestartFractalJulia() : void
RefreshFractalJulia(rez:double, imz:double) : void
Update() : void
OnSubmitJulia() : IEnumerator
UpdateMandelbrotRenderingValues() : void
UpdateJuliaRenderingValues() : void
ListenerFractal(fractal:FractalGPU, coordinatesListener:CoordinatesListener, text:TextMeshProUGUI) : IEnumerator
InterfaceController
mandelbrotNumber : TextMeshProUGUI
juliaNumber : TextMeshProUGUI
numImagesJuliaText : TextMeshProUGUI
fractalMandelbrot : MandelbrotCPU
fractalJulia : JuliaCPU
clMandelbrot : CoordinatesListener
clJulia : CoordinatesListener
eventSystem : GameObject
zoomInMandelbrot : Button
zoomOutMandelbrot : Button
refreshMandelbrot : Button
upMandelbrot : Button
downMandelbrot : Button
leftMandelbrot : Button
rightMandelbrot : Button
zoomInJulia : Button
zoomOutJulia : Button
refreshJulia : Button
upJulia : Button
downJulia : Button
leftJulia : Button
rightJulia : Button
textZoomM : TMP_InputField
textPanXM : TMP_InputField
textPanYM : TMP_InputField
textZoomJ : TMP_InputField
textPanXJ : TMP_InputField
textPanYJ : TMP_InputField
realPartJulia : TMP_InputField
imaginaryPartJulia : TMP_InputField
parallelToggle : Toggle
sliderNumImagesJulia : Slider
«const» format : string = "F14"
rezM : double
imzM : double
rezJ : double
imzJ : double
inputFields : List<TMP_InputField>
inputFieldsJulia : List<TMP_InputField>
previousValues : List<string>
previousValuesJulia : List<string>
allowEnter : bool
allowEnterJulia : bool
calculateJuliaImages : bool = false
reverse : bool = false
defaultZoom : double
defaultMovement : double
tooltipCorroutine : Coroutine
Awake() : void
Start() : void
OnEnable() : void
OnDisable() : void
OnEnableCoroutine() : IEnumerator
DrawImagesJulia() : IEnumerator
ToggleParallel(value:bool) : void
OnValueChangedSlider(slider:Slider) : void
ZoomInMandelbrot(defaultZoom:double) : void
ZoomOutMandelbrot(defaultZoom:double) : void
UpMandelbrot(defaultMovement:double) : void
DownMandelbrot(defaultMovement:double) : void
LeftMandelbrot(defaultMovement:double) : void
RightMandelbrot(defaultMovement:double) : void
RefreshFractalMandelbrot() : void
ZoomInJulia(defaultZoom:double) : void
ZoomOutJulia(defaultZoom:double) : void
UpJulia(defaultMovement:double) : void
DownJulia(defaultMovement:double) : void
LeftJulia(defaultMovement:double) : void
RightJulia(defaultMovement:double) : void
RefreshFractalJulia() : void
RefreshFractalJulia(rez:double, imz:double) : void
ZoomInFractal(fractal:Fractal, rez:double, imz:double) : void
ZoomOutFractal(fractal:Fractal, rez:double, imz:double) : void
ResetInputFieldsMandelbrot() : void
ResetInputFieldsJulia() : void
Update() : void
OnSubmitJulia() : IEnumerator
OnSubmit() : IEnumerator
UpdateMandelbrotRenderingValues() : void
UpdateJuliaRenderingValues() : void
ListenerFractal(fractal:Fractal, coordinatesListener:CoordinatesListener, text:TextMeshProUGUI) : IEnumerator
Clamp(value:double, min:double, max:double) : void
Clamp(value:double, min:double, max:double, source:string) : void

