Mandelbrot Maps: Rebuilding for a responsive cross-device experience

Freddie Bawden

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2020

First created by Iain Parris in 2009, the Mandelbrot Maps project aimed to create a browser-based real-time fractal viewer. Since then, others have ported the application to Android to provide an experience for touch devices. Today, most browsers support touch directly — by rebuilding the application for the modern web, it can support touch and desktop devices without the need to maintain separate applications for each target.

The application was rebuilt using Javascript and WebAssembly to create a responsive cross-device application which is usable on desktop and mobile devices. The application supports a user interface that allows for intuitive control through multiple input sources and a renderer which uses web workers to parallelize the workload. The renderer is able to draw fractals magnified 10,000,000 times in under a second on popular browsers. A user experience survey based on the System Usability Scale showed our system exceeded the industry standard: it scored 77.8 out of 100, where a rating of 70 or higher is considered acceptable. The survey also demonstrated that our user interface is accessible and easy to learn regardless of the user's mathematical background. These results show that the new version of Mandelbrot Maps is a successful update of the original project. The updated application is available to try at magentahttp://mmaps.freddiejbawden.com

Acknowledgements

Special thanks to:

Philip Wadler My supervisor for this project, who offered great advice and feedback which drove the project forwards.

Siobhan Vickerstaff For helping unscramble my thoughts into understandable words.

The people of Appleton Tower floor 9 For keeping me sane and letting me commandeer their computers to test on

Table of Contents

Chapter 1

Introduction

The Mandelbrot Maps project was originally created by Iain Parris in 2009 [1] (Figure 1.1). The project's aim was to provide a browser based fractal viewer that demonstrated the relationship between the Julia and Mandelbrot fractals. The project has been a staple for undergraduates ever since; the latest version was developed for Android devices and saw great success on the Play Store [2].

A decade on from the release of Parris' application, a number of the dependencies needed for the application have been deprecated. Notably, support for Java Applets, (the framework the application is built on) has been dropped by many browsers [3] [4] [5] [6]. Without support for the framework, the application is no longer usable online.

In addition to losing dependencies, the application was not optimised for touch devices. When Parris originally created the application desktop browsers accounted for 99.33% of all browser traffic [7], making touch controls an unnecessary addition. Today, touch-first devices make up the majority of browser traffic [7]. The lack of touch support was partially solved by moving to an Android application [2]. However, the Android application does not provide an alternative for iOS or desktop users.

I aim to provide a new version of Mandelbrot Maps for all devices by building a modern web application (Figure 1.2). To achieve this I undertook the following:

- Creating a cross-platform rendering engine using WebAssembly and Javascript
- Designing an accessible user interface which can support touch and desktop devices and is usable on many screen sizes
- Evaluating the performance of the application through user surveys and automated testing.

The project was created in parallel with Joao Maio. Both of us focused on different technologies, I used WebAssembly and Javascript to render the fractals while Joao used WebGL, a web-based graphics engine. Throughout the project, we held joint biweekly meetings with our project supervisor to share ideas. However, due to differences in the WebGL and WebAssembly interfaces, no design or code was shared between the projects.

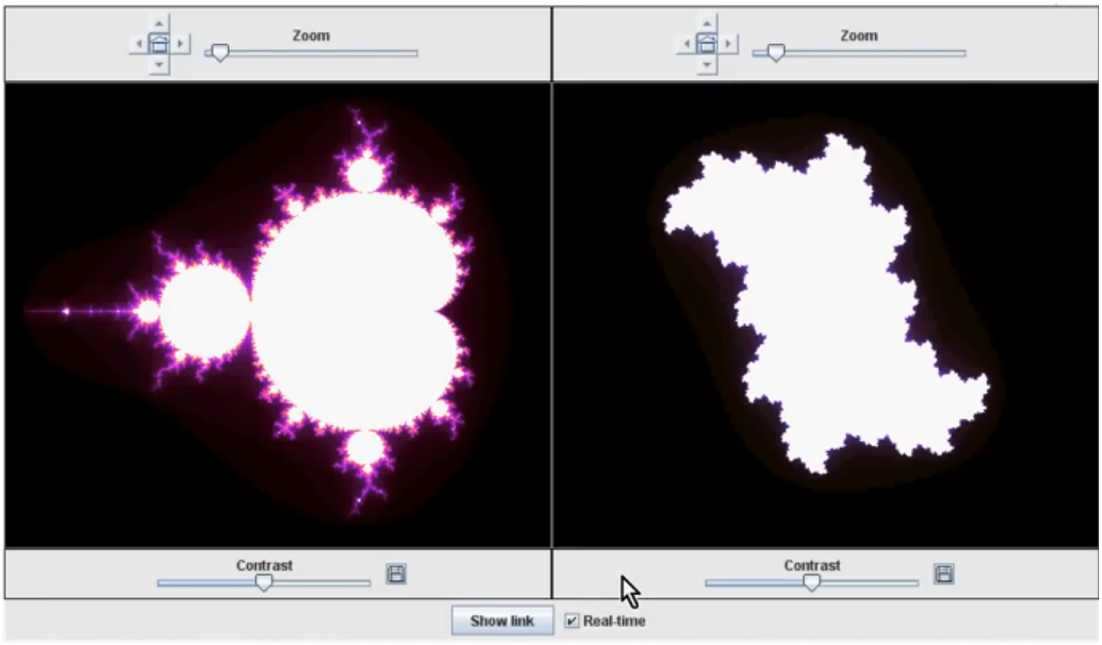
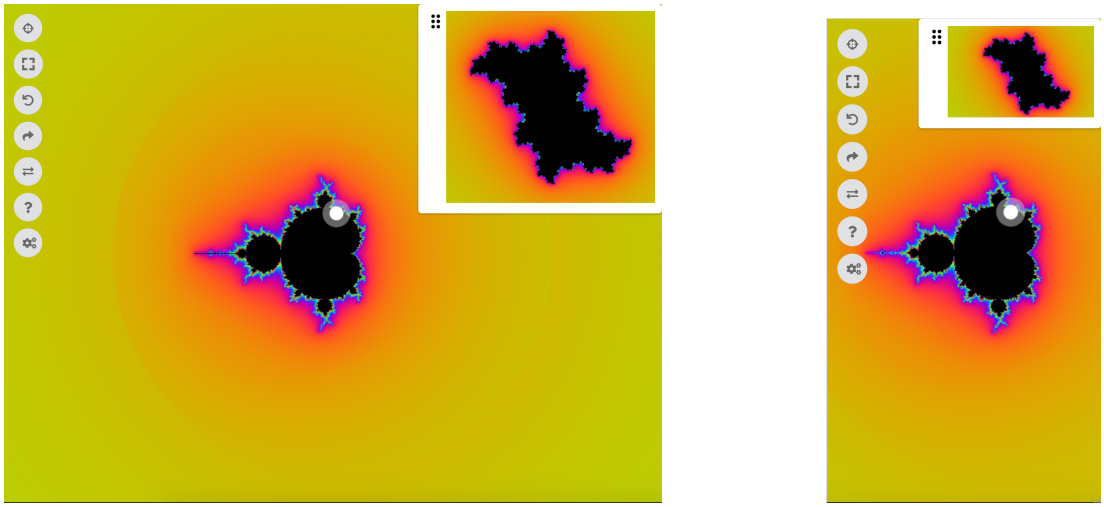


Figure 1.1: Original Mandelbrot Maps application. The original application was built using Java. This meant as browsers dropped Java support the application became inaccessible online. The original application also did not have a touch interface. This is likely due to touch interfaces being a rare use case at time of creation. Touch devices now make up the majority of browser traffic, the updated application must therefore include a touch interface [7]. (Image taken from an instructional video created by Parris [8])



(a) Desktop Interface **(b)** Mobile Interface

Figure 1.2: Updated Mandelbrot Maps application. The updated application is built using WebAssembly and Javascript. The interface was designed to be usable regardless of screen size; we achieved this by taking inspiration from popular mapping applications (Section 3.1). We built intuitive interactions for mouse, multi-touch trackpad, touch and keyboard to allow the user to explore the fractals.

Chapter 2

Background

2.1 Mathematical Background

A detailed understanding of fractals is not needed for this project; however a general background helps justify the design choices made during this project. This section assumes a basic understanding of complex numbers and the complex plane.

2.1.1 Iterative Systems

The fractals shown in the application are examples of iterative systems. In these systems, we repeatedly apply a function to a point, feeding the output of the function as the input for the next iteration. We describe the values $x_0, x_1, x_2, x_3 ...$ generated at each step by $f(x)$ as the orbit of x_0 under f.

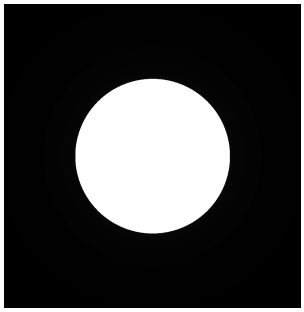
An orbit can either be convergent or divergent depending on its behaviour. Converging orbits approach a fixed point or are contained in a fixed area; diverging orbits escape to infinity. The starting point of a function can change an orbit's behavior. For example, when $-1 \leq x_0 \leq 1$, the orbit under $f(x) = x^2$ converges. When $-1 > x_0$ or $1 < x_0$, the orbit grows to infinity, diverging.

2.1.2 Filled Julia Set

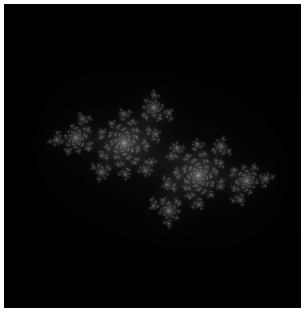
The Filled Julia Set of a complex number c is the set of complex points such that orbit of $f(z) = z^2 + c$ for $z \in \mathbb{C}$, where $\mathbb{C}$ is the set of complex numbers, converges. From this we can derive the Julia set; the boundary of the Filled Julia Set. For simplicity we will use "Filled Julia Set" and "Julia Set" interchangeably.

Plotting these points in the complex plane we can see that the set changes depending on our constant c. Setting $c = 0$ produces a connected structure (Figure 2.1a); $c = -0.67 - 0.4i$ separates the set into disconnected dusty particles (Figure 2.1b); $c = 1.5$ produces a different, yet still connected set (Figure 2.1c).

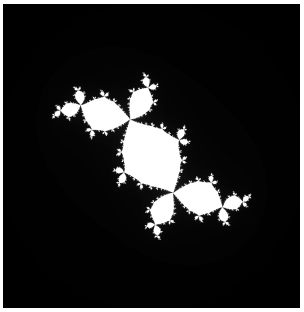
The connectedness of a Julia set is described by Mandelbrot's Criterion: a Julia set is connected if and only if $z_0 = 0$ under $f(z) = z^2 + c$ converges [9]. From this theorem we can derive the Mandelbrot Set: the set of $c \in \mathbb{C}$ for which Mandelbrot's Criterion is true (Figure 2.2)..



(a) $c = 0$ produces a connected set.



(b) $c = -0.67 - 0.4i$ produces a disconnected set.



(c) $c = 1.5i$ produces a different connected set.

Figure 2.1: Julia Sets with different c values. We visualise the Julia set by coloring a point white if it belongs to the set, black if not. We can see the set changes dramatically depending on the starting point.

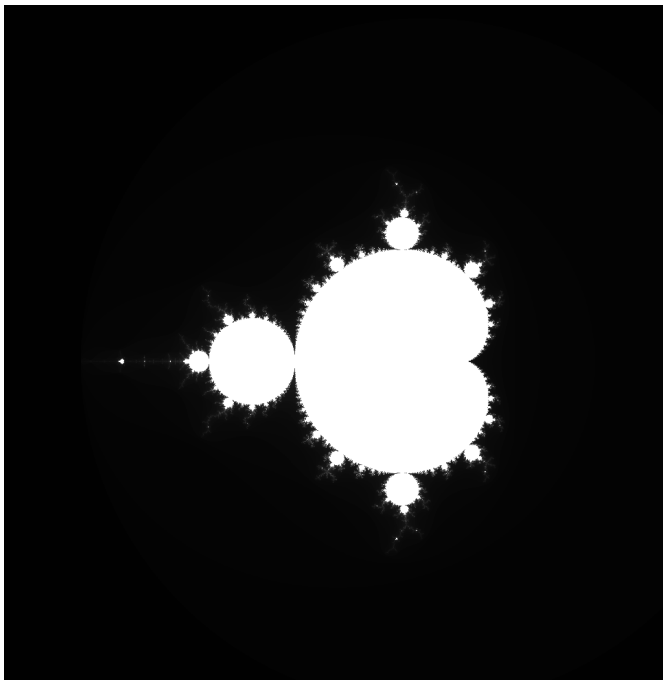


Figure 2.2: The Mandelbrot Set. We visualise the Mandelbrot set in the same way as the Julia set. If a point belongs to the set, it is colored white, otherwise the point is black.

