Real-Time Fractal Rendering

Non-Examined Assessment
Optimised Fractal Rendering in C++

Toby Davis
Hampton School

March 16, 2023

Abstract

Write this bad boi when the thing is finished :)

Summary

1 Project Analysis

1.1 A Brief Introduction to Fractals

A fractal is "a curve or geometrical figure, each part of which has the same statistical character as the whole"[1].

Some fractals are defined by simple equations which exhibit chaotic behaviour. Arguably the most famous fractal, the Mandelbrot Set, is defined by the following iterative equation, where $Z_0 = 0 + 0i$ and C is the initial value in the complex plane.

$$Z_{n+1} = Z_n^2 + C \quad : \quad Z_n, c \in \mathbb{C} \tag{1}$$

For a given point C to be in the Mandelbrot Set, the value of Z_n must remain bounded (i.e. not diverge to infinity) after the iterative series is repeated infinitely many times. This approach is used in most iterative fractal equations.

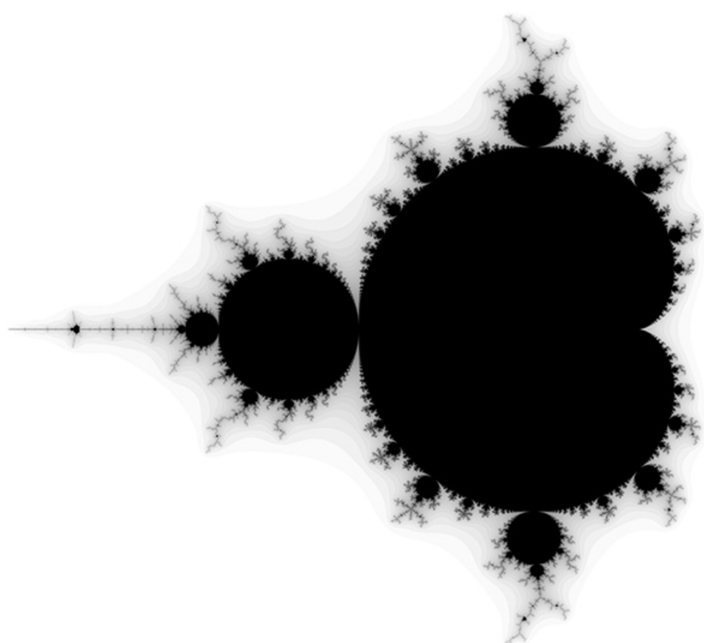


Figure 1: The standard Mandelbrot fractal[2]

Additionally, fractal variations can be created by changing the generating equation slightly. For example, changing the r in the Mandelbrot equation ($Z_{n+1} = Z_n^r + C$) yields the following fractals.

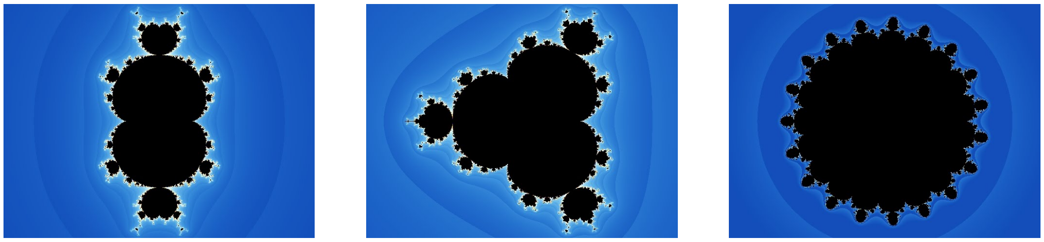


Figure 2: (left) $r = 3$, (centre) $r = 4$, (right) $r = 20$

Other famous fractals include the Sierpiński Triangle, the Julia Set, Hilbert Spirals, etc. All are defined either by infinitely-recursive self-similar patterns or repeated equations.

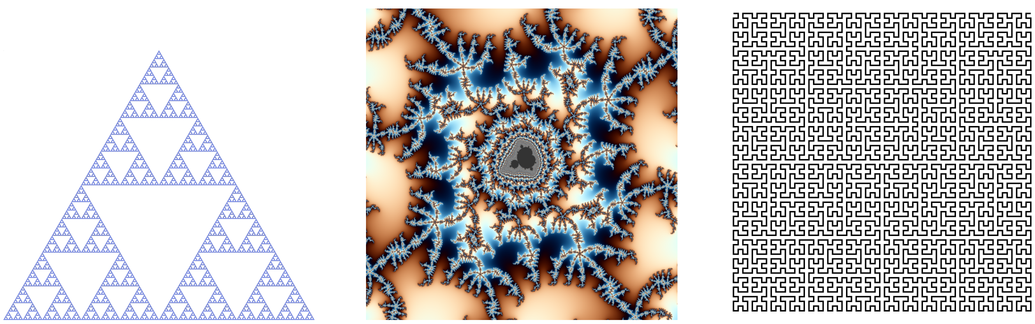


Figure 3: (left) Sierpiński-Triangle, (centre) Part of the Mandelbrot Set, (right) Hilbert Curve

1.2 Defining the Problem

Fractals have been the subject of much debate and curiosity throughout history. However, due to the computational requirements of generating them, research into them was minimal until the rise of the electronic computer.

The newfound processing power allowed increasingly detailed images to be generated, and mathematicians could better understand fractals' underlying equations and seemingly chaotic nature.

With the power of modern computers, it is possible to render some fractals in real-time and explore them to great depths, though there are still technical, physical and monetary hurdles to clear.

Many fractal rendering programs exist online; however, most are incomplete, inefficient applications not designed for high performance and increased zoom factors. While many high-quality applications exist, the best ones are often quite expensive, making them inaccessible to most potential users. For example, some extremely advanced software costs almost £80[3].

Technological Limitations The further you zoom into a fractal, the smaller the numbers you have to deal with. At lower zoom levels, this doesn't pose much of an issue, as 64 bit or even 32 bit floating point numbers often have the required precision to render an image accurately. At higher zoom levels, however, the precision of the numbers used affects the image quality.

When zoomed in far enough, floating point rounding errors start to cause certain pixel positions to merge into one, causing unattractive "blocky" patterns in the image. Eventually, these blocks will consume the entire image, and no more detail can be seen.

To get around this, it is possible to use high-precision floating point data types. However, since these are processed in software, not hardware, they are orders of magnitude slower than normal number types, which can make the rendering process impractically slow.

Some techniques can be taken to optimise the performance of high-performance number types. For example, it can be proven that Z_n will diverge to infinity if $|Z_n| > 2$ at any point. Additionally, advanced algorithms can mix fixed and multi-precision arithmetic to decrease the number of operations performed in software.

Program Limitations Many fractal renderers do just that; render fractals. They don't support any render export features and do not allow for saving, reloading or sharing render configurations.

Some programs have methods to save the rendered fractals as image files but often have limiting export settings and don't support many resolutions. Some programs allow the current position and zoom level, among other information, to be exported to a file, allowing interesting fractal locations to be shared easily.

Precision vs Performance To increase rendering performance, most implementations of fractal renderers use 32- or 64-bit floating point numbers. Since operations on these data types are performed directly by hardware, they are highly efficient. Unfortunately, 64-bit floating point values can only accurately represent around 15 decimal places, so zooming in far enough will exceed this precision and cause visual glitches.

To circumvent this issue, it is possible to use multi-precision floating point types capable of representing hundreds, thousands or even millions of bits, allowing for near-infinite zooms. These numbers, however, are implemented in software and are many orders of magnitude slower than standard floating point types. It is possible to use multi-precision floating point arithmetic for sufficiently optimised programs, though the performance will be abysmal.

Furthermore, some areas of different fractals require a considerable number of iterations before a reasonable amount of detail can be obtained. As a result, potentially millions of calculations must be done to determine the colour for a single pixel.

The two main issues above become even more extreme when combined with the goal of near-infinite zooming. Due to the nature of many fractals, the number of iterations required to get high levels of detail in areas close to the border of the fractal increases with zoom. Additionally, deeper zooms need higher precision numbers to represent all the points accurately. Combine these, and the result is a slow, inefficient program.

1.3 The End User

Mathematics Teachers and Professors could use the program to assist in their lessons and provide students with an interactive resource to help with homework and further their understanding. This could dramatically increase students' engagement in studies and inspire them to pursue further degrees in mathematics. Additionally, less well-off schools could afford and use the software if the program is free and open source, increasing its accessibility.

Researchers and General Acedmia could use the high precision, deep zooms and fast renders to further their studies on the properties of fractals. Furthermore, the more advanced fractal export tools could be used to share the exact configurations of the areas they explore to accelerate the peer-review process.

Anyone Interested in Mathematics could explore fractals' beauty and share the rendered images with their friends and family. If the program is suitably intuitive, even young children would be able to use it, potentially inspiring an interest in mathematics.

1.4 Analysis of Existing Programs

1.4.1 David J. Eck's Online Mandelbrot Renderer [4]

David Eck's online Mandelbrot rendering program implements many nice-to-have features, including the ability to export the current render settings as an XML file, intuitive controls and various configuration settings. The user can easily change the colour palette, image resolution, the number of threads to use, and more.

This implementation also supports a multi-precision floating point type, which allows the renderer to zoom in "infinitely". Unfortunately, this renderer is written in Javascript rather than a faster language like Java or, better yet, C++; as a result, it can take a long time to render an image, especially at higher resolutions and quality settings.

Interface The interface for this renderer is quite primitive and, while intuitive, is not pleasant to use. The box select to zoom in can be frustrating to use and often results in poor framing since it zooms in immediately after releasing the mouse.

The status indicator shows the current render progress and is extremely limited. It shows the current pass of the renderer, the precision it is using and the number of rows completed, but there is nothing showing the estimated time remaining, the speed at which it is rendering, or the time elapsed since the render started.

Configuration While the renderer allows for the major settings to be changed, such as the maximum number of iterations to perform, the colour palette and the number of threads to use, many of the more advanced settings cannot be changed. For advanced applications, it is often useful to know *exactly* where the frame is centred in fractal space. An arguably more important feature is the ability to specify a location and scaling factor directly, instead of zooming in manually.

On the other hand, the ability to revert to default settings, combined with the lists of predefined settings for each option, make the program much more accessible for the less experienced. This feature is a must-have for a program aimed at a wide audience.

Render Quality While the user can specify the resolution at which to render the image, it must fit on the screen you are using. You cannot render a higher-resolution image to a file, for example. Furthermore, there are no options to configure anti-aliasing with this renderer, which means the image quality may not be as high as required for some purposes.

1.4.2 XaoS.js Online Mandelbrot Renderer [5]

The Fractal Foundation *XaoS.js* Mandelbrot renderer is relatively primitive, using only machine-precision floating point arithmetic and no options to configure the fractal. On the other hand, the renderer is intuitive, with a simple click-to-zoom interface, making it ideal for less knowledgeable users who want to explore fractals.

However, the *XaoS.js* renderer does implement some complex rendering algorithms. It maintains existing pixel information from the previous frame and uses it to refine the next one, creating a smoother transition progressively. Unfortunately, these rendering techniques result in significant artefacts when sufficiently zoomed in (though the limited number of iterations performed means the fractal is unrecognisable at this point).

1.4.3 XaoS Offline Fractal Renderer [6]

The Fractal Foundation *XaoS* fractal renderer is a much more advanced, offline version of the previously examined program. Since it is free and open-source, many people have contributed code and developed its rendering algorithms. As a result, it can render fractals much faster than other programs. However, the advanced techniques used cause artefacts to appear in the final renders, making it unsuitable for academic or research purposes since it is not a true likeness of the fractal.

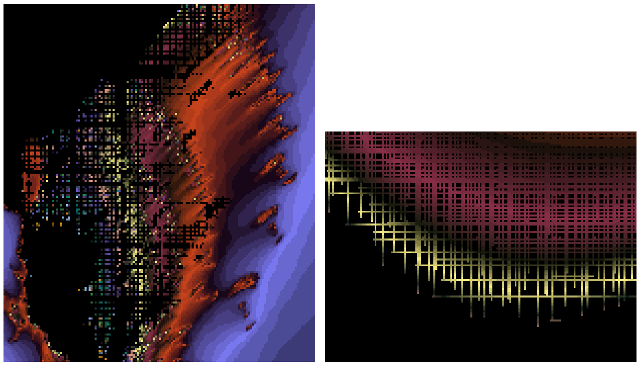


Figure 4: Artefacting in XaoS' renderer

Interface *XaoS* has the same intuitive controls as the online implementation, but all the configuration options are hidden in awkward menus at the top of the screen.

Configuration With enough searching, almost every parameter about the fractal can be changed, including filters such as edge detection and anti-aliasing. This is incredibly useful for advanced users, as it enables them to adjust the appearance of the fractal to their exact needs, emphasising the features they are investigating.

File Export *XaoS* allows the user to export a configuration file or save the current render as an image. The configuration file contains the information required to reconstruct the image in the renderer, simplifying the sharing of fractals between users. The image export saves the current pixels of the screen to a file, meaning the image's resolution is limited to the size of the window. This is unfortunate for those who may want to save a high-resolution image but cannot make the window large enough to support this.

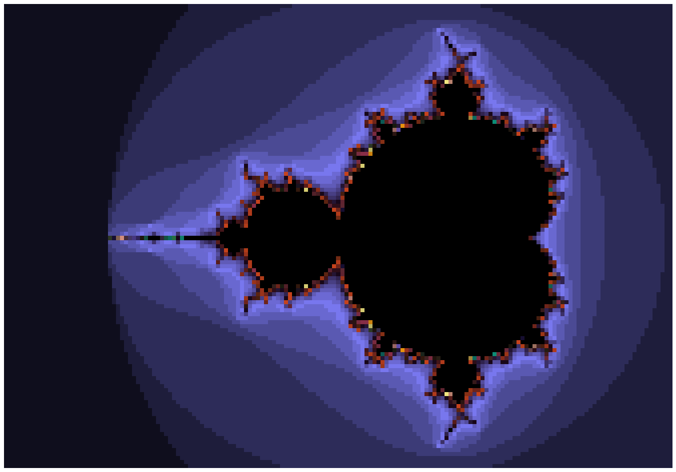


Figure 5: A low-quality image saved from XaoS

Supported Fractals The *XaoS* renderer supports 25 common fractals by default and can render simple user-defined fractals. While nice to have, this feature is unnecessary for a program to implement, assuming it supports at least two fractal types. Additionally, the user-defined fractals tend to render much slower than the built-in ones since optimised algorithms can be developed for them.

Precision *XaoS* uses fixed-precision arithmetic to perform calculations; hence, you cannot zoom into the fractals indefinitely. Given the performant nature of the program, it is unfortunate that this is not a feature since it could outcompete most other rendering programs.

