



UNIVERSITY OF  
LIVERPOOL

COMP39X

2019/20

## Rendering Complex Polynomial Fractals in Real-Time

Student Name: Thomas Moscrip

Student ID: 200853225

Supervisor Name: Paul Dunne

DEPARTMENT OF COMPUTER SCIENCE

University of Liverpool  
Liverpool L69 3BX





UNIVERSITY OF  
LIVERPOOL

COMP39X

2019/20

Rendering Complex Polynomial Fractals in Real-Time

DEPARTMENT OF COMPUTER SCIENCE

# Contents

<b>1</b>	<b>Abstract</b>	<b>1</b>
<b>2</b>	<b>Introduction</b>	<b>1</b>
<b>3</b>	<b>Aims &amp; Objectives</b>	<b>1</b>
<b>4</b>	<b>Julia and Fatou sets</b>	<b>2</b>
4.1	The Pixel Game . . . . .	2
4.2	Normalising Escape Time . . . . .	3
4.3	Polynomial Formulae . . . . .	3
<b>5</b>	<b>Design</b>	<b>4</b>
5.1	Final Design . . . . .	4
5.2	Changes From Original Specification . . . . .	5
5.3	Tools & Libraries . . . . .	6
5.3.1	WebGL . . . . .	6
5.3.2	React . . . . .	6
5.3.3	Chrome Developer Tools . . . . .	6
5.3.4	Git . . . . .	6
5.4	User Interface . . . . .	6
<b>6</b>	<b>Implementation</b>	<b>9</b>
6.1	OpenGL Pipeline . . . . .	9
6.2	Reusable & Modular JavaScript with React . . . . .	11
6.3	State Management . . . . .	15
6.4	The Julia/Fatou Fragment Shader . . . . .	16
<b>7</b>	<b>Testing, Feedback &amp; Evaluation</b>	<b>20</b>
7.1	Automated Testing . . . . .	20
7.2	Manual Testing . . . . .	20
7.3	User Evaluation . . . . .	21
<b>8</b>	<b>Improvements</b>	<b>23</b>
8.1	GUI . . . . .	23
8.2	State Management . . . . .	24
8.3	Planning . . . . .	24
8.4	Testing . . . . .	25

8.5	Research . . . . .	25
8.6	Complimentary Programming Paradigms . . . . .	25
<b>9</b>	<b>Extensions</b>	<b>26</b>
9.1	Quaternions and Julia Sets in Higher Dimensions . . . . .	26
9.2	Complex Iterative Formula . . . . .	26
9.3	Animation with Lissajous Curves . . . . .	27
9.4	Newton Fractals . . . . .	28
9.5	Mandelbrot Set . . . . .	28
<b>10</b>	<b>BCS Project Criteria</b>	<b>29</b>
<b>11</b>	<b>Self-Reflection</b>	<b>29</b>
<b>12</b>	<b>Conclusion</b>	<b>30</b>
<b>A</b>	<b>React Components</b>	<b>34</b>

# 1 Abstract

This project explores methods of creating visually appealing and distinct visualisations of fractal shapes through the repeated iteration of complex quadratic polynomials. Two complimentary sets are generated by this process, discovered by Julia, 1918 and Fatou, 1917. While there is little of particular note about this mathematics behind this process, computers grant the capability to visualize the sets produced and explore the beauty of their recursive, self-similar nature that was not possible for decades after their discovery.

Using the parallel processing powers of graphics processing units (GPUs) utilised within a web browser through the WebGL API, imagery is generated in real-time. In this report, the process involved in using the API is examined in addition to implementing a browser-based GUI to interface with it and provide parameters influencing the generated visuals.

## 2 Introduction

Two motivations that influenced me to undertake this project over others available are my interest in creative coding, and an opportunity to work with a number of specific technologies I had prior interest in.

Those who may find the application useful are teachers and students as a demonstration of the mathematic field of complex analysis, in addition to visual artists as a resource for generating graphics, as a framework for their own graphical applications or as inspiration from the beauty of fractal imagery.

## 3 Aims & Objectives

1. Explore modern web development technologies & patterns. I have chosen to focus on the React library and WebGL API.
2. Design and implement a responsive using appropriate HTML and flexible CSS.
3. Build a highly modular and reusable application. The framework I aim to create is relevant to projects I will undertake in the future, and potentially others given the popularity of similar applications. The finished project, along with the source code and documentation, will be made freely available shortly after its conclusion.
4. Develop specific technical skills for my career. React is the most in-demand front-end web library (freeCodeCamp, 2019). WebGL and similar APIs allow unique graphics-heavy experiences to be delivered through web browsers. I hope that by gaining experience with these, my employment opportunities will be enhanced.
5. Develop non-technical skills related to development including planning, documentation, testing and time management. I believe that these areas are lagging behind my technical skills and wish to pay extra attention to them to balance my personal skill set.

## 4 Julia and Fatou sets

The Julia and Fatou sets are constructed through classification of points on the complex plane under iteration, as described by French mathematicians Julia, 1918 and Fatou, 1917. For all points  $z$  on the complex plane, a constant point  $C$  and a polynomial function  $f^{0 \rightarrow \infty}(z)$ , there are two sets we can place  $z$  into:

1. The magnitude of  $z_\infty$  remains bounded
2. The magnitude of  $z_\infty$  tends towards  $\infty$

Points which fall into the first category are part of the Julia set, those in the latter make up the complimentary Fatou set. While this project does not delve deeply into the mathematics and theory behind these sets which resides in the realms of complex analysis, it instead focuses on creating visualisations through computational methods. A thorough investigation of the theory behind the complex analysis of these unique sets is given by Alligood et al., 1997.

### 4.1 The Pixel Game

Computers struggle with the notion of infinity, which occurs twice in the described process prior. To address this, a limit is placed on the maximum number of iterations (MI) and an "escape radius" (ER) is defined, past which a point is assumed to tend towards  $\infty$ . The classification becomes:

1.  $z$  belongs to the Julia set if its magnitude does not exceed ER at any number of iterations up to MI
2. Otherwise,  $z$  belongs to the Fatou set

This computational approach has been named the "Pixel Game" (Peitgen, Jürgens, and Saupe, 1992). Renders of Julia/Fatou sets through this method usually depict the Julia set as a solid colour and the Fatou set as range of colours. Using the iteration count when a point passes the escape radius, it is assigned an "escape time" (ET) in the range  $[0, 1]$  through:  $ET = \frac{iterations}{MI}$ . Figure 1 shows a comparison of the Julia/Fatou sets for the iterative function  $f^{0 \rightarrow 50}(z^2 + c)$ ,  $ER = 4$ .  $ET = 0$  is mapped to white,  $ET = 1$  to black.

Algorithm 1 computes the escape time for the quadratic form. This approach is expanded upon in the implementation section.

---

**Algorithm 1** The Pixel Game

---

```
1: procedure PIXEL-GAME( $z, C, escapeRadius, maxIter$ )
2:   iters  $\leftarrow$  0
3:   while iters < maxIter & mag( $z$ ) < escapeRadius do
4:      $z \leftarrow z^2 + C$ 
5:     iters  $\leftarrow$  iters + 1
6:   end while
7:   return iters/maxIter
8: end procedure
```

---

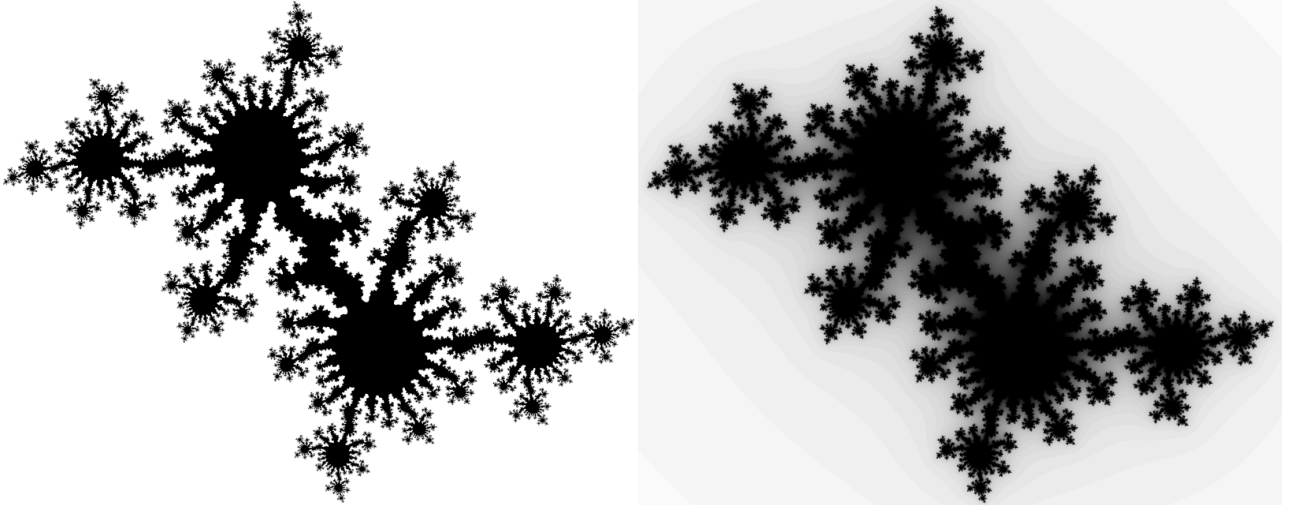


Figure 1: Rendered Julia set (left) and combined Julia/Fatou sets (right)

## 4.2 Normalising Escape Time

The iteration count determining the colour of a pixel in the Fatou set is returned as an integer. When mapped into the range  $[0, 1]$ , the value jumps up in discrete intervals. The result of this is an unpleasing banded appearance noticeable in Figure 1. The pure mathematical analysis of these sets found in complex analysis theory does not suffer from such restrictions.