2.1.3 Mandelbrot Set

The Mandelbrot Set is the set of complex points such that the orbit of the function $f(z) = z^2 + c, c \in \mathbb{C}$, where $z_0 = 0$, converges. As z_0 is constant, there exists only one Mandelbrot Set.

The Mandelbrot Set provides a *map* of Julia Sets, when a point c in the Mandelbrot set converges, the Julia Set of c is connected. Conversely, when a point diverges, the corresponding Julia Set is disconnected.

2.2 Technical Background

2.2.1 Rendering Mandelbrot and Julia Sets

Rendering the Mandelbrot and Julia sets requires knowing whether a point converges or not. This is challenging as no closed form solution has been found to determine a points behavior for either set. We must therefore iterate each point individually to determine its behavior. However, as convergent points can bounce around a fixed area infinitely we cannot know if a point has converged after a finite number of iterations. We can therefore only approximate the curves when rendering.

The Escape Time Algorithm presents a simple method for performing the approximated calculation. Using the fact that the orbit of the function, $f(z) = z^2 + c, c \in \mathbb{C}$, will always diverge when $|z_n| > 2$, we iterate each point on the complex plane until its absolute value is greater than 2, or we reach an iteration limit.

Placing an iteration limit guarantees that any point within the Mandelbrot Set will be shown as such, however points outside the set that need more iterations than the limit to diverge may be mis-classified as convergent. Raising the iteration limit reduces this inaccuracy, but impacts performance due to the additional iterations.

2.2.2 Multithreading In The Browser

In browsers, events such as button presses and API calls occur sequentially due to script execution using a single thread. This causes computationally heavy processes to slow all other script execution, meaning the page will be unresponsive [10].

A common remedy for slow web pages is to use an external server to which web pages can offload heavy computation. While this solution prevents computation from blocking interaction, the response time will increase due to the network transfer time. As my application relies on quick interactions, this is not a viable solution.

Web workers were introduced to the HTML5 standard as another method to counter single threaded script execution. They allow for intensive tasks to be offloaded to a separate thread while keeping the main thread responsive [11]. Web workers are ideal for rendering fractals as they allow us to perform the escape time algorithm without blocking the main thread. I use web workers to parallelize the escape time algorithm.

2.2.3 WebAssembly

Parris' application was build using Java Applets; a system which allowed for Java Bytecode to be run in a separate process from the web client. Java Applets also provided hardware acceleration and access to additional computation resources, meaning it was a popular choice for intensive applications. However, mounting concerns surrounding key dependencies needed to run Java Applets led to support being dropped by popular browsers [3] [4] [6] [5]. As Java applets departed, a new standard arrived to keep the web running quickly.

WebAssembly is a new coding standard which provides a portable binary format that focuses on high performance in the browser. Its compiled nature makes it much faster to parse than an equivalent Javascript file. WebAssembly also makes it far easier to write performant code through finer control over memory and by utilising common hardware optimisations [12].

I chose to compile Rust (a language focused on performance and safety [13]) into WebAssembly using *wasm-pack* [14]. This decision was motivated by Rust's functionality for exposing methods to Javascript through the *wasm-bindgen* library [15]. This library made integrating WebAssembly into the application more simple. Writing in Rust and then compiling to WebAssembly rather than manually writing the WebAssembly binary also improves maintainability for future contributors. I use WebAssembly to provide a fast experience across browsers.

2.2.4 React

React is an open source Javascript framework maintained by Facebook along with a large community of independent developers and companies [16]. React is a popular option when creating modern web applications: Github hosts 3.4 million projects that use React [17]. A React application is built out of multiple components each of which manage their own state. These components are arranged in a hierarchical structure to allow for efficient rendering of the application. This hierarchy also makes it easier to create dynamic web pages; when a component's state is updated, React re-renders the updated component along with all its children. I use React to organise code and increase maintainability for future contributors.

Chapter 3

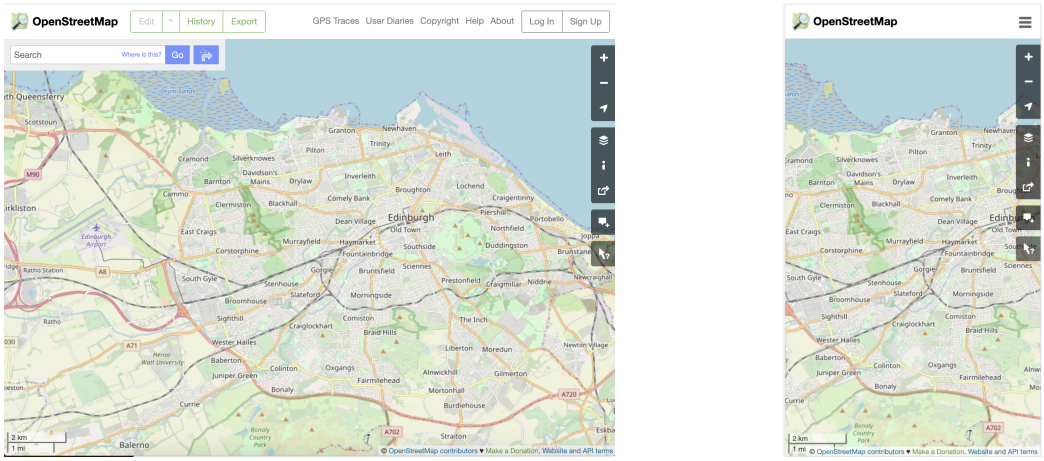
Implementation

3.1 User Interface

Parris's application was created for desktop computers and could not scale to fit mobile screens. To create a cross-device experience, I redesigned the user interface so that it is usable on many screen sizes. To see how similar applications create responsive user interfaces, I evaluated popular mapping applications, OpenStreetMap (Figure 3.1) and Google Earth (Figure 3.2).

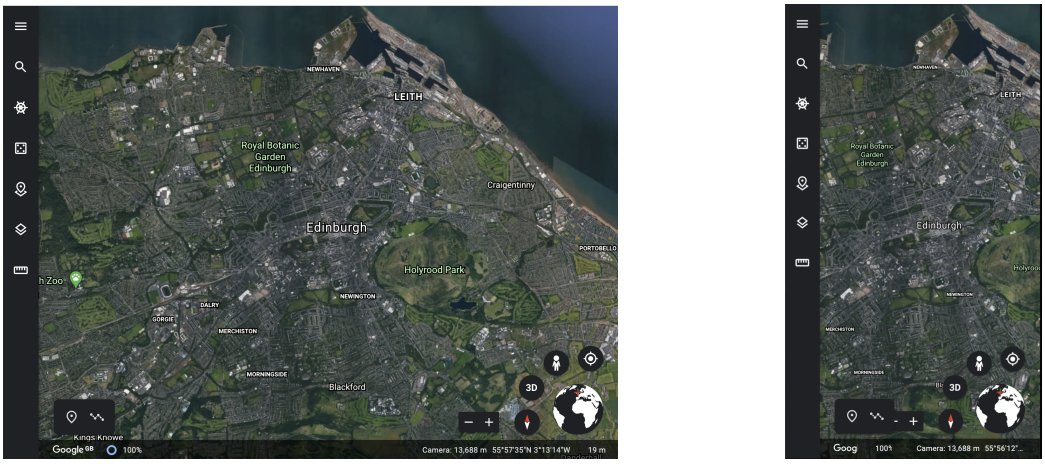
Both mapping applications have a similar structure; they display a fullscreen map and provide additional functions through a small iconographic menu (upper right in Figure 3.1 and upper left in Figure 3.2). This approach means the interface does not change much when scaled down to a mobile screen which provides the user with a consistent experience: a key principle in building cross-device user interfaces [18].

My application's design takes inspiration from these mapping applications (Figure 3.3). The user interface presents the user with a fullscreen fractal view and provide additional features using a sidebar menu. By calling upon other mapping applications, my application is able to create a consistent experience across devices.



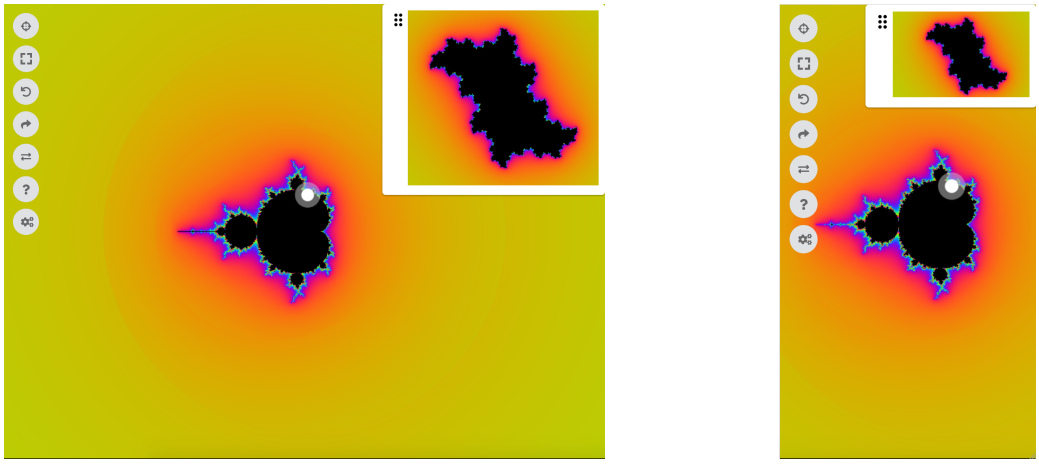
(a) Desktop Interface **(b)** Mobile Interface

Figure 3.1: OpenStreetMap Interface. OpenStreetMap is a community driven mapping application with over 6 million registered users. It allows users to view, annotate and share highly detailed maps. [19].



(a) Desktop Interface **(b)** Mobile Interface

Figure 3.2: Google Earth Interface. Google Earth provides an extensive set of tools for exploring satellite imagery. [20]



(a) Desktop Interface **(b)** Mobile Interface

Figure 3.3: Redesigned Mandelbrot Maps Interface. Both OpenStreetMap and Google Earth display a fullscreen map and put controls to the side of the window (upper right in Figure 3.1, upper left in Figure 3.2). The updated application mirrors this layout to create a consistent experience. I used the Semantic UI design system to create the user interface [21]. This provides premade interface elements to which I gave interactive functions.

3.2 Viewer Options

Parris's original application placed the Mandelbrot and Julia fractals side-by-side (Figure 3.4). This layout works well on desktop but suffers on smaller screens. The recent Android implementation created by Alasdair Corbett solves this issue by shrinking one of the fractals into a corner, freeing up space for the larger (Figure 3.5). As the updated application is to be used on both desktop and mobile browsers, I added both arrangements to the application, along with a fullscreen mode to allow users to explore one fractal at a time (Figure 3.6).

Users can cycle through view modes using the "Change view button" and use the "Swap Fractal" button to change which fractal is minimized. The viewers are able to adapt to the screen they are being used on, for example the side-by-side mode stacks the fractals vertically when the user is using a portrait display (Figure 3.7a). The detached viewer adapts by shrinking in order to not obscure the primary fractal (Figure 3.7b).