Figure A.2: Class Diagram for Interface Controller Package

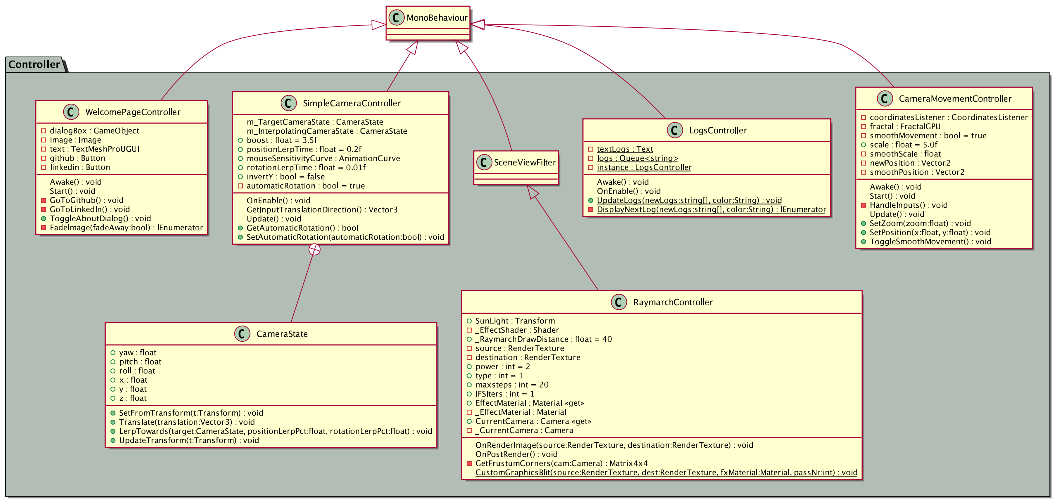

MonoBehaviour
Controller
WelcomePageController
dialogBox : GameObject
image : Image
text : TextMeshProUGUI
github : Button
linkedin : Button
Awake() : void
Start() : void
GoToGithub() : void
GoToLinkedIn() : void
ToggleAboutDialog() : void
FadeImage(fadeAway:bool) : IEnumerator
SimpleCameraController
m_TargetCameraState : CameraState
m_InterpolatingCameraState : CameraState
boost : float = 3.5f
positionLerpTime : float = 0.2f
mouseSensitivityCurve : AnimationCurve
rotationLerpTime : float = 0.01f
invertY : bool = false
automaticRotation : bool = true
OnEnable() : void
GetInputTranslationDirection() : Vector3
Update() : void
GetAutomaticRotation() : bool
SetAutomaticRotation(automaticRotation:bool) : void
SceneViewFilter
LogsController
textLogs : Text
logs : Queue<string>
instance : LogsController
Awake() : void
OnEnable() : void
UpdateLogs(newLogs:string[], color:String) : void
DisplayNextLog(newLogs:string[], color:String) : IEnumerator
CameraMovementController
coordinatesListener : CoordinatesListener
fractal : FractalGPU
smoothMovement : bool = true
scale : float = 5.0f
smoothScale : float
newPosition : Vector2
smoothPosition : Vector2
Awake() : void
Start() : void
HandleInputs() : void
Update() : void
SetZoom(zoom:float) : void
SetPosition(x:float, y:float) : void
ToggleSmoothMovement() : void
CameraState
yaw : float
pitch : float
roll : float
x : float
y : float
z : float
SetFromTransform(t:Transform) : void
Translate(translation:Vector3) : void
LerpTowards(target:CameraState, positionLerpPct:float, rotationLerpPct:float) : void
UpdateTransform(t:Transform) : void
RaymarchController
SunLight : Transform
_EffectShader : Shader
_RaymarchDrawDistance : float = 40
source : RenderTexture
destination : RenderTexture
power : int = 2
type : int = 1
maxsteps : int = 20
IFSiters : int = 1
EffectMaterial : Material «get»
_EffectMaterial : Material
CurrentCamera : Camera «get»
_CurrentCamera : Camera
OnRenderImage(source:RenderTexture, destination:RenderTexture) : void
OnPostRender() : void
GetFrustumCorners(cam:Camera) : Matrix4x4
CustomGraphicsBlit(source:RenderTexture, dest:RenderTexture, fxMaterial:Material, passNr:int) : void

Figure A.3: Class Diagram for Controller Package

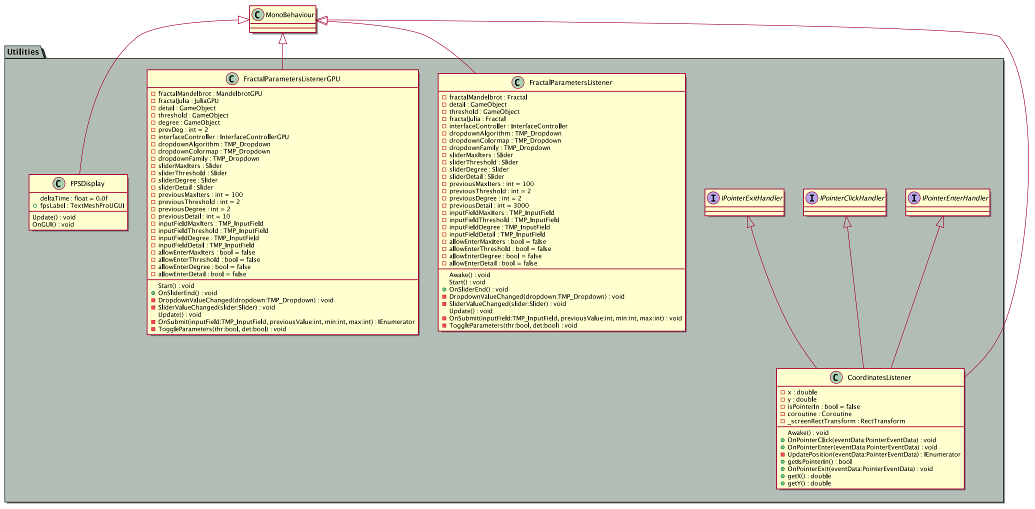

MonoBehaviour
Utilities
FPSDisplay
deltaTime : float = 0.0f
fpsLabel : TextMeshProUGUI
Update() : void
OnGUI() : void
FractalParametersListenerGPU
fractalMandelbrot : MandelbrotGPU
fractalJulia : JuliaGPU
detail : GameObject
threshold : GameObject
degree : GameObject
prevDeg : int = 2
interfaceController : InterfaceControllerGPU
dropdownAlgorithm : TMP_Dropdown
dropdownColormap : TMP_Dropdown
dropdownFamily : TMP_Dropdown
sliderMaxIters : Slider
sliderThreshold : Slider
sliderDegree : Slider
sliderDetail : Slider
previousMaxIters : int = 100
previousThreshold : int = 2
previousDegree : int = 2
previousDetail : int = 10
inputFieldMaxIters : TMP_InputField
inputFieldThreshold : TMP_InputField
inputFieldDegree : TMP_InputField
inputFieldDetail : TMP_InputField
allowEnterMaxIters : bool = false
allowEnterThreshold : bool = false
allowEnterDegree : bool = false
allowEnterDetail : bool = false
Start() : void
OnSliderEnd() : void
DropdownValueChanged(dropdown:TMP_Dropdown) : void
SliderValueChanged(slider:Slider) : void
Update() : void
OnSubmit(inputField:TMP_InputField, previousValue:int, min:int, max:int) : IEnumerator
ToggleParameters(thr:bool, det:bool) : void
FractalParametersListener
fractalMandelbrot : Fractal
detail : GameObject
threshold : GameObject
fractalJulia : Fractal
interfaceController : InterfaceController
dropdownAlgorithm : TMP_Dropdown
dropdownColormap : TMP_Dropdown
dropdownFamily : TMP_Dropdown
sliderMaxIters : Slider
sliderThreshold : Slider
sliderDegree : Slider
sliderDetail : Slider
previousMaxIters : int = 100
previousThreshold : int = 2
previousDegree : int = 2
previousDetail : int = 3000
inputFieldMaxIters : TMP_InputField
inputFieldThreshold : TMP_InputField
inputFieldDegree : TMP_InputField
inputFieldDetail : TMP_InputField
allowEnterMaxIters : bool = false
allowEnterThreshold : bool = false
allowEnterDegree : bool = false
allowEnterDetail : bool = false
Awake() : void
Start() : void
OnSliderEnd() : void
DropdownValueChanged(dropdown:TMP_Dropdown) : void
SliderValueChanged(slider:Slider) : void
Update() : void
OnSubmit(inputField:TMP_InputField, previousValue:int, min:int, max:int) : void
ToggleParameters(thr:bool, det:bool) : void
IPointerExitHandler
IPointerClickHandler
IPointerEnterHandler
CoordinatesListener
x : double
y : double
isPointerIn : bool = false
coroutine : Coroutine
_screenRectTransform : RectTransform
Awake() : void
OnPointerClick(eventData:PointerEventData) : void
OnPointerEnter(eventData:PointerEventData) : void
UpdatePosition(eventData:PointerEventData) : IEnumerator
getIsPointerIn() : bool
OnPointerExit(eventData:PointerEventData) : void
getX() : double
getY() : double