1.5 Program Requirements

1. Rendering
2. Configuration
3. Interface and Movement
4. Import and Export
5. Installation

6. Performance

ID	Description	Importance
1.1	The program can render a fractal correctly	**HIGH**
1.2	Fractals can be rendered at high resolutions without artifacting	**HIGH**
1.3	The fractal can be coloured to bring out details	**HIGH**
1.4	Colouring algorithms can be isolated from the fractal rendering process, allowing different algorithms to be implemented more easily	**HIGH**
1.5	Anti-aliasing can be used to reduce noise and produce a cleaner image	**MEDIUM**
1.6	The background colour of the fractal can be changed	LOW
1.7	Fractal algorithms are optimised for the data type used	**MEDIUM**
1.8	Simple optimisations are made to accelerate the rate at which fractals are rendered	**MEDIUM**
1.9	Images can be rendered with high-precision floating point types, allowing for “infinite” zooms	**HIGH**
2.1	The area of the fractal currently being rendered, as well as the zoom factor, can be changed	**HIGH**
2.2	The number of threads used to render the fractal can be changed	**MEDIUM**
2.3	The maximum number of iterations allowed can be changed	**HIGH**
2.4	The bailout value can be changed	LOW
2.5	The anti-aliasing factor can be changed	**HIGH**
2.6	The image size can be changed independently of the image resolution	**HIGH**
2.7	The image resolution can be changed	**HIGH**
2.8	The colouring algorithm can be changed and customised	**MEDIUM**
2.9	Settings can be reset to default values	**HIGH**
2.10	The option to undo/redo changes to settings	LOW
2.11	Floating point precision can be customised	**HIGH**
2.12	Different fractals can be rendered	**MEDIUM**
2.13	Each fractal has predefined default settings which are loaded when a new fractal is selected	LOW

2.14	Settings are loaded from a JSON file at program startup	LOW
3.1	There is a graphical user interface (GUI)	**HIGH**
3.2	The GUI is fast and responsive	**HIGH**
3.3	Similar settings and options are contained in a single window which can be moved around the screen	**MEDIUM**
3.4	Input and numeric information fields should handle data to the current precision used by the program	**HIGH**
3.5	Not all menus are shown initially, and settings with different complexities can be shown or hidden	**MEDIUM**
3.6	Different workspaces can be selected from a menu, configuring the windows and settings shown for different levels of understanding – beginner, intermediate, advanced	LOW
3.7	The area to zoom into can be selected with the mouse	**HIGH**
3.8	The zoom box does not need to match the aspect ratio of the fractal	LOW
3.9	The zoom box can be moved and scaled after its creation, with the option to apply the zoom after the user is happy with it	**MEDIUM**
3.10	The current render progress, render time, render speed and estimated time remaining are displayed	**HIGH**
3.11	There is a history of previous frames rendered which can be reverted to	LOW
3.12	There is a way to zoom back out of the fractal	**HIGH**
3.13	The fractal should be rendered progressively, allowing the user to see roughly what is being rendered without having to wait for the full image	LOW
3.14	The current location can be copied to the clipboard easily	LOW
3.15	Any numeric input fields should accept scientific input formats	**HIGH**
4.1	The render configuration settings can be loaded from a JSON file at runtime	**MEDIUM**
4.2	The render configuration settings can be saved to a JSON file, allowing for easy sharing	**MEDIUM**
4.3	Images can be saved to a file	**HIGH**
4.4	The saved images can have a user-defined filetype, and are not limited to, for example, `*.png`	LOW

4.5	Images can be rendered separately from the main GUI, allowing incredibly high-resolution images to be saved	**POSSIBILTIY**
5.1	Program compiles on Windows	**HIGH**
5.2	Program compiles on MacOS	**HIGH**
5.3	Program compiles on Linux	**HIGH**
5.4	Program compiles with `msvc`	**HIGH**
5.5	Program compiles with `gcc/g++`	**HIGH**
5.6	Program compiles with `clang`	**HIGH**
5.7	Program is easy to compile from source with CMake (i.e. `cmake --build . --config Release`)	**HIGH**
5.8	Prebuilt executable is available for Windows	**MEDIUM**
5.9	Prebuilt executable is available for MacOS	**MEDIUM**
5.10	Prebuilt executable is available for Linux	**MEDIUM**
6.1	The program can render fractals in a reasonable time (under 2 seconds) on a single thread with machine word precision	**HIGH**
6.2	Multiple threads can be used to accelerate the rendering process	**HIGH**
6.3	The rendering algorithm runs on a separate thread to the GUI, ensuring the interface continues to refresh quickly	**HIGH**
6.4	Where possible, calculations are optimised to suit the data type being operated on	LOW
6.5	Some simple optimisations are implemented to accelerate the rendering of the fractals	**MEDIUM**
6.6	Low-quality images can be rendered quickly with multi-precision data types	**MEDIUM**

2 Design Phase

2.1 Third Party Libraries

Cinder [7] *Cinder* is a free, open-source graphics engine for C++. It provides a simple way to access OpenGL, ImGui and other tools, such as image loading and saving, optimised rendering in 2D and 3D, and more. I am using *Cinder* for this project instead of doing all the graphics processing with raw OpenGL because it dramatically simplifies the code and reduces the scope for hard-to-fix bugs.

This project uses a modified version of *Cinder* with updated libraries and a few extra features. Most significantly, this modified version includes a much newer version of ImGui and an altered build configuration to fix common compile errors on some platforms.

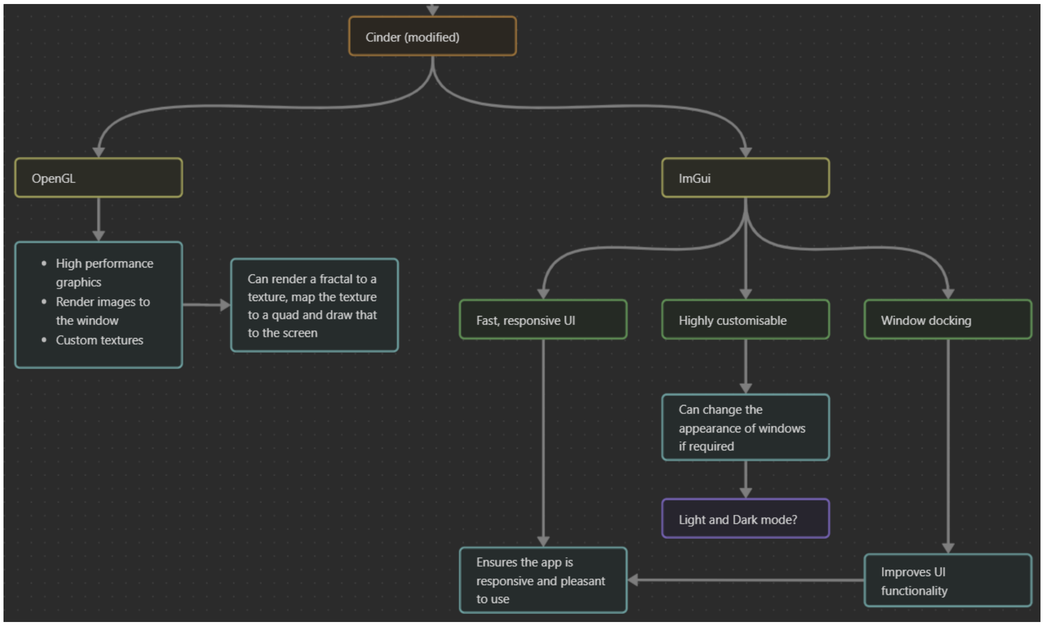
LibRapid [8] *LibRapid* is a high-performance library for mathematical applications, including optimised vector classes, complex number types and general mathematical functions. However, this library's most helpful feature is its support for MPIR and MPFR, which are highly-optimised multi-precision implementations. This will allow floating point calculations with more than 64 bits.

Incorporating an efficient multi-precision implementation into the project could allow for "infinite" fractal zooms since traditional floating-point limitations would no longer constrain the software.

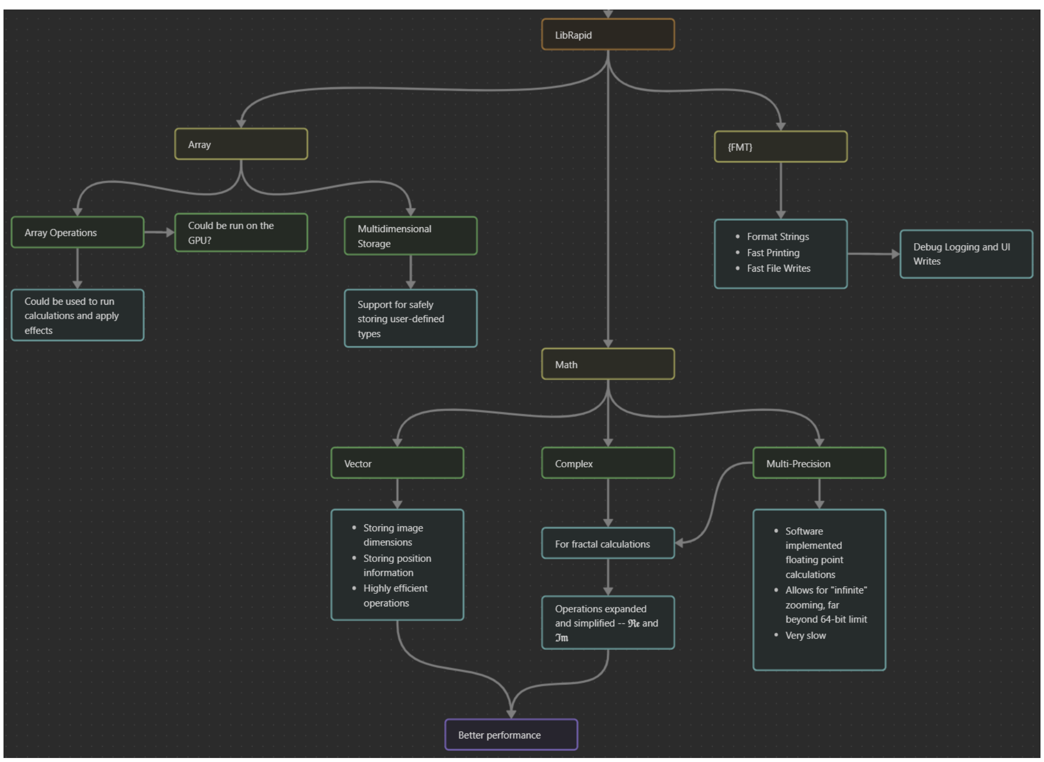
Another feature of *LibRapid* used heavily in this project is the compiler- and system-agnostic macro definitions. Useful features like inlining of functions, no-discard specifiers and more are not implemented by all compilers and sometimes work differently on different operating systems. LibRapid implements macros which automatically detect the relevant information and define the most suitable replacement. This isn't strictly required for the project, but it might result in a slight performance improvement and can help reduce bugs.

Cinderbox Both of the afore mentioned libraries are packaged with *Cinderbox* for simple integration into *CMake* projects.

2.2 Library Heirarchy



Cinder (modified)
OpenGL
ImGui
High performance graphics
Render images to the window
Custom textures
Can render a fractal to a texture, map the texture to a quad and draw that to the screen
Fast, responsive UI
Highly customisable
Window docking
Can change the appearance of windows if required
Light and Dark mode?
Ensures the app is responsive and pleasant to use
Improves UI functionality



LibRapid
Array
{FMT}
Array Operations
Could be run on the GPU?
Multidimensional Storage
Could be used to run calculations and apply effects
Support for safely storing user-defined types
Format Strings
Fast Printing
Fast File Writes
Debug Logging and UI Writes
Math
Vector
Complex
Multi-Precision
Storing image dimensions
Storing position information
Highly efficient operations
For fractal calculations
Operations expanded and simplified -- $\mathfrak{Re}$ and $\mathfrak{Im}$
Software implemented floating point calculations
Allows for "infinite" zooming, far beyond 64-bit limit
Very slow
Better performance

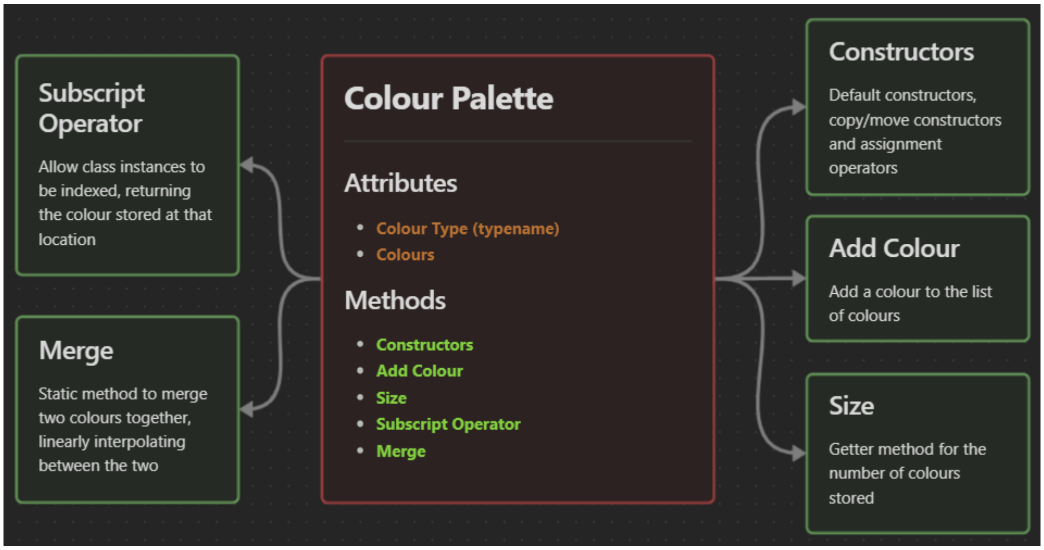
2.3 The Debug Logger

Since the program is written in C++and is a GUI application instead of a console application, there will not be a usable standard output to which debug information can be printed. To circumvent this issue, I will use a debug logger instance to write information to a file.

The logger's constructor will take a file path relative to the executable and attempt to open the specified file. If the document does not exist, it will be created for the user. The destructor will ensure that all buffers are flushed, and the file is closed. Without these checks, the program may terminate without saving the changes to be written to the file, and the debug log might be incomplete or corrupt. The logger will also have a priority level, optimising logging in release builds, as the user does not need all the information. For example, the logger could be configured to write only errors to the file in release mode. Finally, the logger has a function which enables the user to send data to be written to the file. New lines should be formatted appropriately, and logs should be timestamped. In addition to these functions, I will create a macro that captures the log statement's line number and filename, making tracebacks easier and faster during development.

2.4 Colour Palettes

A simple class containing a list of colours and a few helper methods is helpful for storing the colours and gradients used by the fractal rendering process. This class simplifies the act of colour palette generation and usage throughout the software, reducing bugs and improving the rate of development.



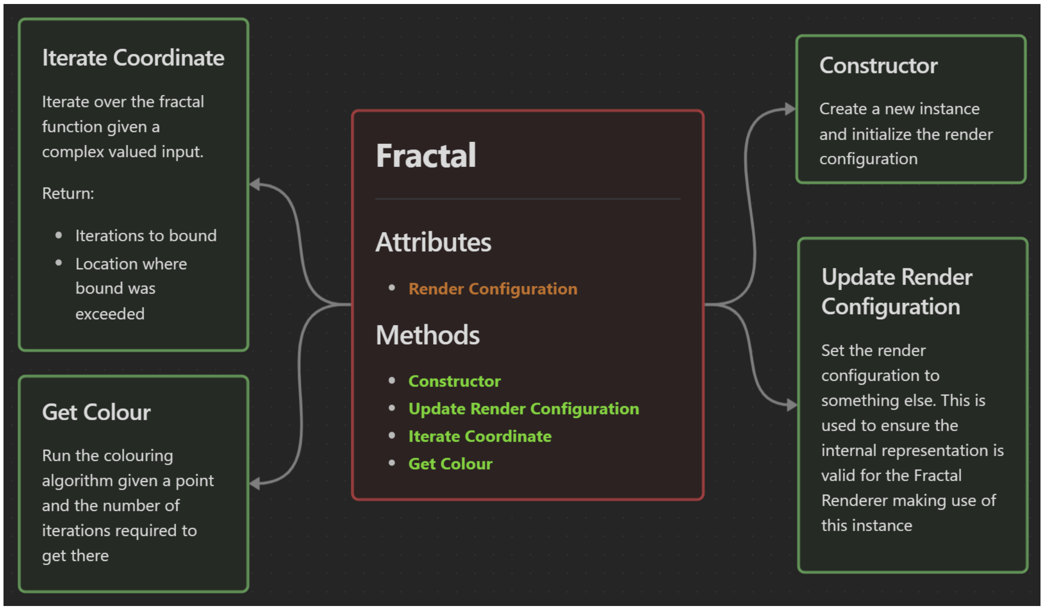
The colour merging function is a very simple static method of the class, which linearly interpolates between the colour's red, green and blue components.

$$\left\{ \quad t \times (R - r) + r, \quad t \times (G - g) + g, \quad t \times (B - b) + b \quad \right\} \tag{2}$$

Where t is the interpolation factor and $0 \leq t \leq 1$.

2.5 The Fractal Class

To support multiple fractal equations at runtime, each fractal will be implemented as a class inheriting from a main parent type. This is the fractal data type. It defines the functions required to iterate the fractal's equation from a given starting value, the logic to generate a colour from a starting point, an endpoint, and the number of iterations required to get there.



