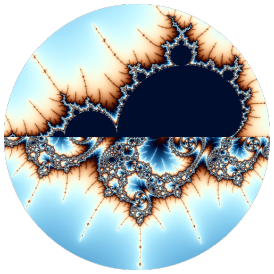


Mandelbrot Maps

WebGL Application for Exploring Fractals

João Filipe Maio

MInf Project (Part 1) Report

Master of Informatics
School of Informatics
University of Edinburgh

April 19, 2020

Abstract

Mandelbrot Maps is a running project in the School of Informatics – it was originally a Java applet for visualisation and manipulation of fractals, specifically the Mandelbrot and Julia sets. Unfortunately, the original is no longer supported by today's browsers, and the more recent implementations are restricted to a single platform, namely Android.

This report showcases a new version of the project, which leverages advanced processing capabilities of mobile devices in gesture support and compute power using their GPU (Graphics Processing Unit), and pairs them with performance-focused technologies like React and WebGL, to open the application to more devices than ever before.

Additionally, I explain the process behind the design and implementation of the application, both for the user interface (UI) and the renderer, and evaluate it with resort to user feedback provided by students from the School of Informatics. Finally, I introduce an outline for further development next year.

The application is available at the following URL:

jmaio.github.io/mandelbrot-maps

Source code is available on GitHub:

github.com/JMaio/mandelbrot-maps

Acknowledgements

I would like to thank

Philip Wadler, my project supervisor, for his guidance and feedback on my work.

Asmita Dulan for lending me her knowledge of React.

My parents for their constant love and support.

Table of Contents

Chapter 1

Introduction

1.1 An Ode to the Modern Web

Mandelbrot Maps was originally a Java applet, which was created by Iain Parris in 2008 (Figure 1.1). Intended to be used on the web at the time, it was unfortunately based on a technology which would not stand the test of time a decade later:

> *Oracle announced in January 2016 that Applets would be deprecated in Java SE 9, and removed from Java SE 11 (18.9).* – Oracle [1]

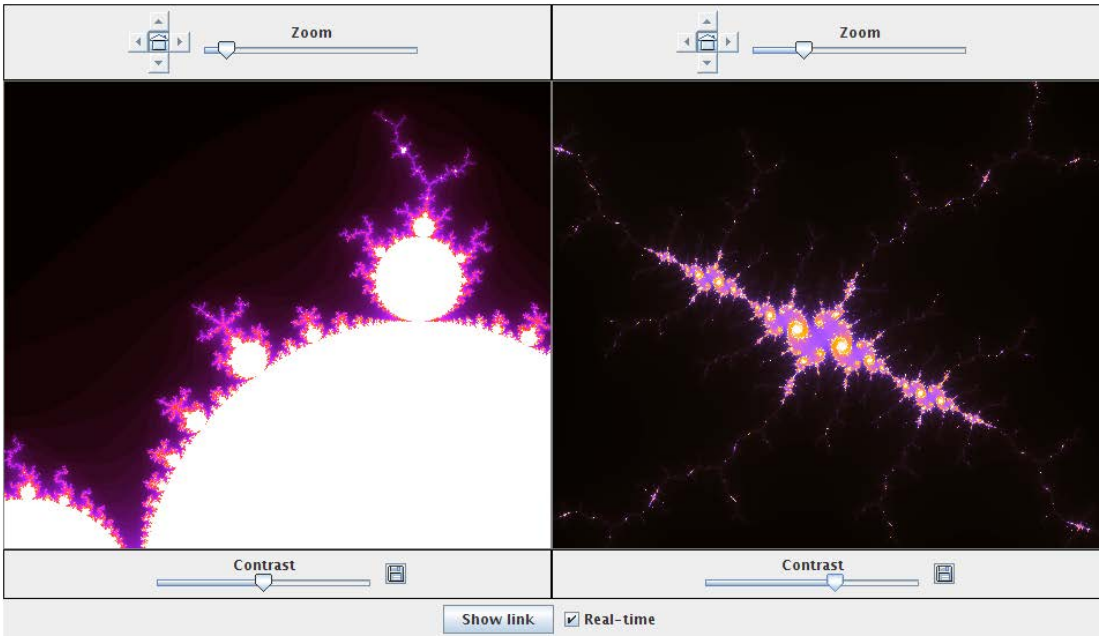


Figure 1.1: A screenshot of Parris' original application (from the original paper).

Java Applets were introduced as part of the Java language in 1995, with Java version `1.0` being made publicly available in January 1996 [2]. The principal appeal of Java applets was their interactivity at a time when the web was mostly static, which, paired with good performance, made it a popular choice for early web applications. This was,

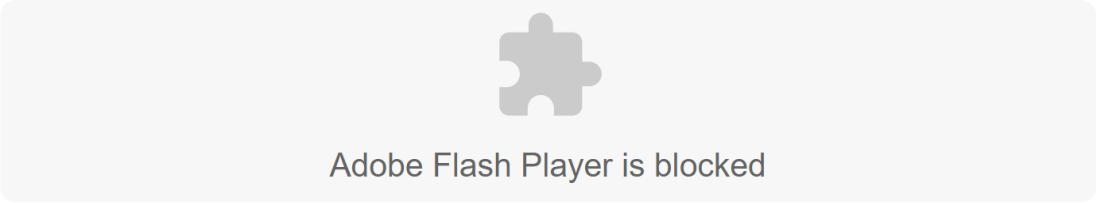


Figure 1.2: The policy of web browsers today is to block Flash content by default.

however, not the only technology of the kind. *Adobe Flash* was another competing technology which had once dominated the web, yet is now almost entirely phased out:

> *But as open standards like HTML5, WebGL and WebAssembly have matured over the past several years, most now provide many of the capabilities and functionalities that plugins pioneered and have become a viable alternative for content on the web.* – Adobe [3]

Adobe Flash was introduced in January 1996 under the name of *Macromedia Flash*, prior to its acquisition by Adobe in 2005. Its high popularity stemmed from its advanced support for vector graphics, 3D graphics, embedded audio, video and raster graphics [4], which meant that it could be used for a number of different applications, from online 'Flash' games, to video and music players. YouTube famously used Flash for its video player until it began supporting the HTML5 `<video>` tag in 2010 [5], and by 2015 had made it the default choice over Flash [6].

Developments in mobile technology were also a driving force for standards such as HTML5 to include more advanced functionality, partly due to the novelty of touch-screen devices which provided a new way to browse the web. Apple was one of the companies pioneering in this space with their 2007 release of the original iPhone [7]. It supported what was, at least at the time, "advanced web browsing", but had a major omission in that Flash was not allowed on the iPhone in any way. The reasoning behind this is that Apple's Terms of Service prohibit unauthorised applications from running on their devices [8].

As Flash was basically a development platform which allowed for developers to create and publish their own applications online, it may have enabled the formation of an app market parallel to Apple's App Store. This market would have been unregulated, giving developers the ability to publish any content without the need to pass any of Apple's stringent checks. The potential for an unregulated parallel market – and arguably also the financial detriment it could cause to Apple – meant that it was never in Apple's interest to allow Flash to make its way to their mobile operating systems.

Apple was not the only company that fought to keep Flash out of the browser, with others like Google and Microsoft joining in, citing that the implementation of Flash had severe security flaws by nature of running its own processes within the web browser, potentially allowing unauthorised access to users' data [9, 10] (Figure 1.2). Newly found vulnerabilities were constantly hurting Flash's reputation to the point where even Adobe decided that it should no longer be maintained, and instead moved its efforts to support open standards like HTML5.

With the gap that Flash was leaving on the web, it was necessary to leverage other technologies to achieve similar functionality to what Flash had provided in interactive content distribution for years. HTML5, JavaScript, WebGL, and WebAssembly are some of the technologies which came to fill in that gap, and have matured over the years to the point where they are now considered fully-featured [11]. Since these newer technologies were quickly proving to be viable alternatives to replace most of Flash's functionality, Java Applets were also a target, and both technologies began dropping in popularity as the new standards were embraced.

1.2 Project Aim

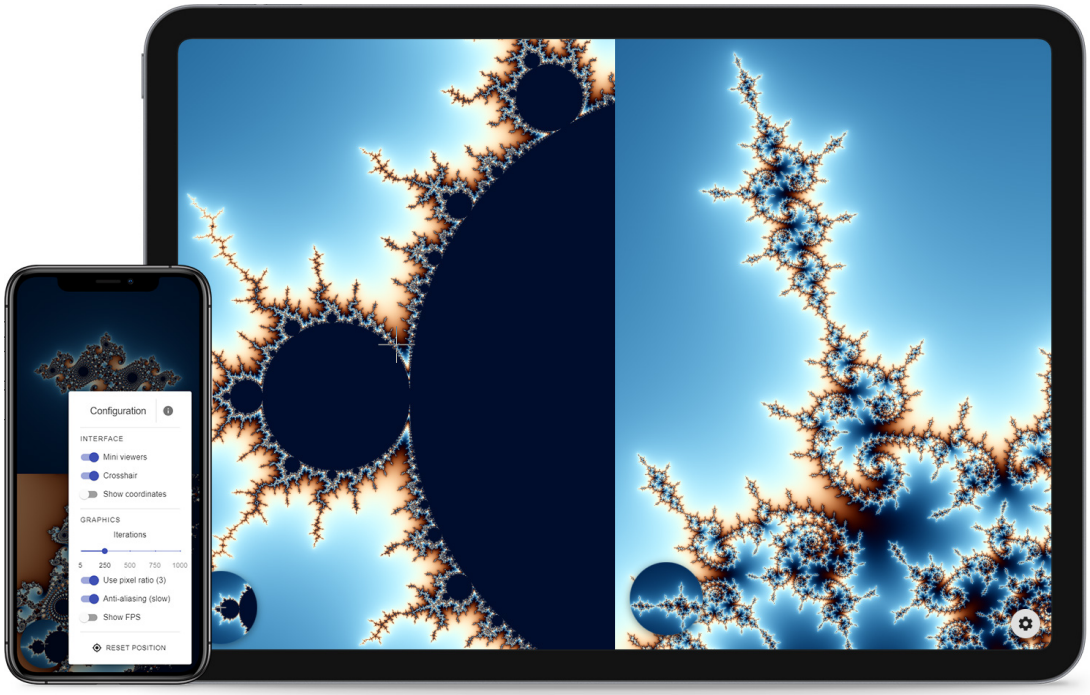


Figure 1.3: Using the Mandelbrot Maps application on an iPhone and iPad (screen images simulated).

Despite the deprecation of Java Applets and Adobe Flash, there are exciting opportunities for the technologies which superseded them. Previous versions of the *Mandelbrot Maps* application have normally only targeted a single computing platform, with Parris' Java applet (now deprecated) targeting large-form-factor devices (like desktops and laptops), and the Android app first created by Alasdair Corbett [12] targeting small-form-factor Android phones and tablets with touchscreens.

The opportunity here is that, after many years of fragmentation on the web with Flash and Java applets, the new open standards are now considered mature, and so can provide a consistent experience across a large portion of devices regardless of platform, physical size, screen resolution, input devices, or other variables.

The aim of this project is to bring Mandelbrot Maps to the modern web, using JavaScript (with React) and WebGL (with a device's Graphics Processing Unit – GPU) to support

more devices than ever before, while providing touchscreen and gesture functionality for mobile devices (Figure 1.3). The application should allow users to easily explore the complex plane and study the intricate relationship between the Mandelbrot set and the Julia set, with no setup required, and independent of device choice. Additionally, providing a smooth, responsive experience, coupled with a polished and intuitive user interface are also key considerations for this project.

1.3 Report Overview

This report is meant as a self-contained introduction and description of the project, outlining the design decisions and rationale behind them. The following chapters are:

Chapter 2 provides a quick catch-up on the topic of fractals and how they are generated.

Chapter 3 reviews applications which have influenced this version of Mandelbrot Maps.

Chapter 4 explains the technology behind the application, along with the implementation decisions and justifications, with both a high-level overview and in-depth detail.

Chapter 5 evaluates the application and analyses the findings from the user survey.

Chapter 6 concludes with an overview, and outlines the work to be done in the second part of the project.

1.4 Project Coordination

This year, both myself and another student, Freddie Bawden, were assigned this project. Since we were both intent on porting the project over to the web, we had initial discussions on designing a *common API* that our application components would implement. This was intended to make the front-ends and renderers inter-compatible, so that we could switch out different parts of the applications easily.

One of the use cases was that at the beginning of the project, while Freddie was developing his renderer, I was developing my user interface (UI), and we could have merged the two parts to let us iterate more quickly. Later on, I would have built my renderer on top of my UI, and Freddie would have developed his UI around his renderer, keeping our projects totally separate while still making them fully compatible with one another. Technically, anyone could have created their own renderer to use with our applications!

In the end, Freddie chose to use WebAssembly to implement his renderer, whereas I chose WebGL for my renderer. We agreed on a basic specification of how to set up our renderers, but unfortunately the technologies that we had chosen were not compatible (mostly due to the way WebGL works), so we were unable to develop this further.

Chapter 2

Fractals

The term *fractal* is generally used to refer to structures which exhibit some kind of self-similarity at different scales, and was introduced by mathematician Benoît Mandelbrot, who wrote about the topic in *The Fractal Geometry of Nature* [13]. Real-life examples of this fractal property can be observed in romanesco broccoli (Figure 2.1), mountains, lightning, peacock feathers, and more [14].

In this section, I take a generally algorithmic approach to generating Mandelbrot and Julia sets, with reduced mathematical rigour. For a more thorough mathematical approach, I'd recommend having a look at the WolframAlpha [15, 16] and Wikipedia [17, 18] pages on Mandelbrot and Julia sets.

Figure 2.1: Romanesco broccoli exhibits fractal properties and is especially interesting because the number of spirals is a Fibonacci number [19] (© Lars Kristensen – flickr.com/photos/133645177@N03/21194171451/)

2.1 The Complex Plane

The mathematics behind the Mandelbrot set is fairly simple and straightforward, even with little knowledge of complex numbers. The main fact to note is that a complex number can be expressed in the form:

$$z = a + bi \quad \text{where } a, b \in \mathbb{R} \text{ (real numbers) and } i = \sqrt{-1} \tag{2.1}$$

An interesting and useful property of complex numbers is that they can be thought of as a function of two real numbers – the *Real* component a and the *Imaginary* component b – so they can easily be visualised on a two-dimensional plot, with the Real component mapped to the horizontal axis and the Imaginary component mapped to the vertical axis. This is referred to as the *complex plane*.

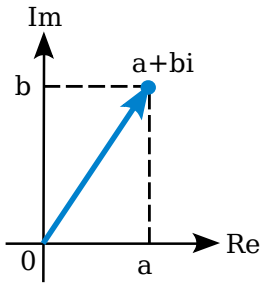


Figure 2.2: A plot of a complex number (© Wolfkeeper at English Wikipedia, CC BY-SA 3.0)

2.2 The Mandelbrot set

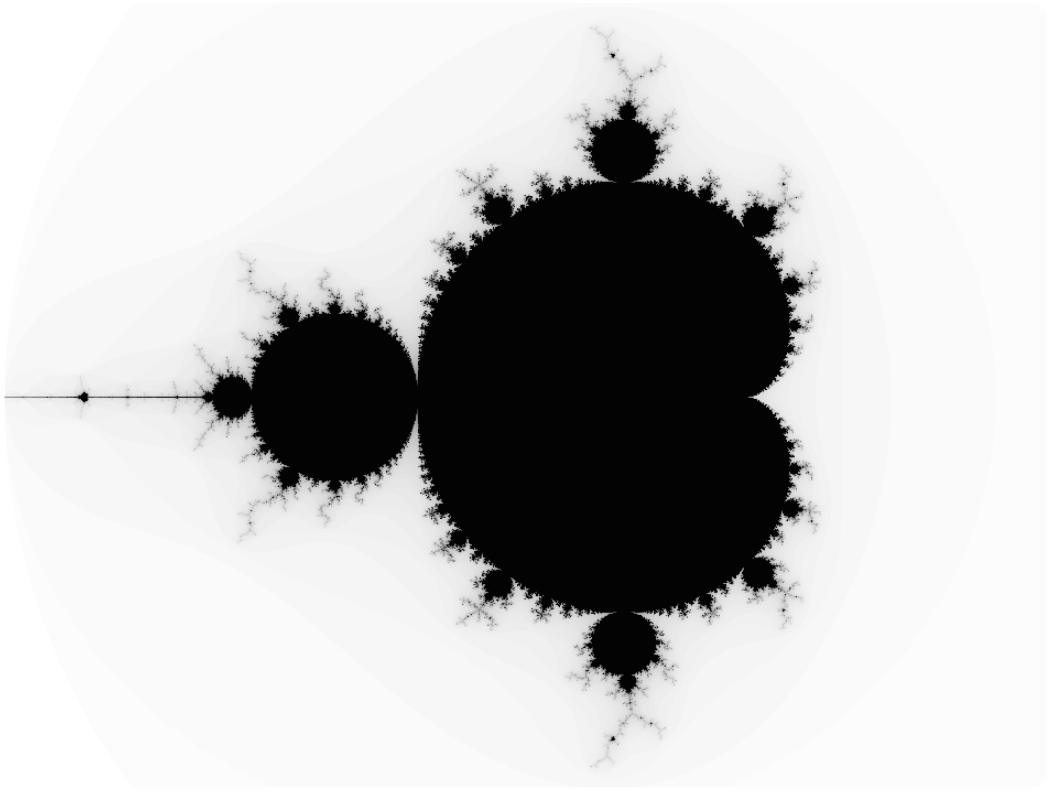


Figure 2.3: The Mandelbrot set, as generated by an early prototype of this application (100 iterations). A gradient effect is achieved by calculating the number of iterations before z diverges and dividing by the maximum iterations to obtain a percentage grey level. 100% grey (black) indicates non-diverging z, with lighter grey indicating fewer iterations until z diverges.