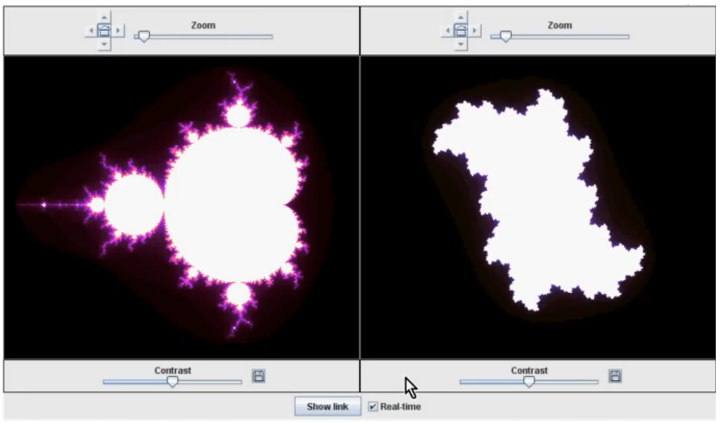


Figure 3.4: Parris's original application. The original application used a side by side view to display the fractals. This works well on desktop however suffers on mobile due to the reduced screen size. (Image taken from instructional video created by Parris [8])

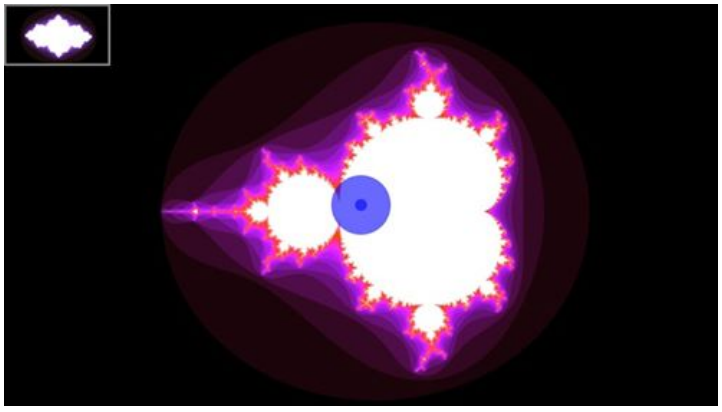
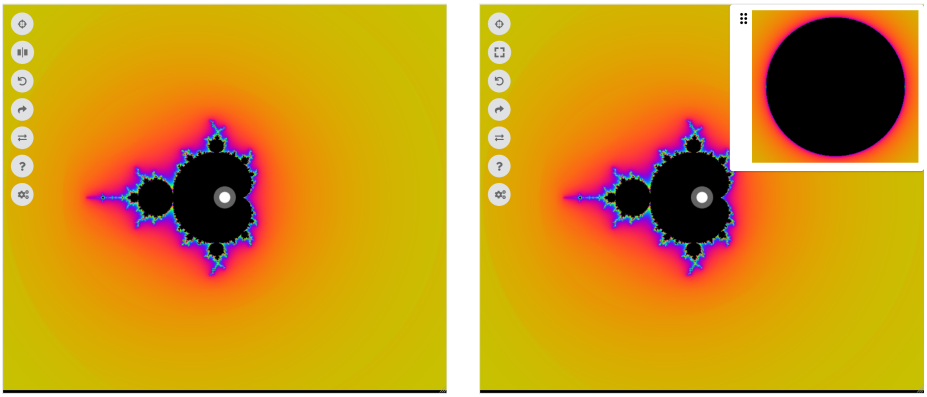
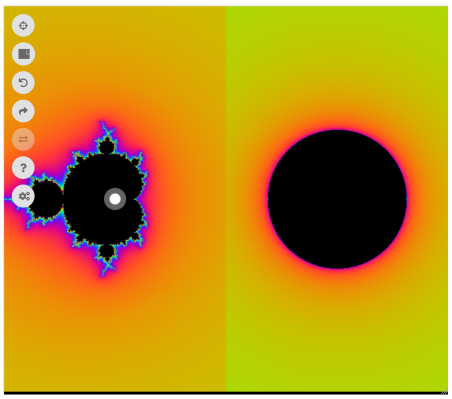


Figure 3.5: Mandelbrot Maps Android app interface. The Mandelbrot Maps app created by Alasdair Corbett makes a mobile friendly experience by shrinking one fractal into the corner, freeing up space for the other [2] (Image taken from application running on my phone)

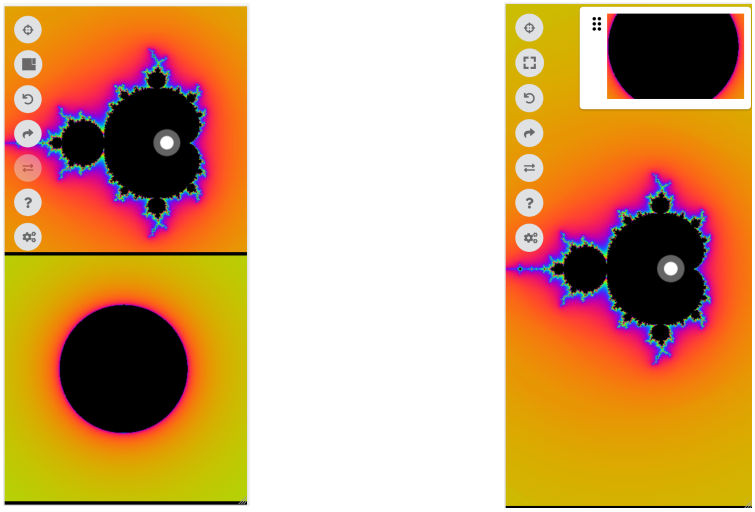


(a) Fullscreen **(b)** Detatched



(c) Side By Side

Figure 3.6: View options provided by the application. The application provides 3 view options, fullscreen, detatched and side by side.



(a) Vertically stacking side by side view **(b) Shrinking the detached viewer**

Figure 3.7: Adapting to mobile screens. The application's interface changes depending on the user's screen size. This makes view modes usable across devices.

3.3 Architecture

The application is structured in a hierarchy to follow React guidelines. At the root of the tree is the application router created by *react-router* [22]. The router loads different components depending on the URL path, for example */app* loads the fractal viewer, while */feedback* loads the feedback form.

3.3.1 State Management

React's hierarchical structure makes it challenging to share state among components. For example, the Mandelbrot viewer must share the current c value to the Julia viewer. A common solution to this problem is using a global state management system. I created my own lightweight state management system inspired by L. Spyna [23].

This system uses the root component to maintain the global state and provides an interface for other components. Updating the state at the root causes the whole component tree to re-render. Re-rendering a component is expensive, therefore each component independently decides whether it needs to update given the new state. This system prevents unnecessary renders.

3.4 User Interaction

The application uses a *canvas* element to display the fractal. The *canvas* element does not have in built functionality for dragging or zooming. I therefore implemented this functionality myself using the browser's interaction events.

A web page can capture interaction events using a listener interface. The application uses these events to enable custom interactions using a mouse, multi-touch trackpad, touch screen or keyboard.

Including keyboard controls is important for accessibility. Users who cannot use a mouse due to physical disability or conditions such as repetitive stress injuries often use the keyboard as their means of interaction with a web page [24]. Implementing bindings for keyboard allows these users to more easily access the application.

Keyboard controls however does not provide an alternative for users using touch controls. Touch accessibility involves creating alternative, single pointer motions for multi-touch gestures, such as a two finger pinch zoom (Section 3.4.2.1). This means that users who are using a stylus or a single finger can still use the application [25]. The interface therefore provides alternative control schemes for those using a single pointer. This could be improved further by adding button inputs for some interactions, such zooming in and out, similar to those found in Google Earth (Figure 3.8).

3.4.1 Dragging

The dragging interaction allows users to pan the fractal. Users using mouse or trackpad can "click and drag"; keyboard users can use the arrow keys and touch users can "touch and drag". These actions mirror the control scheme seen in Google Earth [20].

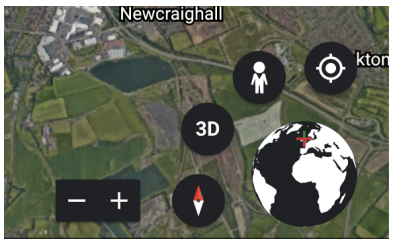


Figure 3.8: Alternative controls for Google Earth. Google Earth provides alternative controls for its complex pinch zoom gesture through + and − buttons [20]. Adding a similar feature to the application in future would help improve its accessibility

In a prototype design, the application rendered the fractal every time the user panned it. This method led to a choppy experience as the browser would have to wait for the render to complete before continuing with the movement. I solved this by moving the previously rendered image with the drag motion; only rendering once the motion was completed. This method led to a smoother experience as moving the image is much quicker than rendering a new one.

I used a similar solution when implementing the Julia Pin interaction: users can drag the Julia Pin on the Mandelbrot fractal to update the Julia set's c value. This movement requires re-rendering the whole Julia set. Rather than re-render the whole image when the user moves the pin, the application performs a low iteration render of the image and then renders at full quality after the interaction is over. This allows for the Julia set to react in real time to the changes in the Julia pin.

3.4.2 Zooming

The zoom interaction allows users to enlarge the fractal. Mouse users can use the scroll wheel, trackpad and touch users can "pinch and zoom" (as recommended by Google's Material Design guidelines [26]) and keyboard users can use the plus and minus keys. User can also double tap to zoom the fractal; this provides a single pointer alternative for touch users. Again, a similar control scheme is seen in Google Earth [20].

The prototype design used a simple zoom which scaled the image around it's centre. This made for a bad user experience as users would have to zoom then re-position the viewer to reach their desired position. Mapping applications solve this by zooming in on a point rather than the centre of the image.

I implemented this solution by defining an anchor point in the world space whose position on screen is maintained after a zoom (Figure 3.9). The anchor point is placed at the centre of the screen when using keyboard, the centre of the pinch on mobile and the pointer position when using trackpad or mouse. This allows the user to zoom in on a point in the fractal without having to re-position the viewer.

When the user zooms, the application must re-render the whole fractal as the pixel size (the area of the fractal one pixel represents) changes. Rendering the whole image is an intensive activity therefore the application scales the previous fractal while the user zooms. After the user has not zoomed for a short time, the fractal is re-rendered.

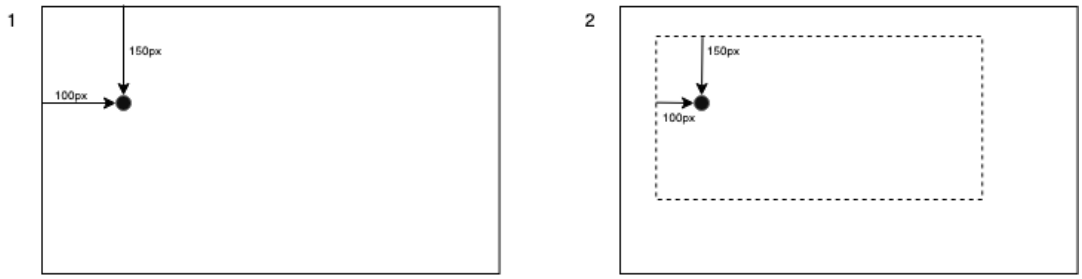
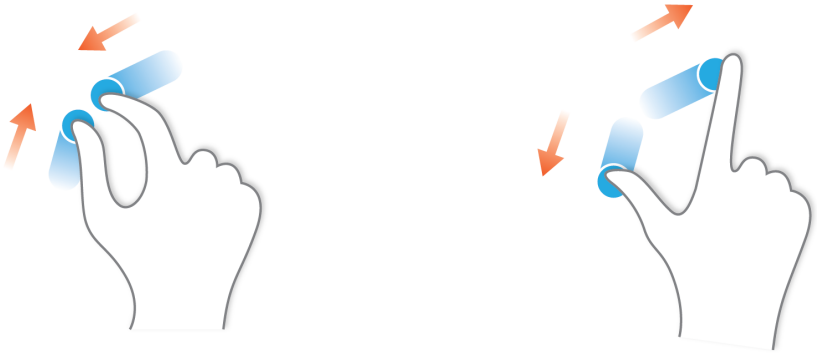


Figure 3.9: The anchored zoom process. We put our anchor point at (100, 150), we then scale the view and reposition the zoomed view such that our anchor point is at (100, 150) in the new frame.

3.4.2.1 Pinch And Zoom

Unlike scrolling using a mouse, browsers currently do not contain a pinch zoom event for touch screen devices (Figure 3.10). I developed a straight-forward system to enable this feature by scaling the image with the velocity of the pinch. This allows for the zoom speed to match the user's pinch, creating a responsive feeling.



(a) Contracting fingers on a point on the screen zooms the image out

(b) Expanding fingers on a point on the screen zooms the image in

Figure 3.10: Pinch Gesture (Image Credit: GRPH3B18 CC BY-SA [27] [28])

3.5 Rendering

Efficient rendering is critical for the usability of the application. If the application cannot update the fractal quickly, the panning and zooming actions will feel unresponsive.

I created two renderers for the application, one using Javascript and one using WebAssembly; this allows us to see if WebAssembly provides a significant improvement in performance compared to Javascript. I analyse the performance of each renderer in Section 4.1.

The escape time algorithm (Section 2.2.1) requires many calculations to render an image. For example, an image of size 1000 x 1000 pixels with a limit of 200 iterations could require as many as 200,000,000 calculations! To prevent the user interface from freezing during execution, the application offloads the rendering work to web workers (Section 2.2.2).

The render system must be able to translate between screen space (defined in pixels) and the world space (a continuous plane). This is achieved by maintaining a pixel size parameter, (the area contained in a single pixel); for example a pixel size of 0.001 represents a 0.001 x 0.001 segment of world space.

Using the pixel size, along with the screen width, screen height and centre point of the fractal, we can to create a bounding box in the world space. This is accomplished through Equation 3.1 which calculates the corners of the bounding box given a pixel size α, screen width in pixels, w, screen height h in pixels and a centre point defined in world space (x_c, y_c).

$$\begin{aligned}(x_1, y_1) &= (x_c - \frac{w}{2}\alpha, y_c - \frac{h}{2}\alpha) \\ (x_2, y_2) &= (x_c + \frac{w}{2}\alpha, y_c + \frac{h}{2}\alpha)\end{aligned} \tag{3.1}$$

After defining the bounding box, the render system steps through each pixel and computes the escape time for the point at the centre of each world space segment. The render system returns an array containing RGBA values for each pixel (discussed further in Section 3.5.4). The array is then compiled into an image and displayed on screen using a *canvas*.