For example, the code below shows the definition of the Mandelbrot fractal class.

```
#pragma once

#include <fractal/genericFractal.hpp>

namespace frac {
    class Mandelbrot : public Fractal {
    public:
        /// Constructor taking a RenderConfig object
        /// \param config RenderConfig object
        explicit Mandelbrot(const RenderConfig &config);
        Mandelbrot(const Mandelbrot &)    = delete;
        Mandelbrot(Mandelbrot &&)     = delete;
        Mandelbrot &operator=(const Mandelbrot &) = delete;
        Mandelbrot &operator=(Mandelbrot &&) = delete;

        ~Mandelbrot() override = default;

        LIBRAPID_NODISCARD std::pair<int64_t, lrc::Complex<LowPrecision>>
        iterCoordLow(const lrc::Complex<LowPrecision> &coord) const
            override;

        LIBRAPID_NODISCARD std::pair<int64_t,
```

```
            lrc::Complex<HighPrecision>>
        iterCoordHigh(const lrc::Complex<HighPrecision> &coord) const
            override;
    };
} // namespace frac
```

2.5.1 Render Box States

An enum of valid states is required to keep track of each render box's current state. This is drawn to the main window on top of the fractal as it renders, providing the user with information about which areas are rendered, which are actively rendering and which areas are yet to be processed.

```
/// Represents the state of a render box
enum class RenderBoxState {
    None, // Not yet assigned a state
    Queued, // Queued to be rendered
    Rendering, // Currently being rendered
    Rendered // Rendered and ready to be written to the image
};
```

2.5.2 Render Boxes

The position information required to render a small area of the main fractal is contained within a `RenderBox` struct. These can be passed to a function inside the `FractalRenderer` class to be processed.

```
/// Stores the pixel-space coordinates of a region to render
struct RenderBox {
    lrc::Vec2i topLeft;
    lrc::Vec2i dimensions;
    RenderBoxState state = RenderBoxState::None;
    double renderTime = 0;
};
```

2.5.3 Render Box Statistics

To calculate the remaining time of the render and the fastest and slowest render box times, the program needs to know precisely how long each box took to render. To improve efficiency, a separate struct is created to store this data.

```
struct RenderBoxTimeStats {
    double min = 0;
    double max = 0;
    double average = 0;
    double remainingTime = 0;
};
```

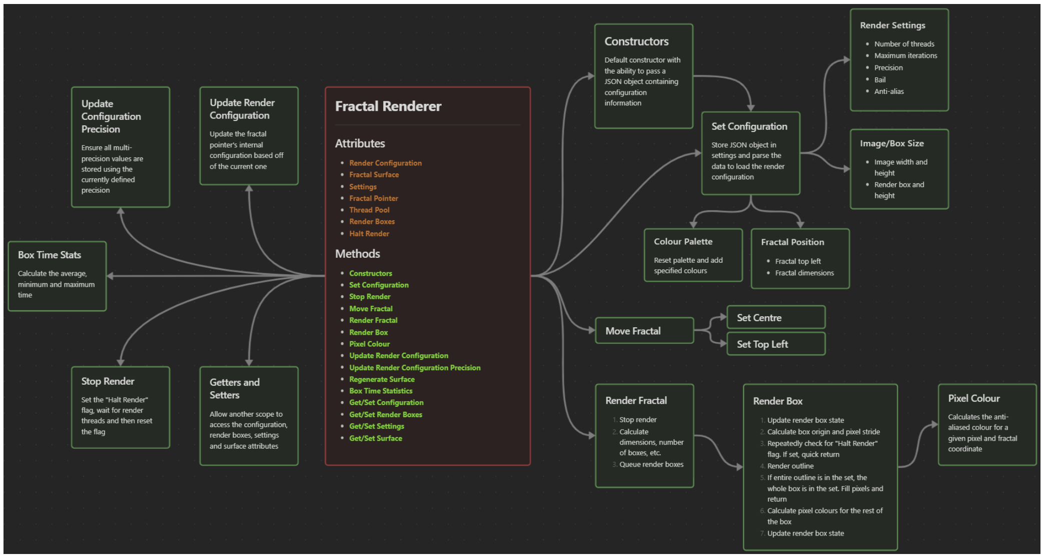
2.5.4 Render Configurations

Arguably the most important helper class, the `RenderConfig` struct contains all the information required for a `FractalRenderer` instance to render an image. The information in this struct can also be saved to a JSON file and shared, allowing people to send specific configurations between users easily.

```
struct RenderConfig {
    int64_t numThreads; // Number of threads to render on (max)
    int64_t maxIters; // Largest number of iterations to allow
    int64_t precision; // Precision (in bits) of floating point types
        used for arithmetic
    LowPrecision bail; // Bailout value
    int64_t antiAlias; // Anti-aliasing factor -- 1 = no anti-aliasing
    lrc::Vec2i imageSize; // Size of the image to render
    lrc::Vec2i boxSize; // Size of sub-regions to render (see RenderBox)
    lrc::Vec<HighPrecision, 2> fracTopLeft;  // The fractal-space center
        of the image
    lrc::Vec<HighPrecision, 2> fracSize;  // The width and height of the
        fractal space
    lrc::Vec<HighPrecision, 2> originalFracSize; // Original size for
        zoom factor calculation
    ColorPalette palette; // The palette to use for rendering the fractal
};
```

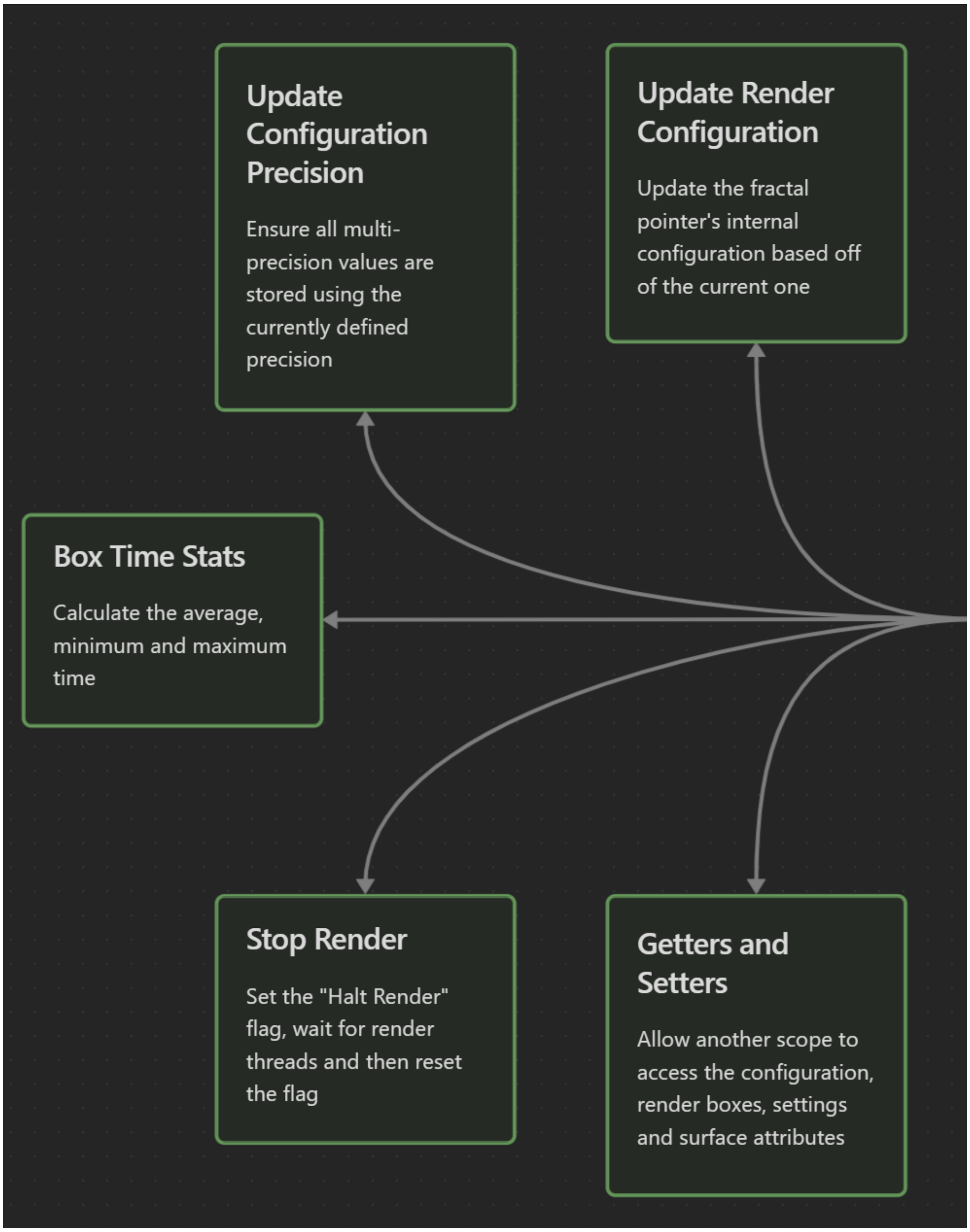
2.6 The Fractal Renderer Class

While it is essential to have a method of calculating the colour of a given point on the fractal (from the fractal class), it doesn't support rendering an entire image. This is the role of the fractal renderer class.



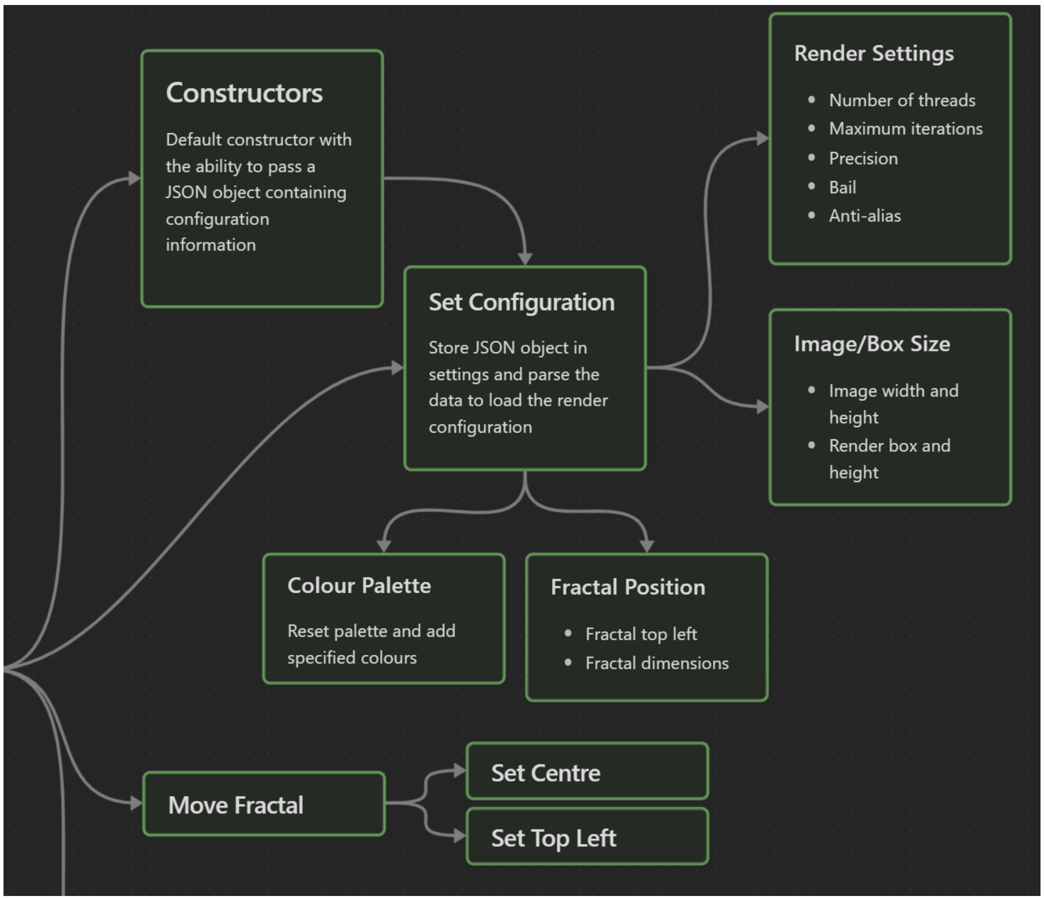
This class implements many functions at various levels of abstraction, allowing performance-critical sections of the code to be run with efficient, parallelised algorithms. At the same time, the high-level interfaces are easy to interact with and use.

2.6.1 Configuration, Getters, Setters and Statistics


Update Configuration Precision
Ensure all multi-precision values are stored using the currently defined precision
Update Render Configuration
Update the fractal pointer's internal configuration based off of the current one
Box Time Stats
Calculate the average, minimum and maximum time
Stop Render
Set the "Halt Render" flag, wait for render threads and then reset the flag
Getters and Setters
Allow another scope to access the configuration, render boxes, settings and surface attributes

These functions operate at a high level, allowing basic access to the information stored by the class. While simple, they are essential for the main program to function correctly, as there would be no way of accessing the rendered image, for example.

2.6.2 Constructors and Configurations



To render a fractal, much information is required about the dimensions of the image, the dimensions of the fractal, the origin in the complex plane, and more.

The fractal renderer class can parse a JSON object and load its configuration. By storing some data as a string instead of a number, it is possible to save and load high-precision numbers as well – this could be used to enable fractal locations with extremely high zoom factors to be saved and shared easily.

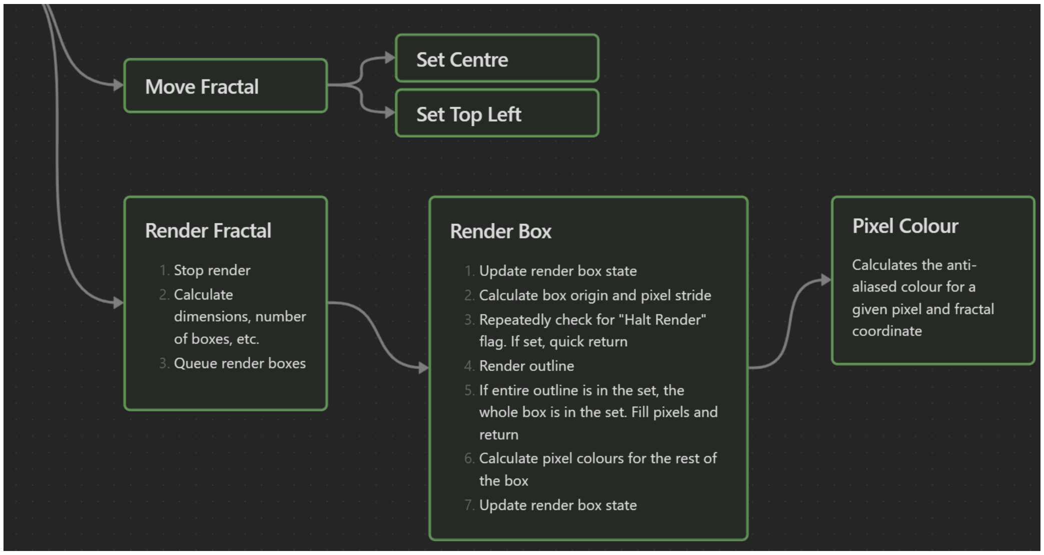
The JSON snippet below shows a highly simplified version of the default configuration used within the software (for a full version, see the code-listing at the end of the document).

```
{
    "renderConfig": {
        "numThreads": 8,
        "maxIters": 500,
        "precision": 64,
        "bail": 65536,
        "antiAlias": 2,
        "imageSize": {
            "width": 800,
            "height": 700
        },
        "colorPalette": [
            {
                "red": 0.5568628,
                "green": 0.23137255,
                "blue": 0.27450982,
                "alpha": 1.0
            },
            {
                "red": 0.88235295,
                "green": 0.8666667,
                "blue": 0.56078434,
                "alpha": 1.0
            }
        ]
    }
}
```

A vast number of configuration options can be configured in the file, including the threading options, render quality settings and even the size of the boxes to render in parallel.

The `loadConfiguration` method also enables the configuration to be changed or re-parsed at runtime, allowing quick and easy updates to the fractal settings.

2.6.3 Rendering Algorithms



The fractal renderer is responsible for creating an image buffer and setting the colour of each pixel in that image to represent the fractal at that location. Doing this efficiently is difficult, so the process is split into many small functions, providing fine-grained control over the algorithm.

Upon requesting the renderer to re-render the image, any existing render threads are halted, dependent image settings are recalculated, and the render-box queue is cleared.

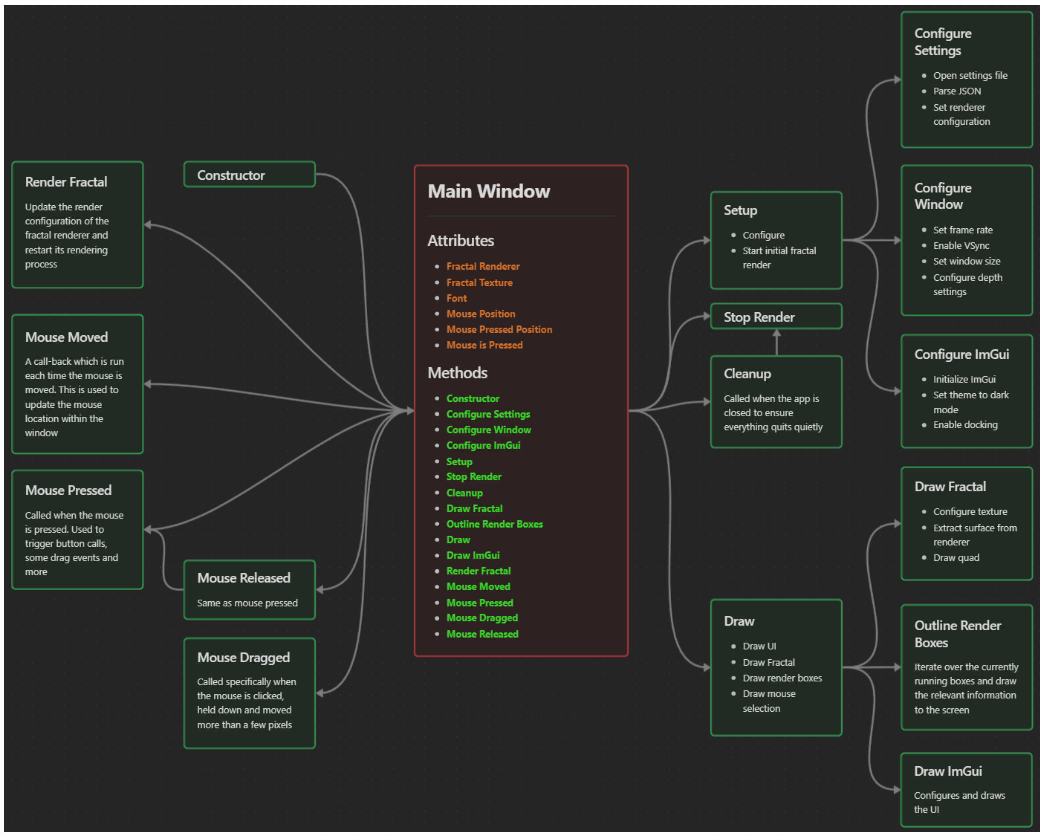
Next, the render boxes are recalculated and pushed back to the render queue. Each box is dequeued by a thread in a thread pool and runs in parallel.