The Mandelbrot set is named after mathematician Benoît Mandelbrot, and is one of the most famous fractals in existence. It is defined for a point in the complex plane in terms of the following quadratic recurrence:

$$z_{n+1} = z_n^2 + c \tag{2.2}$$

Here, $z_0 = 0$ and c is a point on the complex plane. This recurrence is simply the repeated application of a function to its previous output (similar to the Fibonacci sequence). This algorithm is *iterative* given that each z is calculated (*iterated*) from the previous:

$$\begin{aligned}
z_0 &= 0 \\
z_1 &= (z_0)^2 + c & &= 0 + c & &= c \\
z_2 &= (z_1)^2 + c & &= (c)^2 + c & &= c^2 + c \\
z_3 &= (z_2)^2 + c & &= (c^2 + c)^2 + c & &= c^4 + 2c^3 + c^2 + c \\
z_4 &= (z_3)^2 + c & &= \ldots
\end{aligned}$$

The criteria for a point c to be part of the Mandelbrot set is that it *remains bounded* [17], and thus must not diverge under iteration. Calculating this property becomes mostly trivial with the help of a computer: simply input the point to be tested and continue to iterate according to Equation 2.2. To ease these kinds of calculations, c and z can be represented as 2-dimensional vectors, since many computer systems are heavily optimised for vector operations, which is especially true of GPUs [20, 21].

To assert that a point is indeed in the Mandelbrot set, one would technically need to perform this iteration an infinite amount of times. For this reason, it is standard practice to limit the number of iterations to a given maximum I_{max}, normally between 100 to 1000 iterations, since the approximation produced is fast and good enough for visualisation purposes. Joining up the points from each iteration results in a spiral pattern, which can either converge or diverge, as shown in Figure 2.4.

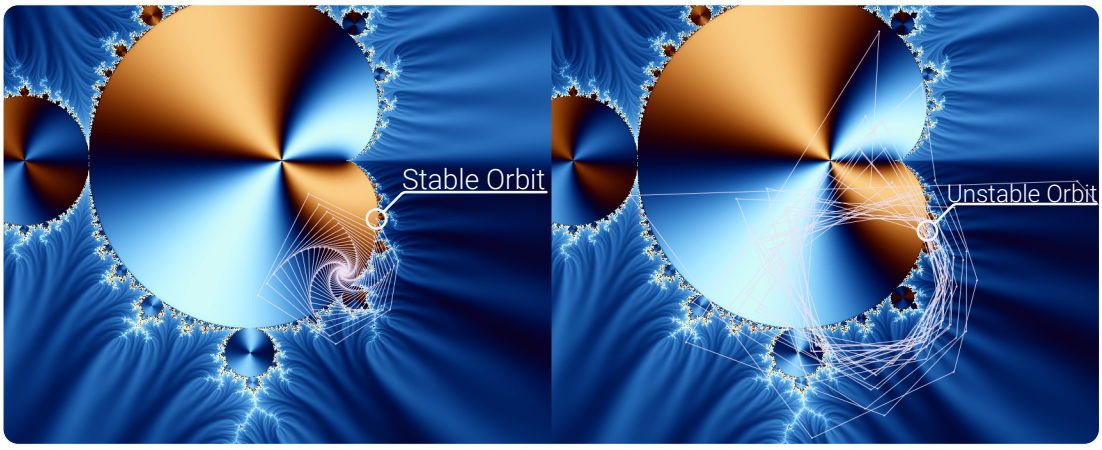


Figure 2.4: Iterating two spatially close points (each highlighted by the white circle) results in radically different outcomes. The traces represent the path from one iteration to the next. The figure on the left has a stable orbit which converges, and thus the initial value of c is in the Mandelbrot set, whereas the one on the right is unstable and diverges, so is not part of the Mandelbrot set. (From DeepFractal.js, Section 3.4)

To optimise the running time of the algorithm, we specify a radial distance at which we know that a given point will diverge. This boundary is usually set at a radius of size 2, where points that fall outside the boundary are guaranteed to diverge. Rather than only wanting a binary *yes* / *no* for whether a point is in the Mandelbrot set, it would also be useful to know how many iterations it has taken for z to diverge.

The ***escape-time algorithm*** is an extension of the basic iterative algorithm that returns the number of iterations taken for z to surpass the radial boundary. To do this, we either return the number of iterations n that have elapsed until z has diverged past the radial boundary, or I_{max} if z has not diverged. That iteration count n can then be used for colouring the plot. By dividing n by I_{max}, it's possible to create regions with *fractional* shading, as seen in Figure 2.3. Any number of arbitrary functions can be applied in this manner to create interesting ways to shade the Mandelbrot set.

2.3 Julia sets

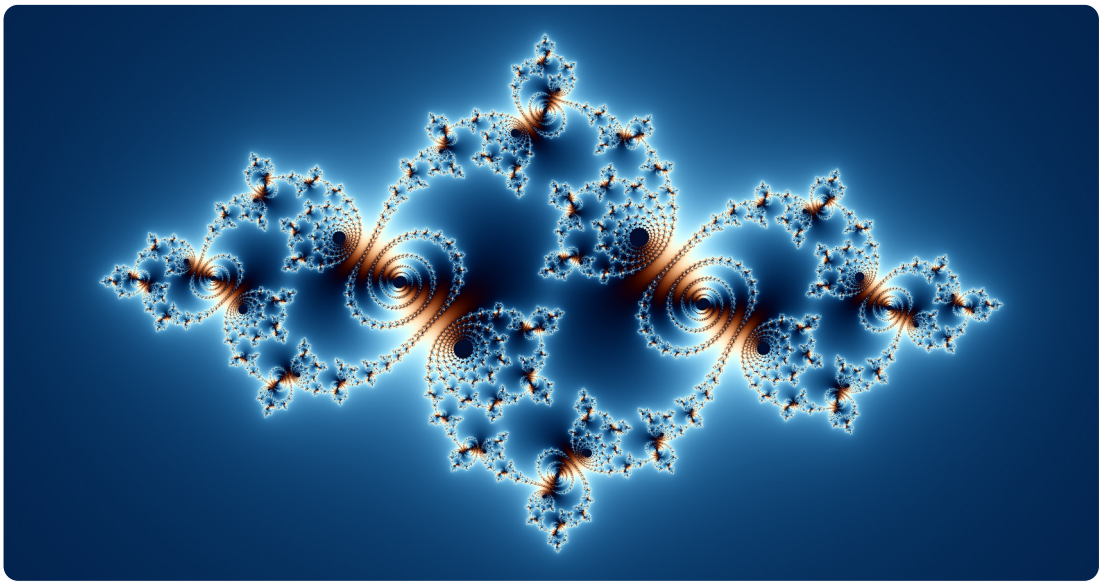


Figure 2.5: An image of a Julia set from Mandelbrot Maps.

Julia sets are named after mathematician Gaston Julia. Note that although there is a single, unique Mandelbrot set, there is one Julia set for each point in the complex plane. The term *Julia set* usually refers to only the boundary of a *filled Julia set*, but we will use the term *Julia set* to refer to a *filled Julia set* for simplicity.

The definition of a Julia set is similar to the Mandelbrot set, so the two have intrinsic connections, meaning that some of Section 2.2 is applicable here as well. A Julia set is also defined in terms of the quadratic recurrence:

$$z_{n+1} = z_n^2 + c \tag{2.3}$$

Here, however, z_0 is the point being tested, and c is a constant point which we can choose to be any point on the complex plane. The Julia set for c consists of all z_0 such

that $z_{n+1} = z_n^2 + c$ does not diverge. Since the Mandelbrot set is only dependent on a single point, we can simply plot it without issue. With a Julia set, however, we have a dependence on c, so we need to have a way to choose that point c. Naturally, since we are plotting the Mandelbrot set on the complex plane, it makes sense to use that same complex plane as a way to pick out c. By having it such that we pick c for the Julia set 'from' the Mandelbrot set, the connection between the two is emphasised. Two interesting connections that I would like to highlight are that:

1. If c is chosen such that it is in the Mandelbrot set, i.e. c does not diverge under iteration, then the associated Julia set is *connected*. This can also be defined in reverse, so the Mandelbrot set is actually the set of points c where the associated Julia set is connected [13, Chapter 19]. Being *connected* implies that the Julia set has no 'breaks' or 'gaps' in its internal structure. Figure 2.5 for example, is not connected and so its value of c is not in the Mandelbrot set, and neither are the examples in Figure 2.6 – they are disconnected.

2. Some regions of a Julia set resemble the original region in the Mandelbrot set where point c was selected – this was proven by Tan Lei and is known as Tan's theorem [22]. A few examples are shown in Figure 2.7.

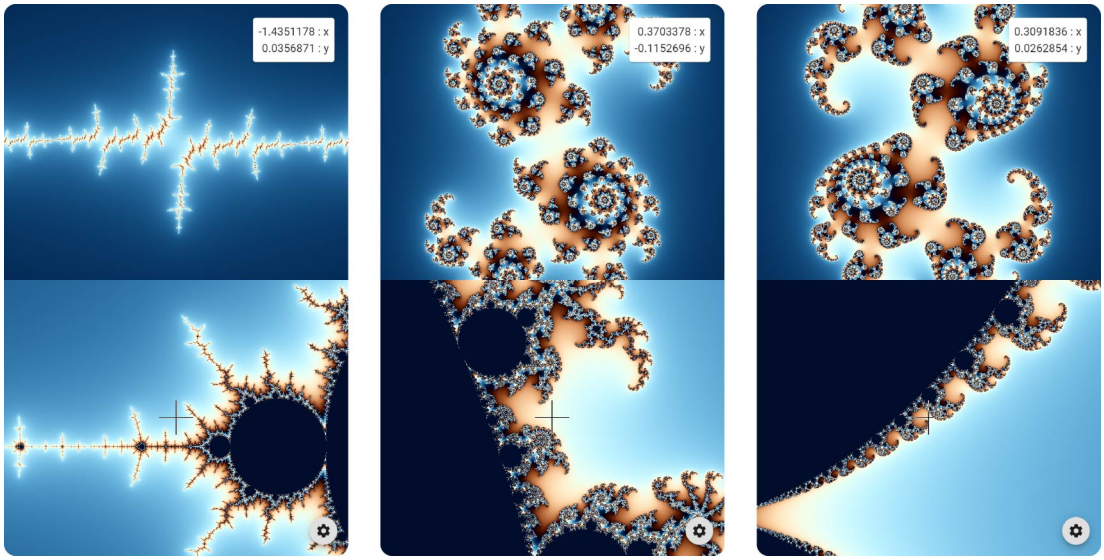


Figure 2.6: Selecting c outside the Mandelbrot set means that the corresponding Julia set is disconnected.

2.4 Parallel Computation and Optimisation

This section describes optimisations we can apply on top of the escape-time algorithm.

This could mean using optimised data structures like matrices – the Python package `numpy` comes to mind – or simply using more low-level languages that already provide fast routines – like C, which is what `numpy` uses to improve performance.

Another optimisation is to enable the algorithm to run in parallel. As it happens, GPUs are perfect for this job, since they are made to handle highly-parallelised workloads.

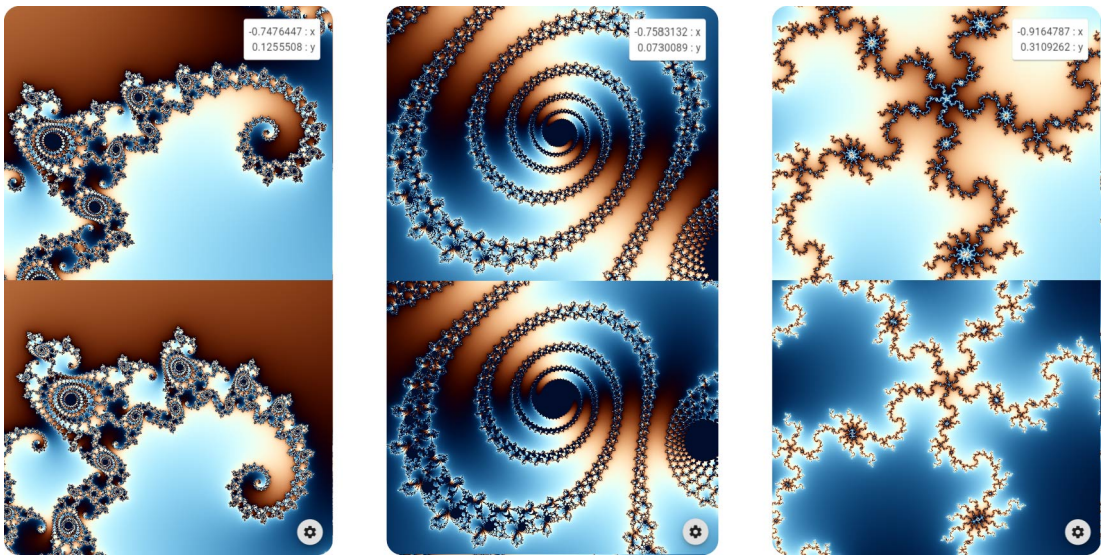


Figure 2.7: Some regions of a Julia set (top) resemble the original region in the Mandelbrot set (bottom) where point c was selected.

GPUs can run programs called *shaders*: small pieces of code that are compiled into highly-optimised code meant specifically for running in parallel.

On top of this, there are other ways to improve performance purely by simplifying the problem. The real number line, and thus by extension, the complex plane, are continuous, meaning that there is an infinite number of points that need to be put through our algorithm. Computer screens on the other hand, they give the appearance of being continuous even though they are really just quantised, finite collections of tiny dots. Thus, a 100×100 grid of pixels can be used to project any part of the Mandelbrot set, with each pixel mapped to a corresponding complex coordinate. This cuts down the work from an infinite amount of points to a mere 10,000 points.

When checking whether z is still within the radial boundary, some arithmetic can be simplified by reducing the number of operations. For this calculation, we can optimise it by removing a square-root operation as follows:

$$z \text{ diverges if: } \quad |z_n| > 2 \equiv \sqrt{|z_n|^2} > 2 \equiv \sqrt{z_n \cdot z_n} > 2 \equiv (\sqrt{z_n \cdot z_n})^2 > 2^2 \equiv z_n \cdot z_n > 4$$

Another optimisation is to bypass calculations in certain regions of the Mandelbrot set. By excluding some regions that are known to not diverge (and so take longest to iterate), we can reduce the calculation time significantly. Two large regions that we exclude are the main cardioid (M1, boundary $c = e^{i\theta}/2 - (e^{i\theta}/2)^2$) and the period-2 bulb (M2, a circle of radius ¼ centred at $(-1,0)$), shown in Figure 2.8. Combined, they represent over 90% of the area of the whole Mandelbrot set [23].