To circumvent this restriction of integers, a fractional component can be introduced to the iteration count, obtained by analysing how far past the escape radius a point goes upon passing. This fractional component in the range  $(0, 1)$ , where points close to the previous band are near lower limit and points close the next band approach the upper limit, can be obtained from the following formula (Quilez, 2007):

$$normalised(n) = n - \frac{\ln \frac{\ln |z_n|}{\ln ER}}{\ln D}$$

where  $n$  is the iteration count,  $z_n$  is the value of  $z$  after it passes the escape radius,  $ER$  is the escape radius and  $D$  is the degree of the iterative polynomial. Following this procedure, the escape time is now calculated through  $ET = \frac{normalised(iterations)}{MI}$ . Figure 2 demonstrates the visual difference produced by this process (observe the "banded" effect on the left).

## 4.3 Polynomial Formulae

As there are an infinite set of values that  $C$  can take there are an infinite set of Julia sets that can be obtained from iterating the quadratic form  $z = z^2 + C$ , however more visually interesting results can be found in the infinite set of polynomials that may be used in the iterative formula.

The application offers the possibility for polynomials of any degree with arbitrary coefficients. Similar to how quadratic Julia sets display twofold symmetry, the iterative formula of  $z = z^5 + C$  shown in Figure 3 shows fivefold symmetry. Figure 4 for the formula  $z = z^4 + z^3 + z^2 + C$ , while not symmetrical across any set of lines, retains the property of scaling self-similarity.



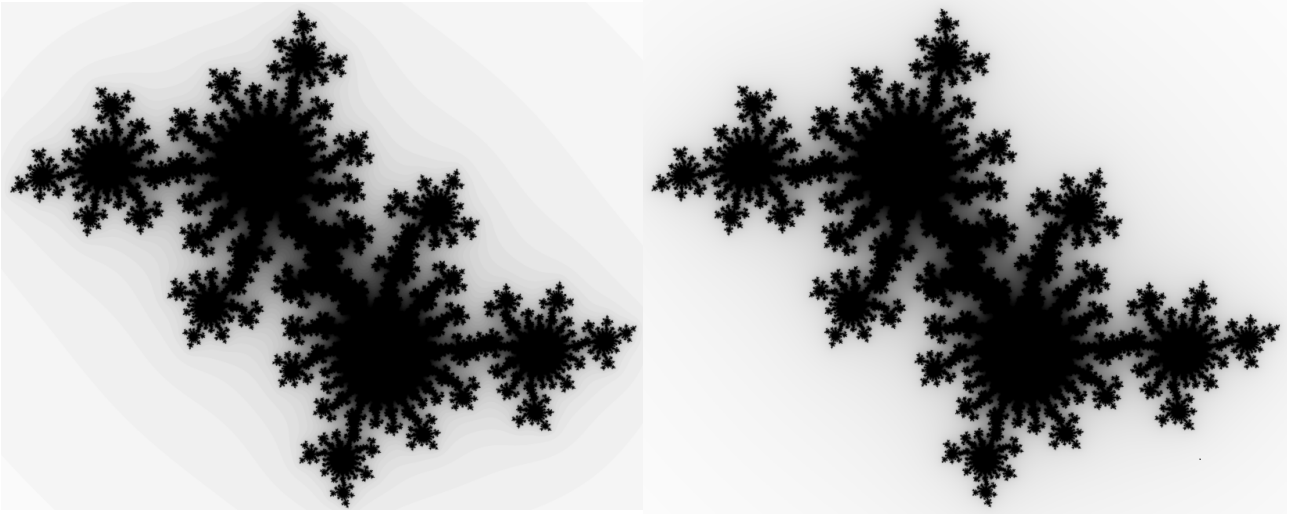


Figure 2: Difference between unnormalised (left) and normalised (right) Fatou sets

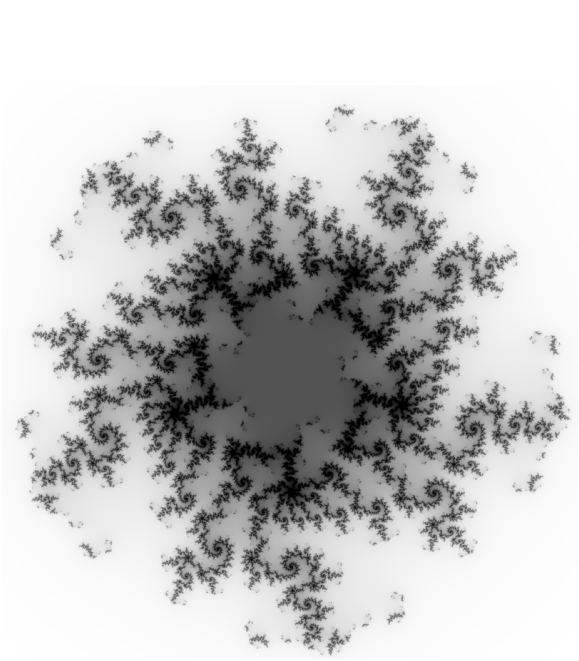


Figure 3: Julia/Fatou sets of the fifth degree

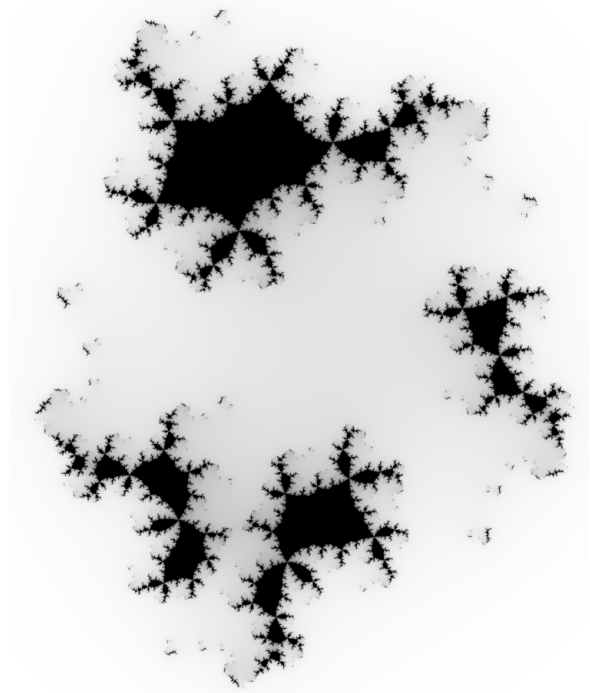


Figure 4: Polynomial Julia/Fatou sets

Further possibilities for related sets that are not implemented in this project are discussed, in addition in the Extensions section.

## 5 Design

### 5.1 Final Design

Figure 5 illustrates the flow of data within the application.

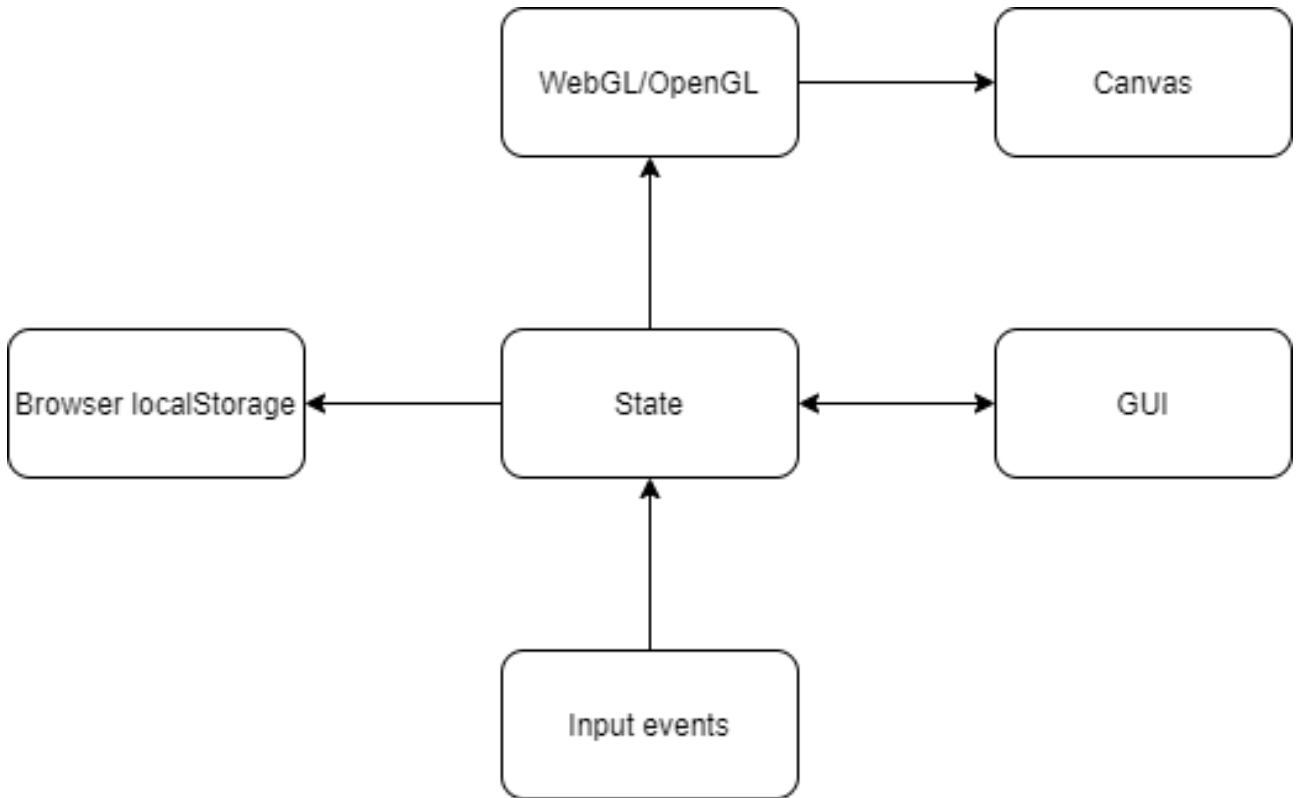


Figure 5: Flow of data within the application

## 5.2 Changes From Original Specification

The project has changed in numerous ways since the initial design specification. Features were dropped to allow more time to be spent on others which contributed more towards achieving the outlined goals & objectives.

Two interfaces were included in the original specification, one optimised for desktops and the other for smaller screens e.g. mobile devices. Instead, a single interface compatible across a variety of common screen sizes was implemented.