To optimise the rendering of each box, we can use the fact that if an outline can be drawn where every point is in the set, every point contained within that outline must also be within the set. Outlining each box before calculating the inner area makes it possible to check whether all points were in the set. If they were, we can quickly fill the rest of the box without calculating the colour of each pixel.

To calculate the colour of each pixel, we first call the `iterCoord` method of the fractal pointer stored to get the number of iterations required to exceed the bailout value (if it is exceeded at all), as well as the first point at which this occurs. This information is then passed to the fractal's colouring algorithm to generate a colour for the pixel. If anti-aliasing is enabled, this process is repeated for multiple points within the pixel, and the resulting colours are averaged. Anti-aliasing produces smoother images that appear to be higher resolution, allowing for faster render times with high-quality results.

The fractal renderer class also implements routines to change the fractal-space coordinates to render. This is used to move the fractal when zooming in or out. For different use cases, there are methods to set the top left coordinate or the image's centre.

2.7 The Main Window



This class is responsible for creating, managing and drawing information to the window and controlling all the other classes mentioned previously.

When the window is created by *Cinder*, the `setup` routine is called, initializing the window's attributes. First, the settings JSON file is loaded, parsed and passed to the fractal renderer member. Next, the window itself is constructed and configured, including the framerate, enabling or disabling vertical syncing

(V-Sync), the window size and OpenGL depth buffer settings. Finally, *ImGui* is configured.

The `MainWindow` class is also responsible for drawing to the window. After setting the background colour, the *ImGui* windows are created and drawn. This produces most of the UI, but some extra parts must be drawn in later. Next, the fractal itself is drawn to the screen. An image texture is created and assigned the fractal image buffer, and a rectangle is drawn with the aforementioned texture. The rest of the UI is now drawn, including the render-box status indicators and the mouse selection.

When the application is requested to close, the `cleanup` routine is called. This signals any existing render threads to halt, ensures all members are correctly destructed and then destroys the window.

The window also has a variety of callback functions which are called when specific triggers occur. For example, functions are called every time the mouse is moved, dragged or pressed.

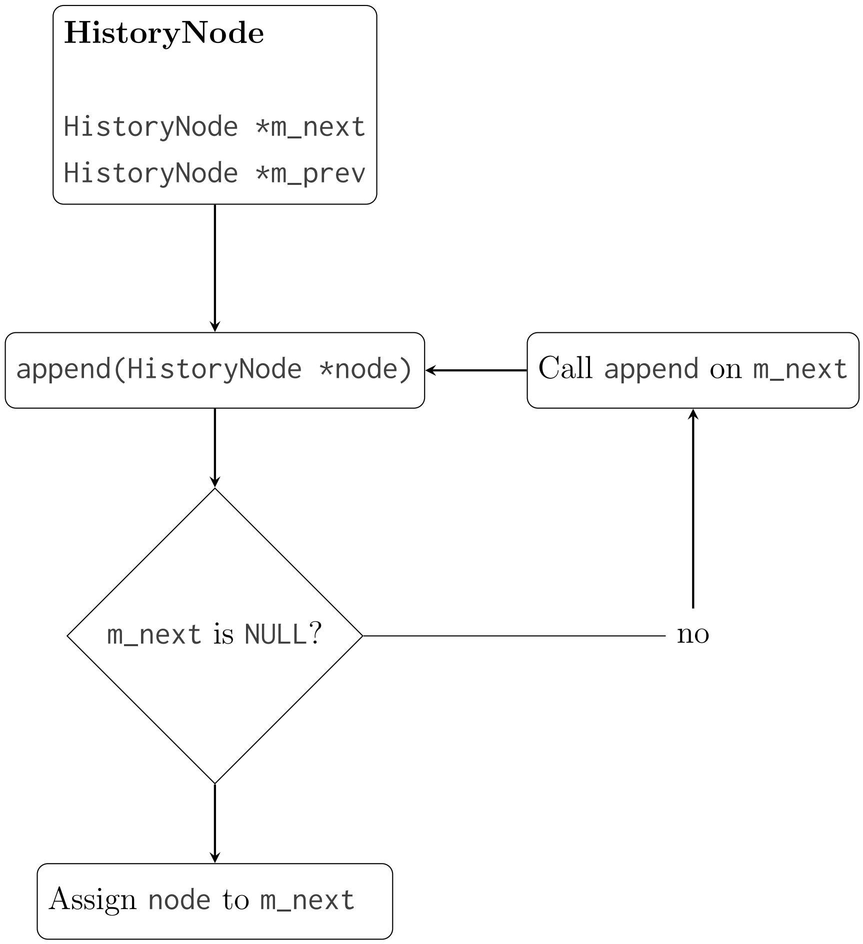
2.8 Additional Features

The features outlined previously comprise the vast majority of the program, but some additional features must also be designed and implemented. These are often the less critical features, though they still impact the user's interaction with the program.

2.8.1 Movement History

While we often take the "undo" and "redo" buttons for granted, they give us a powerful means of reverting unwanted changes. Furthermore, they allow the user to see what adjustments have been made to the fractal between movements, supporting verbal descriptions of how to arrive at fascinating points within the fractal.

Using a linked list, where each node stores a `RenderConfig` object and a `Surface`, it is possible to implement a move history into the program. Whenever the user requests to change a setting or move the origin of the fractal, the current surface and render configuration are appended to the history before the changes are applied.



2.8.2 Improved Selection Area

3 Technical Solution

3.1 Color Palette (colorPalette.hpp)

```cpp
#pragma once

namespace frac {
    class ColorPalette {
    public:
        using ColorType = lrc::Vec<float, 4>;

        ColorPalette()                          = default;
        ColorPalette(const ColorPalette &)      = default;
        ColorPalette(ColorPalette &&)           = default;
        ColorPalette &operator=(const ColorPalette &) = default;
        ColorPalette &operator=(ColorPalette &&) = default;

        /// Append a new colour to the palette
        /// \param color The colour to add
        void addColor(const ColorType &color);

        /// Return the number of colours in the palette
        /// \return The number of colours in the palette
        LIBRAPID_NODISCARD size_t size() const;

        /// Indexing operator (const)
        /// \param index The index of the colour to return
        /// \return The colour at the given index
        const ColorType &operator[](size_t index) const;

        /// Indexing operator (non-const)
        /// \param index The index of the colour to return
        /// \return The colour at the given index
        ColorType &operator[](size_t index);

        /// Linearly interpolate between two colours
        /// \param a First colour
        /// \param b Second colour
        /// \param t Interpolation factor
        /// \return The interpolated colour
        static ColorType merge(const ColorType &a, const ColorType &b,
            float t);

    private:
        std::vector<ColorType> m_colors;
```

```cpp
    };
} // namespace frac
```

3.2 Color Palette (colorPalette.cpp)

```cpp
#include <fractal/fractal.hpp>

namespace frac {
    void ColorPalette::addColor(const ColorType &color) {
        m_colors.push_back(color);
        FRAC_LOG(fmt::format("Adding Color: {} {} {} {}", color.x(),
            color.y(), color.z(), color.w()));
    }

    size_t ColorPalette::size() const { return m_colors.size(); }

    const ColorPalette::ColorType &ColorPalette::operator[](size_t
        index) const {
        if (index > m_colors.size())
            FRAC_ERROR(fmt::format(
              "Index {} out of bounds for ColorPalette with size {}",
                  index, m_colors.size()));
        return m_colors[index];
    }

    ColorPalette::ColorType &ColorPalette::operator[](size_t index) {
        if (index > m_colors.size())
            FRAC_ERROR(fmt::format(
              "Index {} out of bounds for ColorPalette with size {}",
                  index, m_colors.size()));
        return m_colors[index];
    }

    ColorPalette::ColorType ColorPalette::merge(const ColorType &a,
        const ColorType &b, float t) {
        return a + (b - a) * t;
    }
} // namespace frac
```

3.3 Debug Logger (debug.hpp)

```cpp
#pragma once

#define FRAC_LOG(message) \
    ::frac::debugLogger.write(message, ::frac::Priority::Info, FILENAME,
        __LINE__)
#define FRAC_WARN(message) \
    ::frac::debugLogger.write(message, ::frac::Priority::Warning,
        FILENAME, __LINE__)
#define FRAC_ERROR(message) \
    ::frac::debugLogger.write(message, ::frac::Priority::Error,
        FILENAME, __LINE__)

namespace frac {
    // A way to specify the verbosity of the debug logger
    enum class Priority { Info = 0, Warning = 5, Error = 10 };

    /// A logger type that can write debug information to a file
    class DebugLogger {
    public:
        /*
         * Since there should only ever be a single DebugLogger
             instance, delete the
         * majority of the constructors and assignment operators as they
             are not needed
         */

        DebugLogger()                           = delete;
        DebugLogger(const DebugLogger &)        = delete;
        DebugLogger(DebugLogger &&)             = delete;
        DebugLogger &operator=(const DebugLogger &) = delete;
        DebugLogger &operator=(DebugLogger &&)  = delete;

        /// Create a new DebugLogger instance from a filename
        /// \param filename
        explicit DebugLogger(const std::string &filename,
                             Priority priority = Priority::Info);

        /// Close the file stream on destruction
        ~DebugLogger();

        /// Set the priority level of the debugger's logs. Higher
            priorities will be
        /// logged in release mode, while lower priorities will only be
```

```
            logged in debug
        /// mode
        /// \param newPriority The new priority level
        void setPriorityLevel(Priority newPriority) { m_priority = newPriority; }

        /// Write a message to the log file
        /// \param message The message to write
        /// \param priority The priority of the message (see Priority)
        /// \param filename The filename of the file that called this function
        /// \param line The line number of the file that called this function
        void write(const std::string &message, Priority priority,
                   const std::string &filename, int64_t line);

    private:
        std::fstream m_log;     // File stream
        double m_startTime;     // Time at which the logger was created
        Priority m_priority = Priority::Info; // Priority level of the logger
    };

    extern DebugLogger debugLogger;
} // namespace frac
```

3.4 Debug Logger (**debug.cpp**)

```
#include <fractal/fractal.hpp>

namespace frac {
    DebugLogger::DebugLogger(const std::string &filename, Priority priority) {
        m_log.open(filename, std::fstream::out);
        m_startTime = lrc::now();

        m_log << "============[ FRACTAL RENDERER DEBUG LOG ]============\n" << std::endl;
    }

    DebugLogger::~DebugLogger() {
        m_log << "\n============[ FRACTAL RENDERER DEBUG LOG ]============";
        m_log.flush();
        m_log.close();
```

```cpp
}

void DebugLogger::write(const std::string &message, Priority priority,
                        const std::string &filename, int64_t line) {
    if (static_cast<size_t>(priority) < static_cast<size_t>(m_priority)) return;

    double time      = lrc::now();
    constexpr size_t maxFilenameLength = 40;
    std::string truncatedFilename;
    std::string priorityString;
    std::string cleanedMessage;

    if (filename.size() > maxFilenameLength)
        truncatedFilename =
          "..." + filename.substr(filename.size() - maxFilenameLength + 3, maxFilenameLength);
    else
        truncatedFilename = filename;

    switch (priority) {
        case Priority::Info: priorityString = "INFO"; break;
        case Priority::Warning: priorityString = "WARNING"; break;
        case Priority::Error: priorityString = "ERROR"; break;
    }

    // Pad new lines with 26 spaces to align with the beginning of the message
    // in the log file.
    int64_t preambleLength = 26 + maxFilenameLength;
    for (char c : message) {
        cleanedMessage += c;
        if (c == '\n') cleanedMessage += std::string(preambleLength, ' ');
    }

    // Use std::endl here to force-flush the buffer -- this is the only way
    // to ensure that the log is written to disk in the event of a crash
    m_log << fmt::format("[ {:.5f} ] {:>7} {:>{}}:{:0>4} {}",
                         time - m_startTime,
                         priorityString,
                         truncatedFilename,
```

```
                           maxFilenameLength,
                           line,
                           cleanedMessage)
             << std::endl;
    }

#if defined(LIBRAPID_DEBUG) // More reliable than NDEBUG
    DebugLogger debugLogger("./log.txt", Priority::Info); // Log all
        messages
#else
    DebugLogger debugLogger("./log.txt", Priority::Warning); // Only
        warnings and above
#endif
} // namespace frac
```

3.5 Fractal Base Class Implementation (**fractal.hpp**)

```cpp
#pragma once

#include <cinderbox/cinderbox.hh>
#include <librapid>
#include <fstream>
#include <nlohmann/json.hpp>
#include <BS_thread_pool.hpp>

#ifndef FRACTAL_SETTINGS_PATH
# define FRACTAL_UI_SETTINGS_PATH FRACTAL_RENDERER_ROOT_DIR "/settings/settings.json"
#endif

namespace lrc = librapid;

using ThreadPool = BS::thread_pool;
using json  = nlohmann::json;

namespace frac {
    using HighPrecision = lrc::mpf;
    using LowPrecision = double;

    using HighVec2 = lrc::Vec<HighPrecision, 2>;
    using LowVec2 = lrc::Vec<LowPrecision, 2>;
} // namespace frac

#include <fractal/debug.hpp>
#include <fractal/colorPalette.hpp>
#include <fractal/openglUtils.hpp>
#include <fractal/renderConfig.hpp>
#include <fractal/genericFractal.hpp>
#include <fractal/mandelbrot.hpp>
#include <fractal/fractalRenderer.hpp>
#include <fractal/history.hpp>
#include <fractal/mainWindow.hpp>
```

3.6 Fractal Renderer Definition (**fractalRenderer.hpp**)

```cpp
#pragma once

namespace frac {
    class FractalRenderer {
    public:
        FractalRenderer() = default;

        /// Construct a new renderer object from a JSON config object
        /// \param config The JSON config object
        explicit FractalRenderer(const json &config);

        ~FractalRenderer();

        /// Set the fractal renderer config
        /// \param config JSON object
        void setConfig(const json &config);

        /// Stop the renderer gracefully and wait for all threads to rejoin main
        void stopRender();

        /// Set the complex-valued coordinate of the top-left corner of the fractal and
        /// its size
        /// \param topLeft Top-left corner
        /// \param size Size of the fractal
        /// \see moveFractalCenter
        void moveFractalCorner(const lrc::Vec<HighPrecision, 2> &topLeft,
                               const lrc::Vec<HighPrecision, 2> &size);

        /// Set the complex-valued coordinate of the center of the fractal and its size
        /// \param center Center of the fractal
        /// \param size Size of the fractal
        /// \see moveFractalCorner
        void moveFractalCenter(const lrc::Vec<HighPrecision, 2> &center,
                               const lrc::Vec<HighPrecision, 2> &size);

        /// Render the fractal into the fractal surface, and copy that to the
        /// fractal surface to be drawn. This will be executed on a separate thread
        /// in order to keep the UI updating
```

```
        void renderFractal();

        /// Render a sub-section of the fractal, defined by the \p box variable. This is
        /// intended to be used within the call queue to render multiple sections in
        /// parallel
        /// \param box The box configuration
        /// \param boxIndex Box ID (for updating states)
        void renderBox(const RenderBox &box, int64_t boxIndex = -1);

        /// Calculate the colour of a pixel at standard-precision. This implements
        /// anti-aliasing as well
        /// \param pixPos Pixel-space coordinate
        /// \param aliasFactor Anti-aliasing factor
        /// \param step Step size
        /// \param aliasStepCorrect Anti-aliasing step correction
        /// \return Color of the pixel
        ci::ColorA pixelColorLow(const LowVec2 &pixPos, int64_t aliasFactor,
                                 const LowVec2 &step, const LowVec2 &aliasStepCorrect);

        /// Calculate the colour of a pixel at high-precision. See pixelColorLow
        /// \param pixPos Pixel-space coordinate
        /// \param aliasFactor Anti-aliasing factor
        /// \param step Step size
        /// \param aliasStepCorrect Anti-aliasing step correction
        /// \return Color of the pixel
        /// \see pixelColorLow
        ci::ColorA pixelColorHigh(const HighVec2 &pixPos, int64_t aliasFactor,
                                  const HighVec2 &step, const HighVec2 &aliasStepCorrect);

        /// Update the render configuration of the internal fractal pointer
        void updateRenderConfig();

        /// Ensure all values are using the highest precision possible
        void updateConfigPrecision();

        /// Regenerate the surfaces and resize them to fit the image size
```

```cpp
        void regenerateSurface();

        /// Getter method for the render box time statistics
        /// \return Statistics
        LIBRAPID_NODISCARD RenderBoxTimeStats boxTimeStats() const;

        /// Constant getter method for the render configuration
        /// \return Render configuration
        LIBRAPID_NODISCARD const RenderConfig &config() const;

        /// Non-const getter method for the render configuration
        /// \return Render configuration
        LIBRAPID_NODISCARD RenderConfig &config();

        /// Constant getter method for the internal render box vector
        /// \return Render box vector
        LIBRAPID_NODISCARD const std::vector<RenderBox> &renderBoxes() const;

        /// Non-const getter method for the internal render box vector
        /// \return Render box vector
        LIBRAPID_NODISCARD std::vector<RenderBox> &renderBoxes();

        /// Constant getter method for the internal settings object
        /// \return Settings object
        LIBRAPID_NODISCARD const json &settings() const;

        /// Non-const getter method for the internal settings object
        /// \return Settings object
        LIBRAPID_NODISCARD json &settings();

        /// Constant getter method for the internal surface
        /// \return Surface
        LIBRAPID_NODISCARD const ci::Surface &surface() const;

        /// Non-const getter method for the internal surface
        /// \return Surface
        LIBRAPID_NODISCARD ci::Surface &surface();

    private:
        RenderConfig m_renderConfig;  // The settings for the fractal renderer
        ci::Surface m_fractalSurface;  // The surface that the fractal is rendered to
        json m_settings;     // The settings for the fractal
```

```
        std::unique_ptr<Fractal> m_fractal; // The fractal to render
        ThreadPool m_threadPool;   // Pool for render threads

        std::vector<RenderBox> m_renderBoxes; // The state of each render box

        bool m_haltRender = false; // Used to gracefully stop the render threads
    };
} // namespace frac
```