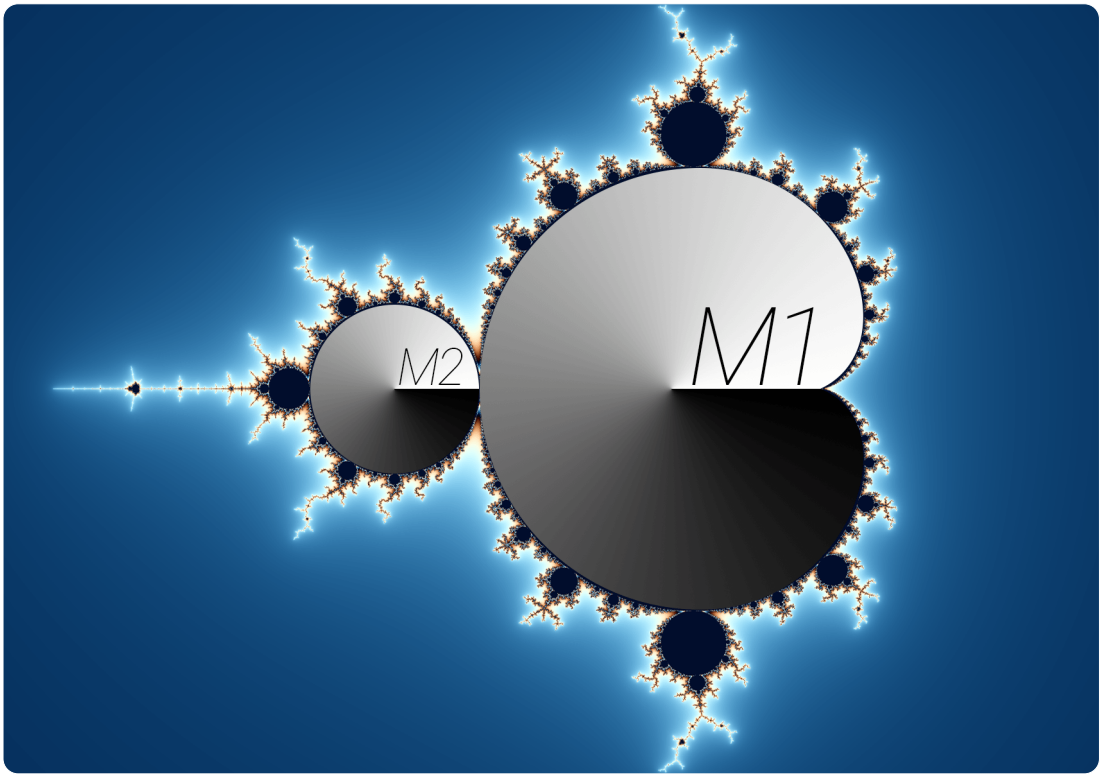


Figure 2.8: An annotated diagram of the main parts of the Mandelbrot set, M1 and M2.

Chapter 3

Existing Fractal Explorers

Drawing inspiration from the original paper by Iain Parris, the following is a selection of some notable applications which have influenced certain aspects of this implementation of Mandelbrot Maps.

The applications discussed dedicate extensive use of screen space to displaying graphics, minimising the user interface to provide a distraction-free, immersive experience.

3.1 Google Maps – The *Gold Standard*

google.com/maps

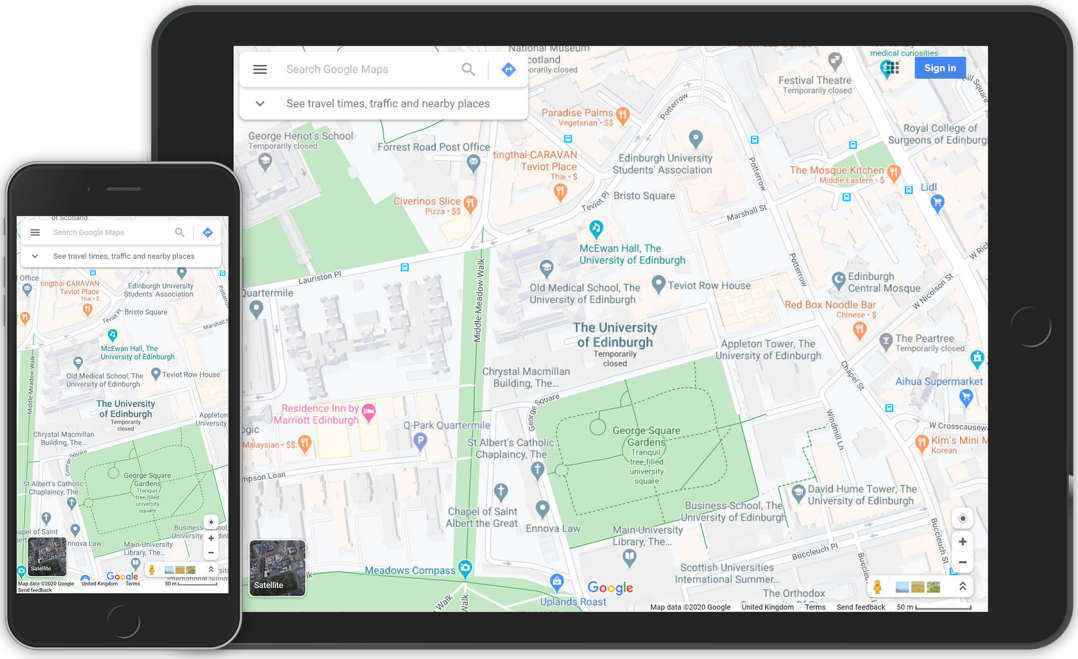


Figure 3.1: Google Maps (web version) running on an iPhone and iPad.

Strictly speaking, Google Maps *could* be considered a fractal explorer – coastlines, for example, are fractal in nature, as Mandelbrot himself writes in *The Fractal Geometry of*

Nature, Chapter 5 – How Long Is the Coast of Britain?. Here, however, we present it for being one of the most popular maps providers, used by millions of people worldwide, and the main source of inspiration behind Parris' original Java applet.

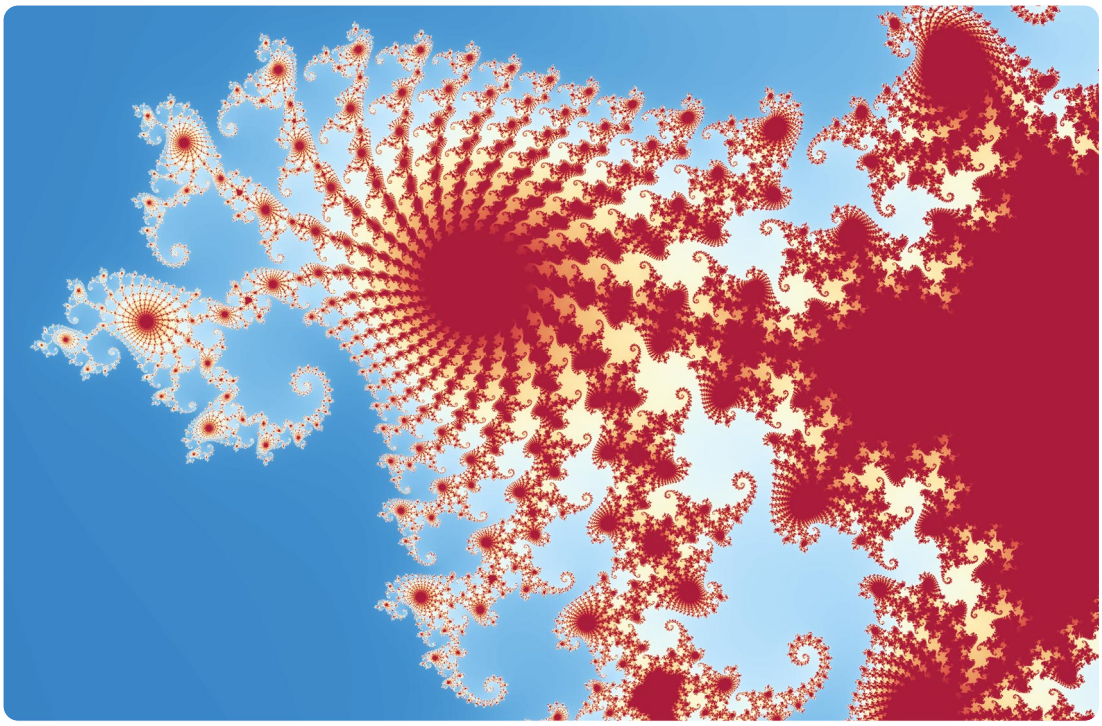
On Google Maps' web interface, the theme is consistent: small, relevant controls pulled to the very corners of the screen in an effort to display as much of the map as possible. It becomes evident from moving around that Google is separating the view into smaller regions, inserting the correct textures at each one as they become available. There is also support for URL parameters, in the format (latitude, longitude, zoom):

`google.com/maps/@55.944781,-3.187282,17z`

The controls are exceptional. On a desktop, dragging to pan is easy, and zooming in is smooth, especially when using a touchpad. On a touchscreen, the process is also smooth, responsive, and accurate. Movements decay away once a user has stopped interacting – they have associated 'momentum', 'velocity', or 'inertia', a feature which is meant to imitate real-world physics and make the interface feel more fluid.

3.2 Mandelbrot Svelte – by `lovasoa` (Ophir Lojkine)

`mandelbrot.ophir.dev`



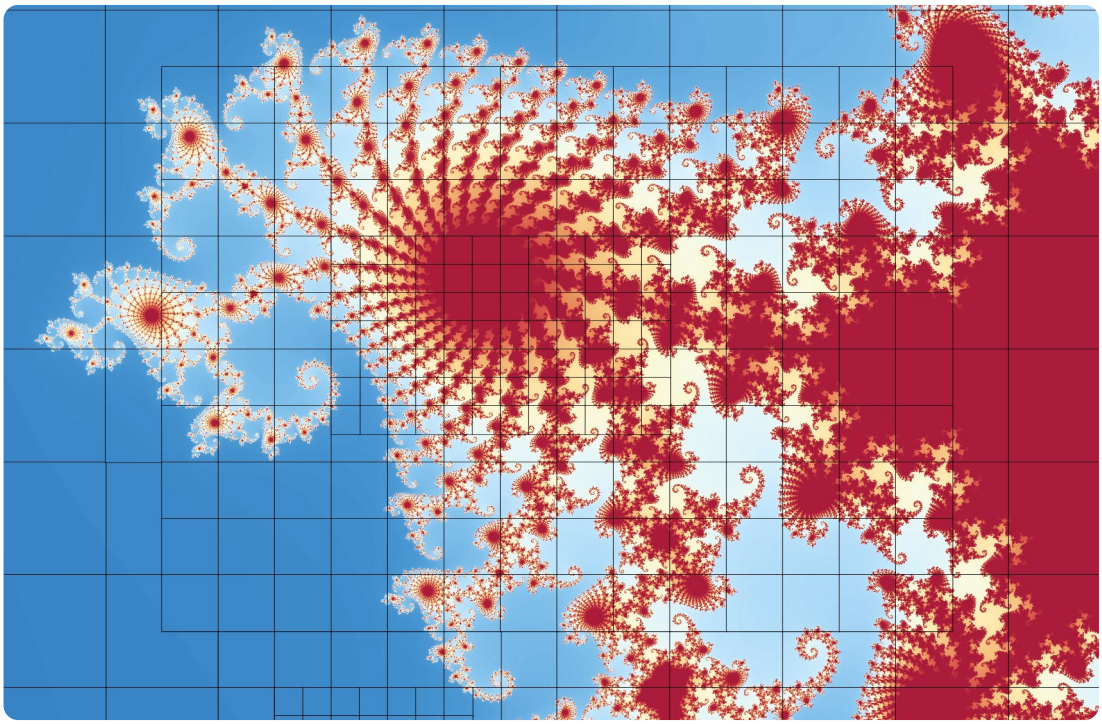
(a) A wide view of "seahorse valley"

Figure 3.2: Mandelbrot Svelte

Mandelbrot Svelte has very good execution of two key aspects: interactivity and a minimal user interface. Both **mouse and touch gestures are supported**, and there is even a form of **movement momentum** present which helps with navigation. Performance is also very good considering that the app runs on JavaScript using only the CPU, since

using the CPU can lead to lower overall performance. The application can also achieve very high levels of zoom before it begins to lose detail.

One of the tricks behind why the app performs so well in these areas is rather clever: a tiling system. Upon analysis of the source code, the application appears to be creating and manipulating small *tiles*: images that are displayed seamlessly, and stitched together to fill the screen, to show the full target image. This approach brings about a natural way to improve rendering performance by parallelising: dividing the relatively large amount of work into smaller, more manageable chunks that can be calculated simultaneously.



(b) Figure 3.2a overlaid with a border at each tile edge. A dynamic tiling system allows for faster rendering by dividing the drawing region into smaller sections, and intelligently manipulating them for better performance.

Figure 3.2: Mandelbrot Svelte (cont'd)

In the documentation, the author indeed confirms that "*the fractal is split into square tiles*", and further explains that the computation behind what is displayed on these tiles is based on the current zoom level, with the transition to new zoom levels triggering a re-computation. Previous tiles are then *cached* and scaled according to the new zoom level to provide an intermediate preview before the new tiles are available. This is not computationally intensive and improves the smoothness of panning and scrolling actions.

In terms of colour, the application features an interesting palette, going from deep blues to deep reds through some lighter shades of orange, yellow, and sky blue. The transitions between different regions are clearly visible, unlike some of the other applications, but they are subtle enough that it should not be considered an issue.

A notable feature is the inclusion of **URL parameters**. Any movement from the user

triggers the URL to update to the current point in the view, specified by the parameters `x`, `y`, and `zoom`, in this format:

```
mandelbrot.ophir.dev/#{"pos":{"x":-0.745,"y":0.125},"zoom":128000}
```

The execution of this feature is very simple, but highly effective, since it does not need any form of user interface other than what a web browser would already offer (in the shape of an address bar) for saving or revisiting a point. This does make entering a point somewhat tricky: if a user wants to manually specify a new region of the view to navigate to, they will have to fiddle with the parameters in the address bar. A potential solution may be to keep the basic implementation while supplying an additional system to allow a user to manually specify a point more easily, like through a dedicated input field. On a side note, the application is linked to on the 'Mandelbrot set' Wikipedia page as an interactive viewer for points in the image gallery.

Although the app does appear to be loaded with sensible settings by default, it unfortunately **does not provide the user with any configuration options**. This includes the maximum iteration count and the colour scheme, neither of which are user-accessible. Another unfortunate omission is the absence of the Julia set. Perhaps not needing to consider any other 'moving parts' is what makes this application perform so well in terms of performance and ease of use.

3.3 Mandelbrot WebGL – by `Syntopia` (Mikael Hvidtfeldt Christensen)

`hvidtfeldts.net/WebGL/webgl.html`

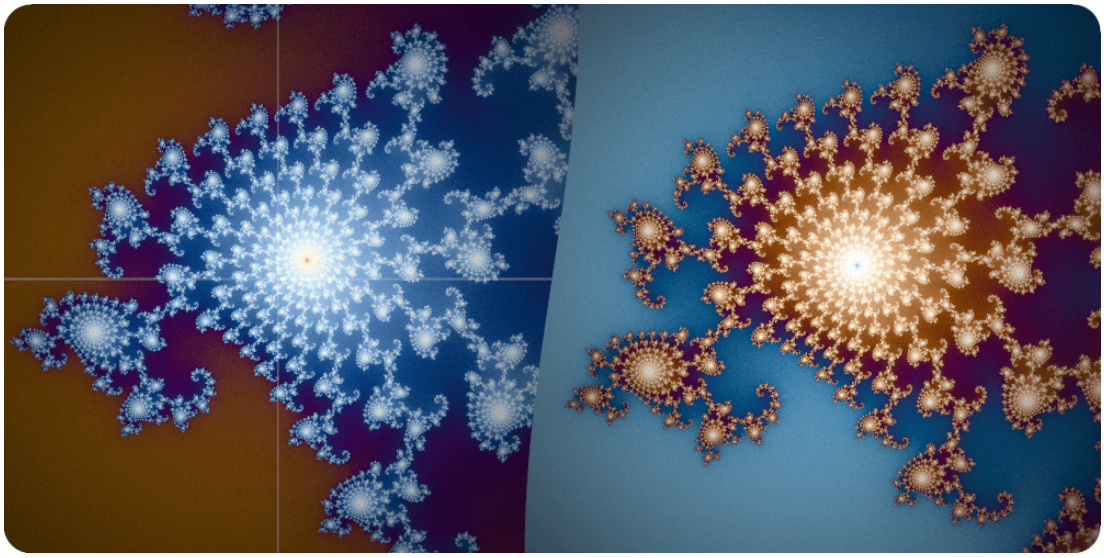


Figure 3.3: Mandelbrot WebGL: similarity of the Mandelbrot set (left) and part of the associated Julia set (right). The crosshair on the left is used to select c for the Julia set.

Mandelbrot WebGL is one of my favourite applications on this list, due to its good performance, radiant colours, and thoughtful user interface. As the name suggests, this application uses WebGL to render the fractals, but also features interesting twists, like a curved boundary between the fractals, and film grain for added effect. Additionally, the

colour scheme is composed of some deep blues, oranges, and purples, and since the left and right views appear to be colour inverses, there is an stark contrast between the two.

