Fractal Visualisation Application

Contents

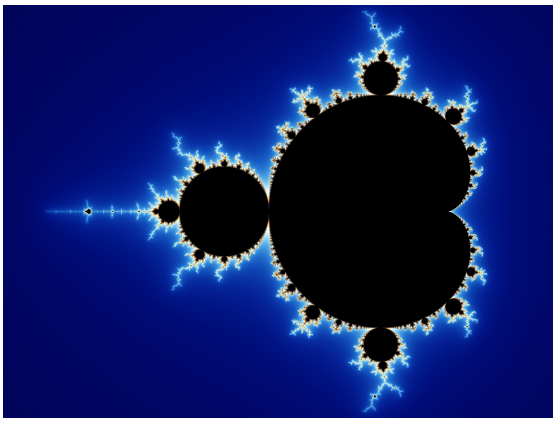
// TODO (FINAL THING)

Analysis

In this project, I intend to create a simple but well-featured graphical interface for exploring fractals, a mathematical phenomenon arising from repeated iterations of mathematical equations. The project will therefore consist of two main parts, the backend (responsible for calculating and producing the fractal itself, and usable from a terminal for more advanced users) and the frontend (responsible for displaying and allowing interaction with the user in real-time, graphical and intended to be easy to use).

Background to the Problem

Fractals



Render of the Mandelbrot Set (created by Wolfgang Beyer, via Wikipedia)

Fractals are a mathematical phenomenon which arise from repeated iteration of a recursive mathematical equation, the most well-known being the Mandelbrot set. The equation responsible for producing it is $z = (z^2) + c$. A fractal is 'rendered' into an image by taking each pixel, calculating a position for it in the complex number space (the horizontal axis becomes the real coordinate, vertical axis becomes the imaginary coordinate), and assigning that complex number to both variables z and c initially. Then, the equation is iterated repeatedly, with c staying constant between iterations and z being recalculated each time. Iteration continues until z tends to infinity (usually defined as having a modulus greater than 2, after which it is guaranteed to tend to infinity), or until some defined limit is reached, at which point the number of iterations completed is counted or reported. The number of iterations performed then becomes the 'value' assigned to that point in the complex number space, and be used to select a colour for a particular pixel.

Fractals are interesting primarily due to their property of self-similarity: no matter how closely you zoom in or how high a resolution image is rendered (i.e. how fine the arbitrary 'pixels' are) there will always be an infinite depth of detail. The complex high-level shapes, such as the recognisable form of the Mandelbrot set, are repeated (although not necessarily identically) in smaller and smaller instances, with many other features such as spirals, ‘cracks’, and more. Thus, it is impossible to represent fractals using vector images, due to the fact that they have an infinite level of edge detail, and effectively an infinite circumference therefore.

Their self-similarity and detail properties has led them to be compared to (and indeed modelled against) natural shapes, such as the growth of fern leaves and coastlines (the coastline paradox is an observation that as a coastline is measured more and more precisely, its circumference seems to continue to increase towards infinity, known as having a fractal dimension). Thus, studying mathematics in this way can be very useful to understand natural phenomena and real-world problems.

This also makes fractals a useful demonstration for mathematics students (or anyone in fact) as discussed further below.

Mathematics

Mathematics is often seen as an uninteresting topic or field, at least for anyone who doesn't already have an interest in it. It tends to be defined by equations and numeracy which many people find boring. However, fractals are an example of how mathematical ideas can apply not only to modelling nature and drawing interesting parallels with the real world, but also in producing aesthetically pleasing and beautiful images which everyone can enjoy and appreciate, perhaps even as an art form. Particularly, their use as an illustration of the applicability of mathematics in a classroom can make the subject more engaging, especially to school students.

Thus, this project aims to, as well as providing a useful fractal visualisation tool, be primarily a teaching aid which can give students the opportunity to experiment with fractals and learn about why they occur at the same time, in an engaging fashion.

The Problem

This brings into focus the problem which the project will address, namely the engagement of students or non-mathematical people with maths in general. Currently, tools for visualising fractals are limited, many are either demos with expensive full versions, outdated, slow, or difficult to use:

- Xaos - a free, interactive piece of software, but has a limited functionality and lacks the functionality to input custom equations and display them as fractals
- FRAX - cheap, but only runs on iOS devices and is thus limited in its processing and usability, making it difficult to render with more computationally expensive parameters and very hard to demonstrate to others
- Fractal to Desktop - less fully featured, simpler to use, but only supports Windows (might be inaccessible for some students who do not have access to a Windows-hosting device)
- JWildfire - although very powerful, also quite complex and has a steep learning curve to it
- Ultra Fractal - incredibly powerful and fully featured, but allows only a 30-day free trial before costing £25-80
- Apophysis - supports a limited range of fractals but lacks help/instructions for users

The user of my application is likely to be one of three main categories:

- A teacher looking to engage and excite students and make maths classes more practical (particular for teaching topics such as complex numbers or iterative/ numerical methods)
- A student learning maths interested in exploring mathematical phenomena or putting them to the test
- Ordinary, non-mathematical people looking to experiment in abstract design or maths

However, my primary **client** for the project will be a teacher from the mathematics department at my school. I interviewed them on their thoughts about the problem and basic requirements for a solution.

“

This tool would be very useful as a form of enrichment to the maths syllabus. I might expect to use it towards the end of term, or early on in the school to show students how exciting and beautiful maths can be, especially given the intriguing nature of fractals. This would also be an interesting extension exercise for sixth form students to investigate applications of complex numbers.

A solution would need to be quick to set up and responsive, to keep students engaged. It should also be visually appealing. There should be the ability to see how the images are generated for sixth form students interested in maths and computing.

”

The Solution

Following my research and a brief discussion with my user, the following requirements are apparent. High level objectives are identified with a '1-xx', low level objectives with '2-xx'.