3.7 Fractal Renderer Implementation (fractalRenderer.cpp)

```
#include <fractal/fractal.hpp>

namespace frac {
    FractalRenderer::FractalRenderer(const json &config) { setConfig(config); }
    FractalRenderer::~FractalRenderer() { stopRender(); }

    void FractalRenderer::setConfig(const json &config) {
        m_settings = config;

        try {
            // Set the default precision
            lrc::prec2(m_settings["renderConfig"]["precision"].get<int64_t>());

            // Load settings from settings JSON object
            m_renderConfig = RenderConfig {
              m_settings["renderConfig"]["numThreads"].get<int64_t>(),
              m_settings["renderConfig"]["maxIters"].get<int64_t>(),
              m_settings["renderConfig"]["precision"].get<int64_t>(),
              m_settings["renderConfig"]["bail"].get<LowPrecision>(),
              m_settings["renderConfig"]["antiAlias"].get<int>(),

              lrc::Vec2i(
                m_settings["renderConfig"]["imageSize"]["width"].get<int64_t>(),
                m_settings["renderConfig"]["imageSize"]["height"].get<int64_t>()),
              lrc::Vec2i(m_settings["renderConfig"]["boxSize"]["width"].get<int64_t>(),
                         m_settings["renderConfig"]["boxSize"]["height"].get<int64_t>()),

              lrc::Vec<HighPrecision, 2>(
                m_settings["renderConfig"]["fracTopLeft"]["Re"].get<float>(),
                m_settings["renderConfig"]["fracTopLeft"]["Im"].get<float>()),
              lrc::Vec<HighPrecision, 2>(
```

```
                m_settings["renderConfig"]["fracSize"]["Re"].get<float>(),
                m_settings["renderConfig"]["fracSize"]["Im"].get<float>()),
              lrc::Vec<HighPrecision, 2>(0, 0),

              ColorPalette(), // Default for now -- colors added later

              m_settings["renderConfig"]["draftRender"].get<bool>(),
              m_settings["renderConfig"]["draftInc"].get<int64_t>()};

            m_renderConfig.originalFracSize = m_renderConfig.fracSize;

            // Load the colour palette from the JSON object
            for (const auto &color :
                 m_settings["renderConfig"]["colorPalette"]) {
                m_renderConfig.palette.addColor(
                  ColorPalette::ColorType(color["red"].get<float>(),
                                          color["green"].get<float>(),
                                          color["blue"].get<float>(),
                                          color["alpha"].get<float>()));
            }

            m_fractal = std::make_unique<Mandelbrot>(m_renderConfig);
        } catch (std::exception &e) {
            FRAC_LOG(fmt::format("Failed to load settings: {}",
                 e.what()));
            stopRender();
        }
    }

    void FractalRenderer::stopRender() {
        m_haltRender = true;
        m_threadPool.wait_for_tasks();
        m_haltRender = false;
    }

    void FractalRenderer::moveFractalCorner(const
        lrc::Vec<HighPrecision, 2> &topLeft,
                                            const lrc::Vec<HighPrecision, 2>
                                                &size) {
        m_renderConfig.fracTopLeft = topLeft;
        m_renderConfig.fracSize = size;
        m_fractal->updateRenderConfig(m_renderConfig);
    }

    void FractalRenderer::moveFractalCenter(const
```

```
        lrc::Vec<HighPrecision, 2> &center,
                                        const lrc::Vec<HighPrecision, 2>
                                            &size) {
        moveFractalCorner(center - size / lrc::Vec<HighPrecision, 2>(2,
            2), size);
    }

    void FractalRenderer::renderFractal() {
        if (m_threadPool.get_tasks_queued() > 0) {
            FRAC_WARN("Render already in progress. Halting...");
            m_haltRender = true;
            m_threadPool.wait_for_tasks();
            m_haltRender = false;
            FRAC_LOG("Render halted");
        }

        FRAC_LOG("Rendering Fractal...");

        m_renderBoxes.clear();
        m_threadPool.reset(m_renderConfig.numThreads);

        // Split the render into boxes to be rendered in parallel
        auto imageSize = m_renderConfig.imageSize;
        auto boxSize = m_renderConfig.boxSize;

        // Round number of boxes up so the full image is covered
        auto numBoxes =
          lrc::Vec2i(lrc::ceil(lrc::Vec2f(imageSize) /
              lrc::Vec2f(boxSize)));

        m_renderBoxes.reserve(numBoxes.x() * numBoxes.y());

        // Iterate over all boxes
        for (int64_t i = 0; i < numBoxes.y(); ++i) {
            for (int64_t j = 0; j < numBoxes.x(); ++j) {
                lrc::Vec2i adjustedBoxSize(
                  lrc::min(boxSize.x(), imageSize.x() - j * boxSize.x()),
                  lrc::min(boxSize.y(), imageSize.y() - i * boxSize.y()));
                RenderBox box {lrc::Vec2i(j, i) * boxSize,
                               adjustedBoxSize,
                               m_renderConfig.draftRender,
                               m_renderConfig.draftInc,
                               RenderBoxState::Queued};

                auto prevSize = (int64_t)m_renderBoxes.size();
```

```

      // Must happen before pushing to render queue
      m_renderBoxes.emplace_back(box);

      m_threadPool.push_task(
        [this, box, prevSize]() { renderBox(box, prevSize); });
    }
  }

  FRAC_LOG("Fractal Complete...");
}

void FractalRenderer::renderBox(const RenderBox &box, int64_t boxIndex) {
  // Update the render box state
  m_renderBoxes[boxIndex].state = RenderBoxState::Rendering;
  const double start   = lrc::now();

  const int64_t inc = box.draftRender ? box.draftInc : 1;

  HighVec2 fractalOrigin = lrc::map(
    static_cast<HighVec2>(box.topLeft),
    HighVec2({0, 0}),
    static_cast<HighVec2>(m_renderConfig.imageSize),
    m_renderConfig.fracTopLeft,
    m_renderConfig.fracTopLeft + static_cast<HighVec2>(m_renderConfig.fracSize));

  HighVec2 step =
    m_renderConfig.fracSize / static_cast<HighVec2>(m_renderConfig.imageSize);

  int64_t aliasFactor = m_renderConfig.antiAlias;
  if (box.draftRender) aliasFactor = 1; // No anti-aliasing for drafts

  HighPrecision scaleFactor =
    HighPrecision(1) / static_cast<HighPrecision>(aliasFactor);
  HighVec2 aliasStepCorrect(scaleFactor, scaleFactor);

  bool blackEdges = true; // Assume edges are black to begin with

  if (m_haltRender) return;

  if (box.draftRender) {
```

```
            for (int64_t py = box.topLeft.y(); py < box.topLeft.y() + box.dimensions.y();
                 ++py) {
                for (int64_t px = box.topLeft.x();
                     px < box.topLeft.x() + box.dimensions.x();
                     ++px) {
                    m_fractalSurface.setPixel(lrc::Vec2i(px, py),
                                              ci::ColorA {0.2, 0, 0.2, 0.5});
                }
            }
        }

        // Render the top edge
        for (int64_t px = box.topLeft.x(); px < box.topLeft.x() + box.dimensions.x();
             px += inc) {
            // Anti-aliasing
            auto pixPos = fractalOrigin + step * HighVec2(px - box.topLeft.x(), 0);

            ci::ColorA pix;

            if (m_renderConfig.precision <= 64) {
                pix = pixelColorLow(pixPos, aliasFactor, step, aliasStepCorrect);
            } else {
                pix = pixelColorHigh(pixPos, aliasFactor, step, aliasStepCorrect);
            }

            if (blackEdges && (pix.r != 0 && pix.g != 0 && pix.b != 0)) {
                blackEdges = false;
            }

            m_fractalSurface.setPixel(lrc::Vec2i(px, box.topLeft.y()), pix);
        }

        if (m_haltRender) return;

        // Render the right edge
        for (int64_t py = box.topLeft.y(); py < box.topLeft.y() + box.dimensions.y();
             py += inc) {
            // Anti-aliasing
```

```
            auto pixPos =
              fractalOrigin + step * HighVec2(box.dimensions.x(), py -
                  box.topLeft.y());

            ci::ColorA pix;

            if (m_renderConfig.precision <= 64) {
                pix = pixelColorLow(pixPos, aliasFactor, step,
                    aliasStepCorrect);
            } else {
                pix = pixelColorHigh(pixPos, aliasFactor, step,
                    aliasStepCorrect);
            }

            if (blackEdges && (pix.r != 0 && pix.g != 0 && pix.b != 0)) {
                blackEdges = false;
            }

            m_fractalSurface.setPixel(
              lrc::Vec2i(box.topLeft.x() + box.dimensions.x() - 1, py),
                  pix);
        }

        if (m_haltRender) return;

        // Render the bottom edge
        for (int64_t px = box.topLeft.x(); px < box.topLeft.x() +
            box.dimensions.x();
             px += inc) {
            // Anti-aliasing
            auto pixPos = fractalOrigin +
                          step * HighVec2(px - box.topLeft.x(),
                              box.dimensions.y() - 1);

            ci::ColorA pix;

            if (m_renderConfig.precision <= 64) {
                pix = pixelColorLow(pixPos, aliasFactor, step,
                    aliasStepCorrect);
            } else {
                pix = pixelColorHigh(pixPos, aliasFactor, step,
                    aliasStepCorrect);
            }

            if (blackEdges && (pix.r != 0 && pix.g != 0 && pix.b != 0)) {
```

```
                blackEdges = false;
            }

            m_fractalSurface.setPixel(
              lrc::Vec2i(px, box.topLeft.y() + box.dimensions.y() - 1),
                  pix);
        }

        if (m_haltRender) return;

        // Render the left edge
        for (int64_t py = box.topLeft.y(); py < box.topLeft.y() +
            box.dimensions.y();
             py += inc) {
            // Anti-aliasing
            auto pixPos =
              fractalOrigin +
              step * HighVec2(box.topLeft.x() - box.topLeft.x(), py -
                  box.topLeft.y());

            ci::ColorA pix;

            if (m_renderConfig.precision <= 64) {
                pix = pixelColorLow(pixPos, aliasFactor, step,
                    aliasStepCorrect);
            } else {
                pix = pixelColorHigh(pixPos, aliasFactor, step,
                    aliasStepCorrect);
            }

            if (blackEdges && (pix.r != 0 && pix.g != 0 && pix.b != 0)) {
                blackEdges = false;
            }

            m_fractalSurface.setPixel(lrc::Vec2i(box.topLeft.x(), py),
                pix);
        }

        if (blackEdges) {
            for (int64_t py = box.topLeft.y() + 1;
                 py < box.topLeft.y() + box.dimensions.y() - 1;
                 py += inc) {
                for (int64_t px = box.topLeft.x() + 1;
                     px < box.topLeft.x() + box.dimensions.x() - 1;
                     px += inc) {
```

```
                m_fractalSurface.setPixel(lrc::Vec2i(px, py),
                                         ci::ColorA {0, 0, 0, 1});
            }
        }
    } else {
        // Make the primary axis of iteration the x-axis to improve cache
        // efficiency and increase performance.
        for (int64_t py = box.topLeft.y() + 1;
             py < box.topLeft.y() + box.dimensions.y() - 1;
             py += inc) {
            // Quick return if required. Without this, the
            // render threads will continue running after the
            // application is closed, leading to weird behaviour.
            if (m_haltRender) return;

            for (int64_t px = box.topLeft.x() + 1;
                 px < box.topLeft.x() + box.dimensions.x() - 1;
                 px += inc) {
                // Anti-aliasing
                auto pixPos = fractalOrigin + step * HighVec2(px - box.topLeft.x(),
                                                              py - box.topLeft.y());

                ci::ColorA pix;

                if (m_renderConfig.precision <= 64) {
                    pix = pixelColorLow(pixPos, aliasFactor, step, aliasStepCorrect);
                } else {
                    pix = pixelColorHigh(pixPos, aliasFactor, step, aliasStepCorrect);
                }

                m_fractalSurface.setPixel(lrc::Vec2i(px, py), pix);
            }
        }
    }

    // Update the render box state
    m_renderBoxes[boxIndex].state = RenderBoxState::Rendered;
    m_renderBoxes[boxIndex].renderTime = lrc::now() - start;
}
```

```cpp
ci::ColorA FractalRenderer::pixelColorLow(const LowVec2 &pixPos, int64_t aliasFactor,
                                          const LowVec2 &step,
                                          const LowVec2 &aliasStepCorrect) {
    ci::ColorA pix(0, 0, 0, 1);

    for (int64_t aliasY = 0; aliasY < aliasFactor; ++aliasY) {
        for (int64_t aliasX = 0; aliasX < aliasFactor; ++aliasX) {
            auto pos = pixPos + step * LowVec2(aliasX, aliasY) * aliasStepCorrect;

            auto [iters, endPoint] =
              m_fractal->iterCoordLow(lrc::Complex<LowPrecision>(pos.x(), pos.y()));
            if (endPoint.real() * endPoint.real() +
                endPoint.imag() * endPoint.imag() <
              4) {
              pix += ci::ColorA(0, 0, 0, 1); // Probably in the set
            } else {
              pix +=
                m_fractal->getColorLow(endPoint, iters); // Probably not in the set
            }
        }
    }

    return pix / static_cast<float>(aliasFactor * aliasFactor);
}

ci::ColorA FractalRenderer::pixelColorHigh(const HighVec2 &pixPos,
                                           int64_t aliasFactor, const HighVec2 &step,
                                           const HighVec2 &aliasStepCorrect) {
    ci::ColorA pix(0, 0, 0, 1);

    for (int64_t aliasY = 0; aliasY < aliasFactor; ++aliasY) {
        for (int64_t aliasX = 0; aliasX < aliasFactor; ++aliasX) {
            auto pos = pixPos + step * HighVec2(aliasX, aliasY) * aliasStepCorrect;

            auto [iters, endPoint] =
              m_fractal->iterCoordHigh(lrc::Complex<HighPrecision>(pos.x(), pos.y()));
```

```
                if (endPoint.real() * endPoint.real() +
                      endPoint.imag() * endPoint.imag() <
                    4) {
                    pix += ci::ColorA(0, 0, 0, 1); // Probably in the set
                } else {
                    pix +=
                      m_fractal->getColorHigh(endPoint, iters); // Probably not in the set
                }
            }
        }

        return pix / static_cast<float>(aliasFactor * aliasFactor);
    }

    void FractalRenderer::updateRenderConfig() {
        m_fractal->updateRenderConfig(m_renderConfig);
    }

    void FractalRenderer::regenerateSurface() {
        FRAC_LOG("Regenerating Surface...");
        int64_t w  = m_renderConfig.imageSize.x();
        int64_t h  = m_renderConfig.imageSize.y();
        m_fractalSurface = ci::Surface((int32_t)w, (int32_t)h, true);
        FRAC_LOG("Surface regenerated");
    }

    void FractalRenderer::updateConfigPrecision() {
        int64_t prec = m_renderConfig.precision;
        HighPrecision highPrecTopLeftX(m_renderConfig.fracTopLeft.x(), prec);
        HighPrecision highPrecTopLeftY(m_renderConfig.fracTopLeft.y(), prec);
        HighPrecision highPrecFracSizeX(m_renderConfig.fracSize.x(), prec);
        HighPrecision highPrecFracSizeY(m_renderConfig.fracSize.y(), prec);
        HighPrecision highPrecOriginalFracSizeX(m_renderConfig.originalFracSize.x(),
                                                prec);
        HighPrecision highPrecOriginalFracSizeY(m_renderConfig.originalFracSize.y(),
                                                prec);

        m_renderConfig.fracTopLeft  = {highPrecTopLeftX,
```

```cpp
            highPrecTopLeftY};
        m_renderConfig.fracSize   = {highPrecFracSizeX, highPrecFracSizeY};
        m_renderConfig.originalFracSize = {highPrecOriginalFracSizeX,
                                           highPrecOriginalFracSizeY};
    }

    RenderBoxTimeStats FractalRenderer::boxTimeStats() const {
        double min = 1e10;
        double max = -1e10;
        double total = 0;
        size_t count = 0;

        for (const auto &box : m_renderBoxes) {
            if (box.renderTime == 0) continue;

            if (box.renderTime < min) min = box.renderTime;
            if (box.renderTime > max) max = box.renderTime;
            total += box.renderTime;
            count += 1;
        }

        double average  = total / (double)count;
        size_t remainingBoxes = m_renderBoxes.size() - count;
        double remainingTime =
          ((double)remainingBoxes * average) / (double)m_renderConfig.numThreads;

        return {min, max, average, remainingTime};
    }

    const RenderConfig &FractalRenderer::config() const { return m_renderConfig; }
    RenderConfig &FractalRenderer::config() { return m_renderConfig; }

    const std::vector<RenderBox> &FractalRenderer::renderBoxes() const {
        return m_renderBoxes;
    }
    std::vector<RenderBox> &FractalRenderer::renderBoxes() { return m_renderBoxes; }

    const json &FractalRenderer::settings() const { return m_settings; }
    json &FractalRenderer::settings() { return m_settings; }

    const ci::Surface &FractalRenderer::surface() const { return
```

```
        m_fractalSurface; }
    ci::Surface &FractalRenderer::surface() { return m_fractalSurface; }
} // namespace frac
```