Starting with the controls, **only mouse / touchpad are supported**, and the app is clearly meant for desktop use. Moving around the view is simple, and zooming works well with a mousewheel, although it's slightly too twitchy on a touchpad from my testing.

Performance is smooth and allows for real-time interaction, immediately displaying changes in the Julia viewer once the the focus point is changed. Since this application has both the Mandelbrot and Julia sets, the author has chosen to have a **fixed crosshair** on the Mandelbrot view as the method of selection for the Julia view. I found this method to be very effective, since other applications, like the Mandelbrot Maps Android app [12], have often relied on a draggable *pin* that is effectively obscured by the user's finger on a mobile device.

There are two sliders which let the user control the maximum number of iterations I_{max}, as well as the number of *samples*. Sampling is explained in more detail in Section 4.2 – for now, think of it as a way to smooth the image to make it look better. Both options allow for a good level of graphical customisation, but also provide a simple way to control performance by giving the user the choice to sacrifice visual fidelity for a faster response, or vice-versa. One of the drawbacks of using WebGL became apparent: when zooming in far enough, the view becomes pixelated. This is an inherent issue due to how WebGL handles floating-point precision, and is discussed further in Chapter 5.

3.4 DeepFractal – by `munrocket`

`deepfractal.js.org`

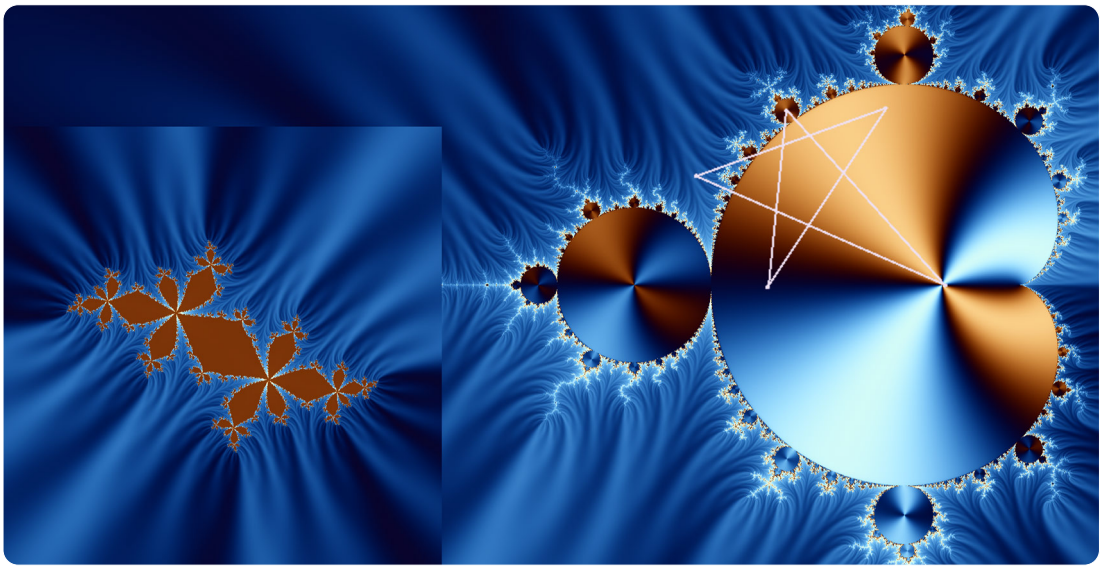


Figure 3.4: DeepFractal: incredible shading. Clicking and holding on a point while pressing `Ctrl` (or with the middle mouse button) provides both the Julia viewer and an iteration trail for that point.

DeepFractal is the most graphically impressive application on this list. It is also one of the more recent applications, so it has some very noteworthy features.

The shading in DeepFractal is phenomenal, and is achieved with WebGL. The trade-off here is that to achieve such good shading, the application is unfortunately not real-time. On faster systems, it's possible to reduce the render time, but even then it is still not quick enough to make it real-time – this could change as GPU technology evolves.

As for controls, DeepFractal's approach is unorthodox. Mouse / touchpad are supported, but it is **not possible to pan around the view** with this set of controls. Left-clicking and right-clicking correspond to zooming in and out. Clicking and dragging, instead of the regular panning action, reveals a box that grows outwards from the origin of the click, to represent where the new transformed viewport will be pointing (including rotation). Using the mousewheel to scroll seems to be inverted (with respect to Google Maps and the other applications), with an upwards scroll zooming out, and a downwards scroll zooming in. The application displays the **iteration trail** and the *Julia set* if a user either clicks with the middle mouse button, or if they click and drag while pressing the 'Control' key.

On touchscreen devices, the application seems to have reduced functionality simply because of the barrier created by having only one input device. Dragging with one finger now makes it possible to pan, although the view does not update until the touch event has ended. Pinching correctly zooms the view in or out as expected, and the application even understands rotation gestures correctly. The iteration trails and Julia set are now impossible to reach, and there is no other way to toggle them otherwise.

Aside from the issues with the controls, there are fairly good positives, with the main one being the **high amount of zoom** available. The author exploits *perturbation theory*: an idea borrowed from quantum physics that has been recently applied to fractal generation, which uses a special algorithm with arbitrary-precision floating-point numbers for faster rendering (see Section 4.2). It appears to also improve the zoom capability of this application compared to Hvidtfeldt's application, bringing WebGL's zoom in line with more conventional CPU renderers.

On a final note, DeepFractal is a ***Progressive Web Application*** (PWA). I provide more detail about PWAs in Section 4.6, but for now, it basically lets DeepFractal function both as a regular webpage, and also makes it possible to install it to any device just like a native application.

3.5 Web Mandelbrot – by `guciek` (Karol Guciek)

`guciek.github.io/web_mandelbrot.html`

Web Mandelbrot is a relatively older project, with the last update dating back to 2014. I have chosen to note it here mostly because it is featured as one of the interactive viewers on the 'Mandelbrot set' Wikipedia page (`en.wikipedia.org/wiki/Mandelbrot_set#Image_gallery_of_a_zoom_sequence`). In fact, as I was looking back to check if it was still there as of 4 April 2020 – it was! – I was pleasantly surprised to find that Mandelbrot Svelte had also been added to the page since I had last visited. One of the reasons why it is listed on that page is likely because it supports **URL parameters**, and so it's possible to share links to interesting locations.

As for the application itself, there isn't much novelty when compared to the previous,

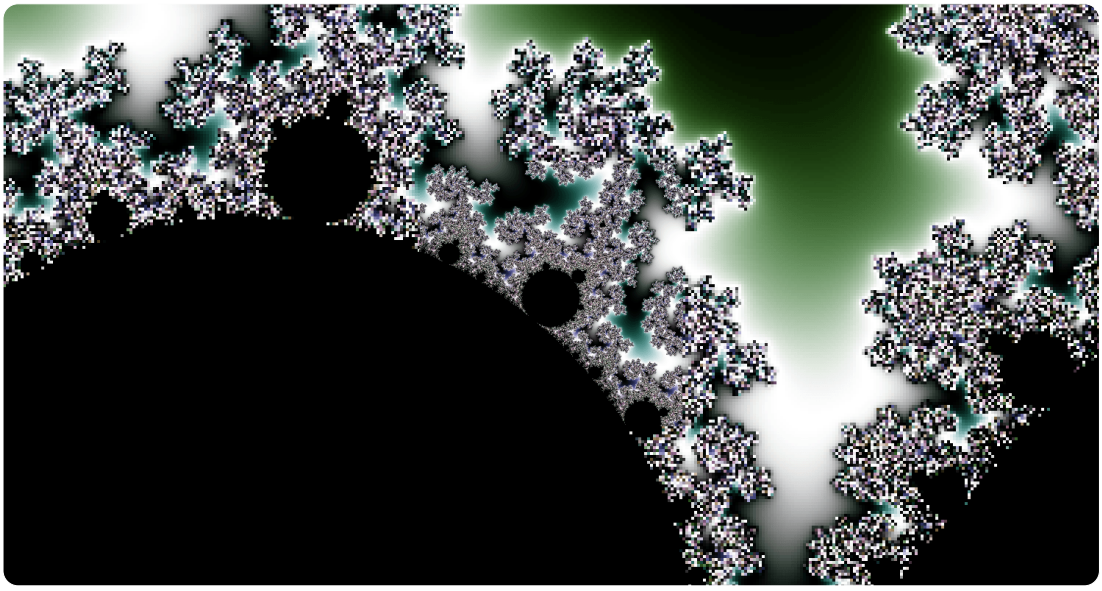


Figure 3.5: Web Mandelbrot: Progressive loading of the Mandelbrot set outwards from centre point: the edges are a preview rendered at 1/4 resolution; a square in the centre is being rendered over the preview, at full resolution.

as expected from its age. One of the notable ideas that it implements is a **progressive rendering system** using the CPU. When first loading the application, and every time a new region is selected, the view is quickly populated by a low-resolution render, outwards from the centre point. This system lets the application be quite fast at initially loading that preview, then it begins to load higher-quality renders. It appears to do this a total of four times, so it seems reasonable to assume that if it is subdividing squares, it renders at the following resolutions: 1/64-res, 1/16-res, 1/4-res, and 1/1 (Full-res). In Figure 3.5, a square portion of the centre is loaded at higher-resolution, giving a pixelated look around the edges. In its defence, the render does look decent once it's finished (after about 60 seconds), with **continuous colouring** that is only partly let down by its colour palette that is perhaps a little too heavy on the dark greens.

For the controls, there is only one (strange) mechanic: "*click any point to zoom in, click near sides to zoom out*". The controls do work, but they are far too simple and leave a lot to be desired.

3.6 webgl-mandelbrot – by `zbendefy`

`github.com/zbendefy/webgl-mandelbrot`

I only found this application after some thorough searching, since it is not directly hosted on the web, but rather through a live preview service from the GitHub repository. This application looks like a standard Mandelbrot set viewer at first, but it has an interesting take on how to display the associated Julia sets: an ***x-ray overlay***. I quite like the idea of providing an overlay like this, since it displays a grid of Julia sets that captures their relation to the Mandelbrot set. It's also possible to convert the view into a grid of hundreds of tiny Julia sets, such that zooming out reveals that this grid approximates the shape of the Mandelbrot set, which feels very satisfying.

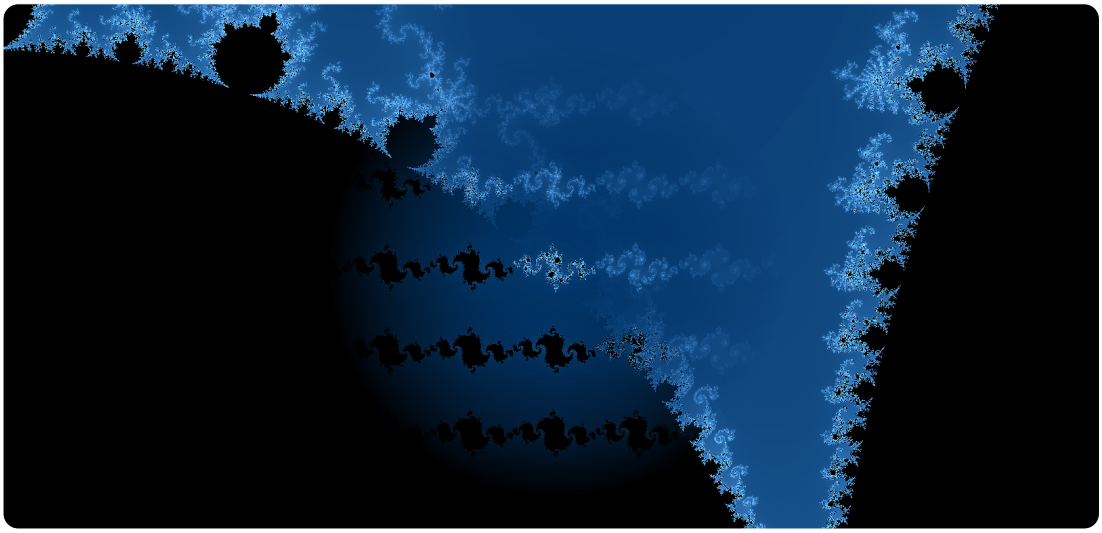


Figure 3.6: webgl-mandelbrot: A grid of small Julia sets overlaid on top of the original Mandelbrot set image.

The app has **basic mouse / touchpad controls**, and supports panning and zooming. Touch input is not very reliable, with only dragging to pan working, even if slightly broken. Pinching to zoom is not supported, and zooms into the page instead of the viewer itself.

3.7 Summary

The applications reviewed were a great source of ideas, many of which I would like to integrate into my own application. These are some of the highlights:

- High performance, ideally real-time
- Simple, intuitive mouse / touchpad and touch controls, with momentum
- Julia set viewer
- Crosshair for Julia set selection
- Configuration (like I_{max})
- URL parameters
- Progressive Web App

Chapter 4

Implementation

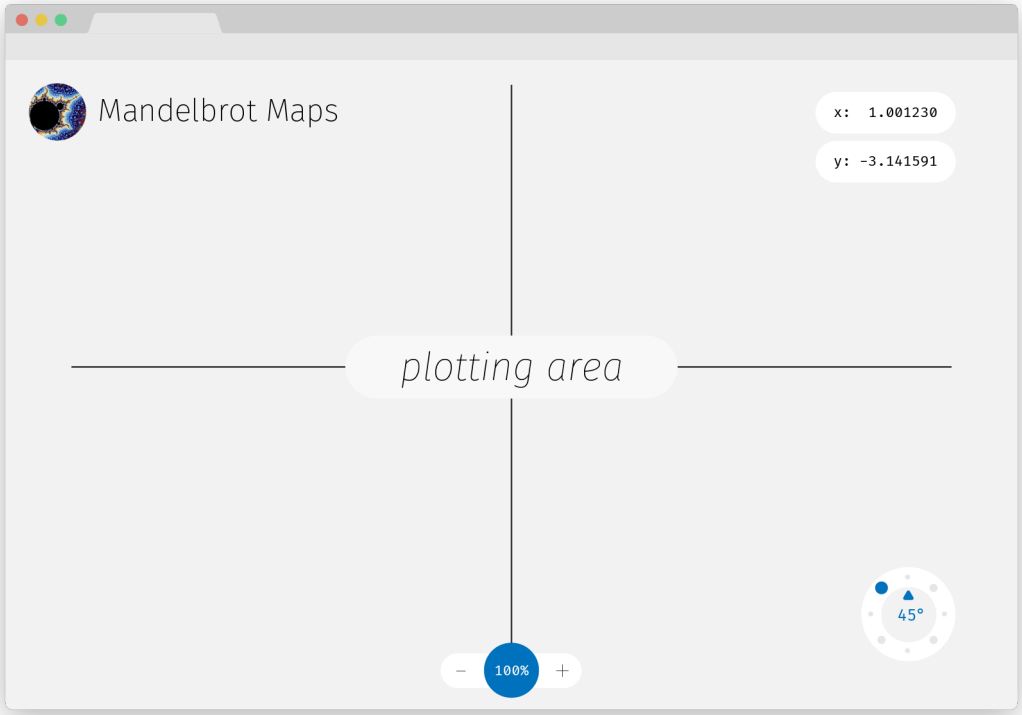


Figure 4.1: First mockup of the application's user interface.

As discussed in Chapter 3, the applications reviewed strive to maximise use of screen space, and this is the guiding principle behind this implementation of Mandelbrot Maps. The application should be informative, and not obtrusive. This chapter focuses on the justifications behind the design decisions for the application, backed by industry standards and with a focus on performance and usability. The flow of development closely resembles the order of the later sections, building up from the rendering, on to the controls, and finally the user interface.

4.1 Technology

4.1.1 React

React (reactjs.org) is advertised as *"a JavaScript library for building user interfaces"*, and is currently one of the most popular JavaScript frameworks with over 7 million weekly downloads from 2020-02-24 to 2020-03-01 on npm, the main repository which distributes JavaScript packages. The next two most popular frameworks are Vue with 1.5 million and Angular with 0.5 million for the same period. React was created by Facebook, and has seen enormous success across the web: a 2018 survey by *npm, Inc.* found that over 60% of 16,000 developers say that they use React [24].

React speeds up development by encouraging the creation and (re-)use of *components* – small, modular pieces that can be *composed* and set up to communicate, creating larger pieces that eventually form the entire user interface. Additionally, React's popularity means that it is generally easy to find answers to questions on development, and since there are hundreds of libraries focused on extending React's functionality, it's usually easy to find one that supports the functionality you need (like using *lodash*'s utility functions, for example).