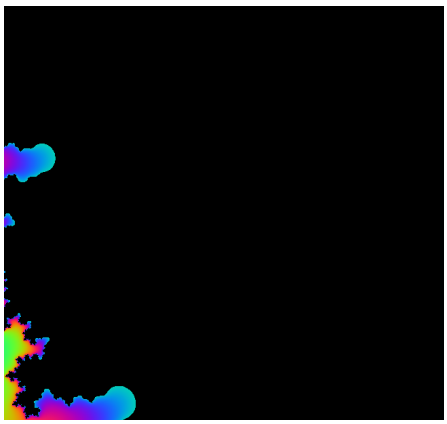
3.5.1 Selecting the Iteration Count

At deep zoom levels, a low iteration count will produce an image that lacks detail (Figure 3.11a). This result is caused by many of the points reaching the iteration count limit and being inaccurately assumed to be convergent. Raising the iteration count reduces the inaccuracy but causes the render time to slow as a result (Figure 3.11b). The renderer must therefore dynamically set the iteration count based on the zoom level to achieve an optimal balance of performance and detail.

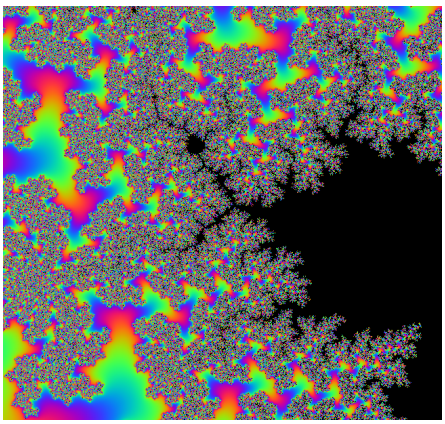
Parris investigated this when creating the original application and found a good solution for varying the iterations with the pixel size (Equation 3.2) [1]. I modified Parris's function through visual inspection of the resultant fractals to better suit my application's coloring function (Equation 3.3).

$$iterations = 54e^{1.23|\ln(pixelSize)|} \tag{3.2}$$

$$iterations = 54e^{0.2|\ln(pixelSize)|} \tag{3.3}$$



(a) Fractal rendered at 100 iterations



(b) Fractal rendered at 500 iterations

Figure 3.11: Impact of iteration count at deep zooms. Rendering the point $(-0.299 - 0.644i)$ at 20000x zoom with a low iteration count causes a loss in detail as shown in Figure 3.11a. By raising the iteration count we add more detail to the image (Figure 3.11b).

3.5.2 Multithreading

Rendering a Mandelbrot fractal is classed as an ”embarrassingly parallelisable” problem; one which can be easily split among different threads [29]. Each pixel can be evaluated independently meaning they can be distributed among threads independently.

A common method of parallel rendering is to divide the image into chunks and distribute them amongst threads [30]. To achieve this, I created a *WebWorkerManager* to coordinate web workers to evaluate chunks in parallel. The *WebWorkerManager* splits the image into n chunks, then distributes the tasks among web workers using a queue and assembles processed chunks into an image.

Finding the optimal number of chunks is challenging. With each call to a web worker there is computational overhead meaning the more chunks *WebWorkerManager* creates, the greater the total overhead is for the process. However, with fewer chunks, the *WebWorkerManager* may not distribute the work evenly, which causes the renderer to have to wait for one thread to finish. I found the optimal number of chunks to be 8, this is discussed further in Section 4.1.1.

3.5.3 Partial Rendering

When the user is panning much of the current image does not need to be recalculated. After the user finishes dragging the renderer copy previously calculated pixel values into the new position and only calculates the newly revealed areas (Figure 3.12). This reduces render time when panning, creating a more responsive experience.

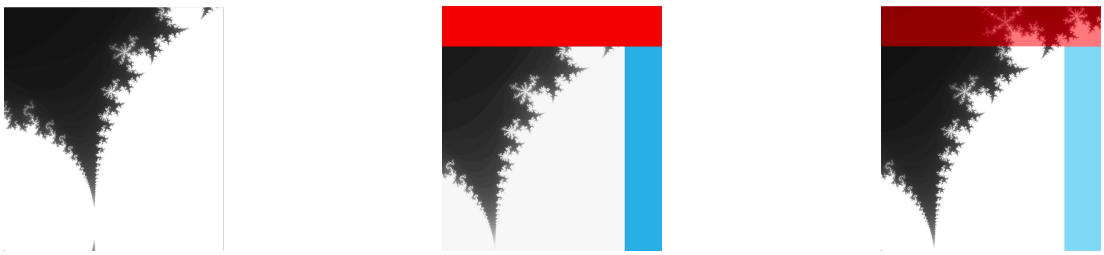
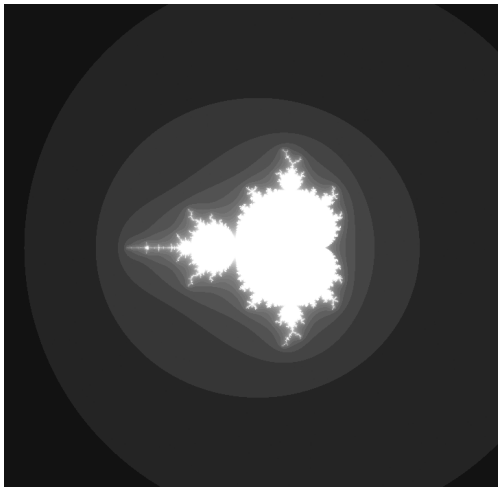


Figure 3.12: Partial Rendering System. When the user pans the image, much of the image does not need to be recalculated. The renderer therefore copies previously calculated pixel values into a new position. The highlighted area is then calculated to produce the new fractal.

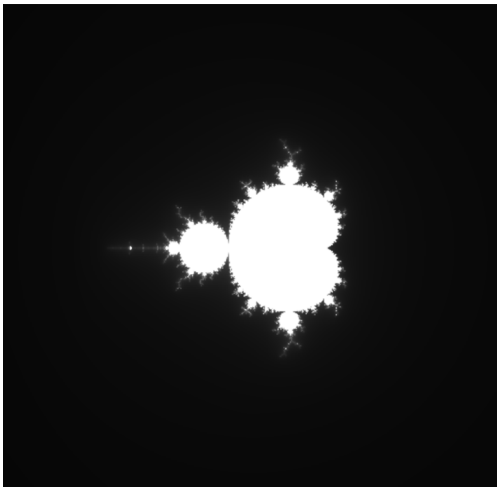
3.5.4 Coloring

A common method of coloring uses the escape time of a point. This method interpolates through a color space as the escape time reaches the maximum. However as the iterations are discrete, images rendered with this method exhibit a haloing effect (Figure 3.13a). To solve this, I used a solution proposed by I. Quilez which uses the magnitude of the point to add a continuous part to the discrete iteration count, making the iteration count continuous (Equation 3.4) (Figure 3.13b) [31].

$$u(z_n) = (i-1) - log_2(log_2(||z_n||)) + 4 \tag{3.4}$$



(a) Result before applying smooth coloring Coloring based off the discrete iteration count caused halos to form around the fractal.



(b) Result after applying smooth coloring. By using the magnitude of a point as a continuous part to the discrete iteration count, we can smooth out the coloring, making it much more appealing.

Figure 3.13: Comparison of coloring based on discrete and continuous iteration count

3.5.4.1 Rainbow Color Scheme

I created a rainbow effect by varying the RGB channels of each pixel using offset sine waves. The sine waves are offset because when the RGB values are equal, the resulting color would be grey. Offsetting the sine waves for each channel means all three channels will never be equal, creating a rainbow effect (Figure 3.14). Figure 3.15 shows the final result.

$$colorvalue = \sin(0.3i + offset) \cdot 127 + 128 \tag{3.5}$$

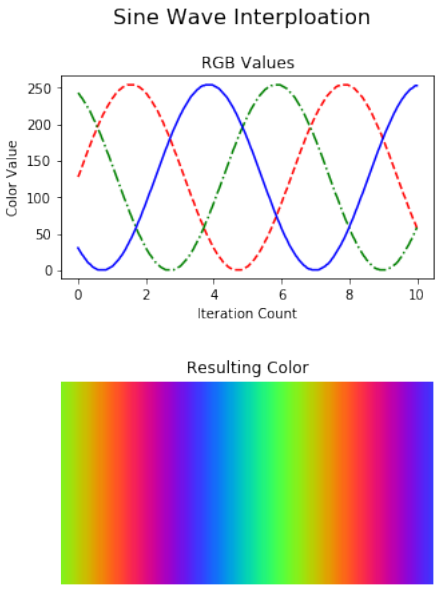


Figure 3.14: Variation of RGB channels producing the rainbow The iteration count is put through three offset sine waves for each color channel, the result is a smoothly changing rainbow effect

3.5.4.2 Striped Color Scheme

To create a striped pattern the renderer passes the smoothed iteration count through a cosine function (Equation 3.6) which is used to interpolate the RGB channels between 0 and 255. We multiply the smooth iteration count by 2π to cycle the color with each iteration creating the striped effect (Figure 3.16).

$$colorvalue = \frac{1 + \cos(2\pi \cdot i)}{2} \tag{3.6}$$

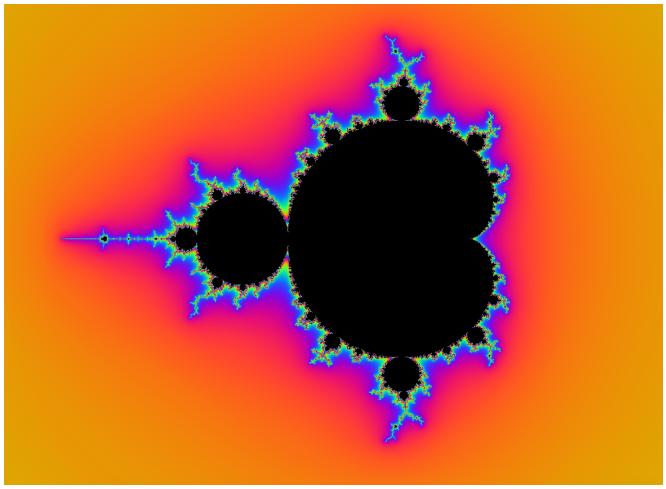


Figure 3.15: Rainbow Color Scheme Passing the iteration count through sine waves produces a rainbow pattern. The renderer colors areas black if they are part of the set to make them stand out against the colorful background.

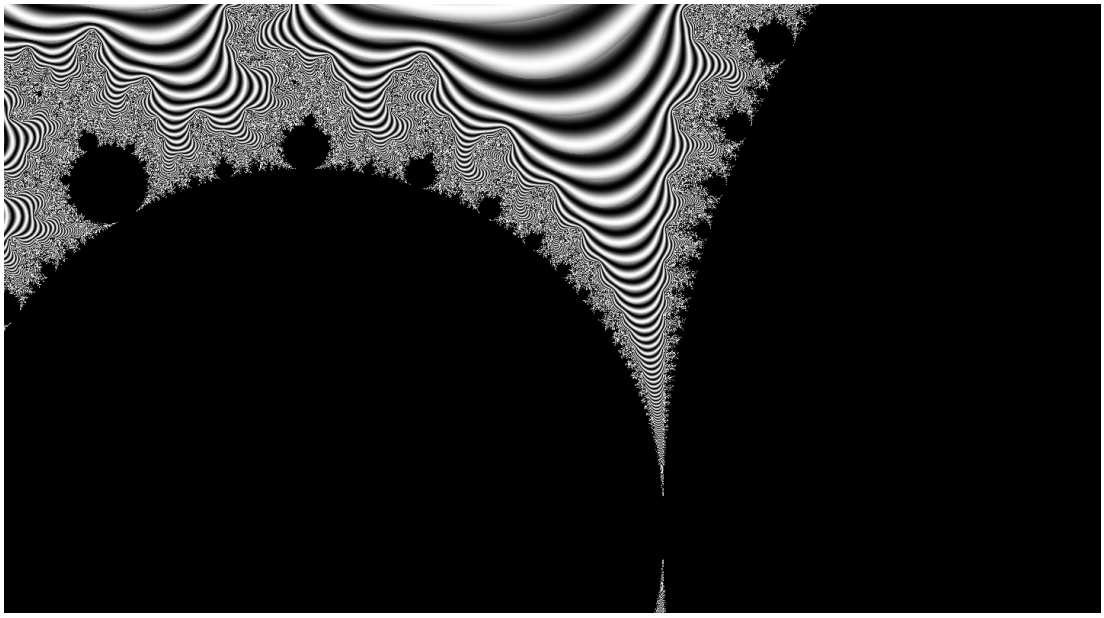


Figure 3.16: Striped Color Scheme The striped coloring scheme gives a unique look to the fractals, adding visual interest in regions where the iteration count is stable

3.6 Link Sharing

The link sharing feature allows users to share to specific areas of the fractal through a URL query string. URL query strings are attached at the end of a URL and contain a list of parameters which can be processed by the application. Users can generate a custom link using the "Share Current Fractal" button. This button encodes the current position and zoom of each fractal in a URL string and displays a dialogue box prompting the user to copy the URL. When the encoded URL is opened by another user, the application parses the values and uses them to configure the viewer to show the shared position.