1-01: The program must be able to produce a rendered fractal at any resolution requested.
1-02: The program must be able to display the output to the user
1-03: The user must be able to move, navigate around, and zoom in and out of the fractal freely
1-04: The user should be able to change the equation being used
    a) The user should be able to input their own iterative equation (i.e. in place of `(z^2)+c`)
    b) The user should be able to select an equation from a list of built-in presets
1-05: The user should be able to save rendered images to a file
1-06: The user should be able to save configured states (i.e. a particular equation with a particular offset, zoom, etc) and restore them
1-07: The program should be able to generate a full screen rendered fractal in under 30 seconds
1-08: The interface must be pleasant to use and intuitive, such that a maths teacher can use it easily without confusion
1-09: The program should offer or come with instructions on how to get started and use each of its features
1-10: The user must be able to change computation and display parameters, such as the iteration limit and colour scheme
1-11: The program must be able to run on both Windows and MacOS machines

2-01: The program should present some kind of preview render which has more minimal parameters (lower resolution and iteration limit) to allow the user to navigate quickly (without having to wait for a full resolution render in between every navigation event)
2-02: The user should be able to navigate using both interface buttons and keyboard keys
2-02: The program should be runnable both as a GUI or as a CLI tool for more advanced users
2-03: The program must be able to parse mathematical expressions from a string including involving imaginary parts and evaluate them efficiently at runtime, including the following operations:
    a) brackets
    b) implicit multiplication
    c) addition
    d) subtraction
    e) multiplication

    f) division
    g) indexing (powers)

2-04: The program must be able to utilise multithreading in order to speed up computation

2-05: The program must be able to store and load render configuration states in a database structure when the user clicks the relevant button

    a) When storing, the program should present a dialog which allows the user to save the render state
    b) The user should be able to name their render state
    c) The database should be stored in an appropriate location on disk
    d) When loading, the program should present the user with a dialog which allows the user to select a saved render state to load

2-06: The GUI must be able to resize and adapt its render resolution to match that of the screen for optimal user experience

2-07: The program should have an instructions area which is easy to find from the main window.

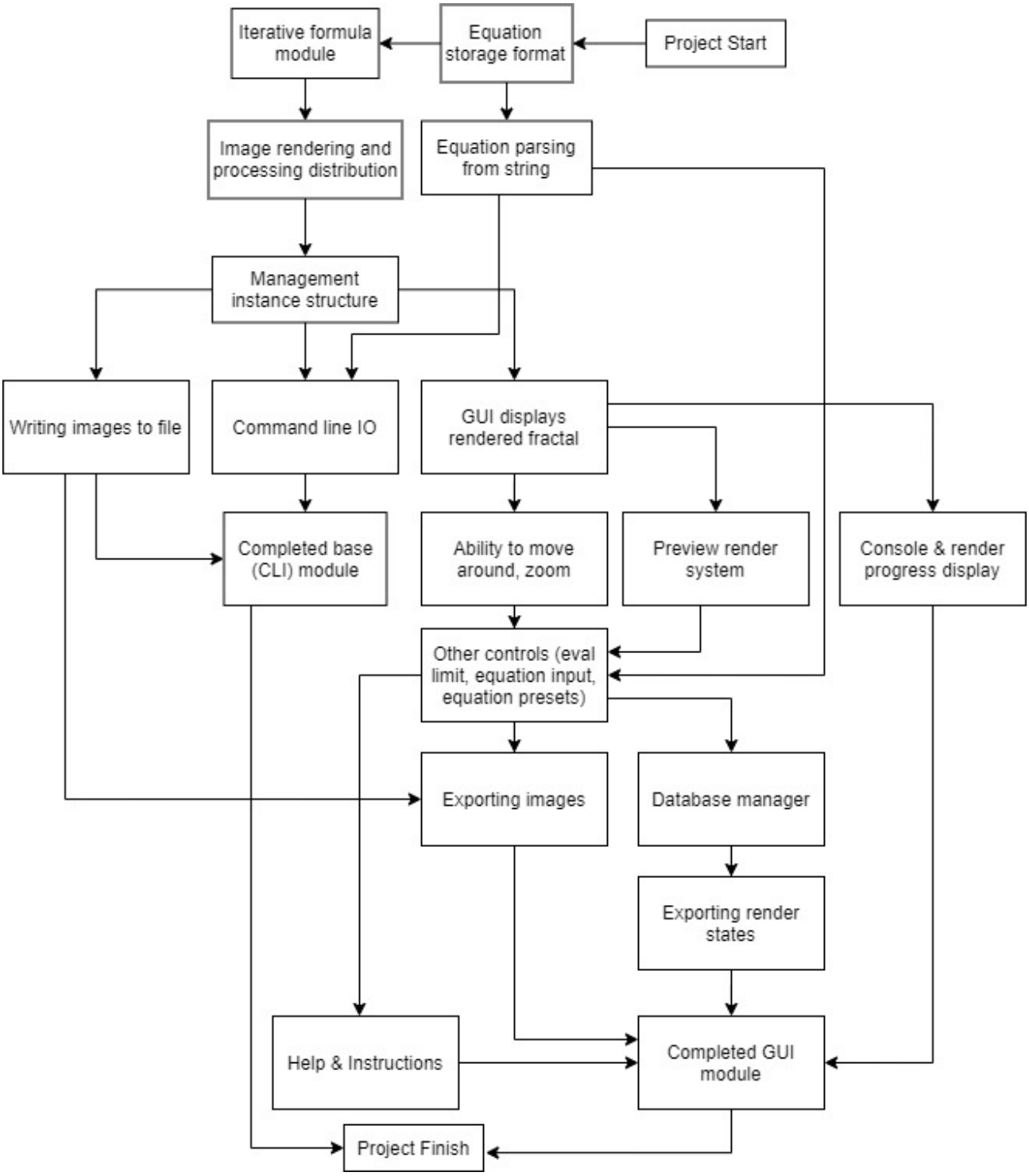
    a) The instructions should explain how to render an image
    b) The instructions should explain how to save and load render states
    c) The instructions should explain how to input custom equations, and also should describe the presets
    d) The instructions should explain the meaning of the colour scheme, zoom, offset and iteration limit parameters
    e) All of these should be explained in a way which a maths teacher or student can understand