A big advantage of React is that it is heavily optimised, producing performant JavaScript courtesy of the Babel JavaScript compiler (babeljs.io). React has had a performance focus since the very beginning, and its popularity has actually fuelled some of that performance:

> *React is now so popular that browsers are optimising for the performance of sites built with it. When React announced Hooks, Chrome instantly optimised the V8 [JavaScript] engine to improve the performance of array destructuring.* [25]

Along with the framework, it's possible to run a simple utility called *create-react-app* which provides a template for a blank project that makes it quick to start prototyping. React is therefore used as the basis of the application's user interface.

4.1.2 Material-UI

I have chosen to design according to Google's Material Design (material.io) specification: a well-established set of guidelines with an ethos of being functional and accessible, and has been widely adopted in the industry. The scope of adoption includes many popular websites (like YouTube, Google Calendar, WhatsApp Web) and, more broadly, Google's Android operating system, which lets developers use standardised *Material* components to build their applications.

At the time of writing, an official *Material Design for Web* implementation exists, but it appears to be very limited in terms of number of components and documentation available. Regardless, another alternative is *Material-UI* (material-ui.com): an open-source implementation of Material Design components, specifically meant to be used with React. Material-UI has a multitude of components available with thorough, helpful documentation, and a powerful API which allows for easy customisation.

Although the user interface in Mandelbrot Maps was likely to be very minimal, I believe that there are clear advantages in using standardised components, especially because most users will immediately be familiar with their look and feel. The goal is to reduce development time in creating the user interface while still delivering a great user experience, and using Material Design is perfect for this.

4.1.3 React Use Gesture

Given the importance of having intuitive controls, it was important to make a conscious and informed decision before committing to any single library to handle the gesture system of the application. A possible candidate was HammerJS, which I used briefly to create an early proof-of-concept for assesssing the feasibility of implementing this project in *vanilla* (regular) JavaScript. Needless to say, as the required functionality for this project became more complex, the choice of React over vanilla JavaScript was clearly optimal in my case.

With this in mind, I began to search for gesture libraries that aligned with the React development style, and came across two projects which are closely related: *react-spring* (`react-spring.io`) and *React Use Gesture* (`use-gesture.netlify.com`). react-spring is a "*spring-physics based animation library*", and React Use Gesture is a gesture library which facilitates the handling of mouse and touch events by enriching them with added information like pointer velocities and relative movement deltas. Both libraries were created by the same authors, and are now maintained by the same organisation (react-spring).

React Use Gesture allows for different controls – specifically mouse / trackpad and touch-based inputs – to be harmonised by merging their logic into simple, high-level actions. react-spring's animation capabilities can then be integrated with these gestures to perform realistic physics-based animations in response to user input – something like Google Maps' 'momentum' dragging could easily be achieved with this combination of libraries. Additionally, React Use Gesture benefits from highly-optimised performance, providing fast updates even on mobile devices, which should give users confidence in using the system.

4.1.4 WebGL

Finally, WebGL (`khronos.org/webgl`) is the key piece of technology which makes this experience possible. Mozilla describes WebGL as "*a JavaScript API for rendering high-performance interactive 3D and 2D graphics within any compatible web browser without the use of plug-ins*" [26].

As explained earlier, GPUs are perfectly suited to the task of computing the visualisation of the Mandelbrot set, and WebGL provides an easy way to interface with the GPU (the *hardware*) and have access to that high performance. This is done with programs called *shaders*, which instruct the GPU to output an image to the screen. Section 4.2 goes into more detail about how rendering happens with WebGL.

4.2 Rendering

Rendering is a term used in computer graphics to refer to the process of creating an image from a model (a mathematical description of a scene). In this section, I'll be walking through the process, starting with a blank canvas and finishing with the resulting *shaders*.

4.2.1 Prototyping in JavaScript

Becoming familiar with the basics of WebGL was challenging, but manageable with incremental objectives. To begin, I researched how to use JavaScript's canvas API, but without WebGL for the moment. The initial goal was to start with a simple JavaScript renderer that would be used in place of a more advanced renderer until it was available.

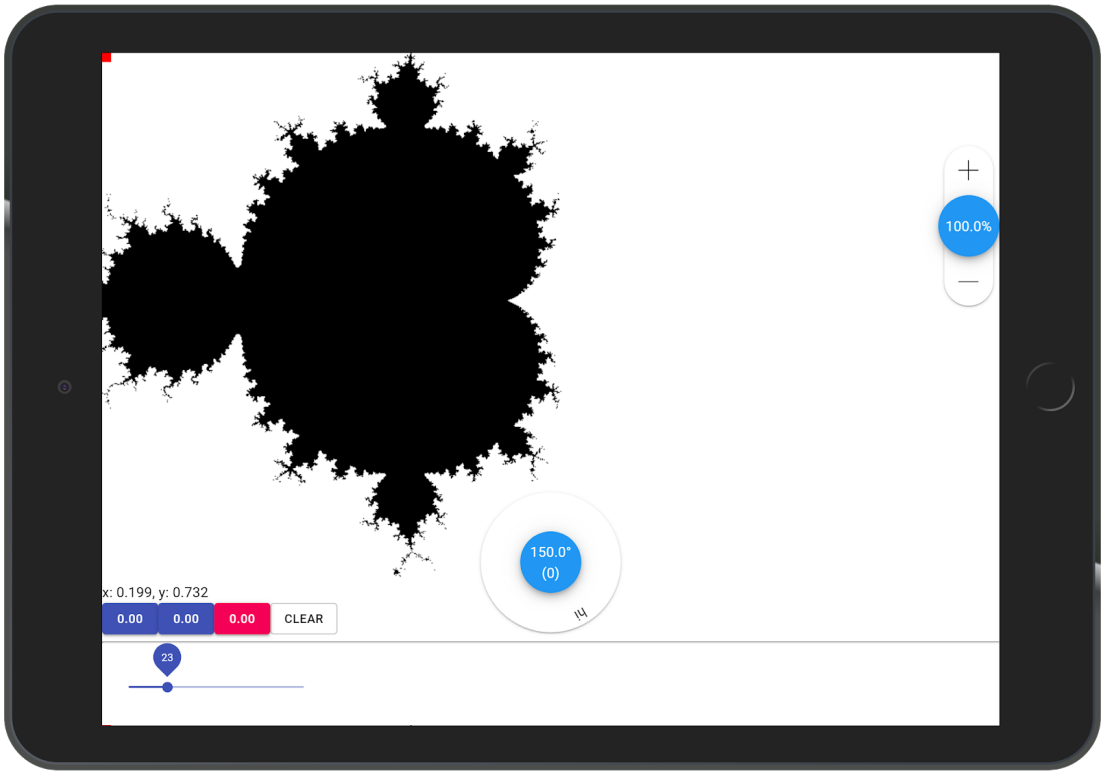


Figure 4.2: A very early version of the application, showing the black / white Mandelbrot renderer in JavaScript.

To create a simple renderer, I experimented with `CanvasRenderingContext2D`: an API which exposes a basic 2D plane that can be controlled with JavaScript. This canvas context has multiple functions available to draw to the canvas, including text, lines, and basic shapes like rectangles and ovals. It's also possible to draw images directly to the canvas.

With this basic rendering context, each pixel of the canvas was mapped to a coordinate on the complex plane. As a brute-force solution, the program looped over all the sets of coordinates and iterated according to the escape-time algorithm (Section 2.2). After the iteration process had completed, a rectangle with a width and height of 1 pixel was

placed at that point on the canvas, and coloured either black if it did not diverge, or white if it did diverge. This was horribly slow, taking about 4 seconds for a relatively small 300×300 pixel frame, and producing results as shown in Figure 4.2.

Despite the underwhelming results, this prototype was useful in helping to devise the control scheme for a fixed canvas since it closely approximated the final product. It also provided a visual indication of whether the controls were affecting the renderer in the intended way – for example, I discovered that the controls had to be inverted to make the image move with the user's input, as they were otherwise pointing the opposite way.

One of the final improvements made was to implement the escape-time algorithm with fractional grey colouring, producing Figure 2.3. This prototype could probably have been optimised further, but since using JavaScript was not the end-goal, it was only useful as a placeholder. With this in place, I focused my efforts in improving the controls for the first half of the year, during which I also researched what I would need to do to re-implement this renderer in WebGL.

4.2.2 WebGL

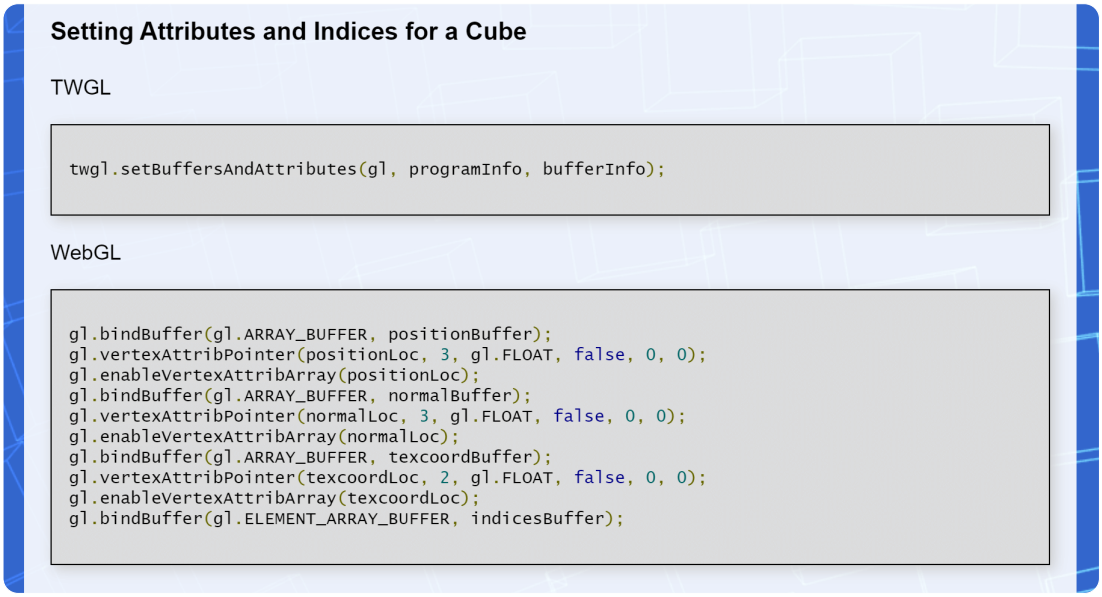


Figure 4.3: TWGL: A Tiny WebGL helper Library. TWGL makes it simple to get started with WebGL, drastically reducing the amount of excess code required (from `twgljs.org`).

The WebGL standard is based on OpenGL ES 2.0 [27]. The way that rendering works in WebGL and OpenGL ES might feel somewhat maths-heavy, but this can be simplified since, for our purposes, we are only interested in 2-dimensional rendering, and so the complexity of 3-dimensional rendering can be avoided. Let us cover the basics of what we need to know to describe our *scene*.

As is common with rendering software, we begin with the idea of a *camera*. Imagine a camera that we have full control over, but the only way that we can move that camera around is by giving it a 4×4 transformation matrix to affect its position and rotation in 3-dimensional space. This camera can be pointed anywhere, and whatever it sees is

then projected onto our 2-dimensional viewport. As you can probably tell, this is quite powerful for 3D applications. To achieve 2D in 3D, all that we need to do is insert a flat plane in front of the view of the camera, and not move either one. The way that I usually visualise this is by thinking of placing a (real) camera in front of a (real) television to represent that flat plane, achieving the same effect.

Now that we have a 2D plane, we need to fill it with information. As previously mentioned, GPUs run small programs called *shaders* (Section 2.4). WebGL shaders are written in GLSL (OpenGL Shading Language), a language with C-like syntax that compiles to efficient parallel code. There are two types that we are interested in: **vertex shaders** and **fragment shaders**. Vertex shaders, as the name indicates, are a way to manipulate vertices – points that join together to create more complex shapes – in our rendering space. In WebGL, shapes can be given an appearance, either with a *texture* – an image, like a brick wall – or a *fragment shader* – a mathematical description of how the shape will be coloured, or *shaded*. We will use a simple vertex shader to project our 2D plane, and a fragment shader to colour it.

Since both the Mandelbrot and Julia set visualisations will feature similar geometry, it's worth noting how WebGL translates from the 2D plane and fragment shaders to the 2D projection of the camera. Our vertex shader would normally be a 4×4 matrix used to translate the vertices in our scene (this is equivalent to moving the camera around). In this case, we set it up to not move the camera in any way, by setting it to be the identity matrix I_4. Vertices in WebGL are declared in sets of three, creating the most basic 2-dimensional geometry: a triangle. Our 2D plane is then actually a pair of triangles, each of which occupies half of the screen.

The projection step for the 2D plane is then performed by taking a *sample* from each pixel of the camera, which maps 1 to 1 with our viewport. This creates a grid system in WebGL which ranges from $[-1, 1]$ on each axis, regardless of size or aspect ratio, where the pixels can be individually addressed. Finally, in what is essentially a loop over all the pixels, each pixel is updated according to the fragment shader, and the next full *frame* is shown once all the pixels have been updated. It was at this point that I began to see a difficulty in making my implementation abide by the planned *common API* (Section 1.4), since the way that WebGL handles drawing to the canvas is built-in, and not modifiable, making it incompatible with regular canvas drawing.

On a final note, I'd like to introduce *anti-aliasing*: the process of smoothing a rendered image by *interpolating* changes between pixels. This is a technique used to improve the quality of an image, either done by *multisampling* (subdividing one pixel into four or more, then averaging them out, also called MSAA) or with other techniques like FXAA (Fast Approximate Anti-Aliasing) [28] or TXAA (Temporal Anti-Aliasing) [29]. The application is able to use multisampling to provide higher quality visualisations on faster machines.

With all the theory down, connecting WebGL to the canvas was a lengthy process, mainly because of how much verbosity there is in WebGL. Luckily, there is a brilliant library called TWGL: A Tiny WebGL helper Library (`twgljs.org`, Figure 4.3). TWGL takes a lot of that verbosity away by providing easy access to essential functions.

4.2.3 Mandelbrot and Julia shaders

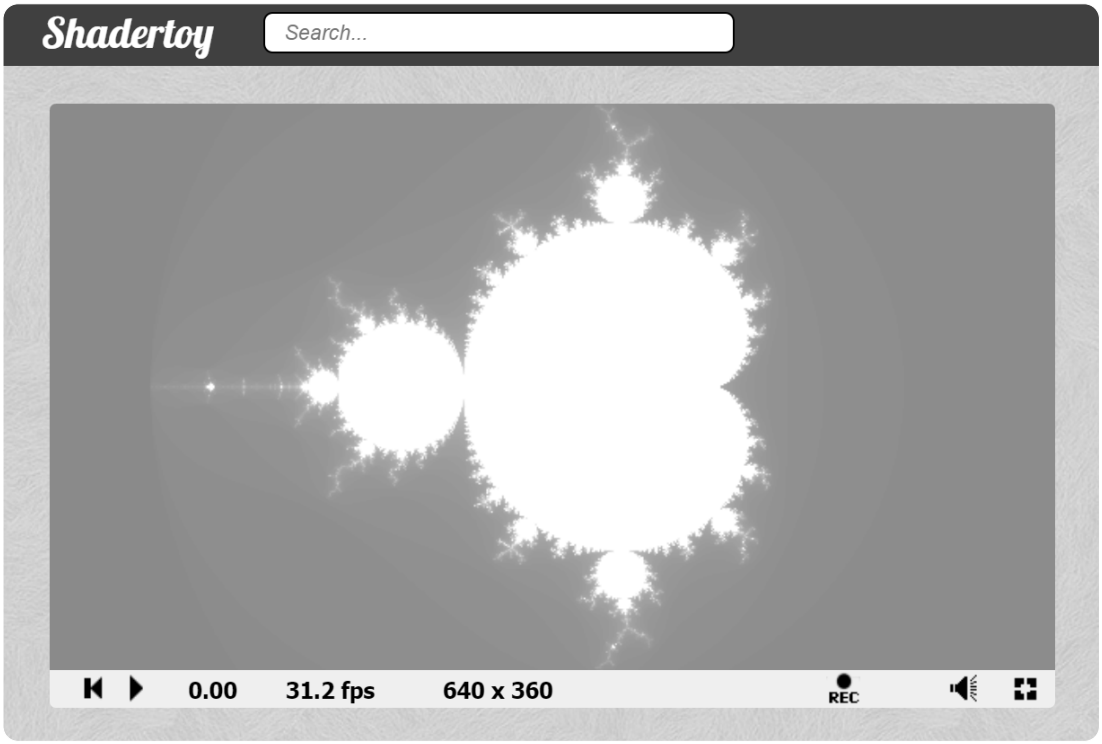


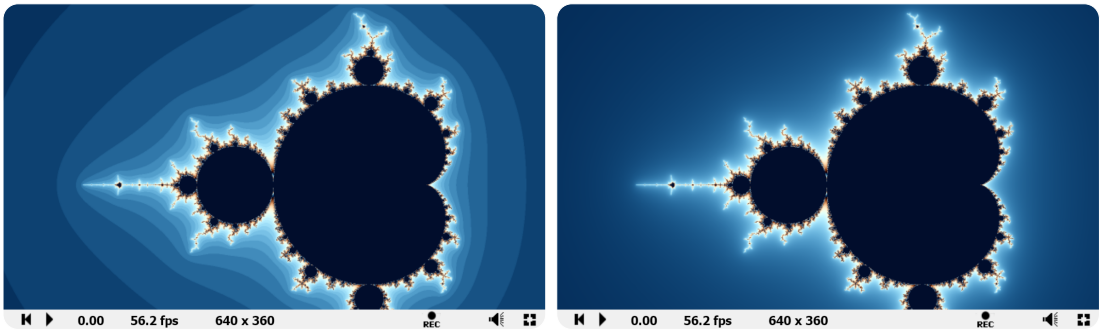
Figure 4.4: The first version of my Mandelbrot set WebGL shader in Shadertoy.