Figure A.4: Class Diagram for Utilities Package

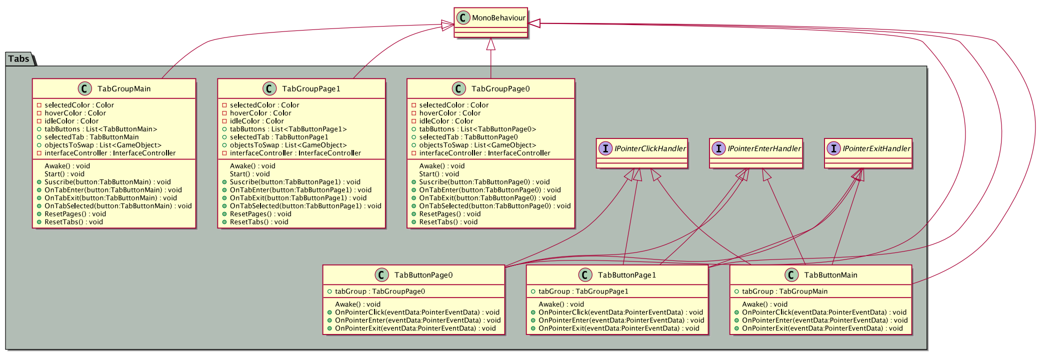

MonoBehaviour
Tabs
TabGroupMain
selectedColor : Color
hoverColor : Color
idleColor : Color
tabButtons : List<TabButtonMain>
selectedTab : TabButtonMain
objectsToSwap : List<GameObject>
interfaceController : InterfaceController
Awake() : void
Start() : void
Suscribe(button:TabButtonMain) : void
OnTabEnter(button:TabButtonMain) : void
OnTabExit(button:TabButtonMain) : void
OnTabSelected(button:TabButtonMain) : void
ResetPages() : void
ResetTabs() : void
TabGroupPage1
selectedColor : Color
hoverColor : Color
idleColor : Color
tabButtons : List<TabButtonPage1>
selectedTab : TabButtonPage1
objectsToSwap : List<GameObject>
interfaceController : InterfaceController
Awake() : void
Start() : void
Suscribe(button:TabButtonPage1) : void
OnTabEnter(button:TabButtonPage1) : void
OnTabExit(button:TabButtonPage1) : void
OnTabSelected(button:TabButtonPage1) : void
ResetPages() : void
ResetTabs() : void
TabGroupPage0
selectedColor : Color
hoverColor : Color
idleColor : Color
tabButtons : List<TabButtonPage0>
selectedTab : TabButtonPage0
objectsToSwap : List<GameObject>
interfaceController : InterfaceController
Awake() : void
Start() : void
Suscribe(button:TabButtonPage0) : void
OnTabEnter(button:TabButtonPage0) : void
OnTabExit(button:TabButtonPage0) : void
OnTabSelected(button:TabButtonPage0) : void
ResetPages() : void
ResetTabs() : void
IPointerClickHandler
IPointerEnterHandler
IPointerExitHandler
TabButtonPage0
tabGroup : TabGroupPage0
Awake() : void
OnPointerClick(eventData:PointerEventData) : void
OnPointerEnter(eventData:PointerEventData) : void
OnPointerExit(eventData:PointerEventData) : void
TabButtonPage1
tabGroup : TabGroupPage1
Awake() : void
OnPointerClick(eventData:PointerEventData) : void
OnPointerEnter(eventData:PointerEventData) : void
OnPointerExit(eventData:PointerEventData) : void
TabButtonMain
tabGroup : TabGroupMain
Awake() : void
OnPointerClick(eventData:PointerEventData) : void
OnPointerEnter(eventData:PointerEventData) : void
OnPointerExit(eventData:PointerEventData) : void

Figure A.5: Class Diagram for Tabs Package

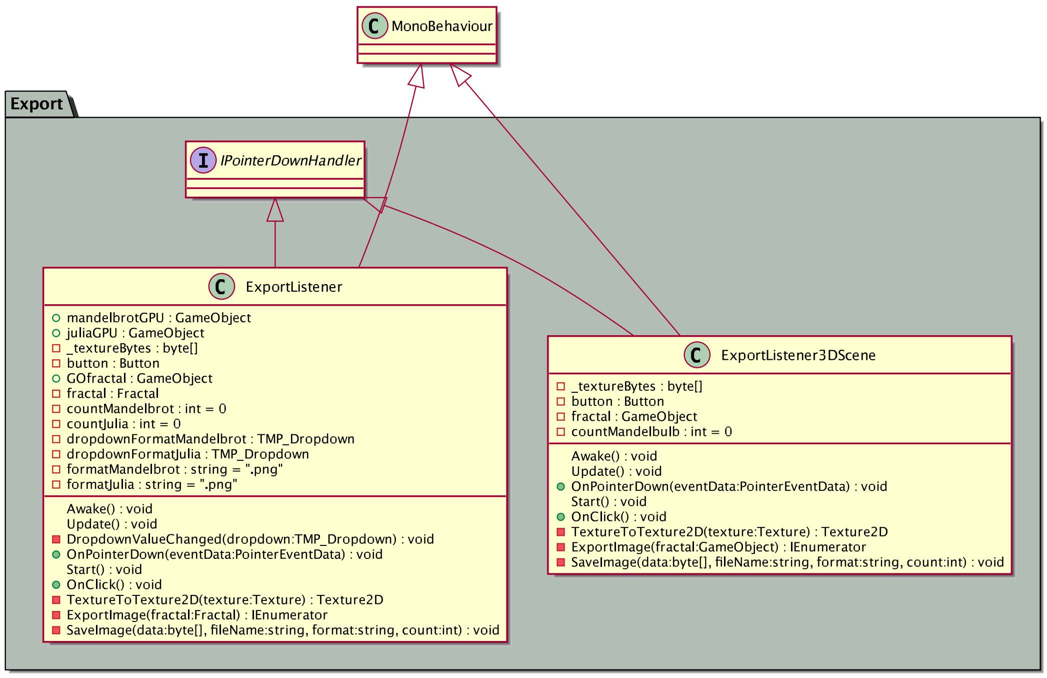
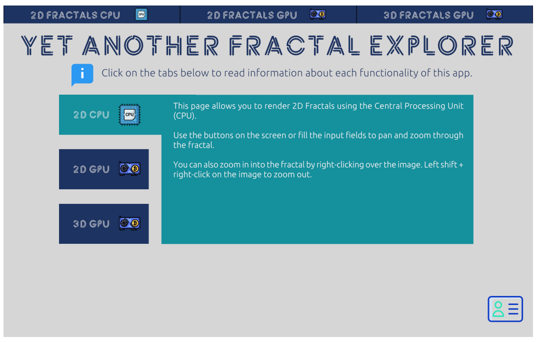