2-08: The program should have a sensible requirement of memory to run (less than 500MiB) at any one time, which should remain consistent even if the program is open for a large period of time (dependent on the resolution being rendered at)

2-09: The program should display its rendering progress (if there is a render currently happening) and its current status, to the user

2-10: The program should lock controls such as iteration limit, offset and zoom whilst a render is occurring

Project Plan



Overview of how I plan to carry out the implementation for the project

Necessary Libraries

The solution to the problem necessitates the use of a graphical user interface. For this, a GUI library is necessary in order to produce a consistent experience across platforms.

A wide range of solutions are available for this. My criteria for a package were as follows:

- free to redistribute
- free to use for commercial purposes
- configurable inside another project
- compatible with the target language, in this case C++
- cross-platform
- providing features such as displaying images, buttons, text

The potential solutions which I identified included Qt, GTK+, wxWidgets, FLTK, BoostUI, and gtkmm. However, many of these did not fit my criteria fully, so I finally chose to use 'raylib', a standalone header-only graphics library which meets all the requirements I set out with.

The C++ standard library fortunately has a built-in complex number handling library. However, during my analysis phase I initially experimented with configuring and using multiple-precision complex number libraries. Multiple-precision arithmetic, or arbitrary precision arithmetic, are methods of performing arithmetic which is not limited by the word size of the host system. The complex numbers involved in generating fractals must be very precise, so I considered using arbitrary precision arithmetic libraries such as:

- MPFR (Multiple Precision Floating-point Reliable) - provides extensible floating point data types which can be sized to any necessary precision
- GMP (GNU Multiple Precision) - a similar library but which determines to provide better performance
- MPC (Multiple Precision Complex) - library which builds off of GMP to provide complex number arithmetic at arbitrary precision
- Boost multiple precision complex - the Boost implementation of the same functionality

However, I encountered a number of problems with all or most of these during my experimentation:

1. They caused a huge performance hit when benchmarked compared with the C++ standard complex number library, on the scale of orders of magnitude slower (which is to be expected with any arbitrary precision arithmetic realistically), which would cause significant problems due to the number of calculations necessary in this project and the intended real-time nature of it.
2. They required the target machines to have libraries installed in order to use the project and application, which also would require configuring host machines (going against the aim of the project to provide a simple, standalone, easy to install and use application)

As a result I decided to stick with the C++ built-in complex number library, the only tradeoff being the limited maximum precision of calculations as a result. This has the effect of effectively limiting the resolution or zoom depth possible with the application, as at a certain combination of resolution and zoom, pixels will no longer be able to be uniquely identifiable and their complex coordinates will be rounded to the nearest floating-point number which can be stored. In order to minimise this, in my application I will aim to use the highest precision float possible (‘long double’ in C++).

Design

High Level Design

As mentioned in the analysis section, I decided to allow both command line and GUI operation of my application. As such, my design can be effectively split into two main parts:

- the fractal generator (rendering environment)
- the GUI

The fractal generator is the core of the application which does the actual processing and creation of the rendered image. It will be accessible from both command line and GUI modes, whereby either mode can create a rendering environment with the necessary parameters (zoom, offset, equation, etc) which can be used for rendering universally.

In CLI mode, this can be done by parsing command line arguments and passing them into the rendering environment as render parameters.

In GUI mode, this can be done instead by simply reading out values from the interface which the user can manipulate (e.g. the user presses a button which zooms in and this command is processed by changing a parameter in the rendering environment).

The GUI is effectively a separate module which provides the graphical interface to the user and simply modifies and calls into the rendering environment provided by the core module and displays the results on the screen, as well as other features such as saving images and accessing the database of render parameters.

The rendering environment will be **encapsulated** in a class which can be instantiated to create an independent, contained environment for generating fractals. It will contain class variables which can be assigned or retrieved via getters and setters. It will also provide functionality to render fractals and to parse textual equations (see below), **hiding all the code** necessary to do this from anything which utilises the rendering environment (e.g. the GUI module).

The GUI module will also be **encapsulated** in a class, which is host to the entire graphical system. It will be responsible for creating and managing the main application window, handling button presses and key events, and will keep track of the rendering environments (two will be used, one for the low resolution preview render, the other for the main, full-resolution render). The GUI class will contain various methods for handling presses for each button, navigation events like moving the viewport, as well as the main-loop function which will manage drawing the interface each frame. The only externally accessible method will be the GUI class's main function, which will perform setup such as intialising the class, creating the GUI and starting the graphics main-loop, from which point the GUI will occupy the main program thread until the user closes the window, when the program will exit.

The ‘int main(int, char**)’ function, which is called when the program starts, will contain code to handle either a console start or a GUI start. For a console start, it will parse the command line arguments, configure a rendering environment, perform the render, save the result to a file, and exit. For a GUI start, instead it will simply construct a GUI class and transfer control to the graphics main method.

Additional Modules

Equation Parser

Other smaller subsystems will include the equation-parsing module, a set of static, **stateless** functions to convert a mathematical expression (e.g. "(z^2)+c", which represents the Mandelbrot set fractal) stored as a string into a form which can be evaluated rapidly at runtime. This module will define a number of data types to represent tokens which make up the parts of the equation.

More information about the equation storage and parsing is below, where the algorithm used and data structures required are described. In order to hide as much code as possible from other modules, only the outer-most of these functions will be accessible to the rest of the program, which will simply be called to perform the entire parsing and conversion operation and will return an equation object (see below).