Equipped with the algorithm to generate the Mandelbrot and Julia sets, and extensive knowledge of WebGL, we now tackle the actual rendering part. During the search for any resource that may prove helpful, I encountered Shadertoy (`shadertoy.com`): a brilliant website that enables quick prototyping of WebGL shaders, and is host to some of the most amazing creations on the web. Shadertoy was a great place to learn the basics, allowing me to experiment with different features of WebGL and learn about fragment shaders.

After some tinkering, I was able to make a simple Mandelbrot set renderer just like the JavaScript prototype, which I then extended to also display fractional grey colouring. The result is shown in Figure 4.4.

Looking for inspiration to add colour to the shader, I researched different colouring techniques, and was amazed to find an excellent shader already in Shadertoy (Figure 4.5), made by one of the creators of the site, Inigo Quilez. This shader has tones of light blue and orange, and the addition of smooth iteration colouring is very aesthetically pleasing. The author provides a link to an article explaining the mathematics behind this continuous colouring, which involves applying two consecutive logarithm operations to obtain a fractional iteration count $L = l - \log_2(\log_2(z \cdot z))) + 4.0$, where l is the integer iteration count before z diverges.

Given how much I like this shader, and that it's licensed in such a way that easily allows adapting the source material (CC BY-NC-SA 3.0), I felt that adapting it with attribution was the best course of action. Thus, the shader that I have used in Mandelbrot Maps is



(a) Standard (b) Smooth

Figure 4.5: 'Mandelbrot - smooth', a shader by Inigo Quilez on Shadertoy, showcasing smooth iteration colouring of the Mandelbrot set (shadertoy.com/view/4df3Rn).

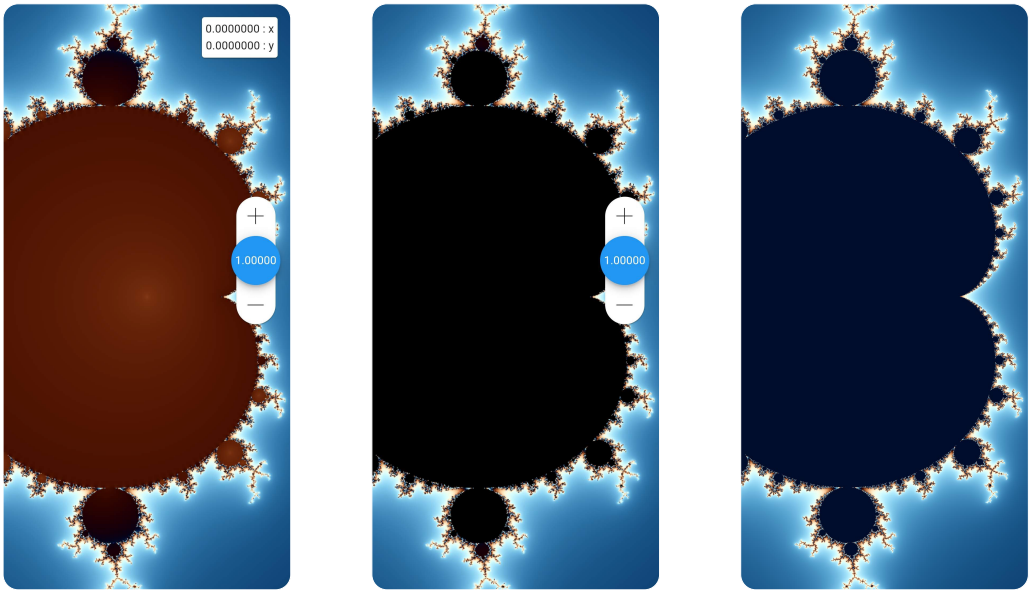
built on top of the 'Mandelbrot - smooth' shader by Inigo Quilez. The modifications are not extensive, but they include adapting the shader to respond to the controls that I had previously implemented in the first half of the year, adding a crosshair, and making some settings like the number of iterations I_{max} and anti-aliasing configurable. For the Julia set shader, the algorithm is quite similar, as I've noted before. This means that the two shaders are almost carbon copies of each other, with the only difference being that the Julia set shader has a parameter c to specify its constant.

One of the issues that unexpectedly appeared was a strange shading only on mobile devices. As illustrated in Figure 4.6a, there is a red / orange tint inside the main cardioid, radiating from the origin. I found that the source of the problem was the chained $\log_2$ operations. The official OpenGL documentation states that for $\log_2(x)$, "*the result is undefined if* $x \leq 0$" [30]. I also found a Stack Overflow post discussing a similar issue, pointing to the fact that in GPUs, hardware implementations of functions like `log2` and `exp2` generally use approximate algorithms. I solved this by skipping the `log2` calculation for $|z| < 1$, preventing a $\log_2(0)$ operation from occurring and breaking the shader. When I next re-checked the Shadertoy page, the shader had been updated to skip points inside the main cardioid and the period-2 bulb as described in Section 2.4.

4.3 Controls

4.3.1 Pan

For the panning control, a user can use either their mouse / touchpad by clicking and dragging, or drag with their finger to cause a translation of the view. Additionally, by using react-spring, I have implemented a form of momentum once a user has stopped dragging, to keep the view moving according to the velocity of their movement, and decaying away like in Google Maps. To achieve this, the end velocity needs to be set based on the current zoom level of the viewer, since a movement on the screen with velocity v would generate the same physical velocity every time, because the scale of the screen remains constant. The further a user has zoomed in, the smaller the scale becomes, making this velocity $v_{movement} \propto \frac{1}{zoom}$.



(a) $\log_2$ approximated on ARM (b) Skipping $\log_2$ for $|z| < 1$ (c) New shader

Figure 4.6: Strange shading in main bulb (Figure 4.6a), fixed by skipping $\log_2$ calculation on $|z| < 1$ (Figure 4.6b). An updated shader provides extra optimisations by skipping even more calculation of points that are known to not diverge (Figure 4.6c).

One of the issues faced is that once a very small scale is reached, this number becomes very close to zero. Due to how react-spring works, once a minimum velocity threshold is reached, the animation is stopped instantly, breaking this feature at very high zoom levels. There is currently no easy way to address this issue with this version of the library, but it is very much a living project so it's possible that a fix will be available.

4.3.2 Zoom

For zooming, both mouse / touchpad are supported via mousewheel events, which includes two-finger gestures on a precision touchpad. On a touchscreen or touchpad, pinching to zoom is also supported. Zoom is simply a multiplier that works like a normal camera's zoom function. Zoom levels for both viewers are limited to the range $[0.5, 100000]$. Zooming in any further would result in a pixelated view, which is a limitation of WebGL as described in Section 4.2.2.

I had also intended for zooming to have associated momentum, just like Google Maps, and even though this has been implemented, zooming gestures currently have slight stuttering at the end, once a user has stopped scrolling. This stuttering is due to the difference in granularity of mousewheel events, which are very coarse (they are emitted in multiples of 100 "*scroll units*") compared to touch events (which are emitted in sub-pixel increments). Some revision will be necessary, and it should be possible to fix in a future version of the application.

4.3.3 Rotate

An initial prototype of the application (Figure 4.2) was created with support for rotating the view, using a two-finger rotation gesture. Although the current code has been fitted with basic support for rotation, this feature could have been potentially confusing given the lack of visual cues to represent the rotation of the axes. For this reason, I have decided against implementing rotation, and instead chosen to focus on the more basic functionality, with potential to revisit it in future.

4.4 User Interface (UI)

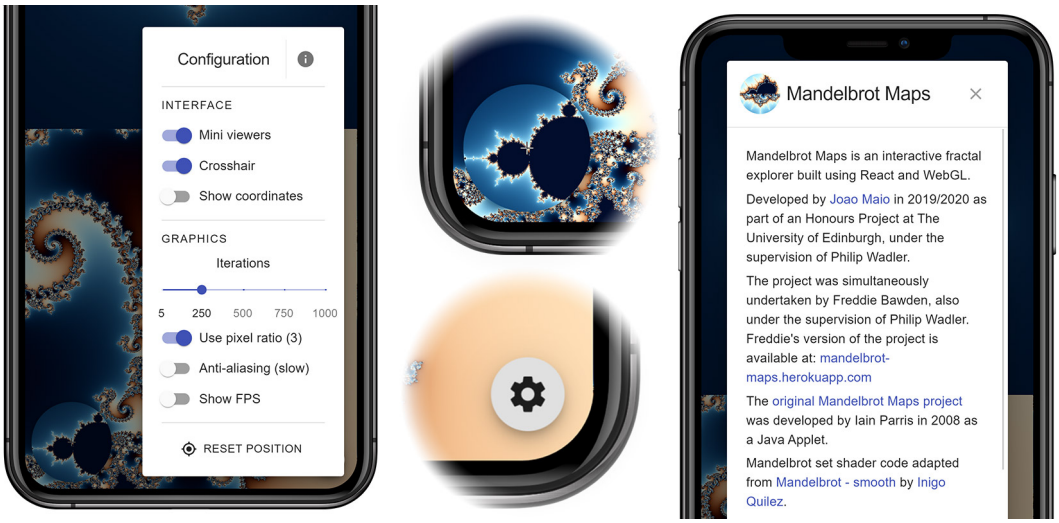


Figure 4.7: Focused view of the main UI elements in Mandelbrot Maps: configuration menu (left), mini viewer and settings button (centre top and bottom), and project information screen (right).

As might be evident from Figure 4.1 and Figure 4.2, I had originally intended for there to be some controls on the page, mostly to provide more granular ways to control the viewers (other than with mouse or touch inputs), and also to account for users without access to certain mechanics like a scroll wheel.

In the initial prototypes of the application, a small linear white and blue control is visible, which was meant to function as a *zoom lever*, with a user holding and dragging up or down to zoom in or out of the view, mimicking a digital camera's zoom lever. There is also a circular control that was meant to represent rotation of the view. In the end, it felt as though the controls may be useful on a desktop, but that they were taking up too much space on a smaller screen, so they were pulled from the final version. In a future version, I would like to re-add them, giving users the choice of whether to have them visible.

The final version of the UI was heavily simplified, with only a single icon on the bottom-right of the screen which expands to give access to the settings menu (Figure 4.7). The configurations options are shown in Table 4.1. Currently, it's possible to click the *info* button next to the *Configuration* text, bringing up a dialog with information about the

project. In line with standard React practice, all parts of the UI are modular components, easily configured via `props` (*properties*) attributes. A diagram of the system is shown in Figure 4.8.

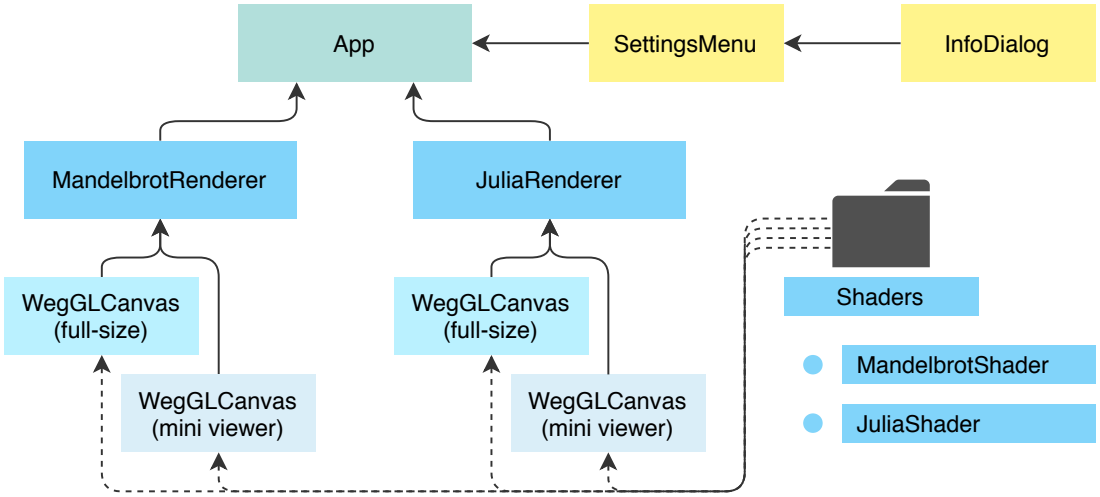


Figure 4.8: A view of how the different React components are composed in Mandelbrot Maps.

Later on, I was finding myself lost at certain points when zoomed in quite far. I turned to Google Maps to see if Google had thought of something that may help me, and indeed they had an element which had an interesting behaviour: the mode switcher on the bottom-left is a button which toggles between *Map* and *Satellite* mode, and also acts like a large tile, displaying a preview of the current location in that other mode.

I had the idea of implementing a similar element inspired partly by the mode switcher, and partly by a common UI element in videogames: a mini-map. The result is a circular overlay in Mandelbrot Maps, which I have dubbed a *mini viewer*, that displays the current position in the fractal at a zoom level of 1, placing the main view into context. Since this could also function as a button, it now acts as a zoom reset button. If clicked, the view animates, zooming out to the normal level. To reduce clutter, a mini viewer will only become visible if the view is zoomed above a certain threshold. If the zoom level is low enough to see the full structure of the fractal, the mini viewer becomes progressively more transparent, until it is fully invisible.

To fully utilise screen space, the application adjusts to the aspect ratio of the target device ($^{width}/_{height}$). If the device has a width equal to or larger than its height (aspect ≥ 1), the views are laid out in two columns: *landscape mode*; if the width is smaller than the height (aspect < 1), the layout is reduced to a single column with two rows: *portrait mode*. The devices most likely to be in portrait mode are mobile phones, so the Mandelbrot set is shown on the bottom to enable easier control of the Julia set on top. Bringing the controls and other UI elements closer to the bottom of a mobile display improves reachability of the controls for one-handed use [31].

Option	Description
Mini viewers	Small circular overlays on bottom-left corner of each viewer, showing a preview of the current position at zoom level of 1
Crosshair	A plus-shaped (+) indicator in the centre of the Mandelbrot set viewer, for selecting the parameter c for the Julia set
Show coordinates	Toggles the visibility of the coordinates of the Mandelbrot set on the top-right corner of the screen
Iterations	A slider which controls the maximum iteration parameter I_{max}, updating both views immediately
Use pixel ratio	Toggles the use of the device pixel ratio: a multiplier to account for the device's pixel density – a high pixel ratio will increase the quality of the render
Anti-aliasing (slow)	Toggles the use of anti-aliasing, which improves the quality of the render; this is explicitly marked as "slow" because it heavily increases the load on the GPU, causing slow performance
Show FPS	Toggles the FPS (frames per second, or how many times the viewer is updating per second) meter on the top-left of the Mandelbrot viewer – a higher number indicates better performance
Reset Position	Returns both views to the origin and resets zoom level

Table 4.1: Available configuration options in Mandelbrot Maps. These are accessible from the settings menu on the bottom-right corner of the application.

4.5 Compatibility

Optimising for different kinds of systems is not always straightforward, given the high degree of variability in hardware and software out in the world. For this reason, the frameworks used were already chosen with maximum compatibility in mind.

WebGL is an industry standard that should run on any recent machine with a browser, since GPUs, even if very basic, are readily available on just about any hardware that is able to output a display signal. To create the UI, I have used React because it already has integrated methods to ensure maximum compatibility. For handling user inputs, both with mouse / touchpad and touchscreen, I have used React Use Gesture since it is a tested solution that reduces my development time, as opposed to creating a solution from the ground up, which would also be prone to error.