MonoBehaviour
Export
IPointerDownHandler
ExportListener
mandelbrotGPU : GameObject
juliaGPU : GameObject
_textureBytes : byte[]
button : Button
GOfractal : GameObject
fractal : Fractal
countMandelbrot : int = 0
countJulia : int = 0
dropdownFormatMandelbrot : TMP_Dropdown
dropdownFormatJulia : TMP_Dropdown
formatMandelbrot : string = ".png"
formatJulia : string = ".png"
Awake() : void
Update() : void
DropdownValueChanged(dropdown:TMP_Dropdown) : void
OnPointerDown(eventData:PointerEventData) : void
Start() : void
OnClick() : void
TextureToTexture2D(texture:Texture) : Texture2D
ExportImage(fractal:Fractal) : IEnumerator
SaveImage(data:byte[], fileName:string, format:string, count:int) : void
ExportListener3DScene
_textureBytes : byte[]
button : Button
fractal : GameObject
countMandelbulb : int = 0
Awake() : void
Update() : void
OnPointerDown(eventData:PointerEventData) : void
Start() : void
OnClick() : void
TextureToTexture2D(texture:Texture) : Texture2D
ExportImage(fractal:GameObject) : IEnumerator
SaveImage(data:byte[], fileName:string, format:string, count:int) : void

Figure A.6: Class Diagram for Export Package

Appendix B

Application Screenshots

In this appendix we present some additional screenshots about the application developed explained on Chapter 8, Section 8.1.



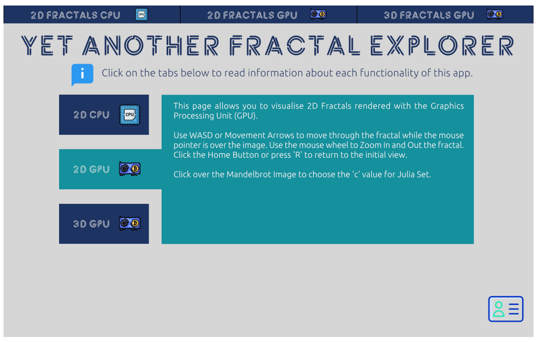
Click on the tabs below to read information about each functionality of this app.

This page allows you to render 2D Fractals using the Central Processing Unit (CPU).

Use the buttons on the screen or fill the input fields to pan and zoom through the fractal.

You can also zoom in into the fractal by right-clicking over the image. Left shift + right-click on the image to zoom out.

(a) Information tab about 2D CPU Scene



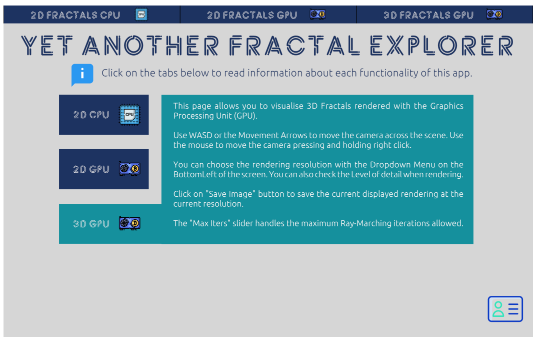
Click on the tabs below to read information about each functionality of this app.

This page allows you to visualise 2D Fractals rendered with the Graphics Processing Unit (GPU).

Use WASD or Movement Arrows to move through the fractal while the mouse pointer is over the image. Use the mouse wheel to Zoom In and Out the fractal. Click the Home Button or press 'R' to return to the initial view.

Click over the Mandelbrot Image to choose the 'c' value for Julia Set.

(b) Information tab about 2D GPU Scene



Click on the tabs below to read information about each functionality of this app.

This page allows you to visualise 3D Fractals rendered with the Graphics Processing Unit (GPU).

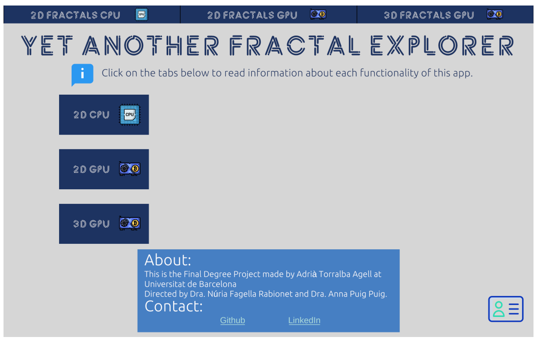
Use WASD or the Movement Arrows to move the camera across the scene. Use the mouse to move the camera pressing and holding right click.

You can choose the rendering resolution with the Dropdown Menu on the BottomLeft of the screen. You can also check the Level of detail when rendering.

Click on "Save Image" button to save the current displayed rendering at the current resolution.

The "Max Iters" slider handles the maximum Ray-Marching iterations allowed.

(c) Information tab about 3D GPU Scene

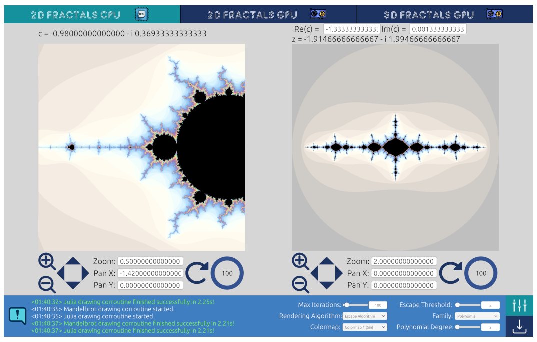


Click on the tabs below to read information about each functionality of this app.