Database

Another element of the application is the ability to save details of configuration states (effectively saving the configuration of the rendering environment, along with a name), via a simple **two-table database** stored in CSV format. This will have its own standalone module, with a class which can be used to manage, update, remove, or create records and handle saving and loading them automatically, again **encapsulating** the database system. This will involve two other data types, representing a record from each table. One table will contain configuration profiles, the other will contain the details of users. The configuration profile table will include fields for:

- Record ID (primary key)
- Config Profile Name
- Fractal X Offset
- Fractal Y Offset
- Fractal Zoom
- Fractal Equation
- Fractal Iteration Limit
- Palette Number
- User ID (foreign primary key of user profile table)

Meanwhile, the user profile table will contain fields for:

- User ID (primary key)
- User Name

The relationship between the configuration profile table and the user profile table will be many-to-one, since many configuration profiles can reference a single user profile (but only one user profile can be referenced by a particular configuration profile).

Evaluator

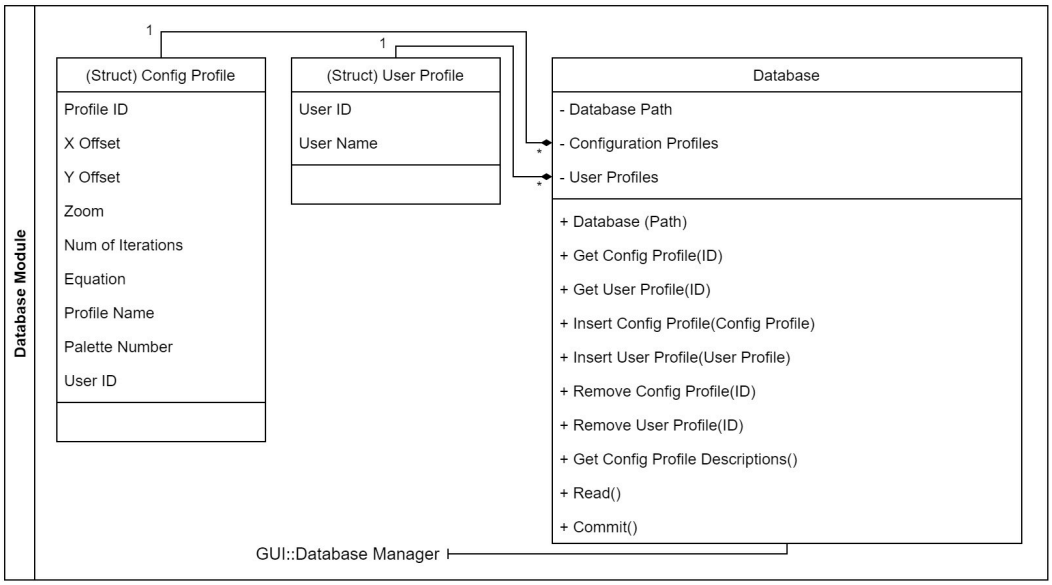
The final module will be the evaluator, which contains the code to handle and evaluate equations parsed by the equation parser. It will **hide the code** to actually evaluate values for any given complex coordinate given, providing functionality to do this which the rendering environment can call for every pixel which needs evaluating. In short, a function will substitute values for variables such as z and c into the stored equation and calculate the result in the form of a complex number. A second function will repeatedly perform this 'single iteration' as described in the analysis section (where the calculation is performed iteratively until the z value of the equation tends to infinity or until the specified iteration limit is reached), and an integer representing the number of iterations performed will be returned (which can then be used to colour each pixel).

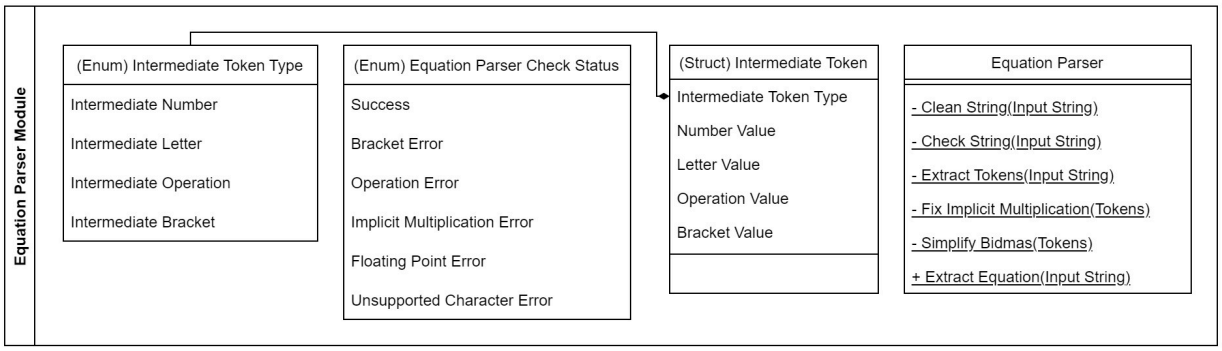
This will also be wrapped in a class to **encapsulate** as the process effectively as possible.