3.6.1 Link Cards

The Open Graph protocol allows links to become rich objects when shared; these can include a description, title and image which are defined in the HTML page's meta data. I created a link card for Mandelbrot Maps to provide more context about the application and make links more attractive when shared on social media (Figure 3.17).

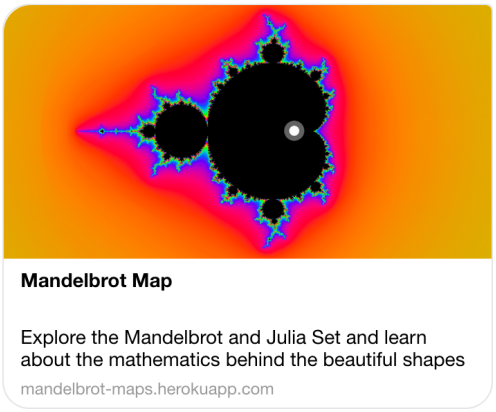


Figure 3.17: Open Graph link as displayed on Facebook Messenger. The rich content of OpenGraphs gives the user more context about the shared URL and makes the URL more visually appealing to the user.

3.7 Tutorial

In previous iterations, tutorials were only available through external videos [8] [32]. I aimed to increase accessibility by adding a tutorial into the application as a more efficient method of teaching the user how to use it.

I provide a tutorial through a static help page. This demonstrates how to use the application through text prompts and animation of the application (Figure 3.18). When users visit the site for the first time, a modal appears which prompts them to view the tutorial before starting. Once the user closes the modal (Figure 3.19), the application stores a flag in the browser to prevent it from displaying again.

I also included an in-app help box so the users can be reminded what the controls are without having to open a new page (Figure 3.20)

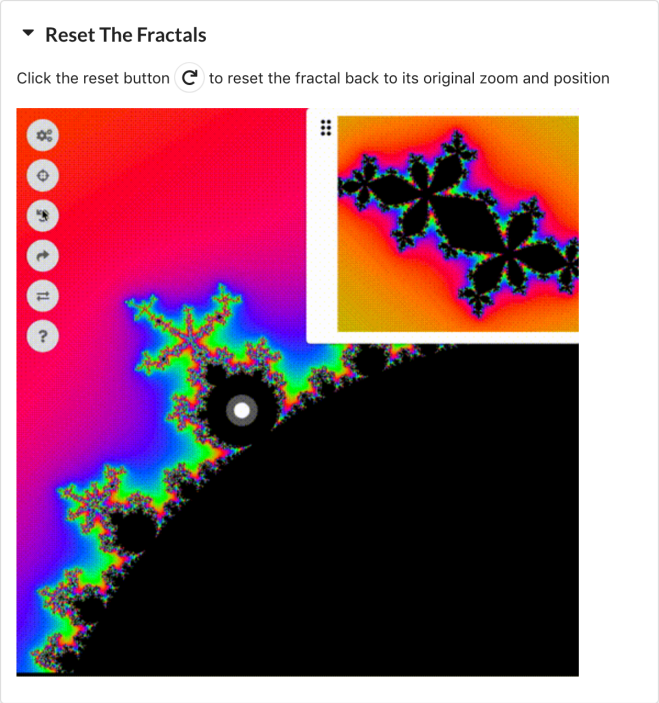


Figure 3.18: Help Card. The help cards are displayed in the tutorial page and use animated GIFs to explain how to use the application. This card shows an animation of the mouse pointer clicking the reset button and how the fractal changes after.

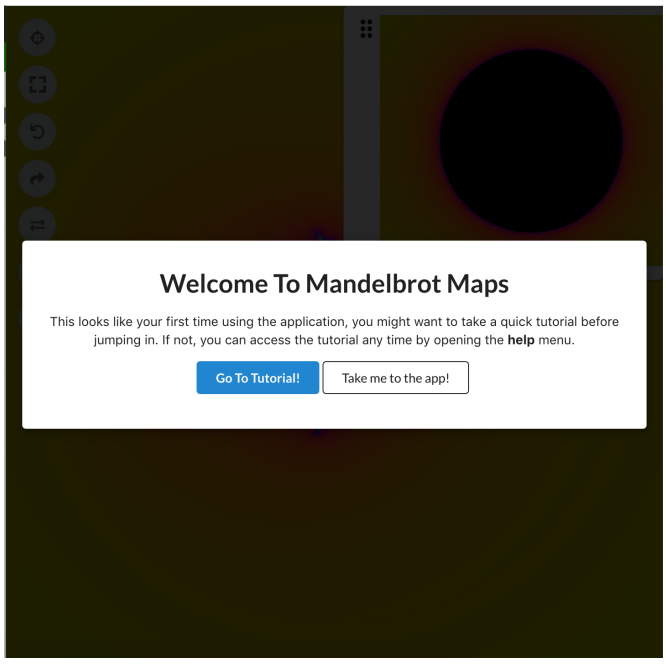


Figure 3.19: The tutorial prompt which is displayed to new users.

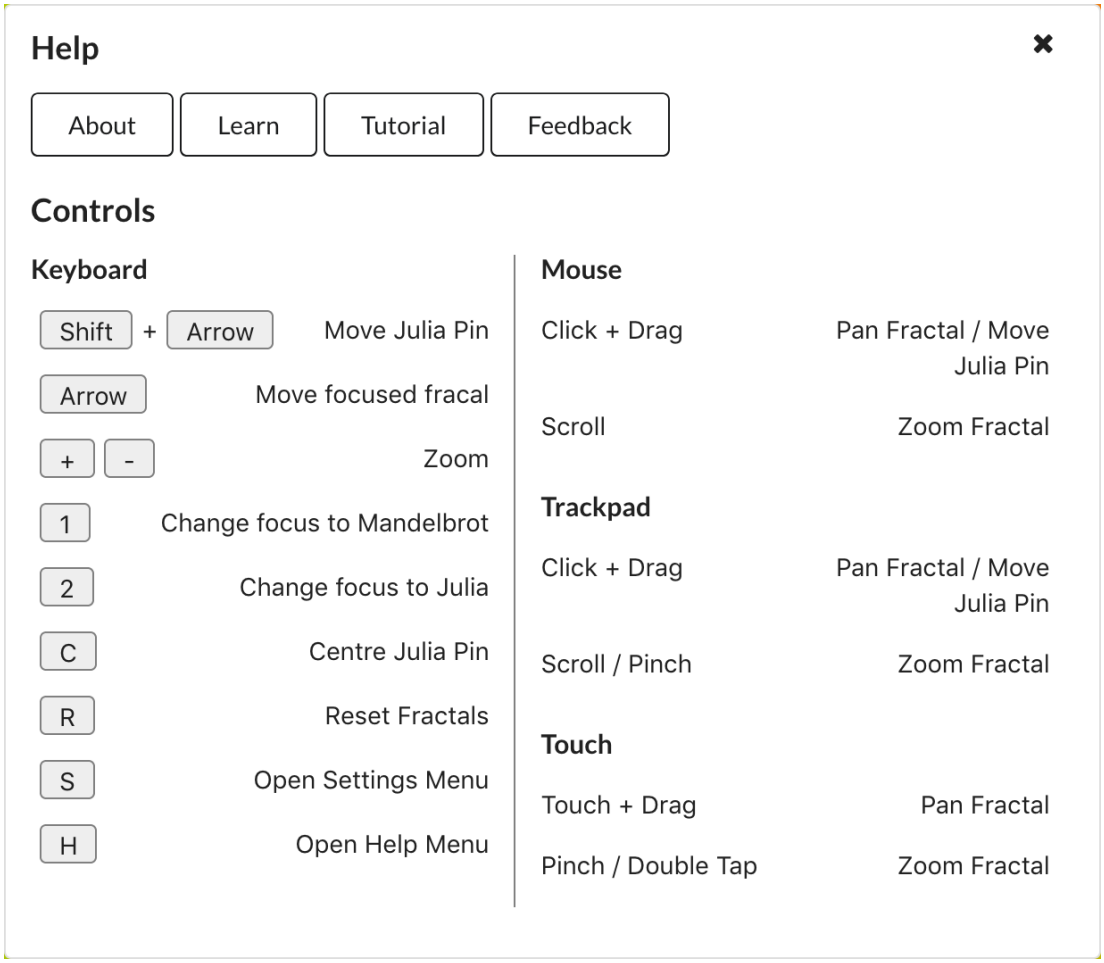


Figure 3.20: In-App Help Box. This pop up provides a quick reference for the application's controls as well as links to the static pages holding the tutorial.

Chapter 4

Evaluation

4.1 Performance

I tested the application using Puppeteer; a high-level API which autonomously controls a browser [33]. The tests used Puppeteer to load the webpage, specify the parameters under test and record performance metrics. Performance metrics were captured using the HTML5 User Timing API which allows web pages to measure the time between events: we measure the time between the start (the triggering action) and end of a render (image being drawn on screen) [34]. To ensure consistency across tests, I locked the screen size at 1280x720 pixels for each test.

Puppeteer is built for use within Chromium browsers, however an experimental package, *puppeteer-firefox*, allows for Firefox browsers to be tested with the Puppeteer API [35]. It is important to test on both these browsers as they use different engines meaning performance metrics will differ.

4.1.1 Impact of number of chunks

As discussed in Section 3.5.2, there is a trade-off between evenly balancing work and the number of calls to the web worker. To explore this, I ran two experiments: one at 1x zoom, and one at the maximum zoom level. During each experiment, I incremented the number of chunks by doubling it, performing 50 renders of the Mandelbrot fractal at each step. To enable this, we bind the ”Q” key to force a render of the Mandelbrot Set; Puppeteer can then automatically press ”Q” repeatedly to trigger renders. I ran both experiments using WebAssembly and Javascript renderers on Firefox and Chrome.

Figure 4.1 shows the result of rendering at 1x zoom. In both browsers, we see a steady decrease in render time for both Javascript and WebAssembly as the number of chunks increases to 4. This is expected as the browsers on the test machine allowed for 4 concurrent web workers, therefore with each increase we are utilising more parallel computation power. We also see an increase in WebAssembly render time as the number of chunks rises past 32. Unlike Javascript, WebAssembly has a high invocation cost: the more chunks, the more WebAssembly is invoked which raises the render time.

At 1x zoom there is no significant change in render time between 4 to 32 chunks, whereas at maximum zoom we see a more pronounced difference. Figure 4.2 shows the render time at the maximum zoom. On Firefox, there is a decrease of 500ms for WebAssembly, however there was no significant difference between Javascript and WebAssembly or across chunk sizes for Chrome at this zoom level.

I chose to use 8 chunks for the application as this provided a good compromise between load balancing and overhead.

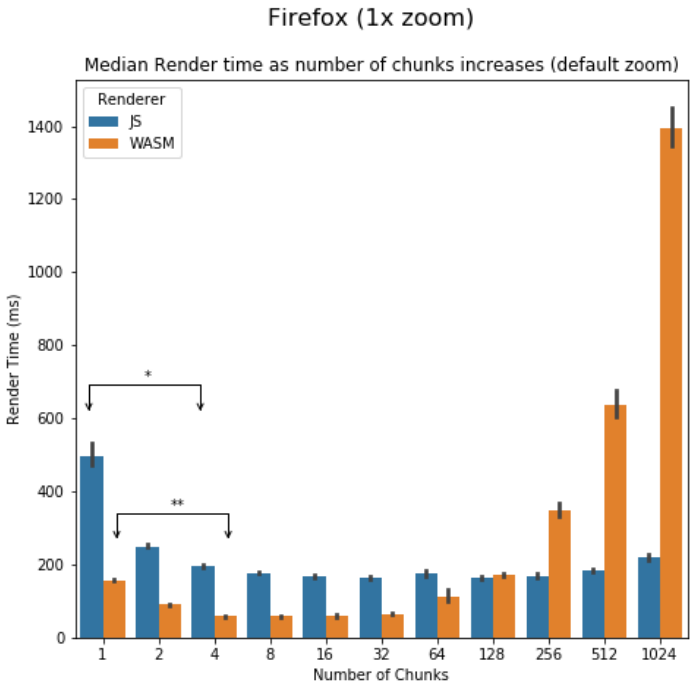
4.1.2 Overall Performance

I set goals for the application's performance based off of how users perceive delay. A delay of 1000ms is thought to be the maximum time a user can maintain flow of thought while waiting [36]. The application will therefore be successful in this regard if it can respond in under one second to user interaction. To evaluate this, I ran tests to capture the overall performance of the application. This involved testing at incremental zoom levels, from 1x to 10^{12}x, zooming by 10 times each step into the point $0+0i$. At each step, the test rendered the Mandelbrot Set 50 times. I again tested on Firefox and Chrome with both WebAssembly and Javascript.