3.8 Generic Fractal Definition (`genericFractal.hpp`)

```cpp
#pragma once

namespace frac {
    class Fractal {
    public:
        /// Constructor taking a RenderConfig object
        /// \param config
        explicit Fractal(const RenderConfig &config);
        Fractal(const Fractal &)   = delete;
        Fractal(Fractal &&)      = delete;
        Fractal &operator=(const Fractal &) = delete;
        Fractal &operator=(Fractal &&)  = delete;
        virtual ~Fractal()      = default;

        /// Configure a new set of options for the fractal
        /// \param config The new RenderConfig to use
        virtual void updateRenderConfig(const RenderConfig &config);

        /// Iterate over a complex-valued coordinate and return the value at which the
        /// coordinate exceeds the given threshold or reaches the desired number of
        /// iterations. The return value also includes the number of iterations it took
        /// to reach the return coordinate.
        /// \param coord The initial complex-valued coordinate
        /// \return <iterations, resulting coordinate (low precision)>
        virtual std::pair<int64_t, lrc::Complex<LowPrecision>>
        iterCoordLow(const lrc::Complex<LowPrecision> &coord) const = 0;

        /// See iterCoordLow(const lrc::Complex<LowPrecision> &coord) const
        /// \param coord The initial complex-valued coordinate
        /// \return <iterations, resulting coordinate (high precision)>
        /// \see iterCoordLow(const lrc::Complex<LowPrecision> &coord) const
        virtual std::pair<int64_t, lrc::Complex<HighPrecision>>
        iterCoordHigh(const lrc::Complex<HighPrecision> &coord) const = 0;

        /// Apply the colouring algorithm given the result of an `iterCoord` call
        /// \param coord Resulting coordinate
```

```
        /// \param iters Number of iterations
        /// \return Colour of the point
        virtual ci::ColorA getColorLow(const lrc::Complex<LowPrecision> &coord,
                                       int64_t iters) const;

        /// See getColorLow(const lrc::Complex<LowPrecision> &coord, int64_t iters) const
        /// \param coord Resulting coordinate
        /// \param iters Number of iterations
        /// \return Colour of the point
        /// \see getColorLow(const lrc::Complex<LowPrecision> &coord, int64_t iters) const
        virtual ci::ColorA getColorHigh(const lrc::Complex<HighPrecision> &coord,
                                        int64_t iters) const;

    protected:
        RenderConfig m_renderConfig;
    };
} // namespace frac
```

3.9 Generic Fractal Implementation (**genericFractal.cpp**)

```
#include <fractal/fractal.hpp>

namespace frac {
    Fractal::Fractal(const RenderConfig &config) : m_renderConfig(config) {}

    void Fractal::updateRenderConfig(const RenderConfig &config) { m_renderConfig = config; }

    ci::ColorA Fractal::getColorLow(const lrc::Complex<LowPrecision> &coord, int64_t iters) const {
        using Col = ColorPalette::ColorType;

        // float logZN  = lrc::log(lrc::abs(lrc::Complex<float>(coord.real(), coord.imag()))) / 2;
        // float nu  = lrc::log(logZN / lrc::LN2) / lrc::LN2;
        // float iteration = static_cast<float>(iters) + 1 - nu;
        // const auto &palette = m_renderConfig.palette;
        // Col color1   = palette[static_cast<size_t>(iteration) % palette.size()];
```

```
    // Col color2   = palette[(static_cast<size_t>(iteration) + 1) % palette.size()];
    // Col merged   = ColorPalette::merge(color1, color2, lrc::mod(iteration, 1.0f));
    // return {merged.x(), merged.y(), merged.z(), 1};

    // Nice gradient
    double s1 = (double)iters -
                lrc::log2(lrc::log2((float)coord.real() * (float)coord.real() +
                                    (float)coord.imag() * (float)coord.imag())) +
                4;
    Col color = 0.5 + 0.5 * lrc::cos(3.0 + s1 * 0.15 + lrc::Vec3d(0.0, 0.6, 1.0));
    return {(float)color.x(), (float)color.y(), (float)color.z(), 1};

    // Cool stepped gradients
    // return {(iters % 10) / 10.f, 0, 0, 1};
    // return {(iters % 11) / 11.f, (iters % 23) / 23.f, (iters % 31) / 31.f, 1};
    // return {(iters % 2) / 2.f, (iters % 3) / 3.f, (iters % 7) / 7.f, 1};
}

ci::ColorA Fractal::getColorHigh(const lrc::Complex<HighPrecision> &coord,
                                 int64_t iters) const {
    using Col = ColorPalette::ColorType;

    // float logZN  = lrc::log(lrc::abs(lrc::Complex<float>(coord.real(), coord.imag()))) / 2;
    // float nu  = lrc::log(logZN / lrc::LN2) / lrc::LN2;
    // float iteration = static_cast<float>(iters) + 1 - nu;
    // const auto &palette = m_renderConfig.palette;
    // Col color1   = palette[static_cast<size_t>(iteration) % palette.size()];
    // Col color2   = palette[(static_cast<size_t>(iteration) + 1) % palette.size()];
    // Col merged   = ColorPalette::merge(color1, color2, lrc::mod(iteration, 1.0f));
    // return {merged.x(), merged.y(), merged.z(), 1};

    // Nice gradient
```

```
        double s1 = (double)iters -
                    lrc::log2(lrc::log2((float)coord.real() * (float)coord.real() +
                                        (float)coord.imag() * (float)coord.imag())) +
                    4;
        Col color = 0.5 + 0.5 * lrc::cos(3.0 + s1 * 0.15 + lrc::Vec3d(0.0, 0.6, 1.0));
        return {(float)color.x(), (float)color.y(), (float)color.z(), 1};

        // Cool stepped gradients
        // return {(iters % 10) / 10.f, 0, 0, 1};
        // return {(iters % 11) / 11.f, (iters % 23) / 23.f, (iters % 31) / 31.f, 1};
        // return {(iters % 2) / 2.f, (iters % 3) / 3.f, (iters % 7) / 7.f, 1};
    }
} // namespace frac
```

3.10 History Definition(**history.hpp**)

```
#pragma once

namespace frac {
    class HistoryNode {
    public:
        HistoryNode()         = default;
        HistoryNode(const HistoryNode &)   = delete;
        HistoryNode(HistoryNode &&)      = delete;
        HistoryNode &operator=(const HistoryNode &) = delete;
        HistoryNode &operator=(HistoryNode &&)  = delete;
        ~HistoryNode()        = default;

        /// Append a new node to the end of the linked list
        /// \param node The node to append (can contain links to other nodes)
        void append(HistoryNode *node);

        /// Free all child nodes following this one
        void killChildren();

        /// The number of nodes in the list, iterating forwards from this node
        /// \param prevSize The number of nodes in the list before this one (default to
        /// zero) \return The number of nodes in the list
        LIBRAPID_NODISCARD size_t sizeForward(size_t prevSize = 0) const;

        /// The number of nodes in the list, iterating backwards from this node
        /// \param prevSize
        /// \return The number of nodes in the list
        /// \see sizeForward
        LIBRAPID_NODISCARD size_t sizeBackward(size_t prevSize = 0) const;

        /// The next node in the linked list. This may be ‘nullptr‘, so always check
        /// the value is valid
        /// \return The next node in the linked list
        LIBRAPID_NODISCARD HistoryNode *next() const;

        /// The previous node in the linked list. See ‘next()‘ for more information
```

```
        /// \return The previous node in the linked list
        /// \see next
        LIBRAPID_NODISCARD HistoryNode *prev() const;

        /// Iterate backwards until a node with an invalid parent is found (i.e. the
        /// first node in the linked list)
        /// \return The first node in the linked list
        LIBRAPID_NODISCARD HistoryNode *first();

        /// return the last node in the linked list
        /// \return
        /// \see first
        LIBRAPID_NODISCARD HistoryNode *last();

        /// Update the configuration and surface members of this node
        /// \param config New configuration
        /// \param surface New surface
        void set(const RenderConfig &config, const ci::Surface &surface);

        /// See `set()`
        /// \param config New configuration
        /// \see set
        void setConfig(const RenderConfig &config);

        /// See `set()`
        /// \param surface New surface
        /// \see set
        void setSurface(const ci::Surface &surface);

        /// Getter method for the configuration instance stored
        /// \return RenderConfig
        LIBRAPID_NODISCARD const RenderConfig &config() const;

        /// Getter method for the surface instance stored
        /// \return ci::Surface
        LIBRAPID_NODISCARD const ci::Surface &surface() const;

        /// Non-const getter method for the configuration instance stored
        /// \return RenderConfig
        LIBRAPID_NODISCARD RenderConfig &config();

        /// Non-const getter method for the surface instance stored
        /// \return ci::Surface
        LIBRAPID_NODISCARD ci::Surface &surface();
```

```cpp

    private:
        HistoryNode *m_next = nullptr;
        HistoryNode *m_prev = nullptr;

        RenderConfig m_config;
        ci::Surface m_surface;
    };

    class HistoryBuffer {
    public:
        HistoryBuffer()         = default;
        HistoryBuffer(const HistoryBuffer &)    = delete;
        HistoryBuffer(HistoryBuffer &&)     = delete;
        HistoryBuffer &operator=(const HistoryBuffer &) = delete;
        HistoryBuffer &operator=(HistoryBuffer &&)  = delete;

        ~HistoryBuffer();

        /// Append a new point to the history buffer
        /// \param config The settings for the fractal renderer
        /// \param surface A saved copy of the fractal surface
        void append(const RenderConfig &config, const ci::Surface
            &surface);

        /// Undo the last operation
        bool undo();

        /// If possible, redo the last operation
        bool redo();

        /// Return the number of elements in the history buffer
        /// \return Number of elements
        LIBRAPID_NODISCARD size_t size() const;

        /// Return the first buffer item (a HistoryNode pointer)
        /// \return First item in the buffer
        LIBRAPID_NODISCARD HistoryNode *first() const;

        /// Return the last buffer item (a HistoryNode pointer)
        /// \return Last item in the buffer
        LIBRAPID_NODISCARD HistoryNode *last() const;

    private:
        HistoryNode *m_listHead = nullptr;
```

```
            HistoryNode *m_currentNode = nullptr;
        };
} // namespace frac
```

3.11 History Implementation (history.cpp)

```
#include <fractal/fractal.hpp>

namespace frac {
    void HistoryNode::append(HistoryNode *list) {
        if (m_next)
            m_next->append(list);
        else
            m_next = list;
    }

    void HistoryNode::killChildren() {
        if (m_next) m_next->killChildren();
        delete m_next;
        m_next = nullptr;
    }

    size_t HistoryNode::sizeForward(size_t prevSize) const {
        if (m_next)
            return m_next->sizeForward(prevSize + 1);
        else
            return prevSize;
    }

    size_t HistoryNode::sizeBackward(size_t prevSize) const {
        if (m_prev)
            return m_prev->sizeBackward(prevSize + 1);
        else
            return prevSize;
    }

    HistoryNode *HistoryNode::next() const { return m_next; }
    HistoryNode *HistoryNode::prev() const { return m_prev; }

    HistoryNode *HistoryNode::first() {
        if (m_prev)
            return m_prev->first();
        else
            return this;
    }
```

```

HistoryNode *HistoryNode::last() {
    if (m_next)
        return m_next->last();
    else
        return this;
}

void HistoryNode::set(const RenderConfig &config, const ci::Surface &surface) {
    m_config = config;
    m_surface = surface;
}

void HistoryNode::setConfig(const RenderConfig &config) { m_config = config; }

void HistoryNode::setSurface(const ci::Surface &surface) { m_surface = surface; }

const RenderConfig &HistoryNode::config() const { return m_config; }
const ci::Surface &HistoryNode::surface() const { return m_surface; }
RenderConfig &HistoryNode::config() { return m_config; }
ci::Surface &HistoryNode::surface() { return m_surface; }

HistoryBuffer::~HistoryBuffer() {
    m_listHead->killChildren();
    // No need to delete m_currentNode, since it will be killed recursively
    LIBRAPID_ASSERT(!m_listHead->next() && !m_listHead->prev(),
                    "HistoryBuffer is not empty");
    delete m_listHead;
}

void HistoryBuffer::append(const RenderConfig &config, const ci::Surface &surface) {
    auto list = new HistoryNode;
    list->set(config, surface);
    if (m_listHead) {
        m_listHead->append(list);
        m_currentNode = m_listHead->last();
    } else {
        m_listHead = list;
        m_currentNode = list;
    }
```

```
    }

    bool HistoryBuffer::undo() {
        if (m_currentNode->prev()) {
            m_currentNode = m_currentNode->prev();
            return true;
        }
        return false;
    }

    bool HistoryBuffer::redo() {
        if (m_currentNode->next()) {
            m_currentNode = m_currentNode->next();
            return true;
        }
        return false;
    }

    size_t HistoryBuffer::size() const {
        if (!m_listHead) return 0;
        return m_listHead->sizeForward();
    }

    HistoryNode *HistoryBuffer::first() const { return m_listHead->first(); }
    HistoryNode *HistoryBuffer::last() const { return m_listHead->last(); }
} // namespace frac
```

3.12 Main Window Definition (`mainWindow.hpp`)

```cpp
#pragma once

namespace frac {
    class MainWindow : public ci::app::App {
    public:
        /// Nothing passed to the constructor
        MainWindow() = default;

        void configureSettings();

        /// Set up the main window, making sure it's the right size and that the frame
        /// rates are set correctly
        void configureWindow();

        /// Configure ImGui, setting up the style and enabling docking
        void configureImGui();

        /// Set up the window, configure ImGui and initialize the fractal rendering
        /// surfaces
        void setup() override;

        /// Halt all render threads and wait for them to join main
        void stopRender();

        /// Run on shutdown to gracefully exit
        void cleanup() override;

        /// Render the fractal to the screen (from the FractalRenderer surface)
        void drawFractal();

        /// Outline each render box (if active) to show their current states
        void outlineRenderBoxes();

        /// Called every frame
        void draw() override;

        /// Draw the UI
        void drawImGui();