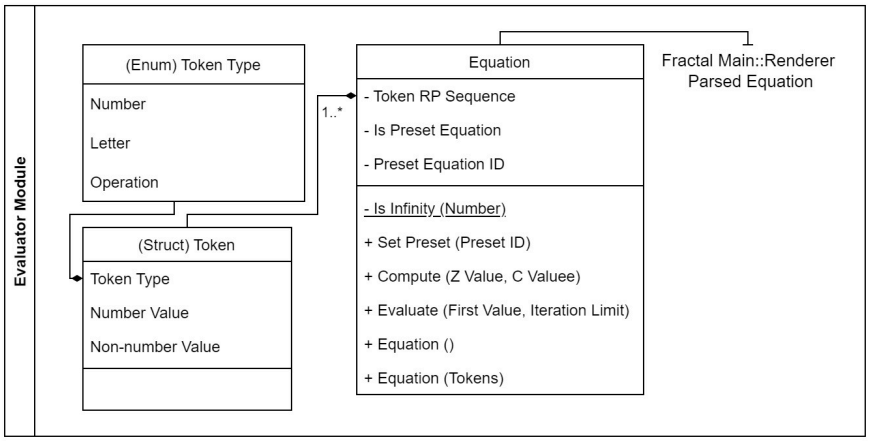
There will also likely be a utilities module which contains miscellaneous functions which multiple other modules might be using, such as a delay/wait procedure, fetching a system path, or retrieving information about an equation preset.