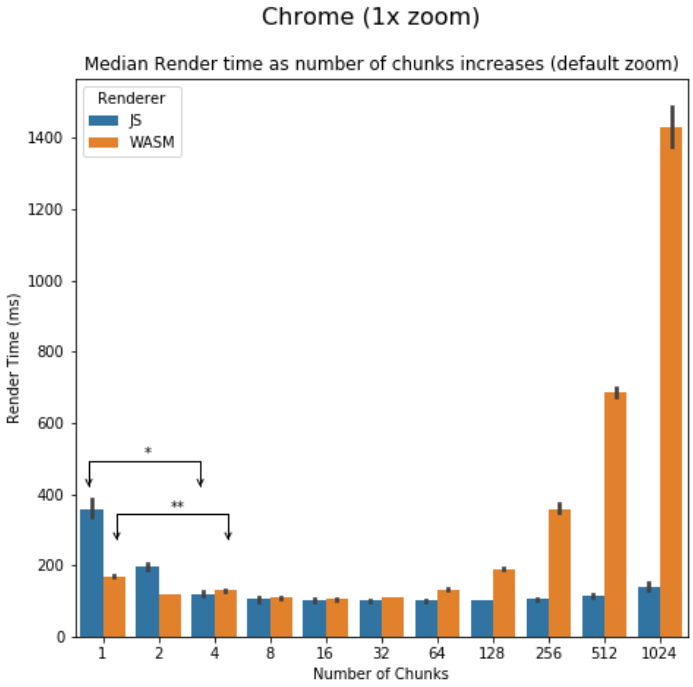
For each zoom level I tested the the worst case - where every pixel reaches the maximum iteration count. I had to modify the renderer to test this as there is no location where all pixels reach the maximum iteration count at 1x and 10x zoom. I achieved this by temporarily removing the termination condition when a point exceeds a magnitude of 2 (Section 2.2.1) forcing each point to iterate to the maximum iteration count. By testing the worst case, we can establish an upper bound for the render time for each zoom level.

Figure 4.3 shows how the renderer performs in these tests. In both cases, we see an increase in render time for both the Javascript and WebAssembly renderer as we zoom deeper into the fractal. This is expected as we increase the iteration count with the zoom level. Firefox shows an improvement in render time when using WebAssembly, whilst Chrome shows no significant difference between Javascript and WebAssembly. The difference in performance is in part a result of the two browsers' WebAssembly and Javascript compilers.

Both Chrome and Firefox manage to stay under the bound of 1000ms as the zoom level rises to 10^7. However, the application fails to maintain this as the zoom level grows further. Rather than limit the user from zooming into these deeper areas, we follow the recommendation of J. Nielson and show a work indicator in the bottom of the screen (Figure 4.4). This gives the user feedback that the viewer is still functioning, making the wait more tolerable [36].

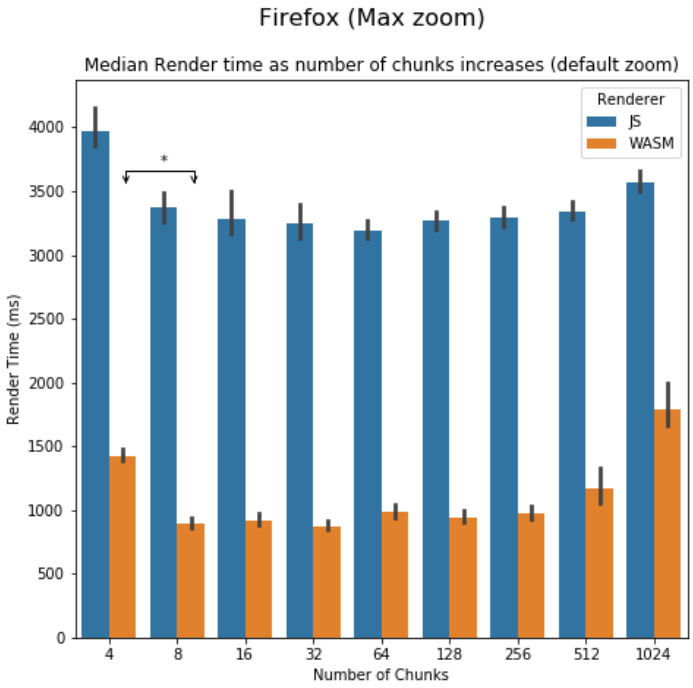


(a) Render times for Firefox 65. We see an average decrease of 300ms and 100ms for Javascript and WebAssembly respectively. *$p < 0.001$, **$p < 0.001$

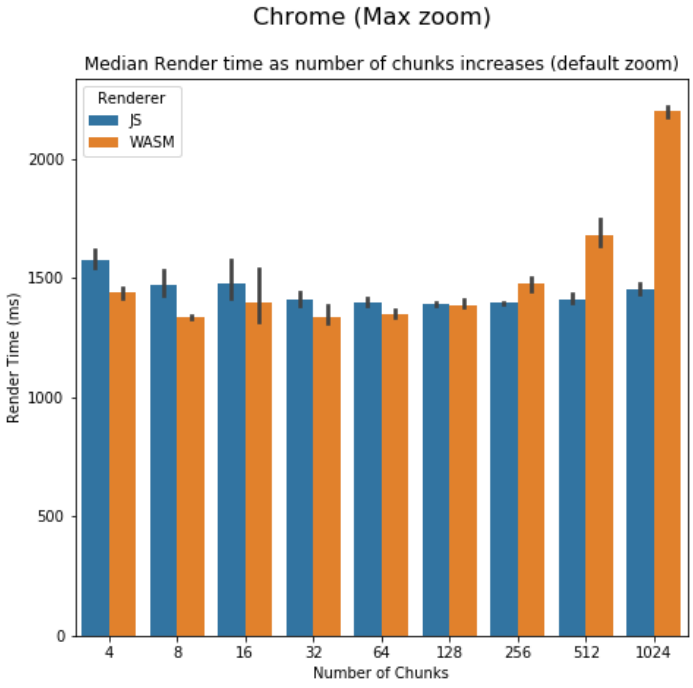


(b) Render times for Chrome 78.0.3882.0. We see an average decrease of 240ms and 60ms for Javascript and WebAssembly respectively. *$p < 0.001$, **$p < 0.001$

Figure 4.1: Results for 1x zoom. Both tests show a significant decrease in render time when the number of chunks increases from 1 to 4. This is due to the most browsers allowing for 4 concurrent web workers, meaning with each new chunk we are able use more parallel computing power. We also see an increase in WebAssembly render time as the number of chunks rises past 32. This is caused by a high invocation cost for WebAssembly, adding overhead to the render.

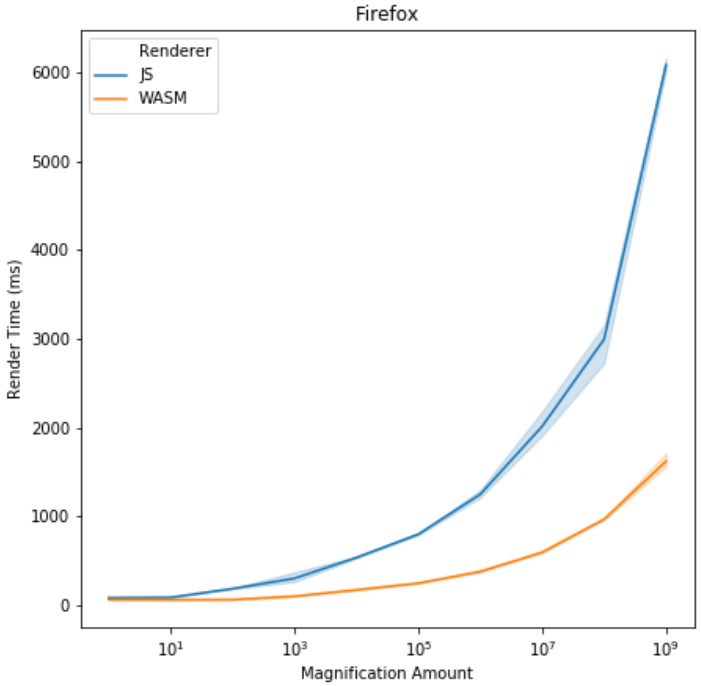


(a) Render times for Firefox 65.0. There is a significant decrease of 500ms when increasing the chunk number from 4 to 8. $^{*}p \simeq 0.00479$

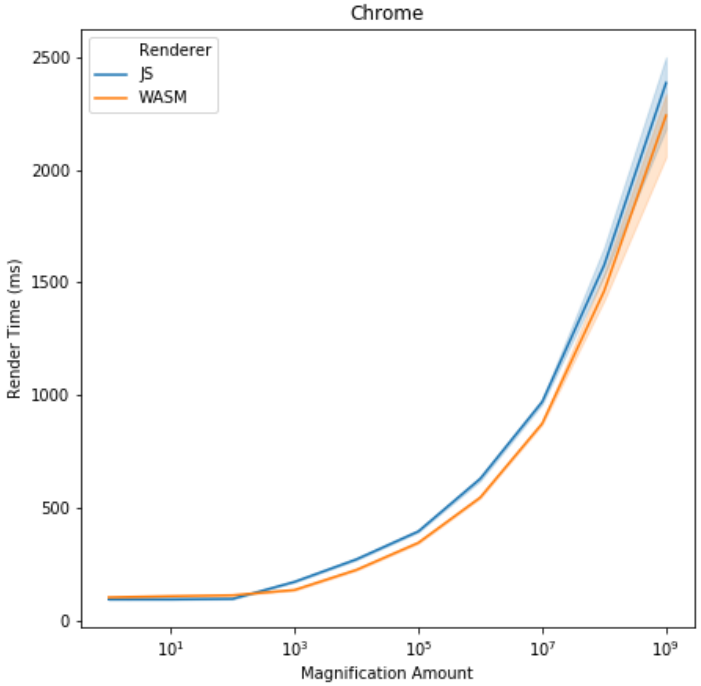


(b) Render times for Chrome 78.0.3882.0. There was no significant decrease in render time when compared against 4 chunks.

Figure 4.2: Results for maximum zoom. Running the test at a deeper zoom level causes differences in render time to be more pronounced due to the increase in iteration count needed to render them.



(a) Render time for Firefox 65. WebAssembly vastly outperforms Javascript at high zoom levels. At 10^9x zoom, WebAssembly is 375% faster than Javascript. $p < 0.001$



(b) Render time for Chrome 78.0.3882.0. There is no significant difference between WebAssembly and Javascript when using Chrome.

Figure 4.3: Render time as magnification amount increases.. As we zoom in, we increase the number of iterations being performed, therefore we see an increase of in render time in both cases. Firefox and Chrome use different engines to execute scripts which perform different optimizations when compiling: this causes the difference in performance profiles.