```

```
    /// Draw the history window
    void drawHistory();

    /// Update the most recent history item with the current fractal
        configuration
    /// and surface
    void updateHistoryItem();

    /// Move the top left corner of the fractal and set a new size
    /// \param topLeft Top-left corner (complex coordinate)
    /// \param size Size of the fractal (Re, Im)
    void moveFractalCorner(const lrc::Vec<HighPrecision, 2> &topLeft,
                           const lrc::Vec<HighPrecision, 2> &size);

    /// Set the center of the fractal and the dimensions -- see
        moveFractalCorner
    /// \param center Top-left corner
    /// \param size Size of fractal
    /// \see moveFractalCorner
    void moveFractalCenter(const lrc::Vec<HighPrecision, 2> &center,
                           const lrc::Vec<HighPrecision, 2> &size);

    /// Advanced zooming method -- given pixel coordinates for the
        top left and bottom
    /// right of the new area, perform the following:
    /// 1. Copy existing pixels in the specified region to a buffer
    /// 2. Regenerate the surface
    /// 3. Copy the buffer to the fractal surface
    /// 4. Reconfigure the fractal renderer
    /// 5. Trigger another fractal render
    /// \param pixTopLeft
    /// \param pixBottomRight
    void zoomFractal(const lrc::Vec2i &pixTopLeft, const lrc::Vec2i
        &pixBottomRight);

    /// Render the fractal into the fractal surface, and copy that
        to the
    /// fractal surface to be drawn. This will be executed on a
        separate
    /// thread in order to keep the UI updating
    void renderFractal();

    /// Callback for mouse movement (this does not include mouse
        clicks or
    /// drags) \param event The mouse event
```

```
        void mouseMove(ci::app::MouseEvent event) override;

        /// Callback for mouse clicks
        /// \param event The mouse event
        void mouseDown(ci::app::MouseEvent event) override;

        /// Callback for mouse drags
        /// \param event The mouse event
        void mouseDrag(ci::app::MouseEvent event) override;

        /// Callback for mouse releases
        /// \param event The mouse event
        void mouseUp(ci::app::MouseEvent event) override;

        /// Callback for mouse wheel events
        /// \param event The mouse event
        void mouseWheel(ci::app::MouseEvent event) override;

        /// Callback for when a key is pressed, including the modifiers (shift, ctrl, etc)
        /// \param event The key event
        void keyDown(ci::app::KeyEvent event) override;

    private:
        /// Given a starting coordinate and a target end coordinate, find the largest
        /// possible box with a given aspect ratio that can fit within this region.
        /// \tparam T The type of the coordinates
        /// \param p1 Starting coordinate
        /// \param p2 Target end coordinate
        /// \param aspectRatio Aspect ratio of the box
        /// \return Dimensions of the box
        template<typename T>
        static lrc::Vec<T, 2> aspectCorrectedBox(const lrc::Vec<T, 2> &p1,
                                                 const lrc::Vec<T, 2> &p2,
                                                 float aspectRatio) {
            lrc::Vec<T, 2> correctedBox;
            lrc::Vec<T, 2> delta = p2 - p1;
            if (delta.y() < delta.x() / aspectRatio)
                correctedBox = {delta.x(), delta.x() / aspectRatio};
            else
                correctedBox = {delta.y() * aspectRatio, delta.y()};
            return correctedBox;
```

```
        }

        /// Draw a zoom box at a given point. This includes a transparent box surrounded
        /// by a solid rectangle with a cross in the middle.
        /// \param start The top left corner of the box
        /// \param end The bottom right corner of the box
        void drawZoomBox(const lrc::Vec2f &start, const lrc::Vec2f &end) const;

        FractalRenderer m_renderer;    // The fractal renderer
        ci::gl::Texture2dRef m_fractalTexture; // The fractal texture
        ci::Font m_font = ci::Font("Arial", 24); // The font to use for rendering text
        lrc::Vec2i m_mousePos; // The current position of the mouse in the window
        lrc::Vec2i m_mouseDownPos; // The position of the mouse when it was clicked
        bool m_mouseDown = false; // Whether the mouse is currently down

        HistoryBuffer m_history;
        float m_historyScrollTarget = 0.0f;

        bool m_drawingZoomBox = false;
        bool m_showZoomBox = false;
        bool m_moveZoomBox = false;
        lrc::Vec2i m_zoomBoxStart;
        lrc::Vec2i m_zoomBoxEnd;

        // Values used for ImGui input fields
        std::string m_fineMovementRe;
        std::string m_fineMovementIm;
        std::string m_fineMovementZoom;
    };
} // namespace frac
```

3.13 Main Window Implementation (mainWindow.cpp)

```
#include <fractal/fractal.hpp>

namespace frac {
    void MainWindow::configureSettings() {
        // Load the settings file
        FRAC_LOG(fmt::format("Loading settings from {}", FRACTAL_UI_SETTINGS_PATH));
```

```

        std::fstream settingsFile(FRACTAL_UI_SETTINGS_PATH, std::ios::in);
        if (settingsFile.is_open()) {
            m_renderer.setConfig(json::parse(settingsFile));
            m_history.append(m_renderer.config(), m_renderer.surface());
        } else {
            FRAC_ERROR("Failed to open settings file");
            quit();
        }

        FRAC_LOG("Settings Configured");
    }

    void MainWindow::configureWindow() {
        setFrameRate(-1);    // Unlimited framerate
        ci::gl::enableVerticalSync(true); // Enable vertical sync to avoid tearing
        setWindowSize(1200, 700);  // Set the initial window size

        // Set up rendering settings
        ci::gl::enableDepthWrite();
        ci::gl::enableDepthRead();
        ci::gl::enableDepth();
        glDepthFunc(GL_ALWAYS);

        FRAC_LOG("Window Configured");
    }

    void MainWindow::configureImGui() {
        ImGui::Initialize();
        ImGui::StyleColorsDark();
        ImGui::GetIO().ConfigFlags |= ImGuiConfigFlags_DockingEnable;
        ImGui::GetIO().FontGlobalScale = 1.0f;

        FRAC_LOG("ImGui Configured");
    }

    void MainWindow::setup() {
        FRAC_LOG("Setup Called");

        configureSettings();
        configureWindow();
        configureImGui();

```

```
    m_renderer.regenerateSurface();
    renderFractal();

    FRAC_LOG("Setup Complete");
}

void MainWindow::stopRender() { m_renderer.stopRender(); }

void MainWindow::cleanup() {
    stopRender();
    FRAC_LOG("Cleaned Up");
}

void MainWindow::drawFractal() {
    m_fractalTexture =
        ci::gl::Texture2d::create(m_renderer.surface());

    const RenderConfig &config = m_renderer.config();
    double aspect = (double)config.imageSize.x() /
        (double)config.imageSize.y();
    double height = getWindowHeight();
    lrc::Vec2f renderSize(height * aspect, height);

    ci::gl::color(ci::ColorA(1, 1, 1, 1));
    ci::gl::draw(m_fractalTexture, ci::Rectf({0, 0}, renderSize));
}

void MainWindow::outlineRenderBoxes() {
    const RenderConfig &config    = m_renderer.config();
    const std::vector<RenderBox> &renderBoxes =
        m_renderer.renderBoxes();
    for (const auto &box : renderBoxes) {
        switch (box.state) {
            case RenderBoxState::None:
            case RenderBoxState::Rendered: continue;
            case RenderBoxState::Queued:
                ci::gl::color(ci::ColorA(0, 1, 0, 0.2));
                break;
            case RenderBoxState::Rendering:
                ci::gl::color(ci::ColorA(1, 1, 0, 0.2));
                break;
        }

        ci::ivec2 boxPos = box.topLeft;
        ci::ivec2 boxSize = box.dimensions;
```

```
            boxPos.y += getWindowHeight() - config.imageSize.y();
            ci::gl::drawStrokedRect(ci::Rectf(boxPos, boxPos + boxSize),
                1);
        }
    }

    void MainWindow::draw() {
        ci::gl::clear(ci::Color(0.2f, 0.2f, 0.2f));

        drawImGui();
        drawFractal();
        outlineRenderBoxes();
        drawHistory();

        if (m_drawingZoomBox) {
            // Draw an aspect-ratio corrected box
            RenderConfig config = m_renderer.config();
            float aspectRatio = (float)config.imageSize.x() /
                (float)config.imageSize.y();
            lrc::Vec2i correctedBox =
              aspectCorrectedBox(m_mouseDownPos, m_mousePos, aspectRatio);
            auto correctedEnd = m_mouseDownPos + correctedBox;
            drawZoomBox(m_mouseDownPos, correctedEnd);
        }

        if (m_showZoomBox) { drawZoomBox(m_zoomBoxStart, m_zoomBoxEnd); }
    }

    void MainWindow::drawImGui() {
        // Arbitrary constant to make the UI look nice
        constexpr int64_t labelledItemWidth = -120;

        RenderConfig &config = m_renderer.config();
        const json &settings = m_renderer.settings();

        // Fractal Information Window
        json fractalInfo = settings["menus"]["fractalInfo"];
        ImGui::SetNextWindowPos({(float)fractalInfo["posX"],
            (float)fractalInfo["posY"]},
                                ImGuiCond_Once);
        ImGui::SetNextWindowSize(
          {(float)fractalInfo["width"], (float)fractalInfo["height"]},
            ImGuiCond_Once);
        ImGui::Begin("Fractal Info", nullptr);
        {
```

```
            ImGui::Text("Fractal Type: Mandelbrot");

            HighPrecision re = config.fracTopLeft.x() +
                config.fracSize.x() / 2;
            HighPrecision im = config.fracTopLeft.y() +
                config.fracSize.y() / 2;
            HighPrecision zoom = config.originalFracSize.x() /
                config.fracSize.x();

            ImGui::TextWrapped("%s", fmt::format("Re: {}", re).c_str());
            ImGui::TextWrapped("%s", fmt::format("Im: {}", im).c_str());

            std::ostringstream os;
            os << std::fixed << std::setprecision(6) << std::scientific
                << zoom;
            ImGui::TextWrapped("%s", fmt::format("Zoom: {}x",
                os.str()).c_str());

            double maxZoomExponent = config.precision / lrc::log2(10);
            ImGui::TextWrapped(
              "%s", fmt::format("Max Zoom: e+{:.3f}",
                  maxZoomExponent).c_str());
        }
        ImGui::End();

        // Fine movement window
        json fineMovement = settings["menus"]["fineMovement"];
        ImGui::SetNextWindowPos(
          {(float)fineMovement["posX"], (float)fineMovement["posY"]},
              ImGuiCond_Once);
        ImGui::SetNextWindowSize(
          {(float)fineMovement["width"], (float)fineMovement["height"]},
              ImGuiCond_Once);
        ImGui::Begin("Fine Movement", nullptr);
        {
            ImGui::InputText("Re", &m_fineMovementRe);
            ImGui::InputText("Im", &m_fineMovementIm);
            ImGui::InputText("Zoom", &m_fineMovementZoom);

            if (ImGui::Button("Apply")) {
                HighPrecision re, im, zoom, sizeRe, sizeIm;
                scn::scan(m_fineMovementRe, "{}", re);
                scn::scan(m_fineMovementIm, "{}", im);
                scn::scan(m_fineMovementZoom, "{}", zoom);