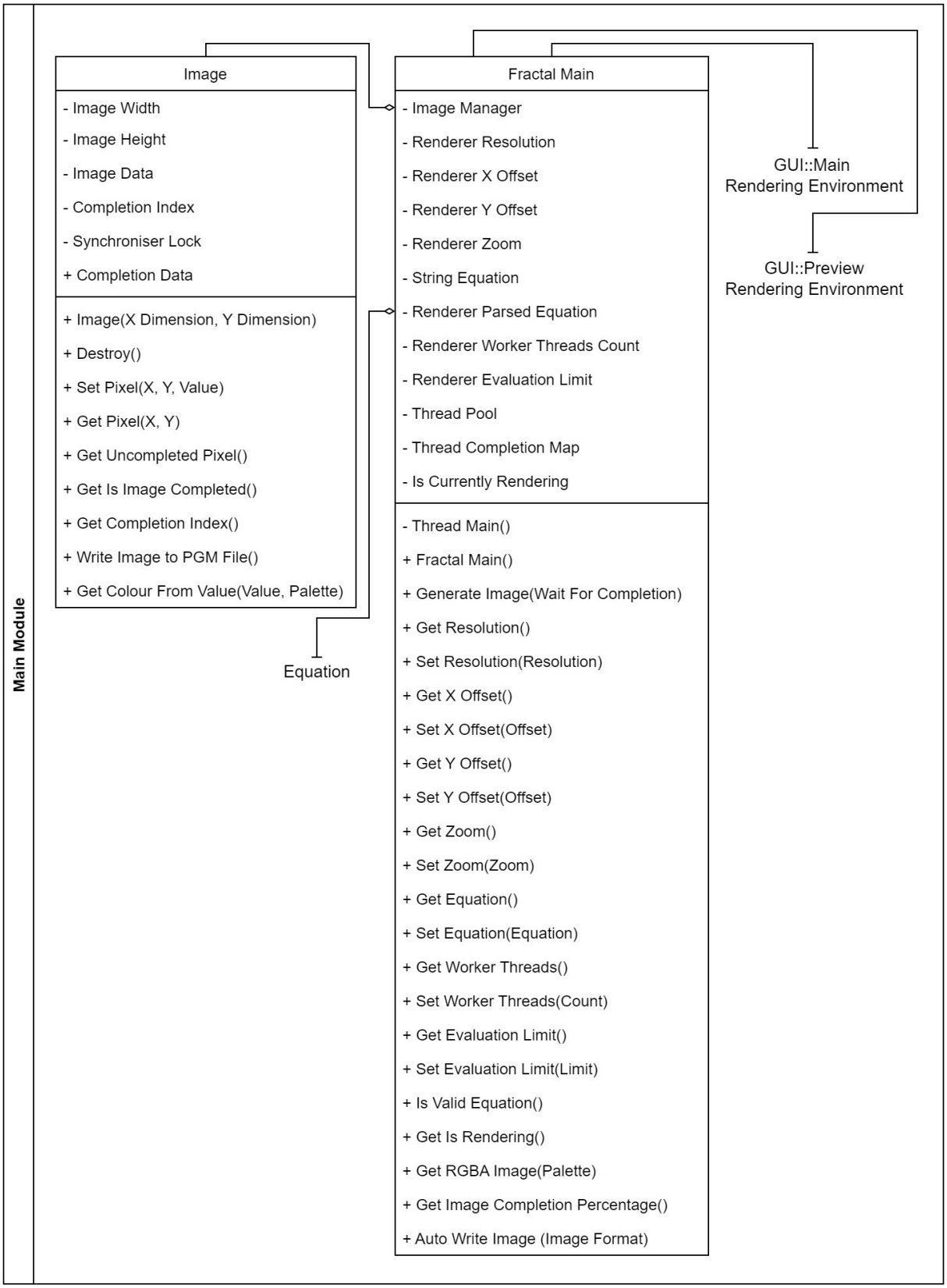
Class Digram

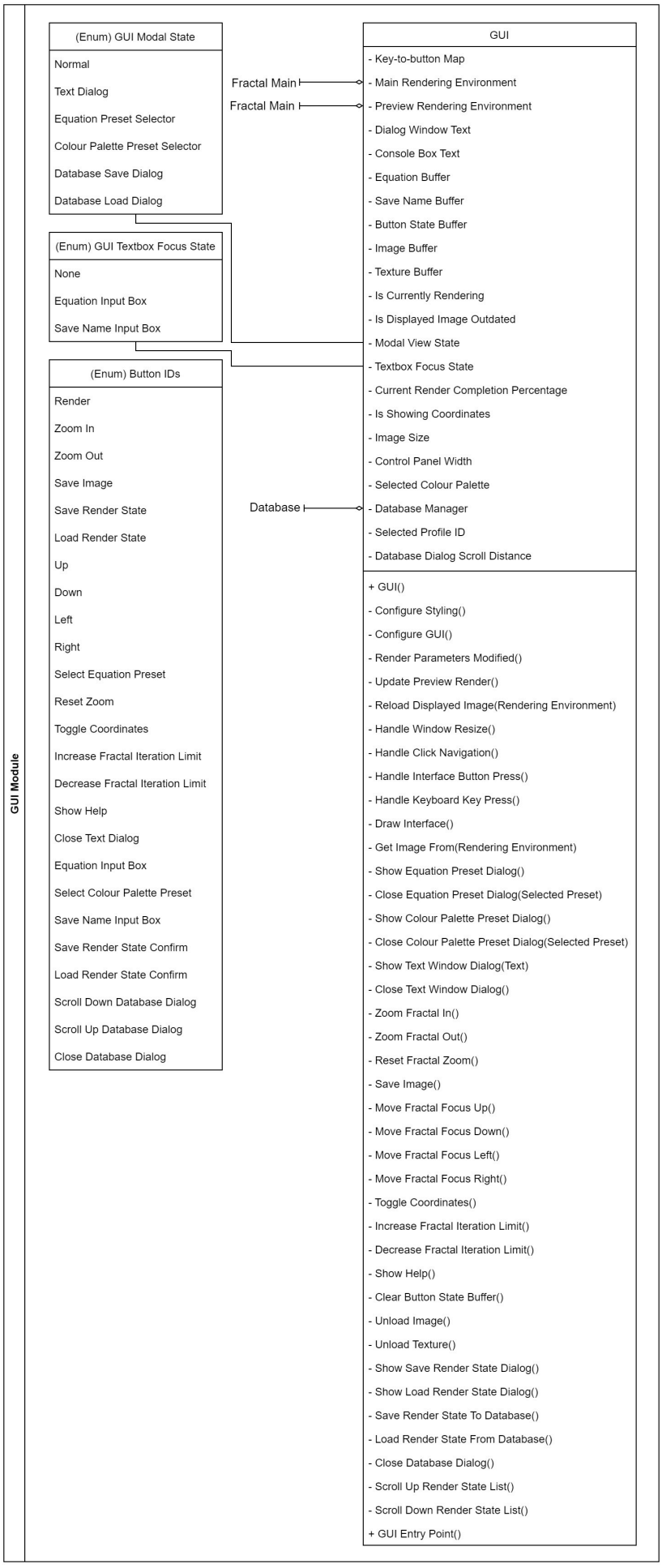
The following pages show comprehensive conceptual class diagrams, which have been separated into modules to make them easier to read. Due to the number of classes, structures and enumerations, the diagram has been split over multiple pages. Connections which split over page borders are shown with a label (as seen with ‘GUI::Database Manager’ which relates to the ‘Database Manager’ field on the ‘GUI’ class).