With respect to these libraries, the system as a whole should be compatible with the vast majority of systems – this is not however, a guarantee. Since development time is limited, it was not feasible to test every single browser on every type of device. As of March 2020, Chrome held some of the highest browser market share across the three main market segments, shown in Table 4.2, so I have given priority to testing on Chrome, both desktop and mobile versions. Testing validated that all of the functionality of the application works as intended in Chrome (version 80).

Market segment	Desktop / laptop	Tablet	Mobile
Chrome	68.50%	43.04%	63.67%
Main competitor	7.59% (Edge)	45.25% (Safari)	26.46% (Safari)
Device market %	40.43%	4.81%	54.76%

Table 4.2: Browser market share statistics, March 2020. Chrome dominates almost every segment, only beaten by Safari in the Tablet market, which represents 4.81% of the devices in these categories. [Source: NetMarketShare (netmarketshare.com), 6 April 2020]

4.6 Progressive Web App (PWA)

From Google's web.dev site about best practices on the web:

> *Progressive Web Apps (PWAs) are built and enhanced with modern APIs to deliver native-like capabilities, reliability, and installability while reaching anyone, anywhere, on any device with a single codebase.*
> web.dev/what-are-pwas

Making Mandelbrot Maps a PWA allows it to be installed to any compatible device just as if it were a native application made specifically for that platform. Some browsers even give PWAs preferential treatment – on Android, Chrome displays large prompts to allow users to install a PWA directly to their system, with no additional work needed from the developer (Figure 4.9). Google has also started "*replacing some Android apps for Chromebooks with Progressive Web Apps. In some cases, PWAs are faster and more functional than their Android counterparts*" [32].

Other than making minor changes to a *manifest* file and serving the application over HTTPS, the process is relatively simple. To address hosting, I have used GitHub Pages: GitHub's free hosting service that uses HTTPS by default and is built into their repository system, since it is fast and does not require a server or backend to support it. Benefits of HTTPS include added security, giving users a high level of confidence that the application is genuine and not malicious.

The template from create-react-app already includes all the necessary files for making a PWA, so I have altered these to make it work with Mandelbrot Maps. I have tested installing the application on Windows 10 and Android 10 with Chrome, and on iOS 13 with Safari. The process is simple on all platforms, and the result is seamless: the icon and the name appear normally, and the app works even if the device loses network connectivity and goes offline. It is shown on the target system just as if it were a native application, as shown in Figure 4.9.

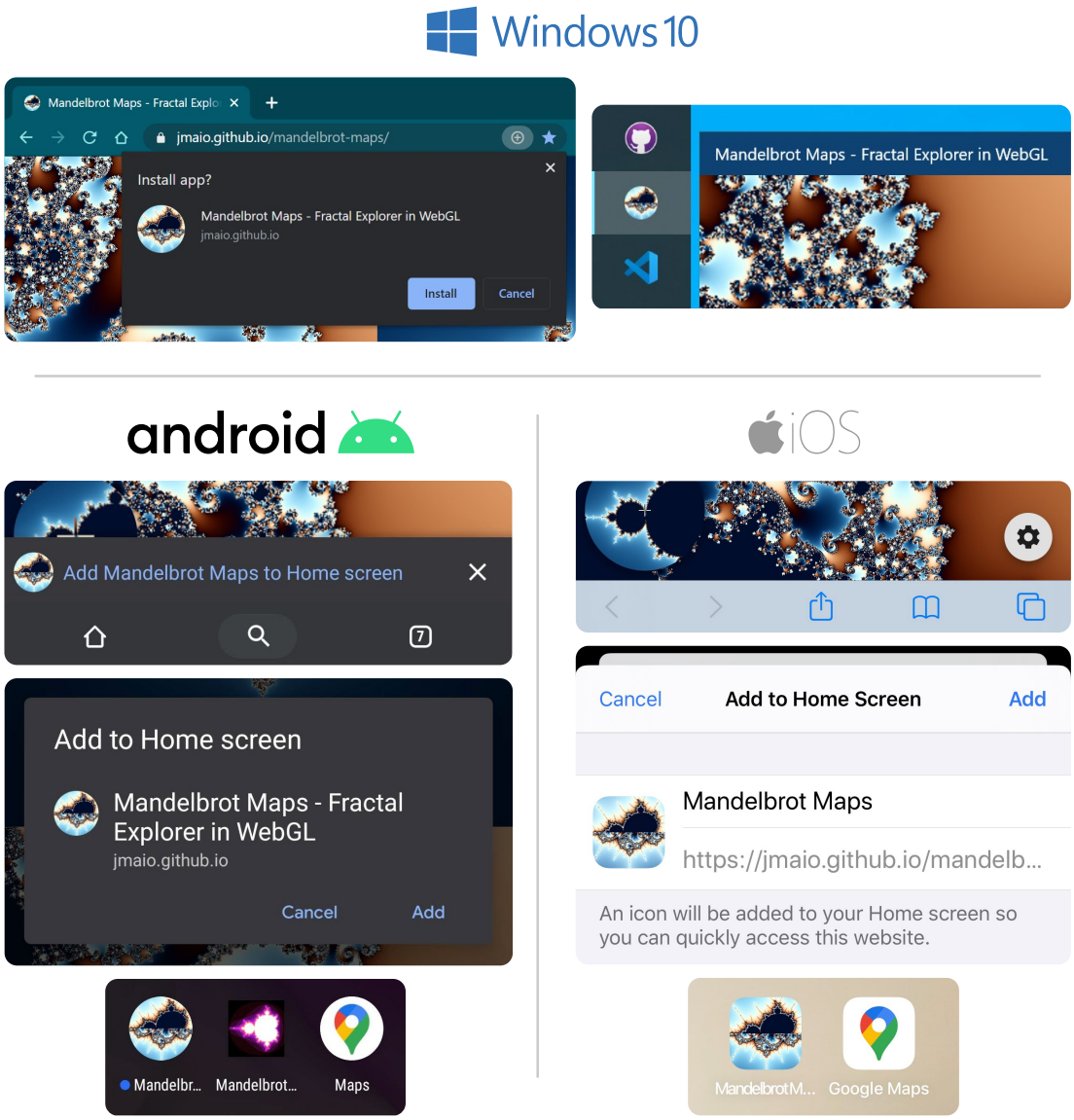


Figure 4.9: Installing Mandelbrot Maps as a PWA on Windows 10, Android 10, and iOS 13. The browser (Chrome on Windows and Android, Safari on iOS) handles installing the application, which looks like a native application and works offline.

Chapter 5

Evaluation

5.1 Performance and Limitations

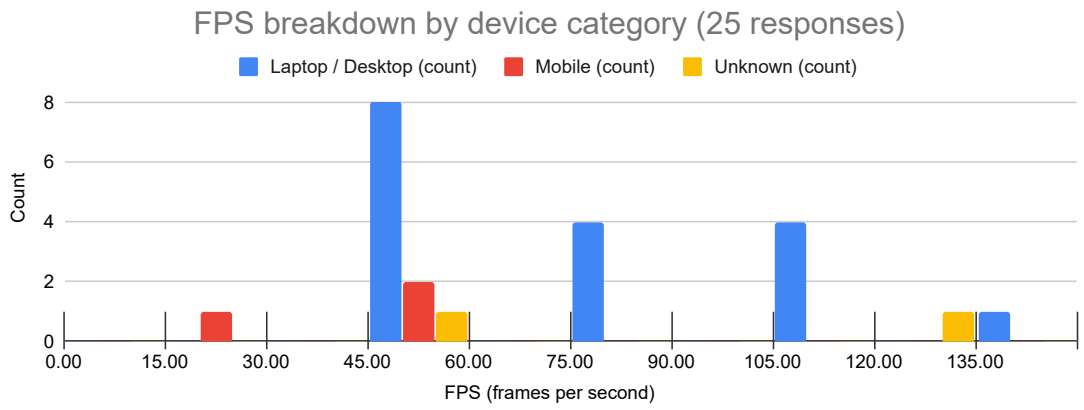


Figure 5.1: Performance histogram of responses, grouped in intervals of 15fps. Most users had performance in the range of 45-60fps. Only 1 out of 25 respondents ($\approx 4\%$) had performance below 30fps, and none had performance below 15fps, clearly highlighting WebGL's performance. Categories are split into Mobile and Laptop / Desktop based on device properties. *Unknown* indicates users with an unspecified device type.

There were high expectations for performance, considering that the WebGL applications reviewed were capable of delivering smooth experiences. From testing, most modern devices that support the application are able to run it easily. By default, the application starts out with a pixel ratio multiplier of 1, a maximum iteration count of 250, and anti-aliasing off to provide a basic, usable experience. It is then up to the user to assess whether their device is able to support the additional graphical settings or not.

Asking users to rank performance is subjective, since users will have different experiences on different devices. Users with high-performance desktops will have a different perception of the application when compared to those using it on a budget smartphone, so I implemented a way to collect a user's device properties during the user survey (shown in Table 5.1), to more accurately build profiles of different devices.

Key	Description	Example
browser	The name of the browser in use	Chrome
browserVersion	The major version number of the browser	80
browserRelease	The full version number of the browser	80.0.3987.149
device	The name of the device (not always provided)	ONEPLUS A6003
os	The operating system of the device	Android
osVersion	The version of the operating system	10
mobile	Whether the device is considered a mobile device	true
platform	The processor technology / architecture (not always accurate)	Linux armv8l
screen	The screen size of the device, as shown to the browser	412 x 869
dpr	The device pixel ratio: a multiplier to account for the device's pixel density	2.625
gpu	The reported graphics processor	Adreno (TM) 630
gpuVendor	The reported vendor (manufacturer) of the GPU	Qualcomm
userAgent	A string with device properties which identifies it to the website	Mozilla/5.0 (Linux; Android 10; ONEPLUS A6003) ...

Table 5.1: Device properties collected on the application. These are available to the survey respondents to copy into the form, providing additional diagnostic information behind the responses to better group them into categories.

One of the more obvious limitations is the low amount of zoom available before the views become pixelated. During testing, I experimented with WebGL's floating-point precision – the three types are `lowp`, `mediump`, and `highp`. WebGL follows the Khronos Group's OpenGL ES specification [33], which dictates that for WebGL (with shaders written in GLSL ES 1.0) the minimum precision for these floating-point types is `highp` (16 bits), `mediump` (10 bits), and `lowp` (8 bits). In practice, these tend to be higher, with many devices exceeding the specification and using 23 bits for `highp` floats.

A new version of WebGL – WebGL 2 – supports GLSL ES 3.0. This newer specification of GLSL ES dictates that `highp` floats have a minimum precision of 23 bits, so it is likely that the devices using 23 bits for `highp` floats are simply conforming to the GLSL ES 3.0 specification. None of these precision types even come close to 32-bit or 64-bit in modern computers, causing this issue. A possible workaround is to use perturbation theory: an idea from quantum physics that has been recently applied to fractal generation for faster, more accurate rendering that can surpass WebGL's floating-point limitation by using arbitrary-precision floats [34].

5.2 Functionality

This version of Mandelbrot Maps implements some of the best parts of the applications reviewed in Chapter 3, bringing them all together in a seamless way. Section 3.7 contained a short list of desired functionality for the final version of the application – of those, the ones that have made it were:

- High performance, ideally real-time
- Simple, intuitive mouse / touchpad and touchscreen controls, with momentum
- Julia set viewer
- Crosshair for Julia set selection
- Configuration (like I_{max}, anti-aliasing)
- Progressive Web App

One of the pieces of functionality that was not implemented is URL parameters, which was left for a future version as it was not considered high-priority, and facilitated development of other interesting features like mini viewers.

5.3 User Survey

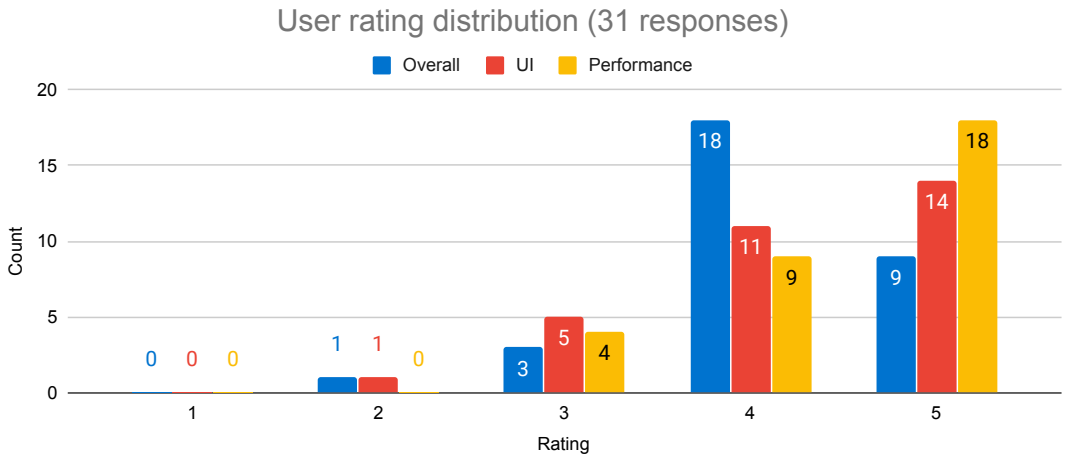


Figure 5.2: User rating statistics indicate positive reception of the application.

I would like to thank those who took part in the survey and provided very useful insight. This short, unsupervised survey consisted of asking participants to explore the app on their own, after which they were asked to answer the questions in Table 5.2 through the Google Forms platform, for easy data collection and analysis.

The rating system used was intended to make it possible to compare against other applications, like those on the Google Play store (which uses a rating of 1 to 5 *stars*: one star being a negative experience and five stars being a positive experience). The results from the survey were positive, with an overall average rating of 4.13 out of

Question	Answer
* How would you rate your overall experience with the application?	1 – 5
Did you encounter any issues or difficulties while using the application?	[Short text]
* Would you be interested in using the application again?	Yes / No / Maybe
What did you like?	[Multiple choice]
What do you think could be improved?	[Multiple choice]
How would you improve these features, or what new features would you add?	[Short text]
Which input methods did you use to explore the fractals? (Which did you prefer?)	Mouse (or Touchpad) / Touchscreen
How easy was it to use the application in terms of user interface? (UI rating)	1 – 5
How would you rate the performance of the application on your device? (Performance rating)	1 – 5

Table 5.2: Questions asked in the user survey. Those marked with an asterisk (*) were required. A 1 – 5 scale corresponds to [1 (Very Negative)] – [5 (Very Positive)].

5.00 and a standard deviation of 0.72 obtained from 31 respondents in total. The most common ratings were 4 (18 users) and 5 (9 users), and they accounted for 87% of the responses. Three users gave a rating of 3, and only one user gave a rating of 2, with no ratings of 1. All the users also gave numerical ratings for the two other metrics: UI and Performance – these were even more favourable with average ratings of 4.23 (std dev 0.84) and 4.45 (std dev 0.72) respectively, indicating that other factors may be lowering the overall rating. A total of 25 respondents completed a benchmarking step asking them to record their performance in terms of frames per second (how quickly the application is updating), shown in Figure 5.1.

Measure	Overall	UI	Performance
Rating (stars / 5)	4.13	4.23	4.45
Std dev	0.72	0.84	0.72

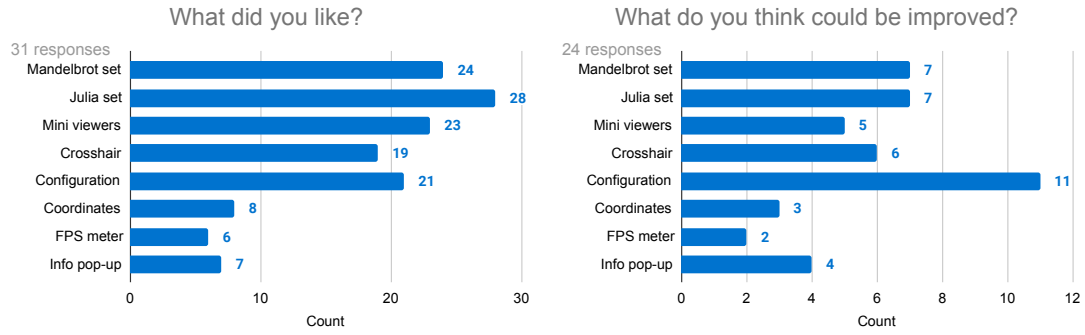
Table 5.3: Ratings for the different metrics from the user survey, based on 31 respondents.

Some users provided positive feedback, reporting that they had either “no issues at all” or “minor issues”. One respondent noted that “*everything ran incredibly smoothly - it looks really nice!*”.