Functionality for saving and loading snapshots of application state was originally planned to be implemented through a back-end server and database. The front-end webpage would have sent requests to:

1. Save current state and receive a response with an identifier key for the database entry
2. Load previously saved state by sending an identifier key

Similar functionality has been replaced with a entirely client-side implementation. State can be saved/restored from the "localStorage" API available in all modern web browsers (Mozilla, 2019b). This alone did not fully capture the original functionality of sharing state outside of a browser instance. In response, the ability to import/export state as a base64-encoded serialisation was implemented.

A secondary canvas displaying additional information that can be obtained from the iteration process from running the calculations on the CPU was planned, for example the orbit points of the value at the cursor's position. Initial testing resulted in performance issues due to

computationally expensive operations being called very frequently, especially when the cursor was being moved. The feature was reassessed and the conclusion that it contributed little to the objectives which focus more on the visual representation than mathematical properties of the fractals.

## 5.3 Tools & Libraries

Language and terminology specific to the tools used to create the application is used throughout the coming sections. Please refer to the glossary for an explanation of these terms and phrases.

### 5.3.1 WebGL

WebGL is a JavaScript API for rendering high-performance interactive 3D and 2D graphics within compatible web browsers by providing bindings to the OpenGL rendering library (Mozilla, 2019a). Due to the computationally-intense nature of generating Julia/Fatou sets in real-time, utilising this API has been essential to realising the project and helped meet objectives 1 and 4.

### 5.3.2 React

React is a declarative, component-based JavaScript library for building user interfaces (Facebook Inc., 2020d). This library has been included to help meet objectives 1, 2, 3 and 4. While web applications of any size can be developed using many approaches, modern libraries introduce abstractions that greatly increase the modularity and reusability of the traditional web technologies of HTML, CSS and JavaScript.

### 5.3.3 Chrome Developer Tools

The Chrome web browser has a section of tools dedicated to debugging and inspecting elements of web applications. The "React Developer Tools" extension introduces tools similar to the one Chrome provides to inspect HTML for React components.

### 5.3.4 Git

Git functions as version control software, allowing snapshots of files and directories to be saved and restored. Using version control software grants peace of mind while iterating on code, as any changes which end up being undesirable can be easily rolled back.

## 5.4 User Interface

Table 1 specifies the elements of the user interface, how the user interacts with them, and their purpose. Each of these elements have corresponding variables within the application's state to manage their values.

Component	Type	Purpose
Pause	Button	Pause/resume the incrementation of the time counter
Toggle menu	Button	Collapse/expand the menu
Show help	Button	Toggle help menu visibility
Help menu	Modal box	Provide usage information
Julia coefficients	String input	Construct iteration formula
Constant point	2x string inputs	Define "C" value for iteration formula
Max iterations	String input	Define maximum iteration count
Escape radius	String input	Define escape radius
Julia smoothing	Checkbox	Toggle iteration count normalisation
Anti-aliasing	Drop-down selector	Select level of multisample anti-aliasing
Time scale	String input	Scale the incrementation of the time counter
Viewport dimensions	2x string inputs	Define width and height of complex plane
Translate	2x string inputs	Define complex plane offset from origin
Lock aspect ratio	Checkbox	Toggle whether viewport transformations scale to aspect ratio
Colour mapping field (colour)	Colour selector	Select a colour for the mapping point
Colour mapping field (position)	String input	Select a position for the mapping point
Mapping curve	Drop-down selector	Control the curve used for interpolating between colour points
Colour model	Drop-down selector	Control the colour model used for interpolating between colour points
Presets	Drop-down selector	Select an existing preset to load
Preset name	String input	Enter a name to save the current state as
Save	Button	Save the current state as value in "Preset name"
Load	Button	Load the selected preset into state
Import	Button	Imports an encoded serialisation of state from the clipboard
Export	Button	Copies an encoded serialisation of the current state to the clipboard
Encoded state	String field	Allows access to clipboard for exporting/importing state

Table 1: User interface components

Figure 6 displays the final GUI. The entire menu is displayed vertically and has its own scrollbar if it happens to go off-screen, but has been cut and placed side-by-side in this image. Related variables have been grouped into nestable folders that can be collapsed and expanded independently by clicking on the group title.



Figure 6: GUI in the final application

Accessibility concerns have been taken into account when designing the GUI. To ensure text is always readable, highly contrasting text colours have been chosen. The first iteration of the GUI featured a semi-transparent background. The issue of colours on the rendering canvas interacting with this leading to difficulty reading text was identified and the background was adjusted to a constant opaque colour.

Element dimensions have been specified in relative "rem" units rather than absolute units e.g. "px", "cm". "rem" units are relative to the font-size of the page's root element W3Schools, 2020. When the user increases the root font-size by changing their browser's default font-size or zooming on the individual page, the width, height and font-size of all GUI elements scale accordingly.

The user can access a dedicated help panel through a prominently placed button in addition to help tooltips being placed next to every GUI element. Since users may not have experience with the terminology used in the GUI, these tooltips are critical for ensuring a high level of usability.

The GUI was heavily inspired by that used in the particles.js demonstration page (Vincent

Garreau, 2017). It was later discovered that this site's interface was provided by a library named dat.GUI (Google Data Arts Team, 2011). The decision to continue implementing the GUI from scratch was made to meet objectives 1, 2, 3 and 4. There was also additional risk involved with bringing in an external GUI library, as the existing one was mostly complete at the time and the challenges that may have been associated with linking the application's state management system to an external library with unknown.

## 6 Implementation

### 6.1 OpenGL Pipeline

OpenGL is a low-level API, requiring numerous steps to achieve any output. To utilise OpenGL, the steps of the "programmable pipeline" (The Khronos Group, 2020) must be implemented. Figure 7 and Table 2 together illustrate this process, which is run by the application to draw each frame.

Operation	Description
Canvas available?	Ensure DOM canvas is present and can accept WebGL rendering contexts
Send browser alert	Trigger a pop-up alerting the user that their browser is incompatible
Build fragment shader code	Insert values from application state and concatenate strings to build GLSL fragment shader code
Bind vertices	Create buffer for storage vertices
Bind vertex indices	Create buffer for storing indices linking vertices to shape primitives
Build vertex shader code	Concatenate strings to build GLSL vertex shader code
Compile shaders	Compile shader source codes for execution on GPU
Create shader program	Create storage for a GPU executed program
Attach shaders to program	Associate vertex shader and fragment shader with the program
Link combined shader program	Create combined compiled program for GPU execution
Error compiling shaders?	Check compile status flags for both shaders
Print errors to console	Output any errors to web browser console
Set shader uniforms	Set varying attributes from application state accessible inside fragment shader
Bind vertex buffer	Place vertex buffer into GPU memory
Bind indices buffer	Place indices buffer into GPU memory
Set coordinate attribute	Associate locations on canvas with vertex locations
Clear canvas	Clear previous frame from canvas
Set render flags	Set OpenGL rendering flags
Draw elements	Draw shape primitives

Table 2: Application frame drawing processes

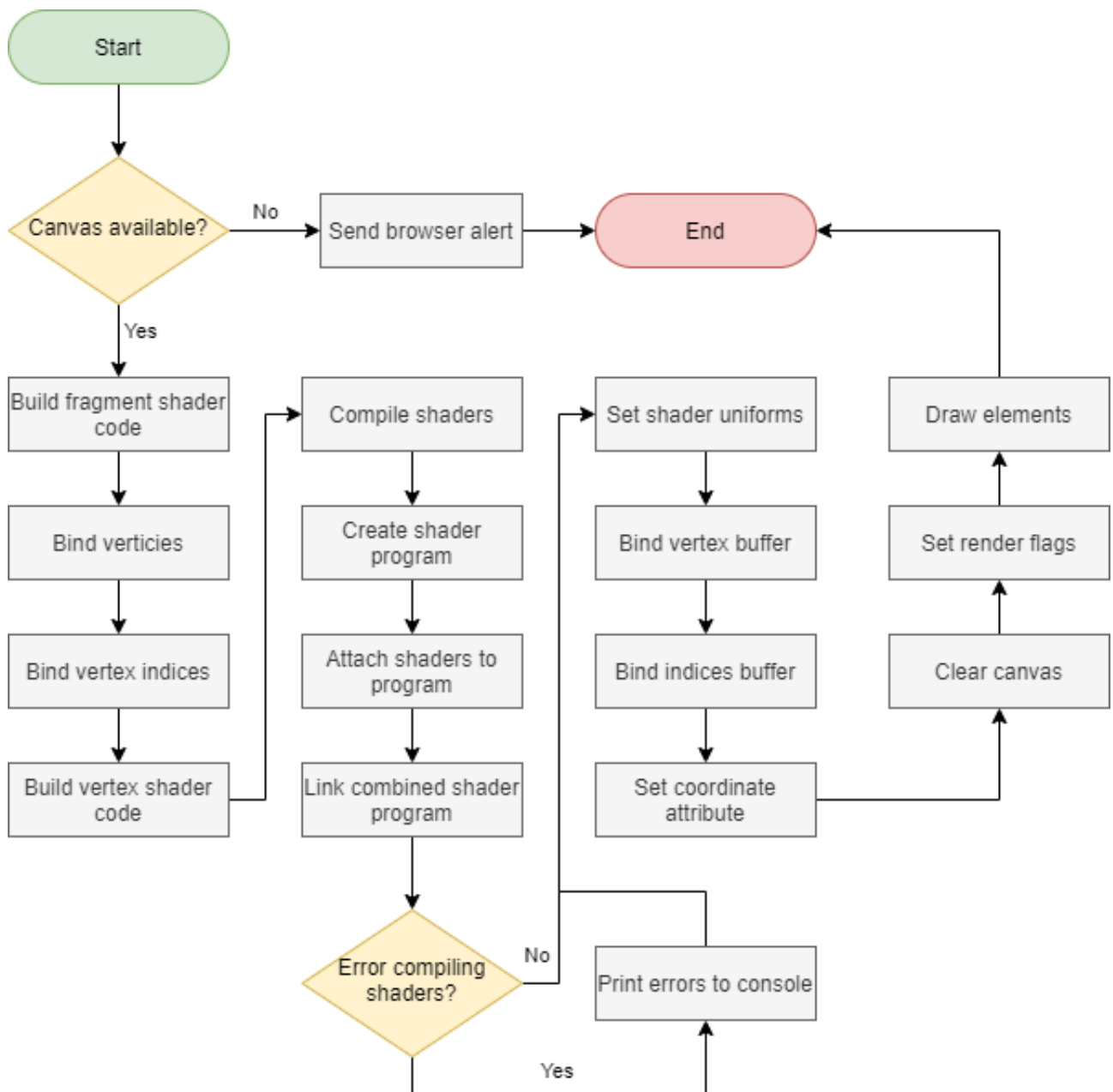


Figure 7: Application frame drawing processes

Everything rendered in OpenGL is composed of triangles. Two right-angled triangles sharing a hypotenuse are positioned across the rectangular canvas, providing a surface for rendering. The term "shader" refers to a piece of code that is executed on a GPU. A "fragment shader" is a piece of code written in the OpenGL Shading Language (GLSL) that determines the colour of each visible pixel of a rendering surface and a "vertex shader" runs once for each vertex to determine its final position. The input to this code is the location of the pixel on the surface, and output is a colour. The entire fractal renderer is implemented as a fragment shader.