```

```cpp
            FRAC_LOG(fmt::format("Received Real Part: {}", re));

            sizeRe = config.originalFracSize.x() / zoom;
            sizeIm = config.originalFracSize.y() / zoom;
            moveFractalCenter(lrc::Vec<HighPrecision, 2>(re, im),
                              lrc::Vec<HighPrecision, 2>(sizeRe,
                                  sizeIm));
            renderFractal();
        }
    }
    ImGui::End();

    // Render configuration
    json renderConfigMenu = settings["menus"]["renderConfig"];
    ImGui::SetNextWindowPos(
      {(float)renderConfigMenu["posX"],
          (float)renderConfigMenu["posY"]},
      ImGuiCond_Once);
    ImGui::SetNextWindowSize(
      {(float)renderConfigMenu["width"],
          (float)renderConfigMenu["height"]},
      ImGuiCond_Once);
    ImGui::Begin("Render Configuration", nullptr);
    {
        static int64_t minThreads = 1;
        static int64_t maxThreads =
            std::thread::hardware_concurrency();
        static int64_t minIters  = 1;
        static int64_t maxIters  = 100000;
        static int64_t minPrecision = 64;
        static int64_t maxPrecision = 1024;
        static int64_t minAntiAlias = 1;
        static int64_t maxAntiAlias = 16;
        static int64_t minDraftInc = 1;
        static int64_t maxDraftInc = 4;

        static int64_t newThreads = config.numThreads;
        static int64_t newIters  = config.maxIters;
        static int64_t newPrecision = config.precision;
        static int64_t newAntiAlias = config.antiAlias;
        static bool newDraftRender = config.draftRender;
        static int64_t newDraftInc = config.draftInc;

        ImGui::PushItemWidth(labelledItemWidth);
        ImGui::SliderScalar(
```

```
              "Threads", ImGuiDataType_S64, &newThreads, &minThreads, &maxThreads);

            ImGui::PushItemWidth(labelledItemWidth);
            ImGui::SliderScalar("Anti Aliasing",
                                ImGuiDataType_S64,
                                &newAntiAlias,
                                &minAntiAlias,
                                &maxAntiAlias);

            ImGui::PushItemWidth(labelledItemWidth);
            ImGui::DragScalarN(
              "Iterations", ImGuiDataType_S64, &newIters, 1, 5, &minIters, &maxIters);

            ImGui::PushItemWidth(labelledItemWidth);
            ImGui::DragScalarN("Precision",
                               ImGuiDataType_S64,
                               &newPrecision,
                               1,
                               0.1,
                               &minPrecision,
                               &maxPrecision);

            if (ImGui::Button("Apply")) {
                stopRender();
                config.numThreads = newThreads;
                config.maxIters = newIters;
                config.precision = newPrecision;
                config.antiAlias = newAntiAlias;
                config.draftRender = newDraftRender;
                config.draftInc = newDraftInc;
                lrc::prec2(newPrecision);
                m_renderer.updateRenderConfig();
                m_renderer.updateConfigPrecision();
                renderFractal();
            }

            ImGui::SameLine();
            ImGui::Checkbox("Draft Mode", &newDraftRender);

            if (newDraftRender) {
                ImGui::SameLine();
                ImGui::PushItemWidth(labelledItemWidth);
                ImGui::SliderScalar("Draft Increment",
```

```
                                  ImGuiDataType_S64,
                                  &newDraftInc,
                                  &minDraftInc,
                                  &maxDraftInc);
            }
        }
        ImGui::End();

        // Render Statistics
        json renderStatistics = settings["menus"]["renderStatistics"];
        ImGui::SetNextWindowPos(
          {(float)renderStatistics["posX"],
              (float)renderStatistics["posY"]},
          ImGuiCond_Once);
        ImGui::SetNextWindowSize(
          {(float)renderStatistics["width"],
              (float)renderStatistics["height"]},
          ImGuiCond_Once);

        RenderBoxTimeStats stats = m_renderer.boxTimeStats();
        ImGui::Begin("Render Statistics", nullptr);
        {
            ImGui::Text("Pixels/s (min): %s", fmt::format("{:.3f}",
                stats.min).c_str());
            ImGui::Text("Pixels/s (max): %s", fmt::format("{:.3f}",
                stats.max).c_str());
            ImGui::Text("Pixels/s (avg): %s",
                        fmt::format("{:.3f}", stats.average).c_str());
            ImGui::Text("Estimated Time Remaining: %s",
                        lrc::formatTime(stats.remainingTime).c_str());
        }
        ImGui::End();
    }

    void MainWindow::drawHistory() {
        const json &settings  = m_renderer.settings();
        const float historyFrameWidth =
            settings["menus"]["history"]["frameWidth"];
        const float historyFrameSep =
            settings["menus"]["history"]["frameSep"];

        const auto windowWidth = (float)getWindowWidth();
        const auto windowHeight = (float)getWindowHeight();
        const RenderConfig &config = m_renderer.config();
        size_t historySize  = m_history.size();
```

```cpp
        HistoryNode *node  = m_history.first();
        int64_t index   = 0;
        int64_t totalHeight  = 0;

        // Draw a bounding box for the frames to sit within
        float boxLeft = windowWidth - historyFrameWidth - historyFrameSep * 2;
        ci::gl::color(ci::ColorA(0.15, 0.15, 0.3, 1));
        ci::gl::drawSolidRect(ci::Rectf(boxLeft, 0, windowWidth, windowHeight));

        while (node) {
            auto texture = ci::gl::Texture2d::create(node->surface());
            float aspect = (float)config.imageSize.x() / (float)config.imageSize.y();
            lrc::Vec2f renderSize(historyFrameWidth, historyFrameWidth / aspect);
            lrc::Vec2f drawPos(windowWidth - historyFrameWidth - historyFrameSep,
                               (renderSize.y() + historyFrameSep) *
                                   (float)(historySize - index - 1) +
                                 historyFrameSep);

            totalHeight += (int64_t)(renderSize.y() + historyFrameSep);
            if (drawPos.y() > windowHeight || drawPos.y() + renderSize.y() < 0) break;

            drawPos.y(drawPos.y() + m_historyScrollTarget);

            ci::gl::color(ci::ColorA(1, 1, 1, 1));
            ci::gl::draw(texture, ci::Rectf(drawPos, drawPos + renderSize));
            ci::gl::color(ci::ColorA(0, 0, 0, 1));
            glu::drawStrokedRectangle(drawPos, drawPos + renderSize, 3);

            node = node->next();
            index++;
        }

        // Limit scrolling -- this is done after drawing to ensure the frame size is taken
        // into account
        m_historyScrollTarget = lrc::clamp(m_historyScrollTarget, 0, totalHeight);
    }
```

```cpp

void MainWindow::moveFractalCorner(const lrc::Vec<HighPrecision, 2> &topLeft,
                                   const lrc::Vec<HighPrecision, 2> &size) {
    m_renderer.moveFractalCorner(topLeft, size);
}

void MainWindow::moveFractalCenter(const lrc::Vec<HighPrecision, 2> &center,
                                   const lrc::Vec<HighPrecision, 2> &size) {
    m_renderer.moveFractalCenter(center, size);
}

void MainWindow::zoomFractal(const lrc::Vec2i &pixTopLeft,
                             const lrc::Vec2i &pixBottomRight) {
    // The vast majority of the calculations must be done at the highest
    // precision possible. Without this, the zooming will not function
    // correctly when the zoom factor exceeds the range of 64-bit precision.

    FRAC_LOG("Moving Fractal...");
    FRAC_LOG(fmt::format("Pixel Coordinates: {} -> {}", pixTopLeft, pixBottomRight));
    updateHistoryItem();

    RenderConfig &config = m_renderer.config();
    ci::Surface &surface = m_renderer.surface();

    // Resize the fractal area
    HighVec2 imageSize = config.imageSize;
    HighVec2 pixelDelta = pixBottomRight - pixTopLeft;

    HighVec2 newFracPos = lrc::map(HighVec2(pixTopLeft),
                                   HighVec2(0, 0),
                                   imageSize,
                                   config.fracTopLeft,
                                   config.fracTopLeft + config.fracSize);

    HighVec2 newFracSize = lrc::map(
      pixelDelta, HighVec2(0, 0), imageSize, HighVec2(0, 0),
```

```
                config.fracSize);

        // Copy the pixels from the selected region to the new region
        // A temporary buffer is required here because, at some point, the loop
        // would be writing to the same pixels it is reading from, resulting in
        // strange visual glitches.
        int64_t imgWidth = config.imageSize.x();
        int64_t imgHeight = config.imageSize.y();
        int64_t index = 0;
        std::vector<ci::ColorA> newPixels(imgWidth * imgHeight);
        auto mouseStartInImageLow =
          pixTopLeft - lrc::Vec2i {0, getWindowHeight() - config.imageSize.y()};
        for (int64_t y = 0; y < imgHeight; ++y) {
            for (int64_t x = 0; x < imgWidth; ++x) {
                int64_t pixX = lrc::map(x,
                                        0.f,
                                        imgWidth,
                                        mouseStartInImageLow.x(),
                                        mouseStartInImageLow.x() + (float)pixelDelta.x());
                int64_t pixY = lrc::map(y,
                                        0.f,
                                        imgHeight,
                                        mouseStartInImageLow.y(),
                                        mouseStartInImageLow.y() + (float)pixelDelta.y());

                newPixels[index++] = surface.getPixel({pixX, pixY});
            }
        }

        // Write the new pixels to the surface
        index = 0;
        for (int64_t y = 0; y < imgHeight; ++y) {
            for (int64_t x = 0; x < imgWidth; ++x) {
                surface.setPixel({x, y}, newPixels[index++]);
            }
        }

        config.fracTopLeft = newFracPos;
        config.fracSize = newFracSize;
        m_renderer.updateRenderConfig();
```

```cpp
    renderFractal();
}

void MainWindow::renderFractal() {
    // updateHistoryItem();
    m_renderer.renderFractal();
    m_history.append(m_renderer.config(), m_renderer.surface());
}

void MainWindow::updateHistoryItem() {
    // Update history buffer surface before re-rendering the fractal
    if (m_history.size() > 0) {
        m_history.last()->setSurface(m_renderer.surface());
        FRAC_LOG("Writing to surface");
    }
}

void MainWindow::mouseMove(ci::app::MouseEvent event) { m_mousePos = event.getPos(); }

void MainWindow::mouseDown(ci::app::MouseEvent event) {
    // Don't capture mouse events if ImGui wants them
    if (ImGui::GetIO().WantCaptureMouse) return;

    m_mouseDown = true;
    m_mouseDownPos = event.getPos();

    // Ensure mouse is within the image
    if (m_mouseDownPos.x() >= 0 &&
        m_mouseDownPos.x() < m_renderer.config().imageSize.x() &&
        m_mouseDownPos.y() >= 0 &&
        m_mouseDownPos.y() < m_renderer.config().imageSize.y()) {
        constexpr size_t offset = 0; // pixel offset from the edge of the box
        if (m_showZoomBox && m_mouseDownPos.x() > m_zoomBoxStart.x() + offset &&
            m_mouseDownPos.x() < m_zoomBoxEnd.x() - offset &&
            m_mouseDownPos.y() > m_zoomBoxStart.y() + offset &&
            m_mouseDownPos.y() < m_zoomBoxEnd.y() - offset) {
            m_moveZoomBox = true;
        } else {
            m_drawingZoomBox = true;
            m_showZoomBox = false;
        }
    }
```

```
}

void MainWindow::mouseDrag(ci::app::MouseEvent event) {
    if (ImGui::GetIO().WantCaptureMouse) return;
    lrc::Vec2i delta = lrc::Vec2i(event.getPos()) - m_mousePos;
    m_mousePos  = event.getPos();

    if (m_moveZoomBox) {
        m_zoomBoxStart += delta;
        m_zoomBoxEnd += delta;
    }
}

void MainWindow::mouseUp(ci::app::MouseEvent event) {
    if (ImGui::GetIO().WantCaptureMouse) return;
    RenderConfig &config = m_renderer.config();
    m_mouseDown   = false;

    // Resize the fractal area
    lrc::Vec2i imageSize = config.imageSize;
    float aspectRatio = (float)imageSize.x() / (float)imageSize.y();

    lrc::Vec2f aspectCorrected =
      aspectCorrectedBox(m_mouseDownPos, m_mousePos, aspectRatio);

    // Persistent zoom area
    if (m_drawingZoomBox) {
        m_zoomBoxStart = m_mouseDownPos;
        m_zoomBoxEnd = m_mouseDownPos + lrc::Vec2i(aspectCorrected);
        m_showZoomBox = true;
        m_drawingZoomBox = false;
    }
}

void MainWindow::mouseWheel(ci::app::MouseEvent event) {
    if (ImGui::GetIO().WantCaptureMouse) return;

    const json &settings = m_renderer.settings();
    if (m_mousePos.x() > settings["menus"]["history"]["frameWidth"]) {
        m_historyScrollTarget +=
          event.getWheelIncrement() *
          settings["menus"]["history"]["scrollSpeed"].get<float>();
    }
}
```

```

    void MainWindow::keyDown(ci::app::KeyEvent event) {
        if (m_showZoomBox && event.getCode() == ci::app::KeyEvent::KEY_RETURN) {
            // Update history and apply zoom box
            // updateHistoryItem();
            zoomFractal(m_zoomBoxStart, m_zoomBoxEnd);
            m_showZoomBox = false;
        } else if (m_showZoomBox && event.getCode() == ci::app::KeyEvent::KEY_ESCAPE) {
            // Cancel zoom box
            m_showZoomBox = false;
        }
    }

    void MainWindow::drawZoomBox(const lrc::Vec2f &start, const lrc::Vec2f &end) const {
        // Translucent inner box
        ci::gl::color(ci::ColorA(1, 0, 0, 0.333)); // Red with alpha
        ci::gl::drawSolidRect(ci::Rectf(start.x(), start.y(), end.x(), end.y()));
        // Small cross in the middle
        ci::gl::color(ci::ColorA(0, 0, 1, 0.333)); // Blue with alpha
        glu::drawCross((start + end) * 0.5f, 5.f, 2.f);
        // Bounding box
        ci::gl::color(ci::ColorA(1, 0, 0, 1)); // Red
        glu::drawStrokedRectangle(start, end, 5);
    }
} // namespace frac
```

3.14 Mandelbrot Fractal Definition (**mandelbrot.hpp**)

```
#pragma once

#include <fractal/genericFractal.hpp>

namespace frac {
    /*
     * No need to document this file, since the class inherits from a generic fractal
     * class and does not implement any novel functions.
     */

    class Mandelbrot : public Fractal {
    public:
        /// Constructor taking a RenderConfig object
        /// \param config RenderConfig object
        explicit Mandelbrot(const RenderConfig &config);
        Mandelbrot(const Mandelbrot &)   = delete;
        Mandelbrot(Mandelbrot &&)    = delete;
        Mandelbrot &operator=(const Mandelbrot &) = delete;
        Mandelbrot &operator=(Mandelbrot &&) = delete;

        ~Mandelbrot() override = default;

        LIBRAPID_NODISCARD std::pair<int64_t, lrc::Complex<LowPrecision>>
        iterCoordLow(const lrc::Complex<LowPrecision> &coord) const override;

        LIBRAPID_NODISCARD std::pair<int64_t, lrc::Complex<HighPrecision>>
        iterCoordHigh(const lrc::Complex<HighPrecision> &coord) const override;
    };
} // namespace frac
```

3.15 Mandelbrot Fractal Implementation (**mandelbrot.cpp**)

```
#include <fractal/fractal.hpp>

namespace frac {
    Mandelbrot::Mandelbrot(const RenderConfig &config) : Fractal(config) {}

    /*
```

```
 * Note that, since this class will be used polymorphically with other classes,
 * these two functions must be implemented separately and cannot be templated, as the compiler
 * would error when trying to identify which function to call. Additionally, splitting the
 * functions in this way allows for more targeted optimisations to be made in some cases.
 */

std::pair<int64_t, lrc::Complex<LowPrecision>>
Mandelbrot::iterCoordLow(const lrc::Complex<LowPrecision> &coord) const {
    LowPrecision re_0 = lrc::real(coord); // Real component (initial)
    LowPrecision im_0 = lrc::imag(coord); // Imaginary component (initial)
    LowPrecision re = 0, im = 0;
    LowPrecision tmp; // Temporary variable for use in the calculation
    int64_t iteration = 0;

    // Bail when larger than this
    double bailout = Fractal::m_renderConfig.bail;

    while (re * re + im * im <= bailout && iteration < Fractal::m_renderConfig.maxIters) {
        tmp = re * re - im * im + re_0;
        im = 2 * re * im + im_0;
        re = tmp;
        ++iteration;
    }

    return {iteration, lrc::Complex<LowPrecision>(re, im)};

    // lrc::Complex<LowPrecision> tempVarThing = coord;
    // while (lrc::abs(tempVarThing) <= bailout && iteration < Fractal::m_renderConfig.maxIters) {
    //  tempVarThing = lrc::pow(tempVarThing, LowPrecision(4)) + coord;
    //  ++iteration;
    // }

    // return {iteration, tempVarThing};
}

```

```
    std::pair<int64_t, lrc::Complex<HighPrecision>>
    Mandelbrot::iterCoordHigh(const lrc::Complex<HighPrecision> &coord)
        const {
        HighPrecision re_0 = lrc::real(coord); // Real component
            (initial)
        HighPrecision im_0 = lrc::imag(coord); // Imaginary component
            (initial)
        HighPrecision re = 0, im = 0;
        HighPrecision tmp; // Temporary variable for use in the
            calculation
        int64_t iteration = 0;

        // Bail when larger than this
        double bailout = 1 << 16;

        while (re * re + im * im <= bailout && iteration <
            Fractal::m_renderConfig.maxIters) {
            tmp = re * re - im * im + re_0;
            im = 2 * re * im + im_0;
            re = tmp;
            ++iteration;
        }

        return {iteration, lrc::Complex<HighPrecision>(re, im)};
    }
} // namespace frac
```

3.16 OpenGL Utilities Definition (**openglUtils.hpp**)

```
#pragma once

namespace frac::glu {
    /// Draw a line from p1 to p2 with a given thickness. The line has a circle drawn at each end
    /// to make it look nice.
    /// \param p1 Line start
    /// \param p2 Line end \param
    /// thickness Line thickness
    void drawLine(const lrc::Vec2f &p1, const lrc::Vec2f &p2, float thickness = 1);

    /// Draw a stroked rectangle with a given thickness -- note this draws the EDGES of
    /// the rectangle, and does not fill the inside \param topLeft Top left corner of the
    /// rectangle \param bottomRight Bottom right corner of the rectangle \param thickness
    /// Line thickness
    void drawStrokedRectangle(const lrc::Vec2f &topLeft, const lrc::Vec2f &bottomRight,
                              float thickness = 1);

    /// Draw a cross at \p center, where each "arm" has length \p radius and thickness
    /// \p thickness
    /// \param center Where to draw the cross
    /// \param radius Radius of the cross
    /// \param thickness Thickness of the cross
    void drawCross(const lrc::Vec2f &center, float radius, float thickness = 1);
} // namespace frac::glu
```

3.17 OpenGL Utilities Implementation (**openglUtils.cpp**)

```
#include <fractal/fractal.hpp>

namespace frac::glu {
    void drawLine(const lrc::Vec2f &p1, const lrc::Vec2f &p2, float thickness) {
        ci::vec3 translation({p1.x(), p1.y(), 0});
        float rotation = lrc::atan2(p2.y() - p1.y(), p2.x() - p1.x());
        float lineLength = static_cast<float>((p2 - p1).mag());
```

```

        ci::gl::pushMatrices();
        ci::gl::translate(translation);
        ci::gl::rotate(rotation);

        // Line segment
        ci::gl::drawSolidRect(ci::Rectf({0.f, -thickness / 2.f}, {lineLength, thickness / 2}));

        // Draw two circles to make it look nice :)
        ci::gl::drawSolidCircle({0, 0}, thickness / 2);
        ci::gl::drawSolidCircle({lineLength, 0}, thickness / 2);

        ci::gl::popMatrices();
    }

    void drawStrokedRectangle(const lrc::Vec2f &topLeft, const lrc::Vec2f &bottomRight,
                              float thickness) {
        // Draw each edge using drawLine, as it allows a thickness to be specified
        drawLine(topLeft, {bottomRight.x(), topLeft.y()}, thickness);
        drawLine({bottomRight.x(), topLeft.y()}, bottomRight, thickness);
        drawLine(bottomRight, {topLeft.x(), bottomRight.y()}, thickness);
        drawLine({topLeft.x(), bottomRight.y()}, topLeft, thickness);
    }

    void drawCross(const lrc::Vec2f &center, float radius, float thickness) {
        ci::vec3 translation({center.x(), center.y(), 0});
        ci::gl::pushMatrices();
        ci::gl::translate(translation);
        ci::gl::lineWidth(thickness);
        ci::gl::drawLine(ci::vec2(0, -radius), ci::vec2(0, radius));
        ci::gl::drawLine(ci::vec2(-radius, 0), ci::vec2(radius, 0));
        ci::gl::popMatrices();
    }
} // namespace frac::glu
```

3.18 Render Configuration Definition (`renderConfig.hpp`)

```cpp
#pragma once

namespace frac {
    /// Represents the state of a render box
    enum class RenderBoxState {
        None, // Not yet assigned a state
        Queued, // Queued to be rendered
        Rendering, // Currently being rendered
        Rendered // Rendered and ready to be written to the image
    };

    /// Stores the pixel-space coordinates of a region to render
    struct RenderBox {
        lrc::Vec2i topLeft;
        lrc::Vec2i dimensions;
        bool draftRender;
        int64_t draftInc;
        RenderBoxState state = RenderBoxState::None;
        double renderTime = 0;
    };

    /// Information about the time taken to render a box
    struct RenderBoxTimeStats {
        double min = 0;
        double max = 0;
        double average = 0;
        double remainingTime = 0;
    };

    /// Contains all the information required to define an image (not including the
    /// fractal type and colouring algorithm)
    struct RenderConfig {
        int64_t numThreads; // Number of threads to render on (max)
        int64_t maxIters; // Largest number of iterations to allow
        int64_t precision; // Precision of floating point types used for arithmetic
        LowPrecision bail; // Bailout value
        int64_t antiAlias; // Anti-aliasing factor -- 1 = no anti-aliasing

        lrc::Vec2i imageSize; // Size of the image to render
        lrc::Vec2i boxSize; // Size of sub-regions to render (see
```

```
            RenderBox)

        lrc::Vec<HighPrecision, 2> fracTopLeft; // The fractal-space
            center of the image
        lrc::Vec<HighPrecision, 2> fracSize; // The width and height of
            the fractal space
        lrc::Vec<HighPrecision, 2> originalFracSize; // Original size

        ColorPalette palette; // The palette to use for rendering the
            fractal

        bool draftRender; // Whether to render the fractal in draft mode
        int64_t draftInc; // Increment for draft rendering
    };
} // namespace frac
```

3.19 Render Configuration Implementation (**renderConfig.cpp**)

```
#include <fractal/fractal.hpp>

namespace frac {

} // namespace frac
```

References

[1] Catherine Soanes and Sara Hawker. *Compact Oxford English Dictionary of current English*. Oxford University Press, 2013.

[2] Paul Bourke, Nov 2002. URL http://paulbourke.net/fractals/mandelbrot/.

[3] Frederik Slijkerman. Ultra fractal: Advanced fractal software for windows and macos, Jul 2022. URL https://www.ultrafractal.com/.

[4] David J Eck. URL https://math.hws.edu/eck/index.html.

[5] URL https://xaos-project.github.io/XaoSjs/.

[6] Fractal software. URL https://fractalfoundation.org/resources/fractal-software/.

[7] Cinder Developers and Toby Davis. Pencilcaseman/cinderbox: A modified version of cinder with extra libraries for improved functionality. URL https://github.com/Pencilcaseman/cinderbox.

[8] Toby Davis. LibRapid: Optimised Mathematics for C++, 1 2023. URL https://github.com/LibRapid/librapid.