About:
This is the Final Degree Project made by Adrià Torralba Agell at Universitat de Barcelona
Directed by Dra. Núria Fagella Rabionet and Dra. Anna Puig Puig.
Contact:

(d) About page

Figure B.1: Images from welcome page of Yet Another Fractal Explorer Application



c = -0.98000000000000 - i 0.36933333333333

Re(c) = -1.33333333333: Im(c) = 0.001333333333
z = -1.91466666666667 - i 1.99466666666667

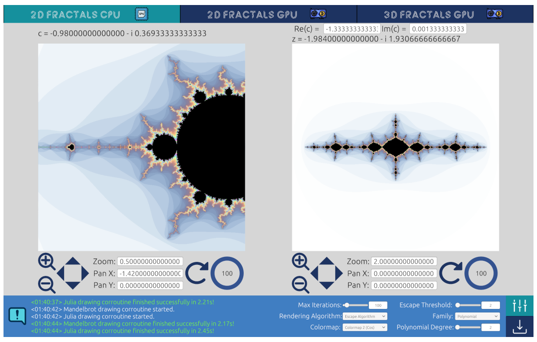
Zoom: 0.50000000000000
Pan X: -1.42000000000000
Pan Y: 0.00000000000000

Zoom: 2.00000000000000
Pan X: 0.00000000000000
Pan Y: 0.00000000000000

```
<01:40:32> Julia drawing corroutine finished successfully in 2.25s!
<01:40:35> Mandelbrot drawing corroutine started.
<01:40:35> Julia drawing corroutine started.
<01:40:37> Mandelbrot drawing corroutine finished successfully in 2.21s!
<01:40:37> Julia drawing corroutine finished successfully in 2.21s!
```

Max Iterations: 100
Rendering Algorithm: Escape Algorithm
Colormap: Colormap 1 (Sin)
Escape Threshold: 2
Family: Polynomial
Polynomial Degree: 2

(a) sin Colormap



c = -0.98000000000000 - i 0.36933333333333

Re(c) = -1.33333333333: Im(c) = 0.001333333333
z = -1.98400000000000 - i 1.93066666666667

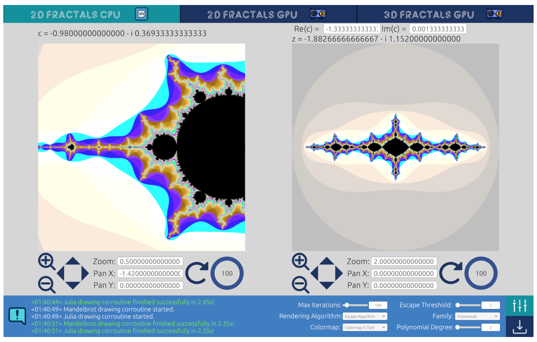
Zoom: 0.50000000000000
Pan X: -1.42000000000000
Pan Y: 0.00000000000000

Zoom: 2.00000000000000
Pan X: 0.00000000000000
Pan Y: 0.00000000000000

```
<01:40:37> Julia drawing corroutine finished successfully in 2.21s!
<01:40:42> Mandelbrot drawing corroutine started.
<01:40:42> Julia drawing corroutine started.
<01:40:44> Mandelbrot drawing corroutine finished successfully in 2.17s!
<01:40:44> Julia drawing corroutine finished successfully in 2.45s!
```

Max Iterations: 100
Rendering Algorithm: Escape Algorithm
Colormap: Colormap 2 (Cos)
Escape Threshold: 2
Family: Polynomial
Polynomial Degree: 2

(b) cos Colormap



c = -0.98000000000000 - i 0.36933333333333

Re(c) = -1.33333333333: Im(c) = 0.001333333333
z = -1.88266666666667 - i 1.15200000000000

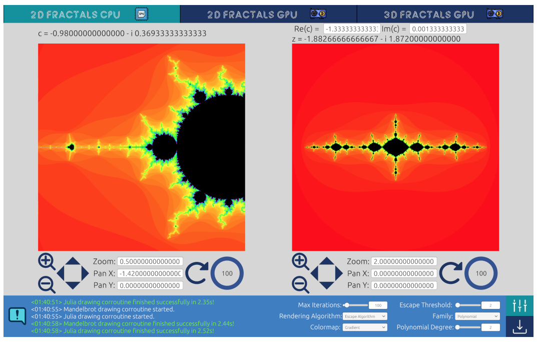
Zoom: 0.50000000000000
Pan X: -1.42000000000000
Pan Y: 0.00000000000000

Zoom: 2.00000000000000
Pan X: 0.00000000000000
Pan Y: 0.00000000000000

```
<01:40:44> Julia drawing corroutine finished successfully in 2.45s!
<01:40:49> Mandelbrot drawing corroutine started.
<01:40:49> Julia drawing corroutine started.
<01:40:51> Mandelbrot drawing corroutine finished successfully in 2.35s!
<01:40:51> Julia drawing corroutine finished successfully in 2.35s!
```

Max Iterations: 100
Rendering Algorithm: Escape Algorithm
Colormap: Colormap 3 (Tan)
Escape Threshold: 2
Family: Polynomial
Polynomial Degree: 2

(c) tan Colormap



c = -0.98000000000000 - i 0.36933333333333

Re(c) = -1.33333333333: Im(c) = 0.001333333333
z = -1.88266666666667 - i 1.87200000000000

Zoom: 0.50000000000000
Pan X: -1.42000000000000
Pan Y: 0.00000000000000

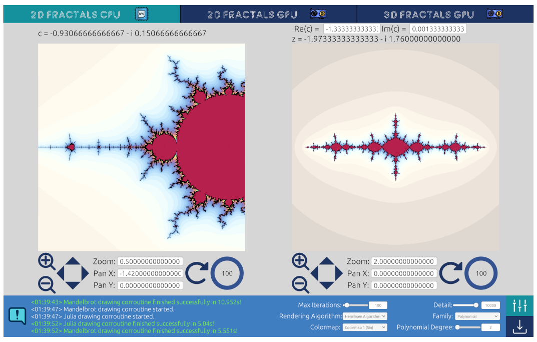
Zoom: 2.00000000000000
Pan X: 0.00000000000000
Pan Y: 0.00000000000000

```
<01:40:51> Julia drawing corroutine finished successfully in 2.35s!
<01:40:55> Mandelbrot drawing corroutine started.
<01:40:55> Julia drawing corroutine started.
<01:40:58> Mandelbrot drawing corroutine finished successfully in 2.44s!
<01:40:58> Julia drawing corroutine finished successfully in 2.52s!
```

Max Iterations: 100
Rendering Algorithm: Escape Algorithm
Colormap: Gradient
Escape Threshold: 2
Family: Polynomial
Polynomial Degree: 2