The application uses two methods for passing additional variables into the fragment shader. "Uniform variables" are the default method for passing variables into a shader. Uniform variables are declared in the shader code and values are bound to them through the OpenGL API for each draw call.

The shader source codes are built from numerous JavaScript functions that each return a string representing a part of the final source code. These functions are passed the application state as a parameter, allowing variables from the application to be hard-coded into the shader. This code generation approach grants greater flexibility and addresses two concerns that uniform variables were unable to address.

Looping constructs are limited in GLSL compared to languages that run on general-purpose CPUs. Only "for" loops with a constant condition and afterthought are guaranteed across all OpenGL-compatible hardware. The reason for this is shader compilers "unroll" loops into a series of consecutive statements, a process reliant on knowing the total number of loop executions at compilation. Looping is used in two areas of the fragment shader: the Julia/Fatou iteration process, and for subdividing pixels to achieve anti-aliasing. Both of these loops are hard-coded during the JavaScript code generation stage rather than being controlled by uniform variables.

The Julia/Fatou iteration coefficients provided in the GUI must be converted into a series of calls to mathematical operations and the degree of the polynomial provided to the iteration normalisation function. String processing capabilities are not part of the GLSL language, therefore these functions are constructed outside of the fragment shader and hard-coded prior to compilation.

## 6.2 Reusable & Modular JavaScript with React

React is a lightweight framework, with few opinions about the structure and design of applications. The user is left to decide the way in which they choose to implement features. One decision it does enforce is "props down, events up", in which child components have their appearance and behaviour customised by properties originating from their parents, and events originating from children are passed up to be handled by parents (Twilio Inc, 2020). This section shows some of the choices I made when implementing the components of my application to meet objectives 1, 2 and 3.

Once the application's design had been created, it needed to be decomposed into individual components. React's developers strongly encourage the paradigm of "composition over inheritance", giving two examples of design patterns: "containment" and "specialisation" (Facebook Inc., 2020a). Both of these patterns result in highly reusable components. Table A.1 shows a set of base components that have been composed into specialised versions.

The containment design pattern involves creating a component that contains other components. The "children" property of a React component refers to any other components inside a JSX tag of the container. Figure 8 shows the "Folder" component that corresponds to the collapsible



groups in the application's GUI implementing this pattern. The alternative should this pattern not have been used would have involved creating individual components for each folder in the GUI containing duplicates of the same HTML structure and toggling logic.

```
1  export default function Folder({ title, children, startClosed }) {
2    startClosed = startClosed ? true : false
3    const [closed, toggle] = useToggle(startClosed)
4
5    return (
6      <li className='folder'>
7        <div className='group'>
8          <ul className={closed ? 'closed' : ''}>
9            <li className='title' onClick={toggle}>
10              {title}
11            </li>
12            {children}
13          </ul>
14        </div>
15      </li>
16    )
17  }
18
19  Folder.propTypes = {
20    title: PropTypes.string.isRequired,
21    startClosed: PropTypes.bool,
22    children: PropTypes.node,
23  }
```

Figure 8: Example of the container pattern

Line 3 of Figure 8 shows a React "hook" – a pattern for creating reusable and customisable blocks of logic. This specific "useToggle" hook, shown in Figure 9 can be used in any component to add a boolean and a function to switch its value between true and false.

It accepts three parameters – an initial value, a callback to execute when the value changes to true, and a callback to execute when the value changes to false.

Line 2 & 7 implement an optional parameters pattern. Unprovided parameters in JavaScript default to undefined. To make the code clearer, these parameter variables are set to null before being accessed if they are not provided.

Line 15 introduces an instance of the useState hook, part of the React library. When called with an initial value, it returns an array containing the stored variable's value and a function to set the value. Components containing useState hooks are notified whenever the value of the inner variable is mutated, triggering a re-render of the component with the updated value.

Lines 17–29 define a function named "toggle" that provides traditional toggle behaviour of

switching between true and false values, and additionally if valid callbacks are provided they are called. The parameter "e" is an event object sent by browser whenever an action occurs within an element, such as being clicked or a value being changed. This is forwarded to the callback functions granting additional flexibility in customising the behaviour of this reusable logic.

On line 31 the "bool" useState inner variable and the defined toggle function are returned to the implementing component, providing access to the functionality encapsulated within useToggle.

```
1  export function useToggle(initial, whenSetToTrueCb, whenSetToFalseCb) {
2    whenSetToTrueCb = whenSetToTrueCb || null
3    if (whenSetToTrueCb !== null && typeof whenSetToTrueCb !== 'function') {
4      throw new Error('whenSetToTrueCb not a function')
5    }
6
7    whenSetToFalseCb = whenSetToFalseCb || null
8    if (whenSetToFalseCb !== null && typeof whenSetToFalseCb !== 'function') {
9      throw new Error('whenSetToFalseCb not a function')
10   }
11
12   if (typeof initial !== 'boolean') {
13     throw new Error('Non-boolean supplied for useToggle initial value')
14   }
15   const [bool, setBool] = useState(initial)
16
17   const toggle = (e) => {
18     if (bool === true) {
19       setBool(false)
20       if (whenSetToFalseCb !== null) {
21         whenSetToFalseCb(e)
22       }
23     } else {
24       setBool(true)
25       if (whenSetToTrueCb !== null) {
26         whenSetToTrueCb(e)
27       }
28     }
29   }
30
31   return [bool, toggle]
32 }
```

Figure 9: Example React hook for providing logic to components

Figure 10 is a presentational component, one that contains no logic and acts as a way to render specialised versions dependent on the parameters it is passed, that demonstrates the "specialisation" pattern. Specialised instances of this component are created by passing properties (parameters on line 10, typed using an optional typing feature of React on lines 27–32).

Lines 11–12 show optional parameters, similar to those discussed in the useToggle example.

Line 16 demonstrates optional rendering. The ternary expression states that if label has a value, a <Label> element will be rendered, otherwise nothing will be (null).

Lines 18–20 renders multiple elements from one variable. For each element in the array, a function is run which returns an Option element populated with data matching the element in the array.

Line 17 shows how the behaviour of components can be customised using composition over inheritance. The onChange property of a <select> HTML element is a function run whenever a value is selected. The <OptionSelector> component is passed a parameter of the same name that is forwarded to the <select> element.

Figure 11 shows the HTML output of an <OptionSelector> specialised to display the options for anti-aliasing.

```
1  import React from 'react'
2  import PropTypes from 'prop-types'
3  import Item from './Item'
4  import Label from './Label'
5
6  function Option({ item }) {
7    return <option value={item}>{item}</option>
8  }
9
10 export default function OptionSelector({ label, tooltip, id, options, value, onChange }) {
11   label = label || null
12   tooltip = tooltip || null
13
14   return (
15     <Item>
16       {label ? <Label htmlFor={id} text={label} tooltip={tooltip} /> : null}
17       <select id={id} name={id} value={value} onChange={onChange}>
18         {options.map((item, idx) => (
19           <Option key={idx} item={item} />
20         ))}
21       </select>
22     </Item>
23   )
24 }
25
26 OptionSelector.propTypes = {
27   label: PropTypes.string,
28   tooltip: PropTypes.string,
29   id: PropTypes.string.isRequired,
30   options: PropTypes.arrayOf(PropTypes.string).isRequired,
31   value: PropTypes.oneOfType([PropTypes.string, PropTypes.number]).isRequired,
32   onChange: PropTypes.func.isRequired,
33 }
```

Figure 10: Example of the specialisation pattern

```

▼<li class="item ">
  ▼<label for="msaa">
    "Anti-aliasing"
    ▼<div class="tooltip">
      ::before
      ▼<span>
        "Level of supersample anti-aliasing applied to render. High levels may result in considerable slowdown"
      </span>
    </div>
  </label>
  ▼<select id="msaa" name="msaa"> == $0
    <option value="1x">1x</option>
    <option value="2x">2x</option>
    <option value="4x">4x</option>
    <option value="8x">8x</option>
    <option value="16x">16x</option>
  </select>
</li>

```

Figure 11: Example React component output

## 6.3 State Management

The majority of the application's state must be accessible in multiple locations, spread out across the DOM tree, so storing state local to individual components is inappropriate and a global store of state is required. One pattern for achieving this without passing a large number of variables and functions down the component tree is through React's Context component (Facebook Inc., 2020b).

React.createContext() returns an object containing two components: a "Provider" and a "Consumer". Once wrapped by a <Provider>, any components inside, including those further down the tree, can access the <Provider>'s "value" property through the React.useContext() hook.

The entire application state is held in an object assigned to the value of a custom <ShaderProvider> component. All leaves at the end of this object tree are setState arrays. The application is wrapped with <ShaderProvider> near the root of the application, essentially creating a global state object which can be accessed and mutated from any component. Using setStates in <ShaderProvider> ensures that components reliant on any part of the state re-render when required.