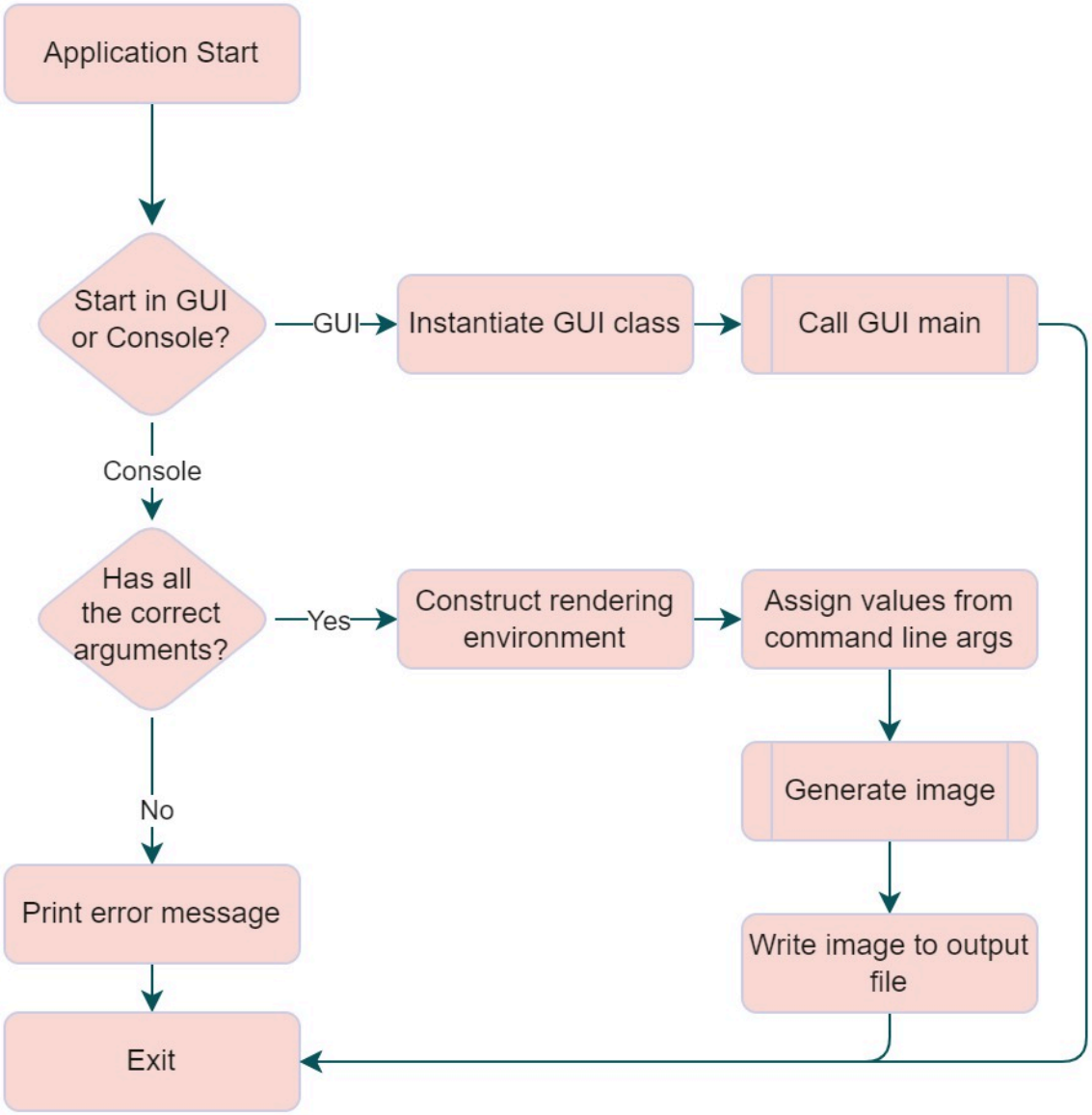


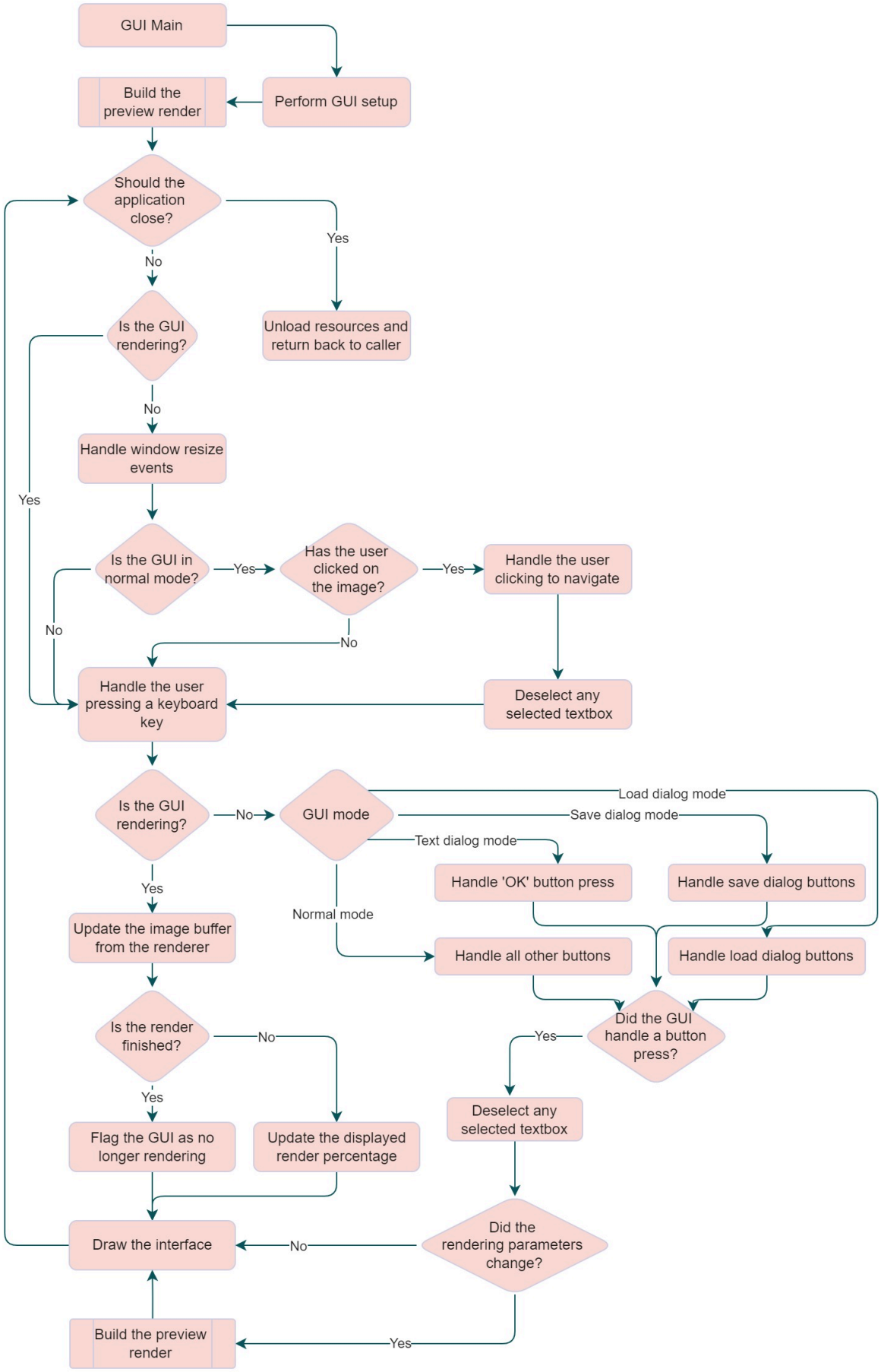


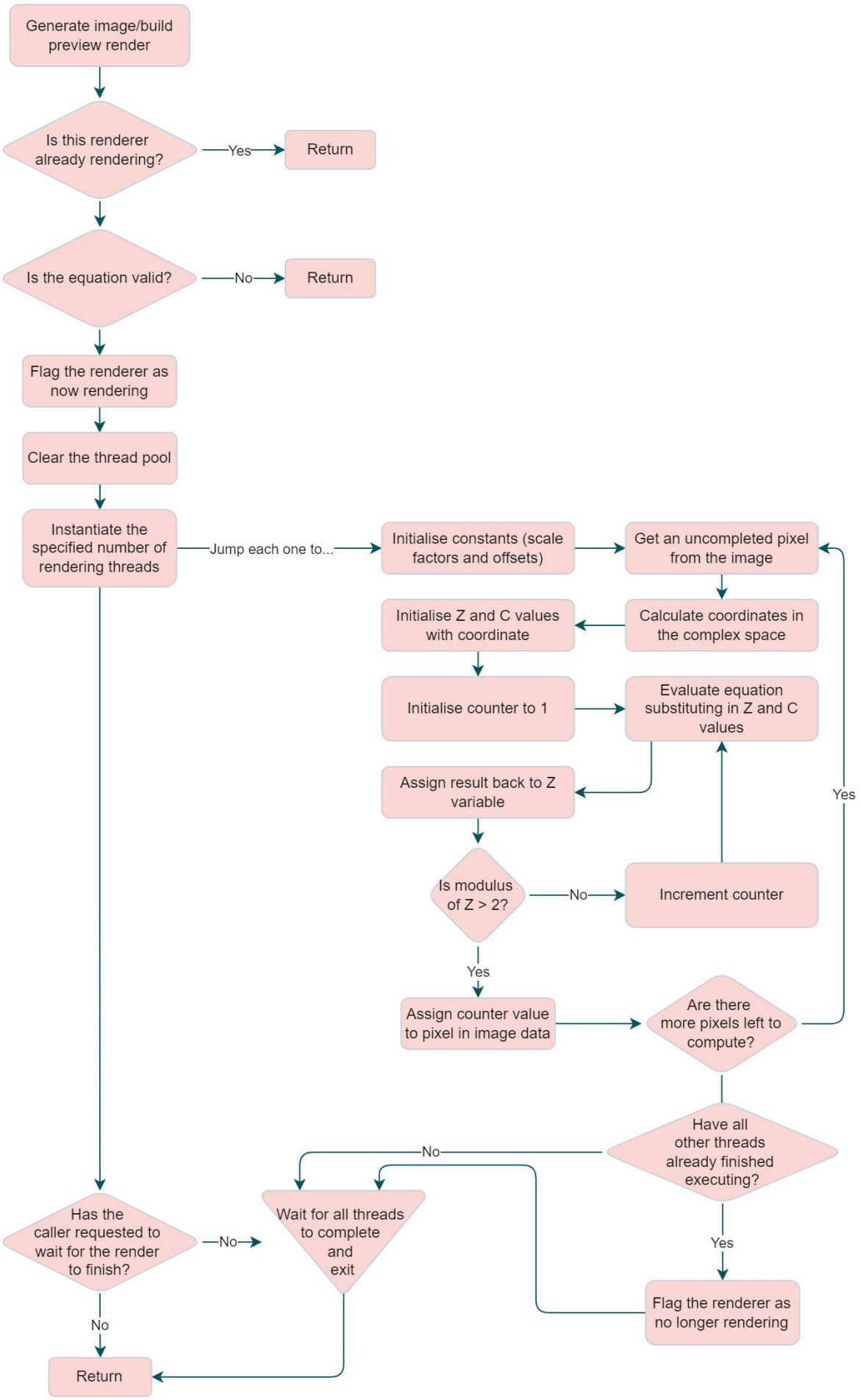

Main Module
Image
- Image Width
- Image Height
- Image Data
- Completion Index
- Synchroniser Lock
+ Completion Data
+ Image(X Dimension, Y Dimension)
+ Destroy()
+ Set Pixel(X, Y, Value)
+ Get Pixel(X, Y)
+ Get Uncompleted Pixel()
+ Get Is Image Completed()
+ Get Completion Index()
+ Write Image to PGM File()
+ Get Colour From Value(Value, Palette)
Equation
Fractal Main
- Image Manager
- Renderer Resolution
- Renderer X Offset
- Renderer Y Offset
- Renderer Zoom
- String Equation
- Renderer Parsed Equation
- Renderer Worker Threads Count
- Renderer Evaluation Limit
- Thread Pool
- Thread Completion Map
- Is Currently Rendering
- Thread Main()
+ Fractal Main()
+ Generate Image(Wait For Completion)
+ Get Resolution()
+ Set Resolution(Resolution)
+ Get X Offset()
+ Set X Offset(Offset)
+ Get Y Offset()
+ Set Y Offset(Offset)
+ Get Zoom()
+ Set Zoom(Zoom)
+ Get Equation()
+ Set Equation(Equation)
+ Get Worker Threads()
+ Set Worker Threads(Count)
+ Get Evaluation Limit()
+ Set Evaluation Limit(Limit)
+ Is Valid Equation()
+ Get Is Rendering()
+ Get RGBA Image(Palette)
+ Get Image Completion Percentage()
+ Auto Write Image (Image Format)
GUI::Main
Rendering Environment
GUI::Preview
Rendering Environment