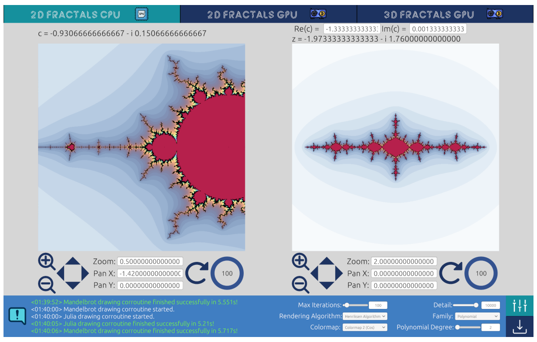
(d) Gradient Colormap

Figure B.2: Results for different colormaps using Escape Algorithm on CPU



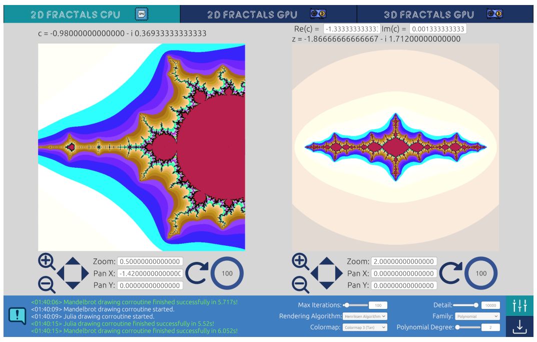
```
c = -0.93066666666667 - i 0.15066666666667
Re(c) = -1.33333333333: Im(c) = 0.001333333333
z = -1.97333333333333 - i 1.76000000000000
Zoom: 0.50000000000000    Zoom: 2.00000000000000
Pan X: -1.42000000000000  Pan X: 0.00000000000000
Pan Y: 0.00000000000000   Pan Y: 0.00000000000000
<01:39:43> Mandelbrot drawing corroutine finished successfully in 10.952s!
<01:39:47> Mandelbrot drawing corroutine started.
<01:39:47> Julia drawing corroutine started.
<01:39:52> Julia drawing corroutine finished successfully in 5.04s!
<01:39:52> Mandelbrot drawing corroutine finished successfully in 5.551s!
Max Iterations: 100              Detail: 10000
Rendering Algorithm: Henriksen Algorithm    Family: Polynomial
Colormap: Colormap 1 (Sin)       Polynomial Degree: 2
```

(a) sin Colormap



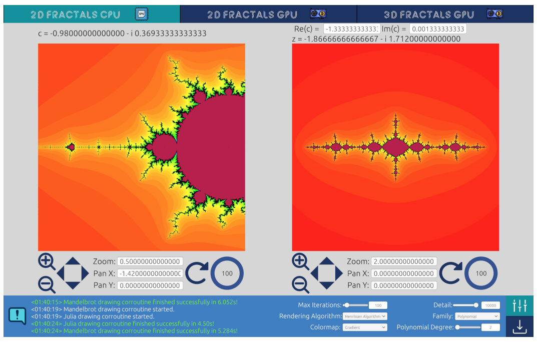
```
c = -0.93066666666667 - i 0.15066666666667
Re(c) = -1.33333333333: Im(c) = 0.001333333333
z = -1.97333333333333 - i 1.76000000000000
Zoom: 0.50000000000000    Zoom: 2.00000000000000
Pan X: -1.42000000000000  Pan X: 0.00000000000000
Pan Y: 0.00000000000000   Pan Y: 0.00000000000000
<01:39:52> Mandelbrot drawing corroutine finished successfully in 5.551s!
<01:40:00> Mandelbrot drawing corroutine started.
<01:40:00> Julia drawing corroutine started.
<01:40:05> Julia drawing corroutine finished successfully in 5.21s!
<01:40:06> Mandelbrot drawing corroutine finished successfully in 5.717s!
Max Iterations: 100              Detail: 10000
Rendering Algorithm: Henriksen Algorithm    Family: Polynomial
Colormap: Colormap 2 (Cos)       Polynomial Degree: 2
```

(b) cos Colormap



```
c = -0.98000000000000 - i 0.36933333333333
Re(c) = -1.33333333333: Im(c) = 0.001333333333
z = -1.86666666666667 - i 1.71200000000000
Zoom: 0.50000000000000    Zoom: 2.00000000000000
Pan X: -1.42000000000000  Pan X: 0.00000000000000
Pan Y: 0.00000000000000   Pan Y: 0.00000000000000
<01:40:06> Mandelbrot drawing corroutine finished successfully in 5.717s!
<01:40:09> Mandelbrot drawing corroutine started.
<01:40:09> Julia drawing corroutine started.
<01:40:15> Julia drawing corroutine finished successfully in 5.52s!
<01:40:15> Mandelbrot drawing corroutine finished successfully in 6.052s!
Max Iterations: 100              Detail: 10000
Rendering Algorithm: Henriksen Algorithm    Family: Polynomial
Colormap: Colormap 3 (Tan)       Polynomial Degree: 2
```

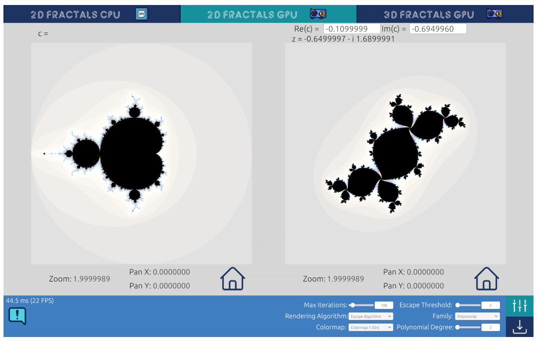
(c) tan Colormap



```
c = -0.98000000000000 - i 0.36933333333333
Re(c) = -1.33333333333: Im(c) = 0.001333333333
z = -1.86666666666667 - i 1.71200000000000
Zoom: 0.50000000000000    Zoom: 2.00000000000000
Pan X: -1.42000000000000  Pan X: 0.00000000000000
Pan Y: 0.00000000000000   Pan Y: 0.00000000000000
<01:40:15> Mandelbrot drawing corroutine finished successfully in 6.052s!
<01:40:19> Mandelbrot drawing corroutine started.
<01:40:19> Julia drawing corroutine started.
<01:40:24> Julia drawing corroutine finished successfully in 4.50s!
<01:40:24> Mandelbrot drawing corroutine finished successfully in 5.284s!
Max Iterations: 100              Detail: 10000
Rendering Algorithm: Henriksen Algorithm    Family: Polynomial
Colormap: Gradient               Polynomial Degree: 2
```