Figure 12 shows a component consuming state from <ShaderProvider>. The context object is imported in line 2 from the file where it is created by React.createContext(). Line 5 shows the value of the context being fetched using the useContext hook. In line 6, the useState array stored in the useModal property of the context object is destructured into the current value and its setter. Line 8 shows the state value used to determine the class name of the div in line 15, setting visibility on the element. The function lines 10-12 is passed to a buttons and closes the modal when clicked by setting the value to false, which causes the component to re-render with a new showModal value, updating the div's classes.

```

1  import React, { useContext } from 'react'
2  import { ShaderContext } from './ModelProvider';
3
4  export default function Modal() {
5    const ctx = useContext(ShaderContext)
6    const [showModal, setShowModal] = ctx.showModal
7
8    const className = showModal ? 'show' : ''
9
10   function closeModal() {
11     setShowModal(false)
12   }
13
14   return (
15 >   <div className={`modal ${className}`}>...
25   )
26 }

```

Figure 12: A React component consuming state from context

Two helper functions have been created to easily save and restore snapshots of application state for the preset load/save/import/export features. The saving function recursively navigates through the state object until a value with a signature matching that of a `useState` array is encountered. The current value of the `useState` array is copied into a new object with the same structure as the application state. The final return value of the function is an object with structure identical to the application state object, except the `useState` arrays have been flattened into simple variables. The restoring function operates in a similar way, except instead of flattening the `useState` arrays, it calls the setter functions within them to set the value of the application state to that stored in the object it is restoring from.

## 6.4 The Julia/Fatou Fragment Shader

The code responsible for rendering output is written in GLSL. The first step in the Julia/Fatou sets implementation was to build support for complex numbers. GLSL does not have native support for complex numbers. The closest datatype available is `vec2` – a vector containing two real numbers.

Functions that perform the following operations on instances of `vec2` as if they are complex numbers, treating the first component as the real and second as the imaginary, have been implemented: addition, subtraction, multiplication, conjugation, magnitude, exponent. Not all of these functions are used in the final application due the exponent operation being introduced at a later time than others, which fulfilled the same requirement while allowing greater flexibility and reducing lines of generated code.

Figure 13 shows the JavaScript function responsible for generating the GLSL polynomial iterate function. Given a string of comma-separated coefficients, it outputs a block of GLSL code that performs the mathematical operations corresponding to the polynomial represented by the string. This block is inserted into the GLSL code for determining the colour of each pixel as part of the iterative process.

```

1  const polyIterate = (coefficients) => {
2      function cmplxExp(exp, coeff) {
3          // Skip all terms with coefficient of 0
4          if (parseFloat(coeff) !== 0) {
5              // When exp = 0, we are handling the C value
6              if (exp === 0) return `z = complexAdd(z, ${coeff}*c);`
7              // Use complexAdd instead of complexPower for exponent 1
8              if (exp === 1) return `z = complexAdd(z, ${coeff}*zPrev);`
9              // Exponent >1, calculate power and add
10             return `z = complexAdd(z, complexPower(${coeff}*zPrev, vec2(${exp}, 0.)));`
11         }
12         // Coefficient === 0, no code needs adding
13         return ''
14     }
15
16     // Remove all whitespace, split into list delimited by commas
17     const coeffList = coefficients.replace(/\s/g, '').split(',')
18     let polySource = ''
19     // Iterate over list of coefficients
20     for (let i = 0; i < coeffList.length; i++) {
21         const exp = coeffList.length - (i + 1) // Get exponent
22         const nextTerm = cmplxExp(exp, coeffList[i]) // Get line of GLSL to add
23         if (nextTerm !== '') {
24             polySource = polySource.concat(nextTerm, '\n') // Add line to source
25         }
26     }
27     return polySource
28 }

```

Figure 13: Code for generating the GLSL polynomial iterate function

Algorithms 2 & 3 with Tables 3 & 4 respectively show pseudocode for the entire rendering process.

Variable	Description
AA	Increment for anti-aliasing loop, $\frac{1}{level}$
FragCoord	Input to the fragment shader via the OpenGL pipeline. Position of the fragment being run through the shader. 2-D coordinates accessed via .xy
FragColor	Output of the fragment shader via the OpenGL pipeline. When execution of the shader ends, the fragment is rendered with the value held here

Table 3: Fragment Shader Main variables

---

**Algorithm 2** Fragment Shader Main

---

```
1: procedure MAIN
2:   if defined AA then
3:      $n \leftarrow 0$ 
4:     for  $x \leftarrow 0; x < 1; x \leftarrow x + AA$  do
5:       for  $y \leftarrow 0; y < 1; y \leftarrow y + AA$  do
6:          $\text{offset} \leftarrow (x, y)$ 
7:          $\text{loc} \leftarrow \text{FragCoord.xy} + \text{offset}$ 
8:          $\text{colour} \leftarrow \text{Julia}(\text{loc})$ 
9:          $n \leftarrow n + 1$ 
10:      end for
11:    end for
12:     $\text{colour} \leftarrow \text{colour}/n$ 
13:  else
14:     $\text{colour} \leftarrow \text{Julia}(\text{loc})$ 
15:  end if
16:   $\text{FragColor} \leftarrow \text{colour}$ 
17: end procedure
```

---

---

**Algorithm 3** Find Pixel Colour

---

```
1: procedure JULIA( $\text{loc}, C$ )
2:    $\text{uv} \leftarrow (\text{loc}/\text{resolution}) - (0.5, 0.5)$ 
3:    $z \leftarrow (\text{viewport} * \text{uv}) + \text{translate}$ 
4:    $\text{result} \leftarrow 0$ 
5:   for  $i \leftarrow 0; i \leq \text{maxIterations}; i \leftarrow i + 1$  do
6:      $z \leftarrow \text{polyIterate}(z, C)$ 
7:      $\text{result} \leftarrow i$ 
8:     if  $\text{mag}(z) > \text{escapeRadius}$  then
9:       break
10:    end if
11:  end for
12:  if  $\text{result} = \text{maxIterations}$  then
13:    return black
14:  else
15:    if useNormalisation then
16:       $\text{result} \leftarrow \text{normalise}(\text{result})$ 
17:    end if
18:     $\text{percent} \leftarrow \text{result}/\text{maxIterations}$ 
19:    return colourmap(percent)
20:  end if
21: end procedure
```

---

Variable	Description
loc	Location of pixel on canvas in pixels
resolution	Resolution of the canvas
uv	Common variable name in graphics programming for representing position on a 2-D plane in a range with magnitude 1.
z	z value of Julia function
result	Number of iterations before escape
maxIterations	Maximum number of Julia iterations to perform
polyIterate	Function that executes the specified polynomial formula to calculate next z value
useNormalisation	Boolean, controls whether to perform escape count normalisation
normalise	Escape count normalisation function
percent	Map iterations before escape from $[0, maxIterations]$ to $[0, 1]$
colourmap	Look up colour from a 1-D texture

Table 4: Find Pixel Colour variables

Algorithm 2 demonstrates an implementation of multisample anti-aliasing (MSAA). Aliasing is a visual artefact inherent to the process of rasterising images that causes a jagged appearance near edges. MSAA smooths these areas by subdividing each pixel, finding the colour for each of these sub-pixels then summing and averaging the results to obtain the colour for each full pixel.

As seen in Figure 14, renders with MSAA enabled are more visually pleasing. Higher levels of MSAA result in smoother edges and colour gradients, albeit with a large performance cost.



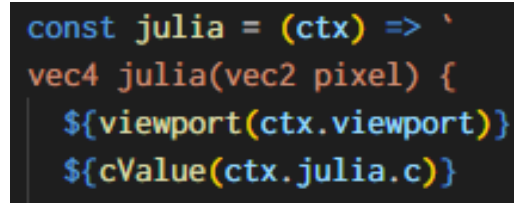
Figure 14: Comparison of no MSAA (left) and 16x MSAA (right)

Individual functions within the fragment shader were stored in the JavaScript portion of the application as either strings, or functions returning strings. These function strings were grouped



and concatenated according to functionality, and these groups again concatenated into the full fragment shader source code string.

The template literal syntax, a new addition to the JavaScript language (Mozilla, 2020), was used heavily in this process. The syntax allows for values, either defined variables or return values of functions, to be concisely inserted into strings in a readable manner. Figure 15 shows variables within the application state (variable *ctx*) being passed to functions enclosed in ``${}`` (template literal insertion syntax) to construct part of the Julia iteration GLSL function.



```
const julia = (ctx) => `
vec4 julia(vec2 pixel) {
  `${viewport(ctx.viewport)}`
  `${cValue(ctx.julia.c)}`
}
```

Figure 15: Template literal syntax

## 7 Testing, Feedback & Evaluation

### 7.1 Automated Testing

One objective in the original project specification was to implement unit tests. These are small test cases targeting small pieces of code that can be isolated in a system SmartBear Software, 2016. While unit testing can be used in UI code to ensure components render correctly, they are more appropriate for logical code that would have been present in the application's back-end system responsible for handling web requests and database access. Due to this feature being removed from the design, the use that unit testing would have brought to the project was greatly lessened.

Implementing unit tests for UI components would have been much more time consuming than performing manual tests that can be passed/failed by quick visual inspection. In the case of applications with large interfaces, testing each individual component by hand becomes unmanageable and the argument for unit tests is much stronger. This was not the case for this project in which the entire interface is visible at first glance.

### 7.2 Manual Testing

A set of manual tests for the GUI were designed to ensure each component functioned as expected.

Component	Test case
All GUI inputs	Field has tooltip containing relevant text
All GUI inputs	Invalid string inputs pause rendering rather than crashing
All GUI inputs	Elements scale according to browser zoom level while remaining usable
First colour point	Value locked to 0
Last colour point	Value locked to 1
GUI folder	Clicking on a title collapses/expands only the clicked folder
Drag zoom	Zoom function doesn't run when area of dragged box = 0
Drag zoom	Zoom function doesn't run if end point not on canvas (e.g. outside of window, on GUI)
Presets selector	Renders correctly when zero presets saved
Preset save button	User asked for confirmation when saving over preset with same name
Preset load button	User notified when attempting to load preset with no matching name found

Table 5: GUI manual test cases

Due to the decoupled nature of presentational and logical concerns throughout the application, once a component had passed tests it was unlikely that errors would occur due to changes elsewhere in the code, bugs known as "regression bugs" (Nir, Tyszberowicz, and Yehudai, 2007).