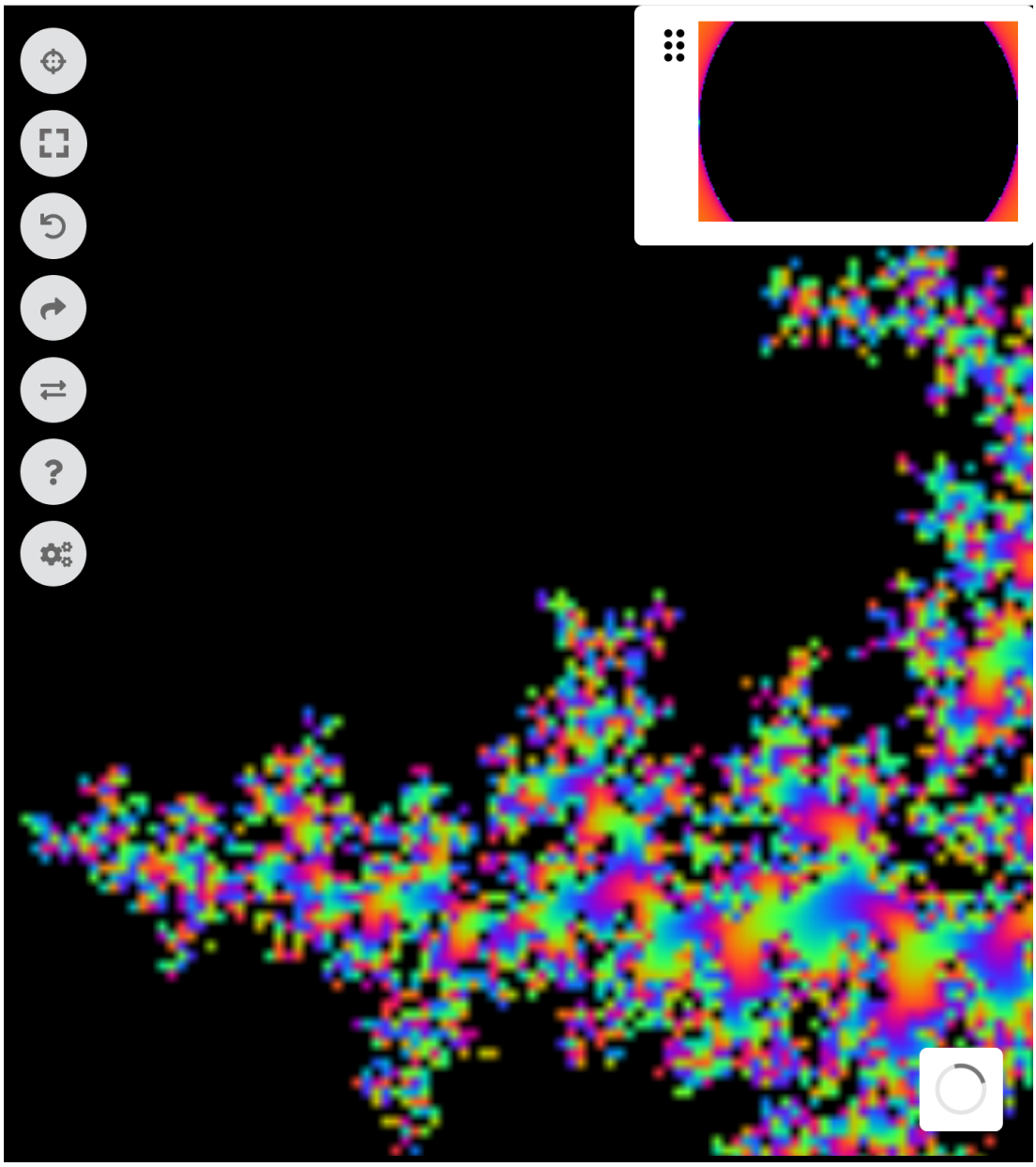


Figure 4.4: Work Indicator (Bottom Right). After a zoom, the renderer may take more than 1000ms to display the updated image. To provide the user with feedback while the system is loading, the application displays a spinner after 1000ms. This component indicates to the user that the application is still running, which makes the wait more tolerable.

4.2 User Experience

I gathered data about the user experience using a survey. The survey was certified according to the Informatics Research Ethics Committee, RT number 2019/17630.

The survey used the System Usability Scale (SUS) to evaluate the application [37]. This is a ten question survey with 5 positively framed questions, (for example, “I thought the system was easy to use”), and 5 negatively framed questions (“I think that I would need the support of a technical person to be able to use this system”). On top of the SUS questions we asked participants to rate their familiarity with the Mandelbrot and Julia sets before using the application, and also asked questions about the application’s performance. The survey is listed in Appendix A.

Participants responded to each question on a Likert scale with 5 options ranging from “Strongly Disagree” to “Strongly Agree”. Each question’s responses were adjusted sit between 0 and 4, where 0 is the worst result and 4 the best for both positive and negative questions (Figure 4.5). Finally, we find the average of each score and sum the results, giving a score out of 40, and then multiply by 2.5 to give a score between 0 and 100. Scores above 70 are deemed acceptable [37].

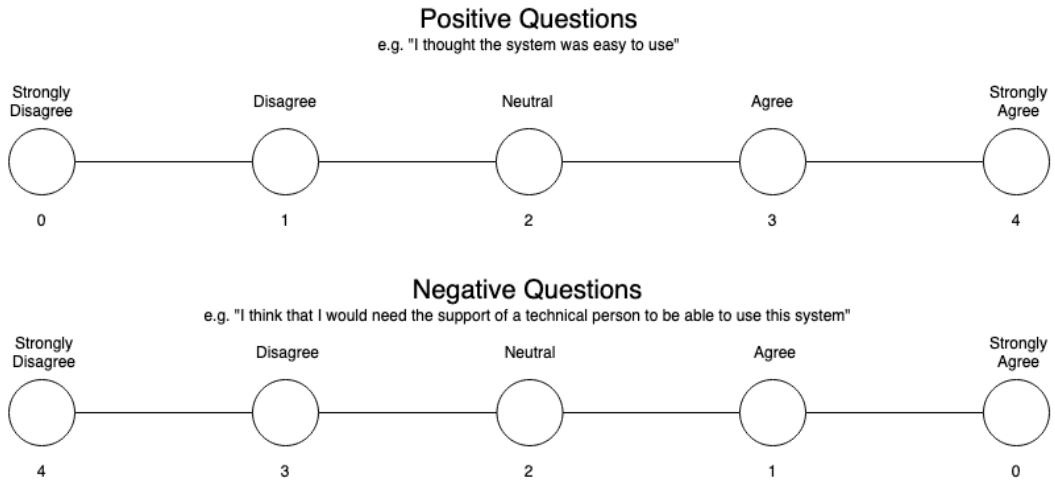


Figure 4.5: The System Usability Scale scoring system. The System Usability Scale adjusts the score depending on whether the question framed in a negative or positive manner. Questions that are positive score higher the closer they are to “Strongly Agree”, while negative questions score higher the closer they are to “Strongly Disagree”.

The survey was distributed amongst student email lists and the project supervisor’s blog, receiving a total of 17 responses. After compiling the responses we found the application’s usability score to be 77.8 out of 100. This is deemed an acceptable score, but indicates there is still room for improvement. There was no significant difference in score between those who said they were familiar with fractals before use and those who were not, showing the application is usable by those without a significant mathematical background.

This survey showed that the application achieved the goal of being accessible. The question “I thought the system was easy to use’; scored a 3.3 out of 4 and “I think that I would need the support of a technical person to be able to use this system” scored a 3.6 out of 4. These results show that users were either able to intuitively use the application or were able to utilise the tutorial to learn the application.

While the survey showed the application succeeded in being accessible, it highlighted other issues. The response to “I found the system was responsive to my input” was scored 2.8 out of 4. This score shows that the system’s performance could be improved further. Use of specialised graphics system, such as WebGL, could solve this. Section 5.1 discusses this further.

There was also a significant difference in responses to the question “I think that I would like to use this system frequently” between those that were familiar with the fractal and those that were not ($p = 0.008$). Users who were familiar averaged a score of 2.5, while those who weren’t averaged 1.14. This shows that users who aren’t familiar with the system view it as a novelty, or that the system is too specialised; options to improve the systems value are discussed in Section 5.2.

Chapter 5

Future Work

5.1 Improving Performance

As mentioned previously (Section 1), this project was completed in parallel with another student; Joao Maio. To avoid overlap in the projects, we agreed to explore different rendering methods. Joao explored rendering using WebGL; a browser based graphics engine. Unlike my WebAssembly and JavaScript renderer, WebGL can use the device's GPU, a highly parallel computation framework, allowing for excellent performance when rendering intensive images. However, WebGL cannot perform deep zooms as it is limited to 23 bit precision [38]; compared to Javascript and WebAssembly's 52 bit precision [39] [40].

A future improvement might be to dynamically switch between a WebGL, and Javascript and WebAssembly renderer. In this system shallow zooms could use the performant WebGL system while deeper zooms use the more precise Javascript and WebAssembly system. This would involve creating a common interface to pass data between both systems, however this may be challenging due to the different ways in which WebGL and Javascript take in data.

5.2 Improving Value

The user evaluation shows that the current application provides a good tool for exploring the Mandelbrot and Julia sets, however lacks value beyond this. This can be improved by expanding the feature set of the application.

5.2.1 Use as an educational tool

The potential for education using the Mandelbrot Maps project has not been explored in previous years. Many online resources exist explaining the fractals but consist mainly of video and text resources. Therefore a future improvement could be to implement an interactive explanation of the fractals. This could include a visualisation of the escape time of a point, or an explanation behind more advanced ideas such as Misiurewicz points (the areas where the Mandelbrot set is self-similar).

5.2.2 Generalize the renderer

The Mandelbrot Set can be generalized to the Multibrot Set, $z \mapsto z^d + c$, where $d \in R$ and $c \in C$. To create the Mandelbrot set, we set $d = 2$ and $c = 0$. Varying d gives unique fractals and could greatly extend the use of the application. This could be taken further by allowing the users to input their own functions, similar to graphing applications such as Desmos (Figure 5.1). This would allow the user to explore other fractal systems such as the "Bugbrot" (Figure 5.2a) or "Simonbrot" sets (Figure 5.2b).

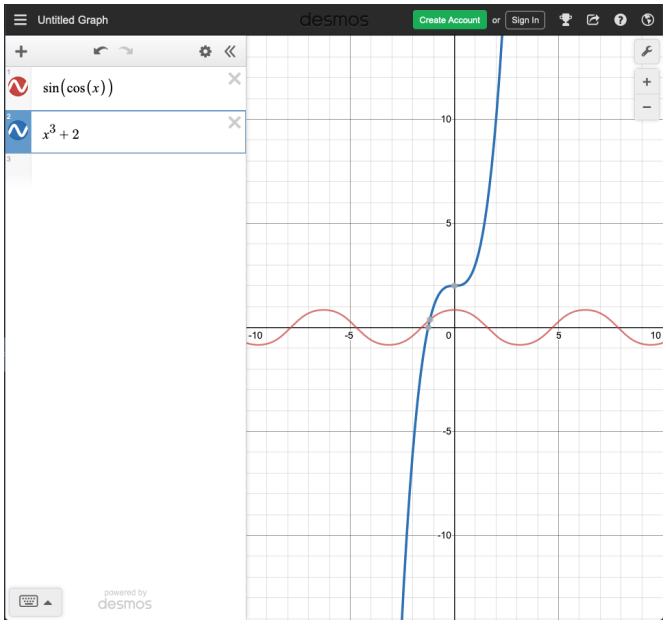
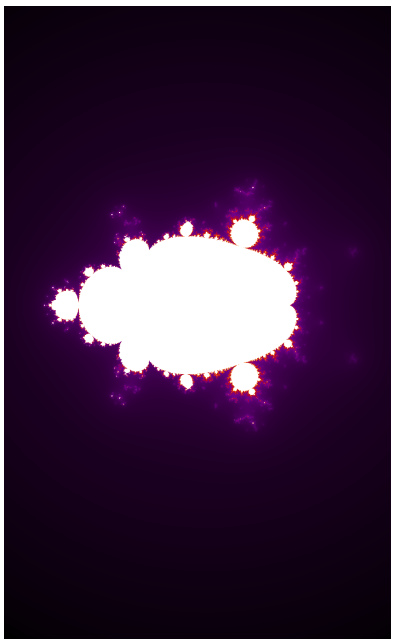


Figure 5.1: Desmos Interface. In Desmos, users can visualise graphs by entering their own formulae [41]. This allows for the application to be extremely flexible as it is not restricted by a set of predetermined visualisations. The current application could be extended to allow the user to input their own fractal functions, greatly increase the utility of the app.

5.2.3 More Rendering Options

We can also increase the value of the application by giving the users more variety in the methods we use to render fractals. The application currently offer three static color options, limiting the potential of the application's visuals. A future extension could see the color options being extended to allow users to customize the palette used. Another extension could see the introduction of other render methods.

The application uses escape time to color the fractal, however many other methods exist. The Triangle Inequality Average, for example yields stripes extruding from the fractal and gives a different insight into the sets [31] (Figure 5.3). The Buddabrot uses a probability distribution to color fractals which adds details into the flat areas of the image (Figure 5.4) [43]. Including these alternative rendering methods will add value by allowing the user to gain different insights into the fractal.