(d) Gradient Colormap

Figure B.3: Results for different colormaps using Henriksen Algorithm on CPU



c =

Re(c) = -0.1099999 Im(c) = -0.6949960
z = -0.6499997 - i 1.6899991

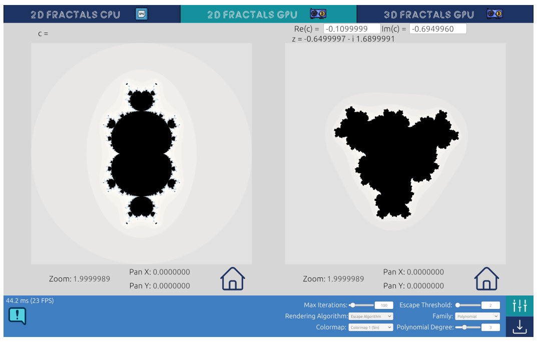
Zoom: 1.9999989 Pan X: 0.0000000 Pan Y: 0.0000000

Zoom: 1.9999989 Pan X: 0.0000000 Pan Y: 0.0000000

44.5 ms (22 FPS)

Max Iterations: 100 Escape Threshold: 2
Rendering Algorithm: Escape Algorithm Family: Polynomial
Colormap: Colormap 1 (Sin) Polynomial Degree: 2

(a) Mandelbrot and Julia sets for $z^2 + c$ on GPU



c =

Re(c) = -0.1099999 Im(c) = -0.6949960
z = -0.6499997 - i 1.6899991

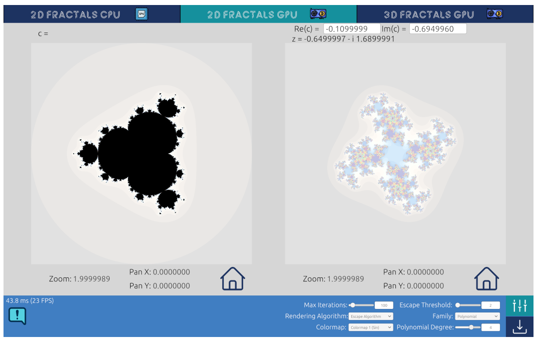
Zoom: 1.9999989 Pan X: 0.0000000 Pan Y: 0.0000000

Zoom: 1.9999989 Pan X: 0.0000000 Pan Y: 0.0000000

44.2 ms (23 FPS)

Max Iterations: 100 Escape Threshold: 2
Rendering Algorithm: Escape Algorithm Family: Polynomial
Colormap: Colormap 1 (Sin) Polynomial Degree: 3

(b) Mandelbrot and Julia sets for $z^3 + c$ on GPU



c =

Re(c) = -0.1099999 Im(c) = -0.6949960
z = -0.6499997 - i 1.6899991

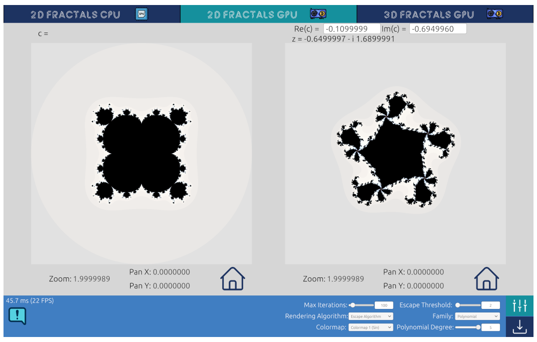
Zoom: 1.9999989 Pan X: 0.0000000 Pan Y: 0.0000000

Zoom: 1.9999989 Pan X: 0.0000000 Pan Y: 0.0000000

43.8 ms (23 FPS)

Max Iterations: 100 Escape Threshold: 2
Rendering Algorithm: Escape Algorithm Family: Polynomial
Colormap: Colormap 1 (Sin) Polynomial Degree: 4

(c) Mandelbrot and Julia sets for $z^4 + c$ on GPU



c =

Re(c) = -0.1099999 Im(c) = -0.6949960
z = -0.6499997 - i 1.6899991

Zoom: 1.9999989 Pan X: 0.0000000 Pan Y: 0.0000000

Zoom: 1.9999989 Pan X: 0.0000000 Pan Y: 0.0000000

45.7 ms (22 FPS)

Max Iterations: 100 Escape Threshold: 2
Rendering Algorithm: Escape Algorithm Family: Polynomial
Colormap: Colormap 1 (Sin) Polynomial Degree: 5

(d) Mandelbrot and Julia sets for $z^5 + c$ on GPU

Figure B.4: 2D GPU Scene rendering $z^d + c$ for $d = 2, 3, 4, 5$

Appendix C

Minimum Requirements

Here we describe the minimum requirements of the application for each platform.

Visit https://github.com/adry26/YetAnotherFractalExplorer for more information.

C.1 Desktop Platform

It requires a i5 Intel CPU at 2.0Ghz (or higher). A Nvidia GT 750M GPU (or higher) 2GB or more of RAM and 150MB of HDD Space. Minimum screen resolution 1280x800.

You can find the latests compiled version for Desktop on https://github.com/adry26/YetAnotherFractalExplorer/releases/latest.

To use this application, just download and run it.

C.2 Mobile Platform

Tested on an iPad Pro 2018. It may work on any iOS device running iOS 10.0 or higher.

Not tested on Android. It may work on any Android device running Android 4.4 KitKat (API level 19) or higher.

Pending to be published on App Store and Play Store. It will be published under the name "Yet Another Fractal Explorer".

C.3 Website Platform

You can find a web version of this project on https://adry26.github.io/YetAnotherFractalExplorer/

It requires a i5 Intel CPU at 2.0Ghz (or higher). A Nvidia GT 750M GPU (or higher) 2GB of RAM or more. Minimum screen resolution 1280x800.

It requires to use a browser compatible with WebGL and have WebGL enabled. Tested on Firefox 77.0.1 for MacOS and Brave Browser for MacOS Version 1.10.90 Chromium: 83.0.4103.97 (Official Build) (64-bit).

Bibliography

[1] Robert Brooks, J Peter Matelski, et al. Collars in kleinian groups. *Duke Mathematical Journal*, 49(1), 1982.

[2] John Milnor. *Dynamics in One Complex Variable*, volume 160. 2006.

[3] Núria Fagella and Xavier Jarque. *Iteración Compleja y Fractales*, volume 1. 2007.