### 7.3 User Evaluation

The project was presented to test users at two points: after the first iteration of the GUI was complete, and when the project was classed feature-complete. After being provided with a short explanation of the application and how to use it, they were given 5–15 minutes experiment with it after which a conversation was held to gather opinions and feedback. Users were divided into two categories based on their level of competency with technology to generate a broader range of feedback as technical users may notice flaws that non-technical users may not, and the latter group may pick up on flaws which more competent users gloss over.

Tables 6, 7 show items of feedback gathered from the conversations. Items in green are those which were addressed, those in red remained resolved.

User	Feedback	Actions
Non-technical	GUI too large for screen (mobile)	Addressed in GUI redesign by reducing width
Non-technical	Meaning of items in GUI unclear	Tooltip texts refined
Technical	Scrollbar on left side of GUI feels unnatural	GUI moved to other side of screen to match expected user interaction patterns. Scrollbar styled differently from native browser scollbars to hint the user that it scrolls the GUI rather than the whole page
Non-technical	Presence of GUI items which require scrolling not obvious	Addressed in GUI redesign by reducing element size and margins

Table 6: Items from first set of feedback

User	Feedback	Actions
Technical	Brightness issues when anti-aliasing enabled on mobile	Believed to be caused by value-clamping behaviour specific to certain vendor implementations of the OpenGL API. An investigation into OpenGL flags for adjusting this default behaviour was performed but none were found.
Technical	Presets folder does not have any items at start	Set of default presets added on initial load if none found
Non-technical	Meaning of items in GUI unclear	Further refinement of tooltip text
Non-technical	Some combinations of values result in black screen	Suggested range of values added to tooltips
Technical	Performance severely degrades over time	Unable to reproduce on development machine, culprit identified as a browser extension test user had enabled
Technical	Enter button to save conflicts with expected behaviour of submitting an input field	Hotkey for screenshot changed, GUI button offering same functionality added
Non-technical	Unclear how to animate renders	Additional information added to help menu, default preset demonstrating animation added

Table 7: Items from second set of feedback

The majority of feedback was able to be addressed. Certain items related to technical matters as opposed to altering or adding small pieces of usability and functionality were unable to be resolved. In hindsight, there was not enough time allocated between when the feedback was gathered and the conclusion of the project. A strategy identified to address this would be to plan more checkpoints after major feature changes and perform external testing to generate more feedback throughout the lifecycle of the project.

## 8 Improvements

### 8.1 GUI

An interesting approach to automatic GUI generation can be found within the Fractal Lab (subblue.com, 2011) application. This website uses the a similar WebGL fragment shader approach to fractal rendering. In the source code for the fragment shaders used in this project,

comments after the declaration of `#define` statements and uniform variables provide a definition for accompanying GUI fields, shown in Figure 16.

```
uniform float power;           // {"label":"Power", "min":-20, "max":20, "step":0.1, "default":8, "group":"Fractal"}
uniform float surfaceDetail;   // {"label":"Detail", "min":0.1, "max":2, "step":0.01, "default":0.6, "group":"Fractal"}
uniform float surfaceSmoothness; // {"label":"Smoothness", "min":0.01, "max":1, "step":0.01, "default":0.8, "group":"Fractal"}
uniform float boundingRadius;  // {"label":"Bounding radius", "min":0.1, "max":150, "step":0.01, "default":5, "group":"Fractal"}
uniform vec3 offset;          // {"label":["Offset x","Offset y","Offset z"], "min":-3, "max":3, "step":0.01, "default":[0,0,0],
"group":"Fractal", "group_label":"Offsets"}
```

Figure 16: Fractal lab GUI-in-comments system

Fractal Lab offers more functionality to users with the ability to write their own GLSL code, as they are able to extend the default code and integrate it into the GUI. While implementing this extension capability is not relevant to the goals of the project, the automatic GUI generation is an interesting and effective approach that would meet objective 3 and one worth investigating for future iterations of the WebGL framework created thus far.

## 8.2 State Management

The approach to web application state management demonstrated in this project is one of many possible examples. In retrospect, the React context approach may have not been the most suitable for the application given the final size, complexity, and goal of flexibility. The Flux pattern utilises a unidirectional data flow that compliments the declarative nature of React applications (Facebook Inc., 2020c).

This pattern was investigated before any code was written, but dismissed as being excessively time consuming for an application of this size due to high levels of boilerplate code required for implementation. Even though care has been taken to decouple state code and presentation code, some cases for state access and mutation lead to code with undesirable qualities. Improvements in this area could have been made by using the Flux pattern to help fulfil objectives 1, 3 and 4.

## 8.3 Planning

The original plan for the project was broken down into a Gantt chart with week-long intervals. One to two weeks was dedicated to working on a specific area before moving onto the next. Approximately 20 tasks were identified in the original plan. This level of granularity was not sufficient for a smooth development process, nor the ordering of tasks. Sticking to the original plan was not possible due to parts of tasks being reliant on parts of later tasks. These dependencies were not identified in the plan and lead to inefficiencies when attempting to follow the original plan.

In the latter half of the project, the approach was adjusted to address concerns identified in the first half. At the end of each working day, a task list was created specifying the tasks to be completed the next. Any tasks projected to take greater than two hours was further decomposed until they met this criteria. These smaller tasks often were concerned with very different parts of the application. As an example, rather than creating the entire presentational side of the GUI and linking it to the application's state as two separate tasks, once a GUI component was created the state functionality was immediately added. This had the additional benefit of making bugs easier to identify and fix. Rather than the same bugs being coded in multiple GUI components and only made apparent at a later stage, if a bug that was likely to occur in a similar component was identified, it was able to be resolved and avoided in the future.

The theme of this change in process was to group tasks by the area of functionality they are related to (e.g. a single GUI button), rather than by the type of functionality they provide (e.g. presentation, state).

## 8.4 Testing

The testing of the application was mostly unplanned. While the project did not suffer greatly as a consequence, this is not good practice relevant to objective 5.

Due to the nature of the application's output being graphical and unpredictable, creating a predefined table of test cases mapping inputs to expected output is difficult, making it hard to identify how testing should take place. Alternate testing methodologies could have been identified and proposed before implementation of the project began.

When interviewing users using the application to generate feedback and suggestions, the information received from this process was completely user-driven. To generate multiple items of feedback about specific aspects of the application, a set of questions could have been devised for each participant to answer in addition to the conversation method that was used. Doing so would have resulted in greater feedback about specific aspects of the application that could be identified for improvement in addition to the general feedback that was received. Structured feedback generated through this process would compliment the broader approach that was taken.

## 8.5 Research

A number of potential features and improvements that could have been part of the original project were discovered late. Not enough research into the features and implementation of similar applications was performed prior to starting the project. There were a few occasions that while searching for a solution to a problem that had been encountered, ideas occurred but were not able to be worked into the project in a way that wouldn't have potentially jeopardised the delivery of the project on time. Had these been discovered before the initial proposal submitted, a more feature-rich and robust project could have been designed and subsequently implemented.

## 8.6 Complimentary Programming Paradigms

The application was created using the functional programming paradigm. This decision was made because React components are functions and functional programming patterns, such as higher-order functions are applicable. I wanted to improve my knowledge of this paradigm as I will be using it in my career and compared to object-orientated and other paradigms, it did not receive as much attention throughout my degree. I came to the realisation that programming paradigms are better treated as tools and used when appropriate rather than being strictly followed.

JavaScript is an untyped language and any variable can hold any type of data, with some exceptions such as typed arrays that are used in the application when interfacing with WebGL. Classes are not supported in the same way they are in object-orientated languages but similar syntax has been added recently to the language that acts as a wrapper around the functional core and gives it an object-orientated appearance.

One area where this could have been utilised is when representing colours and switching between different colour models. Colours are stored in the application as size three arrays of RGB values. When the application requires HSV values for colour map interpolation, a the colour mapping code calls a function to perform this conversion.

This solution is not the most elegant or flexible. It involves hardcoded checks in a number of files and adding additional colour models would be more work than if an object-orientated approach was initially taken. An improved solution would be to create a Colour class that when constructed with a set of values and a flag indicating which colour model these initial values belong to automatically calculates and stores the values for all available colour representation models. Once all code related to accessing these objects has been updated to access colours through the class, adding additional colour models would simply involve extending the class.

## 9 Extensions

### 9.1 Quaternions and Julia Sets in Higher Dimensions

Quaternions are a four-dimensional numerical construct similar to how complex numbers are two-dimensional. The same iterative process can be applied to quaternions to compute sets similar to those of two-dimensional Julia set renders (Quilez, 2001).

Path tracing is a three-dimensional rendering technique that can be implemented in fragment shaders with relative ease and is especially suited to rendering objects definable by mathematical functions. Path tracing works by shooting a ray from each pixel in the viewport and gradually tracing the straight path it follows until an intersection with an object is found. Intersections are detected by testing the distance between a ray's current position and the set of objects in the scene. The ray is moved forwards until this distance is below a certain threshold signalling the boundary of an object is close enough to be considered present at that location.

This approach works in three-dimensional space, so for the case of quaternions only three of the four components are taken into consideration. The fourth can be modulated over time in order to visualise this missing component, resulting in a morphing shape as slices of the four-dimensional construct are rendered in three dimensions. This is similar to the process of modulating the constant value of the two-dimension Julia/Fatou renders, which is a technique that can also be carried over to this case.

Octonions are an extension of Quaternions with eight dimensions and can also be used to generate Julia sets, however visualising shapes higher than four dimensions into three dimensional space comes with its own set of challenges and would require a solution more complex than the time-slicing method applicable to Quaternions.

Almost all of the code required for adding this functionality to the application is already present. Examples of path tracing implemented in fragment shaders exist such as that shown in Quilez's (2012) blog post.

### 9.2 Complex Iterative Formula

While the rendering code has the capability to use complex numbers for exponents and coefficients in the iterative formula, they cannot be input through the GUI. This feature was not taken into account during the specification phase and discovered later on. Leaving the realm

of real exponents and coefficients provides leads to even more interesting nonlinear forms and shapes.