(a) Bugbrot. This image is created using the function $f(z) = ((z^2 + c - 1)(2z + c - 2))^2$

(b) Simonbrot Set. This image is created using the function $f(z) = z^2 \cdot |z|^2 + c$

Figure 5.2: Alternative Fractals. By allowing the user to input their own formulas into the rendering engine the application can show many other fractals apart from Multibrot sets. (Images generated by MandelBrowser app [42])

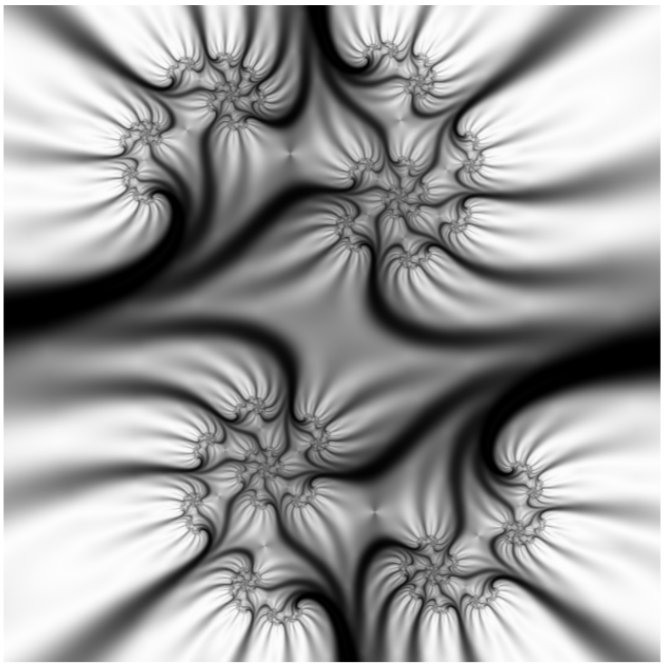


Figure 5.3: The Triangle Inequality Average. This method produces lines extruding from the main fractal. It is created by derving constants from the triangle inequality; two shorter sides of a triangle cannot exceed the longest. Image taken from [31].

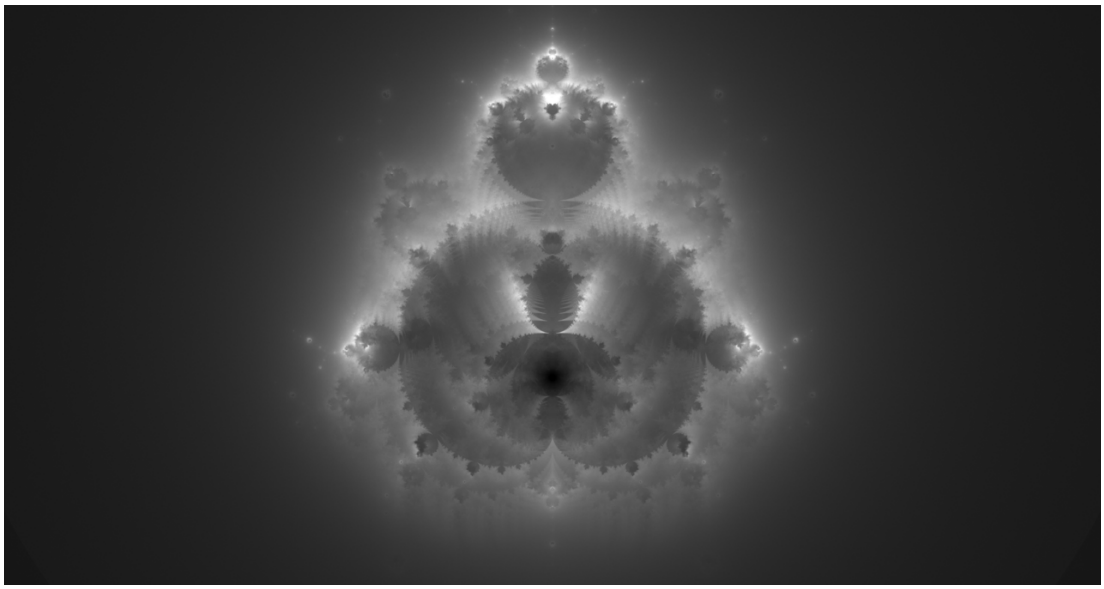


Figure 5.4: Buddhabrot Fractal. This method adds greater detail to the fractal by using a probability distribution of the trajectories of points under iteration. It adds detail to previously flat areas giving a unique insight. (Image created by UnreifeKirsche (Public Domain) [44])

Chapter 6

Conclusion

I updated the Mandelbrot Maps project to run on the web by making use of modern technologies such as WebAssembly and React. Through web workers I was able to significantly decrease the render times in the application. The user interface implements controls for mouse, multi-touch trackpad, touch and keyboard to allow for users on many different devices to use the application intuitively.

I used the System Usability Scale to gain insights from surveys into the user experience and showed that the application exceeded the industry benchmark of 70 out of 100, achieving 77.8. From this survey, I also showed that the system was easy to learn regardless of the user's prior familiarity with the fractals. As well as evaluating user experience, I tested the performance of the application using Puppeteer and saw that the application maintained acceptable performance up to 10,000,000x magnification.

Through user surveys and automated testing I have shown that we succeeded in rebuilding the Mandelbrot Maps project to create a responsive cross-device experience.

The updated version of Mandelbrot Maps presented is available to try at `http://mmaps.freddiejbawden.com`. The source code is hosted at `https://github.com/freddiejbawden/mandelbrotmaps`

Appendix A

Feedback Form

The feedback form contained 12 questions. Questions 1 through 6 ask that the particpants understand their rights as part of the user study. Questions 8 through 17 are standard to the system usability scale test, Question 7 was used to divide users by their knowledge of the fractals prior to use. Question 18 was asked with regards to the system's performances and question 19 asked for general feedback.

Mandelbrot Maps Feedback Form

Please read the information sheet available here before beginning this survey: https://github.com/freddiejbawden/mandelbrotmaps/blob/develop/information_sheet_form.pdf

* Required

Consent

1. I confirm that I have read and understood the Participant Information Sheet for the stud that I have had the opportunity to ask questions, and that any questions I had were answered to my satisfaction *

   Mark only one oval.

   - ◯ Yes
   - ◯ No

2. I understand that my participation is voluntary, and that I can withdraw at any time witho giving a reason. Withdrawing will not affect any of my rights *

   Mark only one oval.

   - ◯ Yes
   - ◯ No

3. I consent to my anonymised data being used in academic publications and presentation

   Mark only one oval.

   - ◯ Yes
   - ◯ No

4. I understand that my anonymised data can be stored fora minimum of two years *

   Mark only one oval.

   - ◯ Yes
   - ◯ No

Bibliography

[1] Parris I. “Fractals don’t have to be scary.” Mandelbrot Maps: Creating a real-time Mandelbrot/Julia fractal explorer [disseration]. University Of Edinburgh; 2009.

[2] Mandelbrot Maps. Alasdair Corbett;. Available from: https://play.google.com/store/apps/details?id=uk.ac.ed.inf.mandelbrotmap.

[3] Java and Firefox Browser. Oracle; [cited January 2020]. Available from: https://java.com/en/download/faq/firefox_java.xml.

[4] Java and Chrome Browser. Oracle; [cited January 2020]. Available from: https://java.com/en/download/faq/chrome.xml.

[5] Java and Safari Browser. Oracle; [cited January 2020]. Available from: https://java.com/en/download/faq/safari.xml.

[6] Windows 10 and Java. Oracle; [cited January 2020]. Available from: https://www.java.com/en/download/faq/win10_faq.xml.

[7] Desktop vs Mobile vs Tablet Market Share Worldwide. StatCounter Global Stats; [updated December 2020; cited January 2020]. Available from: https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet/worldwide/#monthly-200901-201912.

[8] Mandelbrot Maps - Brief Background; 2008. Available from: https://www.youtube.com/watch?v=4AMiUPkapy0.

[9] Yan A, Merenkov S. Asymptotic counting in dynamical systems; 2018. .

[10] Intensive Javascript. Mozilla Foundation; [updated March 2019; cited January 2020]. Available from: https://developer.mozilla.org/en-US/docs/Tools/Performance/Scenarios/Intensive_JavaScript.

[11] Web Workers. HTML Standard. 2020 Apr [cited April 2020];Available from: https://html.spec.whatwg.org/multipage/workers.html#workers.

[12] Portability: Assumptions for Efficient Execution. WebAssembly CG; [cited January 2020]. Available from: https://webassembly.org/docs/portability/#assumptions-for-efficient-execution.

[13] Rust. Rust Programming Language [cited April 2020];Available from: https://www.rust-lang.org/.

[14] Rust and WebAssembly. Rust and WebAssembly [cited April 2020];Available from: https://rustwasm.github.io/.

[15] The wasm-bindgen Guide. The ‘wasm-bindgen‘ Guide [cited April 2020];Available from: https://rustwasm.github.io/docs/wasm-bindgen/.

[16] React – A JavaScript library for building user interfaces. – A JavaScript library for building user interfaces. 2020 [cited April 2020];Available from: https://reactjs.org/.

[17] React Source Code. Facebook; 2020 [cited April 2020]. Available from: https://github.com/facebook/react.

[18] Wäljas M, Segerståhl K, Väänänen-Vainio-Mattila K, Oinas-Kukkonen H. Cross-platform service user experience. Proceedings of the 12th international conference on Human computer interaction with mobile devices and services - MobileHCI 10. 2010 Jan;.

[19] OpenStreetMap. OpenStreetMap. 2020 [cited Feburary 2020];Available from: https://www.openstreetmap.org/about.

[20] Google Earth. Google Earth. 2020 [cited Feburary 2020];Available from: https://www.google.co.uk/intl/en_uk/earth/.

[21] Semantic UI 2.4.2. Semantic UI. 2020 [cited Feburary 2020];Available from: https://semantic-ui.com/.

[22] React Router. ReactTraining; [cited March 2020]. Available from: https://github.com/ReactTraining/react-router.

[23] Spyna L. Manage React State Without Redux. ITNEXT by LINKIT; [updated February 2019; cited December 2019]. Available from: https://itnext.io/manage-react-state-without-redux-a1d03403d360.

[24] Keyboard Compatibility. Web Accessibility Initiative (WAI); 2020 [cited April 2020]. Available from: https://www.w3.org/WAI/perspective-videos/keyboard/.

[25] Understanding WCAG 2.1 - Understanding Success Criterion 2.5.1: Pointer Gestures. Understanding Success Criterion 251: Pointer Gestures. 2018 [cited Feburary 2020];Available from: https://www.w3.org/WAI/WCAG21/Understanding/pointer-gestures.html.

[26] Gestures. Material Design [cited March 2020];Available from: "https://material.io/design/interaction/gestures.html#types-of-gestures".

[27] GRPH3B18. Gestures Pinch; 2011. Available from: https://commons.wikimedia.org/wiki/File:Gestures_Pinch.png.

[28] GRPH3B18. Gestures Unpinch; 2011. Available from: https://commons.wikimedia.org/wiki/File:Gestures_Unpinch.png.

[29] Shavit MHN. Introduction. In: The Art of Multiprocessor Programming, Revised Reprint. Morgan Kaufmann Publishers; 2012. p. 14.

[30] Molnar S, Cox M, Ellsworth D, Fuchs H. A sorting classification of parallel rendering. ACM SIGGRAPH ASIA 2008 courses on - SIGGRAPH Asia 08. 1994 Jul;.

[31] On Smooth Fractal Coloring Techniques [thesis]. Abo Akademi University; 2007.

[32] Mandelbrot Maps - A Quick Start Guide; 2009. Available from: https://www.youtube.com/watch?v=ozC9GBH2i5M.

[33] Puppeteer. Tools for Web Developers [cited March 2020];Available from: https://developers.google.com/web/tools/puppeteer.

[34] User Timing API. MDN Web Docs. 2020 [cited April 2020];Available from: https://developer.mozilla.org/en-US/docs/Web/API/User_Timing_API.

[35] Puppeteer. Puppeteer. 2020 Mar [cited March 2020];Available from: https://github.com/puppeteer/puppeteer/tree/master/experimental/puppeteer-firefox.

[36] Nielsen J. In: Usability engineering. Morgan Kaufmann; 2009. p. 135.

[37] Tullis T, Albert W. In: Measuring the user experience: collecting, analyzing, and presenting usability metrics. Morgan Kaufmann; 2016. p. 137–140.

[38] WebGL best practices. MDN Web Docs. 2020 [cited April 2020];Available from: https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/WebGL_best_practices.

[39] Number. MDN Web Docs. 2020 [cited April 2020];Available from: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number.

[40] Types. Types - WebAssembly 10. 2017 [cited April 2020];Available from: https://webassembly.github.io/spec/core/syntax/types.html#syntax-valtype.

[41] Desmos. Desmos. 2020 [cited April 2020];Available from: https://www.desmos.com/.

[42] Śmigielski T. MandelBrowser. Google Play. 2019 JulAvailable from: https://play.google.com/store/apps/details?id=pl.y0.mandelbrotbrowser&hl=en_GB.

[43] Green M. The Buddhabrot Technique. Buddhabrot fractal method [cited March 2020];Available from: http://superliminal.com/fractals/bbrot/bbrot.htm.

[44] UnreifeKirsche. Buddhabrot Image. Wikimedia Commons. 2008 Nov [cited April 2020];Available from: https://commons.wikimedia.org/wiki/File:Buddhabrot-100I-2000.png.