[4] Lennart Carleson and Theodore W. Gamelin. *Complex Dynamics*, volume 1. 1993.

[5] Robert Devaney. *An introduction to chaotic dynamical systems*. TheBenjamin/CummingsPu- blishing co., 1986.

[6] I. N. Baker. Wandering domains in the iteration of entire functions. *Proceedings of the London Mathematical Society*, s3-49(3):563–576, 1984.

[7] Edward Angel, Dave Shreiner, et al. *Interactive computer graphics: a top-down approach with shader-based OpenGL*. Boston: Addison-Wesley,, 2012.

[8] Homogeneous coordinates. https://en.wikipedia.org/wiki/Homogeneous_coordinates. Last Accessed: June 21, 2020.

[9] James F Blinn. Models of light reflection for computer synthesized pictures. 1977.

[10] Bui Tuong Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6), 1975.

[11] Mandelbulb. https://en.wikipedia.org/wiki/Mandelbulb. Last Accessed: June 21, 2020.

[12] Ifs, iterated function system. https://en.wikipedia.org/wiki/Iterated_function_system. Last Accessed: June 21, 2020.

[13] Vasileios Drakopoulos, N Mimikou, and Theoharis Theoharis. An overview of parallel visualisation methods for mandelbrot and julia sets. *Computers & Graphics*, 27(4):635–646, 2003.

[14] Some algorithms for drawing mandelbrot and julia sets. https://www.mi.sanu.ac.rs/vismath/javier/b2.htm. Last Accessed: June 21, 2020.

[15] Distance estimators for fractals. https://www.iquilezles.org/www/articles/distancefractals/distancefractals.htm. Last Accessed: June 21, 2020.

[16] Tomasz Martyn. Realistic rendering 3d ifs fractals in real-time with graphics accelerators. *Computers & Graphics*, 34(2):167–175, 2010.

[17] Z-buffer. https://en.wikipedia.org/wiki/Z-buffering. Last Accessed: June 21, 2020.

[18] Ray-tracing. https://en.wikipedia.org/wiki/Ray_tracing_(graphics). Last Accessed: June 21, 2020.

[19] Ray-marching. https://en.wikipedia.org/wiki/Volume_ray_casting. Last Accessed: June 21, 2020.

[20] Shadowing on fractals. https://www.iquilezles.org/www/articles/mandelbulb/mandelbulb.htm. Last Accessed: June 21, 2020.

[21] Mandel. http://www.mndynamics.com/. Last Accessed: June 21, 2020.

[22] Mandelbulber. https://www.mandelbulber.com/. Last Accessed: June 21, 2020.

[23] Frax. http://fract.al/. Last Accessed: June 21, 2020.

[24] Xaosjs. https://xaos-project.github.io/. Last Accessed: June 21, 2020.

[25] Unity technologies. https://unity.com/. Last Accessed: June 21, 2020.

[26] Webgl. https://en.wikipedia.org/wiki/WebGL. Last Accessed: June 21, 2020.

[27] Webgl and threads. https://docs.unity3d.com/Manual/webgl-gettingstarted.html. Last Accessed: June 21, 2020.

[28] Ray-marching algorithm. http://jamie-wong.com/2016/07/15/ray-marching-signed-distance-functions/#:~:text=In%20raymarching%2C%20the%20entire%20scene,of%20a%20signed%20distance%20function.&text=Instead%20of%20taking%20a%20tiny,which%20the%20SDF%20provides%20us! Last Accessed: June 21, 2020.

[29] Mandelbulb formulation. http://blog.hvidtfeldts.net/index.php/2011/09/distance-estimated-3d-fractals-v-the-mandelbulb-different-de-approximations/. Last Accessed: June 21, 2020.

[30] Sierpinski tetrahedron. https://en.wikipedia.org/wiki/Sierpi%C5%84ski_triangle. Last Accessed: June 21, 2020.

[31] Menger sponge. https://en.wikipedia.org/wiki/Menger_sponge. Last Accessed: June 21, 2020.

[32] Sierpiński tetrahedron formulation. http://blog.hvidtfeldts.net/index.php/2011/08/distance-estimated-3d-fractals-iii-folding-space/. Last Accessed: June 21, 2020.

[33] Menger sponge formulation. https://www.iquilezles.org/www/articles/menger/menger.htm. Last Accessed: June 21, 2020.

[34] Cpu benchmark specification. https://ark.intel.com/content/www/es/es/ark/products/76086/intel-core-i7-4850hq-processor-6m-cache-up-to-3-50-ghz.html. Last Accessed: June 21, 2020.

[35] Gpu benchmark specification. https://www.notebookcheck.net/NVIDIA-GeForce-GT-750M.90245.0.html. Last Accessed: June 21, 2020.

[36] Macbook pro retina late 2013, 15 inch specs. https://everymac.com/systems/apple/macbook_pro/specs/macbook-pro-core-i7-2.3-15-dual-graphics-late-2013-retina-display-specs.html#macspecs1. Last Accessed: June 21, 2020.

[37] Finite difference method. https://en.wikipedia.org/wiki/Finite_difference_method. Last Accessed: June 21, 2020.

[38] Gpu rank. https://www.videocardbenchmark.net/gpu.php?gpu=GeForce+GT+750M&id=2492. Last Accessed: June 21, 2020.

Figures licences

[39] Elphaba. *Wikipedia* under free license.

[40] J. de Vries. *Learn OpenGl* under CC BY 4.0 license.

[41] J. de Vries. *Learn OpenGl* under CC BY 4.0 license.

[42] Pere Gilabert Roca. *NPR Shading Evaluation on Virtual Reality* under CC BY 4.0 license.

[43] Angel, Edward and Shreiner, Dave and others. *Interactive computer graphics: a top-down approach with shader-based OpenGL* under CC BY 4.0 license.

[44] M. Kraus *Wikipedia* under CC0 license.

[45] B. Smith *Wikipedia* under CC BY-SA 3.0 license.

[46] Henrik. *Wikipedia* under CC BY-SA 4.0 license.

[47] Nvidia. *GPU Gems 2: Chapter 8*

Software Assets Licenses

[48] Gökhan Gökçe. *UnityStandaloneFileBrowser* URL: https://github.com/gkngkc/UnityStandaloneFileBrowser under MIT license.

[49] Ryan. *Unity-native-toolkit* URL: https://github.com/ryanw3bb/unity-native-toolkit under MIT license.

List of Figures

List of Tables