A more robust GUI section for entering the iterative formula would enable this feature. It could be implemented with either a parser for a defined syntax that the user could use to specify exponents and coefficients as a single string, or each of the terms in the polynomial could be constructed from multiple grouped input fields where the real and imaginary components of both the exponent and coefficients. The first approach has the advantage of being more concise while the latter would be quicker to implement and more accessible to users, especially those without experience writing formulas using syntax.

Interesting behaviour can be found in the set of iterative formulae with form  $z_{n+1} = \frac{f(z_n)}{g(z_n)}$  where  $f(z_n)$  and  $g(z_n)$  are both polynomials. Again, code exists that could enable this behaviour and the limitation exists in the GUI. One identified implementation idea was identified, by duplicating the existing iterative formula field and updating the fragment shader code to treat the two inputs as  $f(z_n)$  and  $g(z_n)$ , dividing one by the other. While relatively painless to implement it did not present itself as satisfactory, the main reasoning behind so being the extra computational load. This naïve solution would result in many extra calls to the costly complex exponent function. A more elegant solution could simplify the division into a single polynomial before passing it to the fragment shader with the aim of reducing the number of calculations required.

Finally, unique forms of Julia/Fatou sets can be obtained from iterative formula containing functions e.g. *cos*, *sin* and *exp*. Adding this feature would require research into the complex equivalents of these mathematical functions, implementing them as functions in the code of the fragment shaders, and providing a method of access via additional GUI features.

### 9.3 Animation with Lissajous Curves

Lissajous curves are the family of curves described by the parametric equations:

$$\begin{aligned}x(t) &= A\cos(\omega_x t - \delta_x) \\ y(t) &= B\cos(\omega_y t - \delta_y)\end{aligned}$$

(Wolfram Research Inc., 2020)

To better address the reported issue of users struggling with the animation system, a section in the GUI dedicated to providing parameters to generate a Lissajous curve to modulate the  $C$  value.

An interesting property is that a Lissajous curve will be closed if and only if  $\frac{\omega_x}{\omega_y}$  is rational (Wolfram Research Inc., 2020). When creating graphical animations, the property of closed curves is highly desirable as it guarantees the value being modulated will never have discontinuous jumps that can disrupt the viewer's immersion.

An additional feature that could be implemented alongside a Lissajous curve editor is plotting the orbit and current position of  $C$  over the fractal. This visual representation would help users understand how varying values for  $C$  change the resulting fractal.



## 9.4 Newton Fractals

Another kind of fractal closely related to the ideas and process behind Julia sets is the Newton Fractal, Based on the Newton-Raphson iterative process of finding roots of functions. The Newton-Raphson process generates approximations of roots for a function  $f(x)$  with an initial guess  $x_0$ , with each successive iteration closing in on the true value, using the formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

To generate fractals using this process, the set of roots  $R$  for a function are found. For each point  $z$  in a defined area of the complex plane, we perform the Newton-Raphson process until the distance between  $z_n$  and any item in  $R$  is less than a threshold value. At the point,  $z$  is considered to converge towards that root. Each root is assigned a colour, and a pixel is set to that colour if it converges towards that root. This will create visually unappealing blocks of colour, which can be remedied using a process similar to that of the Fatou set shading. The iteration count before convergence functions similarly to the iteration count before escape, and a normalisation function can be applied to create smooth gradients.

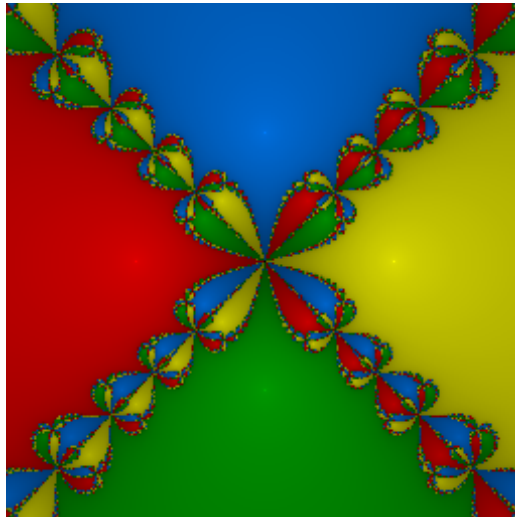


Figure 17: Newton fractal for  $z^4 - 1$  (Tatham, Simon, 2017)

Tatham, Simon, 2017 presents an excellent dissection of the process behind generating Newton fractals on his website where additional explanations and examples can be found.

## 9.5 Mandelbrot Set

The Mandelbrot Set is a fractal closely related to the Julia sets. It follows the same iterative process, however rather than keeping the value of  $C$  constant and varying the starting point  $z_n$ ,  $C$  is assigned the value of the point being iterated and  $Z_n$  is always initialised as 0. Behaviour of points when considering whether they belong to Mandelbrot set follows the same rules as those for the Julia set.

Implementing functionality for rendering polynomial Mandelbrot sets can be achieved with a toggle that switches to an adapted version of the Julia fragment shader. Many components of the GUI can be shared between the two.

## 10 BCS Project Criteria

The six outcomes expected for the project by the Chartered Institute for IT are:

1. An ability to apply practical and analytical skills gained during the degree programme
2. Innovation and/or creativity
3. Synthesis of information, ideas and practices to provide a quality solution together with an evaluation of that solution
4. That your project meets a real need in a wider context
5. An ability to self-manage a significant piece of work
6. Critical self-evaluation of the process

Item 1 is demonstrated within the design and implementation of the project.

Item 2 is demonstrated in the design section, where features that make my application unique from other similar ones.

Items 3 and 5 are demonstrated across the whole project. Starting with the initial theory, a design was created and successfully implemented within a .

Uses for the application are discussed in the introduction, demonstrating item 4.

The following self-reflection addresses item 6.

## 11 Self-Reflection

I could not be more pleased with the journey of the project. I am thankful to have had the opportunity to demonstrate and improve the skills learnt throughout the past few years within the context of a project I feel passionate about. It has been by far the most ambitious and most successful project I have undertaken. I believe my strengths have grown, flaws either addressed or highlighted for improvement with a plan in place, and new areas for personal development discovered. All specified objectives for the project have been met to a high level, with one exception being objective 5 which lags behind the others.

The initial design specification was too broad and I believe had the changes that occurred not been made, the final application could have ended up as two individual pieces of software I may have been disappointed with. By changing course once this came to mind, I was successfully able to focus my work in one direction with the result being one application I am extremely proud of that I want to keep working on in the future. "What focus means is saying no to something that with every bone in your body think is a phenomenal idea, and you wake up thinking about it, but you end up saying no to it because you're focusing on something else" (). Perhaps when I am extending the application, I will improve and implement the originally planned back-end features to elevate the application to a higher level. Time management, a component of objective 5, is not just about making a schedule and sticking to it as I thought prior to the project. Being able to identify when there is not enough time to do everything you want to do and focusing your time in a way that gives the best chance of meeting your most

important goals is arguably more important. I believe the ability to change course when things are not going as you initially hoped is invaluable.

As discussed in the Section 8.3 planning of the project was sub-par, however I do not see this as a failure of the project or disappointing. I expected many difficulties to arise from the planning aspect of the project, having identified it as an area of weakness in my objectives. I have had an opportunity to reflect upon what went wrong in the project plan and learnt valuable lessons that will influence my future work.

Changes in the design limited the potential for software testing, especially in the domain of automated testing which was a specific skill I hoped to develop due to its relevance in software engineering careers. To address this, I plan to follow a "test driven development" methodology for my next personal project.

After looking back at the project as a whole, thoughts about the trade-off between experimentation/iteration and concrete conceptualisation came to mind. In Section 8.5, I wrote about the amount of research conducted prior to beginning the application and whether it was satisfactory. I believe that no matter how much preparation is carried out in advance, further ideas will always come in waves as solutions are implemented. While there are certainly situations where it is highly desirable to seek a complete plan before work is carried out, in the realm of creative and open-ended problems which this project falls into, capitalising on iterative inspiration and experimenting with new ideas can lead to results that surpass initial expectations.

The prospect of implementing improvements and extensions I have identified excites me. As the project is closely related to personal interests of mine, I will be continuing its development for the foreseeable future. Some of these extensions are not restricted to use in this project and will inspire and improve other creating coding endeavours.

In previous creative coding projects, I have encountered limitations related to various areas of knowledge that this project has been able to address. There have been times when my tools were not able to meet performance requirements due to the limitations of general-purpose CPUs. Working with OpenGL and GLSL has provided me with a new tool that provides access to a wide range of new possibilities. I have struggled with creating GUIs that are flexible enough to be implemented across multiple projects without taking away time from the creative coding aspect itself. The framework I have created will be extended and reused in many projects to come and I hope that others will find it similarly useful once the source code has been made public.

## 12 Conclusion

The project has been successful in terms of meeting the outlined objectives, producing a interesting and useful application, and also being a pleasure to undertake. Through the use of specifically selected technologies, the application goes beyond others also accessible through web browsers. Many of the improvements and extensions will be implemented to increase the personal value of the application and to provide online users greater ways to explore the world of fractals.

The next step for the project is stripping out the Julia/Fatou specific code and releasing the GUI and WebGL initialisation components together on GitHub as an open-source framework for quickly creating fragment shader based applications. When it comes to creative endeavours, there should be as little friction as possible in regards to sharing creations. Others who may have faced difficulties sharing their fragment shader based work online without having to invest

significant amounts of time learning how to make it possible now have a solution in place.