When asked whether they would be interested in using the application again, 15 respondents said *Yes*, 12 said *Maybe*, and 4 said *No*. Investigating these numbers reveals that those that answered *Maybe* or *No* had shared concerns regarding a lack of clarity of

what the application offers, and what purpose it serves, apart from "*the fact it's about fractals*". Some of the main issues were:

- no tutorial or advice on what to do or what can be done
- not really understanding how the three [sic] pictures were connected, clicking on the miniviewer making it disappear
- it wasn't quite clear to me at the start that the left window is the selector for the right window
- I was hoping to zoom more than the app let me
- cannot scroll without a mouse
- zooming with the scroll wheel was far slower than I would expect it to be, and there is no way to adjust that
- scrolling was a bit slower than I expected on the Mandelbrot and faster on the Julia
- couldn't find a way to zoom back in using a laptop - figured that it should be possible as you could zoom out using the minimap
- the dragging of the image does not stop when letting go on the mouse clicker
- lots of aliasing
- I was unsure what the abbreviations meant: for example FPS; I was also unsure at what some of the words meant and had to look it up: like for example anti-aliasing



(a) Most functionality was well-received

(b) Improvements suggested

Figure 5.3: Results of multiple-choice questions in the user survey. This was annotated with images of the relevant components to ensure that respondents knew what each option represented. Respondents seemed to highly favour the Julia set, likely because it changes depending on the Mandelbrot viewer. Configuration options were the main part that respondents would like to see improved.

In particular, one of the respondents noted:

> *I selected 'would you be interested in using the application again' - maybe, because I do not have that much practical use for fractals. But potentially depending on the purpose of the application, the browser could display a short introduction window at the beginning introducing the user to fractals and sharing some interesting details?*

It becomes clear from the comments that there needs to be additional information supplied prior to using the application. Even if those that are familiar with fractals are able to work with the application easily, it is not fair to assume that everyone will be able to use it immediately. Some of the improvements that were suggested included:

- more information about the specifics of what is shown and what can be explored
- what's the link between the mandelbrot set and the julia set? why does moving the mandelbrot set around change the julia set?
- more configuration options related to the set (maybe a go to co-coordinate option)
- make the configuration more interactive, or allow the user to input their own equations
- help button or a ? next to each option in the config menu to understand what it means
- scrolling speed options
- keyboard navigation
- allow infinite zoom depth
- make the crosshair a bit smaller and more round in the middle, so it shows more concretely what point it is focusing on without obscuring it
- acceleration on crosshair movement by dragging sometimes makes the cursor move further than I would want it to
- make the minimap remain when zoomed out, as an option to zoom back into a specific area by clicking it on the minimap
- focused view with only Julia or Mandelbrot Set
- select a particular area instead of moving the cursor to find it
- change of colour would be cool
- different colour "themes"? music?

Data on mouse and touchscreen usage was also recorded: 25 users used a mouse or touchpad, 4 used a touchscreen, and 2 used both input methods. Surprisingly, it seems that the application was much more popular on desktops and laptops than mobile phones. However, both respondents who had tried both methods preferred the touchscreen interface, with one noting that they "*seemed to get a more precise position*" and that it was "*easier to get the right area [they] wanted to view*". This could point to a need to tweak mouse / touchpad controls, and perhaps re-introducing the fallback controls mentioned in Section 4.4 for more precise controls.

5.4 Unit Testing

Unit testing is a way to certify that an application is performing as intended. This practice normally works best on applications with highly complex user interfaces by testing the individual components to ensure that they perform their basic tasks correctly. The application includes a basic unit test to ensure that it renders without crashing. Although React has decent support for testing components with libraries like Jest (`jestjs.io/en`), it became increasingly clear that support for `canvas` elements (and therefore WebGL) was limited.

Automated testing software generally uses a *virtual* browser to test the assertions set out in a test suite – this means that only the DOM tree is modelled and no display is available [35]. With regular components, this is not an issue since it is possible to interact with them by using the DOM, but with WebGL it is necessary to have a display to output to. Automated testing of graphical output would not only be slow, it would be impractical because there is no clear answer of what a test should expect.

Eventbrite (`eventbrite.com`) have written about their use of WebGL to "render customisable venue maps" for seat booking on their website in a blog post titled *Automated Cross-Browser Testing for WebGL – It's Not Going to Happen* [36]. When faced with the need to test this flow, they note that it became difficult because of WebGL: to click on a seat in the WebGL view, they would need to specify *the exact coordinate* of that seat on screen, something which could easily change between versions. As a workaround, they turned to Rainforest QA (`rainforestqa.com`), a service which provides testing courtesy of *human QA testers*. As per the post title, automated testing of WebGL is not likely to happen anytime in the near future.

Since unit testing only provides a basic check that the app renders without crashing, I ended up testing the application extensively myself, trying out different use cases and sequences of inputs. No issues were detected, and this was confirmed by the responses in the user survey, with no reports of the application breaking.

5.5 Quality and Compliance Auditing with Lighthouse

To ensure maximum compliance with well-defined web standards, the application has been audited using Lighthouse, a tool developed by Google to assess web page quality:

> *Lighthouse is an open-source, automated tool for improving the quality of web pages. You can run it against any web page, public or requiring authentication. It has audits for performance, accessibility, progressive web apps, SEO* [Search Engine Optimisation] *and more.* [37]

In the Lighthouse audit, Mandelbrot Maps scored highly in all categories, and passed all check for a Progressive Web App. "Best Practices" fails one test advising the use of passive event listeners for scrolling, which can be ignored because there is no scrollable content in the main view. Lighthouse made it simpler to improve the application's accessibility, and to ensure that it meets PWA standards.

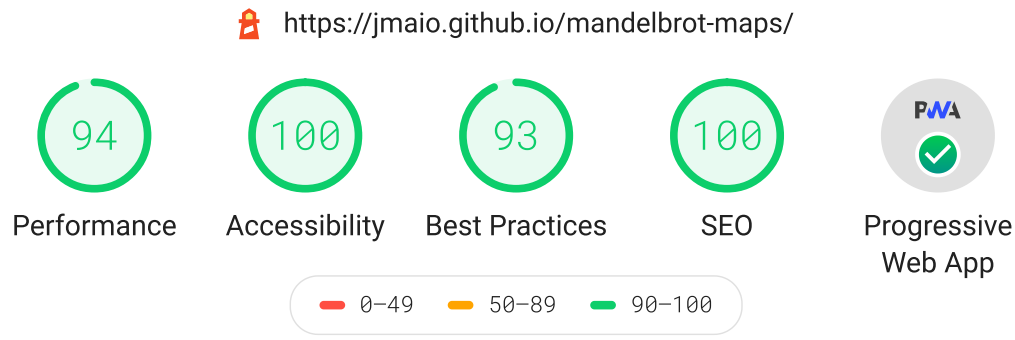


Figure 5.4: Lighthouse audit result for Mandelbrot Maps. Performance can vary wildly between audits; this is obtained on a powerful development machine.

Chapter 6

Conclusion

6.1 Closing Remarks

Hopefully this report has persuaded you to try out the Mandelbrot Maps application. If you have, thank you, and I hope that you have found it interesting, informative, fun, or all of the above!

With this version of the application, we see that WebGL provides an amazing level of performance even on mobile devices, that indeed allows for an interactive, real-time fractal explorer as originally intended. The controls allow for easy exploration of the fractals, and provide a solid base to improve upon.

Additionally, both *core* features like selecting the number of iterations I_{max} and displaying coordinates, and *novel* features like advanced graphics options and mini viewers should make the experience more enjoyable for the end-user. The feature list includes:

- High performance, real-time visualisation of Mandelbrot and Julia sets
- Simple, intuitive mouse / touchpad and touchscreen controls with momentum
- Julia set viewer
- Crosshair for Julia set selection
- Mini viewers
- Coordinate display
- FPS Meter
- Configuration
  - Iteration control (I_{max})
  - Device pixel ratio
  - Anti-aliasing
- Progressive Web App

Evaluating the application showed that respondents rated it favourably, with an average rating of $4.1/5$ based on 31 reviews – the current Android app [12] has an average rating of $3.8/5$ based on 300 reviews as of April 2020.

6.2 Further Work

6.2.1 Graphics

As exemplified by DeepFractal (Section 3.4), it is possible to achieve a high level of zoom even when using WebGL, through the use of perturbation theory. One of the potential improvements would be to bring Mandelbrot Maps in line with this level of zoom. Upgrading from WebGL + GLSL ES 2.0 to WebGL 2 + GLSL ES 3.0 would also be beneficial, and could yield slight improvements.

6.2.2 Different / Interchangeable Shaders

Some thought has gone into this, since it would not be tough to make more shader attributes (like the colour palette) customisable via the user interface. Full shader swaps may be possible by providing different pre-built shaders, or even allowing a user to upload their own shader code. As for practicality, this would likely be a very high-cost addition for not a lot of gain to the average user.

6.2.3 Controls

The controls, while not perfect, are a good base to build upon in the next version. Re-thinking and re-adding some of the control elements like the zoom lever might be useful to users who cannot otherwise use the standard mouse / touch controls. Updating to newer versions of *react-spring* and *React Use Gesture* would also be worthwhile, since they may provide better functionality.

6.2.4 Different Layouts

A part of the app that I would have liked to develop further is to allow for different layouts of the two sets: a picture-in-picture mode similar to DeepFractal (Section 3.4), isolated viewers for screenshots, and perhaps a Julia overlay mode similar to webgl-mandelbrot (Section 3.6).

6.2.5 URL Parameters

URL parameters seem to be a major requirement for the application to be taken more seriously as a fractal viewer. This feature should definitely be centre stage in a future version, although it needs to be designed to be user-friendly and intuitive.

6.2.6 Information

As the survey indicated, many respondents were unsure of the purpose of the application. A future version of the application should invest in a more thorough or more visible introduction for end-users.

6.2.7 Music

As suggested by one of the respondents, it may be interesting to explore an added dimension with a music track as an input to the viewers. Incidentally, a Shadertoy project is available which explores this (shadertoy.com/view/llB3W1), so it should be possible to integrate into the application.

Bibliography

[1] *An Oracle White Paper - Java Client Roadmap Update*. 2018. URL: http://oracle.com/technetwork/java/javase/javaclientroadmapupdate2018mar-4414431.pdf.

[2] *JavaSoft ships Java 1.0*. 1996. URL: https://web.archive.org/web/20070310235103/http://www.sun.com/smi/Press/sunflash/1996-01/sunflash.960123.10561.xml.

[3] *Flash & The Future of Interactive Content*. URL: https://theblog.adobe.com/adobe-flash-update/.

[4] Wikipedia contributors. *Adobe Flash Player — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Adobe_Flash_Player&oldid=935796358. [Online; accessed 18-January-2020]. 2020.

[5] *Flash and the HTML5 <video> tag*. 2010. URL: https://youtube-eng.googleblog.com/2010/06/flash-and-html5-tag_29.html.

[6] *YouTube now defaults to HTML5 video*. 2015. URL: https://youtube-eng.googleblog.com/2015/01/youtube-now-defaults-to-html5_27.html.

[7] *Apple Reinvents the Phone with iPhone*. 2007. URL: https://www.apple.com/uk/newsroom/2007/01/09Apple-Reinvents-the-Phone-with-iPhone/.

[8] Brian X. Chen. *Why Apple Won't Allow Adobe Flash on iPhone*. 2008. URL: https://www.wired.com/2008/11/adobe-flash-on/.

[9] Joshua Rystedt. *Flash is dead. Here's what to do now - Rystedt Creative*. 2018. URL: https://www.rystedtcreative.com/tech-talks/flash-dead/.

[10] Max Slater-Robins. *The long and painful death of Flash*. 2017. URL: https://www.techradar.com/news/internet/the-long-and-painful-death-of-flash-1324425/2.

[11] Simon Owen. *Moving From Flash To HTML, CSS, And JavaScript*. 2018. URL: https://www.smashingmagazine.com/2018/03/from-flash-html-css-javascript/.

[12] Alasdair Corbett. *Mandelbrot Maps – Google Play Store*. URL: https://play.google.com/store/apps/details?id=uk.ac.ed.inf.mandelbrotmaps.

[13] B.B. Mandelbrot, W.H. Freeman, and Company. *The Fractal Geometry of Nature*. Einaudi paperbacks. Henry Holt and Company, 1983. ISBN: 9780716711865. URL: https://books.google.co.uk/books?id=0R2LkE3N7-oC.

[14] Jess McNally. *Earth's Most Stunning Natural Fractal Patterns*. 2010. URL: https://www.wired.com/2010/09/fractal-patterns-in-nature/.

[15] Eric W. Weisstein. *Mandelbrot Set*. [Online; accessed 11-April-2020]. URL: https://mathworld.wolfram.com/MandelbrotSet.html.

[16] Eric W. Weisstein. *Julia Set*. [Online; accessed 11-April-2020]. URL: https://mathworld.wolfram.com/JuliaSet.html.

[17] Wikipedia contributors. *Mandelbrot set — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Mandelbrot_set&oldid=943819946. [Online; accessed 8-March-2020]. 2020.

[18] Wikipedia contributors. *Julia set — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Julia_set&oldid=940023258. [Online; accessed 8-March-2020]. 2020.

[19] Wikipedia contributors. *Romanesco broccoli — Wikipedia, The Free Encyclopedia*. [Online; accessed 12-April-2020]. 2020. URL: https://en.wikipedia.org/w/index.php?title=Romanesco_broccoli&oldid=944891385.

[20] Wikibooks. *GLSL Programming/Vector and Matrix Operations — Wikibooks, The Free Textbook Project*. [Online; accessed 12-April-2020]. 2017. URL: https://en.wikibooks.org/w/index.php?title=GLSL_Programming/Vector_and_Matrix_Operations&oldid=3198898.

[21] OpenGL Wiki. *GLSL Optimizations — OpenGL Wiki*. [Online; accessed 12-April-2020]. 2019. URL: http://www.khronos.org/opengl/wiki_opengl/index.php?title=GLSL_Optimizations&oldid=14548.

[22] Lei Tan. "Similarity between the Mandelbrot set and Julia sets". In: *Comm. Math. Phys.* 134.3 (1990), pp. 587–617. URL: https://projecteuclid.org:443/euclid.cmp/1104201823.

[23] Joe Darcy. *Generics and the Mandelbrot set*. URL: https://blogs.oracle.com/darcy/generics-and-the-mandelbrot-set.

[24] npm, Inc. *This year in JavaScript: 2018 in review and npm's predictions for 2019*. 2018. URL: https://medium.com/npm-inc/this-year-in-javascript-2018-in-review-and-npms-predictions-for-2019-3a3d7e5298ef.

[25] Oliver Williams. *Making the business case for React in 2019*. 2019. URL: https://blog.logrocket.com/making-the-business-case-for-react-in-2019-74463bbb22de/.

[26] *WebGL: 2D and 3D graphics for the web*. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API.

[27] *OpenGL ES - The Standard for Embedded Accelerated 3D Graphics*. URL: https://www.khronos.org/opengles/.

[28] *FXAA Sample*. URL: https://docs.nvidia.com/gameworks/content/gameworkslibrary/graphicssamples/opengl_samples/fxaa.htm.

[29] *Temporal Anti-Aliasing – TXAA Overview — GeForce*. URL: https://www.geforce.com/hardware/technology/txaa.

[30] Khronos Group. *log2 - OpenGL 4 Reference Pages*. URL: https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/log2.xhtml.

[31] Huy Viet Le et al. "Fingers' Range and Comfortable Area for One-Handed Smartphone Interaction Beyond the Touchscreen". In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 2018, pp. 1–12.

[32] Monica Chin. *Google is replacing some Android apps in Chrome OS with web apps*. 2020. URL: https://www.theverge.com/2020/4/13/21219075/google-chrome-os-android-apps-webs-chromebook-replacing-progressive-twitter-youtube-tv.

[33] The Khronos Group. *Khronos OpenGL ES Registry*. URL: https://www.khronos.org/registry/OpenGL/index_es.php.

[34] Kevin I. Martin. *SuperFractalThing – Arbitrary precision Mandelbrot set rendering in Java*. Paper: science.eclipse.co.uk/sft_maths.pdf. URL: http://www.science.eclipse.co.uk/SuperFractalThing.html.

[35] *Testing Overview*. URL: https://reactjs.org/docs/testing.html#tradeoffs.

[36] Amber Rockwood. *Automated Cross-Browser Testing for WebGL- It's Not Going to Happen*. 2018. URL: https://www.eventbrite.com/engineering/automated-cross-browser-testing-webgl-not-going-happen/.

[37] *Lighthouse – Tools for Web Developers – Google Developers*. URL: https://developers.google.com/web/tools/lighthouse.