## References

- Alligood, Kathleen T. et al. (1997). “Chaos: An Introduction to Dynamical Systems”. In: Facebook Inc. (2020a). *Composition vs Inheritance*. Available at: <https://reactjs.org/docs/composition-vs-inheritance.html> (Accessed: 03 May 2020).
- (2020b). *Context*. Available at: <https://reactjs.org/docs/context.html> (Accessed: 01 May 2020).
- (2020c). *In-Depth Overview / Flux*. Available at: <https://facebook.github.io/flux/docs/in-depth-overview/> (Accessed: 06 May 2020).
- (2020d). *React - A JavaScript library for building user interfaces*. Available at: <https://reactjs.org/> (Accessed: 30 April 2020).
- Fatou, Pierre (1917). “Sur les substitutions rationnelles”. In: *Comptes Rendus de l’Académie des Sciences de Paris* 166, pp. 599–601.
- freeCodeCamp (November 2019). *A statistical analysis of React, Angular and Vue*. Available at: <https://www.freecodecamp.org/news/angular-react-vue/> (Accessed: 02 May 2020).
- Google Data Arts Team (2011). *dat.GUI*. Available at: <https://vincentgarreau.com/particles.js/> (Accessed: 03 May 2020).
- Ive, Jony.
- Julia, Gaston (1918). “Mémoire sur l’itération des fonctions rationnelles”. In: *Journal de Mathématiques Pures et Appliquées* 1, pp. 47–246.
- Mozilla (2019a). *WebGL: 2D and 3D graphics for the web*. Available at: [https://developer.mozilla.org/en-US/docs/Web/API/WebGL\\_API](https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API) (Accessed: 26 October 2019).
- (2019b). *Window.localStorage*. Available at: <https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage> (Accessed: 05 May 2020).
- (2020). *Template literals (Template strings)*. Available at: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template\\_literals](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals) (Accessed: 07 May 2020).
- Nir, Dor, Shmuel Tyszberowicz, and Amiram Yehudai (2007). “Locating Regression Bugs”. In: *Proceedings of the 3rd International Haifa Verification Conference on Hardware and Software: Verification and Testing*. HVC’07. Haifa, Israel: Springer-Verlag, 218–234. ISBN: 3540779647.
- Peitgen, Heinz-Otto, Hartmut Jürgens, and Dietmar Saupe (1992). *Fractals for the Classroom: Part Two: Complex Systems and Mandelbrot Set*. 1st ed. Springer-Verlag New York. Chap. 13. ISBN: 978-0-387-97722-5.
- Quilez, Inigo (2001). *3d julia sets*. Available at: <https://www.iquilezles.org/www/articles/juliasets3d/juliasets3d.htm> (Accessed: 06 May 2020).
- (2007). *smooth iteration count for generalized Mandelbrot sets*. Available at: [http://www.iquilezles.org/www/articles/mset\\_smooth/mset\\_smooth.htm](http://www.iquilezles.org/www/articles/mset_smooth/mset_smooth.htm) (Accessed: 28 April 2020).
- (2012). *simple pathtracing*. Available at: <https://www.iquilezles.org/www/articles/simplepathtracing/simplepathtracing.htm> (Accessed: 06 May 2020).
- SmartBear Software (January 2016). *What Is Unit Testing?* Available at: <https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage> (Accessed: 05 May 2020).
- subblue.com (2011). *Fractal Lab - Interactive WebGL Fractal Explorer*. Available at: <https://hirnsohle.de/test/fractalLab/> (Accessed: 06 May 2020).
- Tatham, Simon (February 2017). *Fractals derived from Newton-Raphson iteration*. Available at: <https://www.chiark.greenend.org.uk/~sgtatham/newton/> (Accessed: 09 May 2020).
- The Khronos Group (2020). *Rendering Pipeline Overview*. Available at: [https://www.khronos.org/opengl/wiki/Rendering\\_Pipeline\\_Overview](https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview) (Accessed: 30 April 2020).

- Twilio Inc (2020). *What is React*. Available at: <https://www.twilio.com/docs/glossary/what-is-react> (Accessed: 03 May 2020).
- Vincent Garreau (2017). *particles.js - A lightweight JavaScript library for creating particles*. Available at: <https://vincentgarreau.com/particles.js/> (Accessed: 03 May 2020).
- W3Schools (2020). *CSS Units*. Available at: [https://www.w3schools.com/cssref/css\\_units.asp](https://www.w3schools.com/cssref/css_units.asp) (Accessed: 01 May 2020).
- Wolfram Research Inc. (2020). *Lissajous Curve*. Available at: <https://mathworld.wolfram.com/LissajousCurve.html> (Accessed: 09 May 2020).

## A React Components

Component	Contains	Purpose	Variations
Base64Image	img	Display an image from a given base64 encoded string	Base64 string, encoding, alt text, CSS classes
Button	button	Clickable button	Text, onClick function, CSS classes
Folder	ul, li, div	Nestable, collapsible group of elements in GUI	Text, initial state (open/collapsed)
HelpTooltip	div, span	Icon that can be hovered to display help text	Hover text
InputField	Item, Label, input	Labelled input field in GUI	HTML input type, label text, tooltip text, disabled, onChange function
Item	li	Container for non-folder entries in the GUI	CSS classes
Label	label, HelpTooltip	Provide text label and optional tooltip next to inputs	Label text, tooltip text
OptionSelector	Item, Label, select, option	Drop-down selector field in GUI	Label text, tooltip text, select options, initial value, onChange function
GUI	ul	Container for GUI elements	None

Table A.1: Base components

Component	Contains	Purpose
ColorCurveSelector	OptionSelector	Specialises an OptionSelector for selecting a curve for interpolation between colour points
ColorMapFolder	Folder, ColorPointItem	Container for ColorPointItems, provide logic for handling onChange events of ColorPointItems
ColorMapRender	Item, Base64Image	Encode colour map texture data from state as png, display as element
ColorModelSelector	OptionSelector	Specialises an OptionSelector for selecting a colour model for interpolation between colour points
ColorPicker	InputField	Specialises an InputField for selecting an RGB colour
ColorPosition	InputField	Specialises an InputField for inputting a position for a colour on the colour interpolation map
ColorPointItem	Folder, ColorPicker, ColorPosition	Container for ColorPicker and ColorPosition

Table A.2: Colour mapping components

<b>Component</b>	<b>Contains</b>	<b>Purpose</b>
Coefficients	InputField	Specialises an InputField for inputting polynomial coefficient
ConstantPointX	InputField	Specialises an InputField for inputting constant point real component
ConstantPointY	InputField	Specialises an InputField constant point imaginary component
EscapeRadius	InputField	Specialises an InputField for inputting escape radius
MaxIterations	InputField	Specialises an InputField for inputting max iterations
MSAA	OptionSelector	Specialises an OptionSelector for selecting MSAA level
SmoothingToggle	InputField	Specialises an InputField for enabling/disabling iteration count normalization
TimeScale	InputField	Specialises an InputField for inputting global time scale

Table A.3: Julia function components

<b>Component</b>	<b>Contains</b>	<b>Purpose</b>
HelpButton	Item, Button	Toggle help modal
ScreenshotButton	Item, Button	Save screenshot of canvas
PauseButton	Item, Button	Pause incrementation of time
ToggleMenuButton	Item, Button	Hide/show GUI (except HelpButton, PauseButton, ToggleMenuButton)
Modal	div, typography elements	Provide instructional information to user
CloseModalButton	button	Provides a button to close the modal

Table A.4: Miscellaneous components



<b>Component</b>	<b>Contains</b>	<b>Purpose</b>
ClipboardDOM	Item, input	Provides an input field required to use browser APIs for accessing clipboard
PresetButtons	Item, Button	Specialises Buttons for saving, loading, importing and exporting application state
PresetErrorField	Item, span	Displays errors when using preset functions
PresetFolder	Folder, PresetSelector, PresetNameField, PresetButtons, PresetErrorField, ClipboardDOM	Container for GUI elements related to presets, provides logic to PresetButtons, provides logic for storing/loading from browser localStorage, provides options to PresetSelector
PresetNameField	InputField	Specialises an InputField for inputting a preset name
PresetSelector	OptionSelector	Specialises an OptionSelector for selecting saved presets

Table A.5: Presets components

<b>Component</b>	<b>Contains</b>	<b>Purpose</b>
Dimensions	Folder, ViewportWidth, ViewportHeight	Container for ViewportWidth and ViewportHeight, provides logic for handling changes in those two components
ViewportWidth	InputField	Specialises an InputField for inputting viewport width
ViewportHeight	InputField	Specialises an InputField for inputting viewport height
LockAspectRatio	InputField	Specialises an InputField for enabling/disabling viewport aspect ratio locking
TranslateX	InputField	Specialises an InputField for inputting translation along the real axis
TranslateY	InputField	Specialises an InputField for inputting translation along the imaginary axis

Table A.6: Viewport components

<b>Component</b>	<b>Contains</b>	<b>Purpose</b>
ModelProvider	ShaderProvider	Contains initial state object, functions for loading objects into state and exporting current state to object
ShaderProvider	n/a	Provides access to application state to child components
ShaderCanvas	canvas	Instantiates rendering, provides drag mouse behaviour, right-click behaviour, image saving function, viewport aspect ratio scaling, colour map texture construction
MyGUI	See Figure A.1	Container for GUI elements, provides logic for toggling GUI visibility
App	ModelProvider, ShaderCanvas, Modal, MyGUI	Root element, rendered onto webpage

Table A.7: Root components

```

1  export default function MyGUI() {
2    const [hideMenu, toggleMenu] = useToggle(false)
3
4    return (
5      <GUI>
6        <PauseButton />
7        <ToggleMenuButton onClick={toggleMenu} />
8        <HelpButton />
9        <ScreenshotButton />
10       <div style={{ display: hideMenu ? 'none' : 'inherit' }}>
11         <Folder title='Julia Variables'>
12           <Coefficients />
13           <Folder title='Constant Point'>
14             <ConstantPointX />
15             <ConstantPointY />
16           </Folder>
17           <MaxIterations />
18           <EscapeRadius />
19           <SmoothingToggle />
20           <MSAA />
21           <TimeScale />
22         </Folder>
23         <Folder title='Viewport' startClosed>
24           <DimensionsFolder />
25           <Folder title='Translate'>
26             <TranslateX />
27             <TranslateY />
28           </Folder>
29           <LockAspectRatio />
30         </Folder>
31         <Folder title='Colour Mapping' startClosed>
32           <ColorMapFolder />
33           <ColorCurveSelector />
34           <ColorModelSelector />
35           <ColorMapRender />
36         </Folder>
37         <PresetFolder />
38       </div>
39     </GUI>
40   )
41 }

```

Figure A.1: Structure of MyGUI component