GUI Module
(Enum) GUI Modal State
Normal
Text Dialog
Equation Preset Selector
Colour Palette Preset Selector
Database Save Dialog
Database Load Dialog
(Enum) GUI Textbox Focus State
None
Equation Input Box
Save Name Input Box
(Enum) Button IDs
Render
Zoom In
Zoom Out
Save Image
Save Render State
Load Render State
Up
Down
Left
Right
Select Equation Preset
Reset Zoom
Toggle Coordinates
Increase Fractal Iteration Limit
Decrease Fractal Iteration Limit
Show Help
Close Text Dialog
Equation Input Box
Select Colour Palette Preset
Save Name Input Box
Save Render State Confirm
Load Render State Confirm
Scroll Down Database Dialog
Scroll Up Database Dialog
Close Database Dialog
Fractal Main
Fractal Main
Database
GUI
- Key-to-button Map
- Main Rendering Environment
- Preview Rendering Environment
- Dialog Window Text
- Console Box Text
- Equation Buffer
- Save Name Buffer
- Button State Buffer
- Image Buffer
- Texture Buffer
- Is Currently Rendering
- Is Displayed Image Outdated
- Modal View State
- Textbox Focus State
- Current Render Completion Percentage
- Is Showing Coordinates
- Image Size
- Control Panel Width
- Selected Colour Palette
- Database Manager
- Selected Profile ID
- Database Dialog Scroll Distance
+ GUI()
- Configure Styling()
- Configure GUI()
- Render Parameters Modified()
- Update Preview Render()
- Reload Displayed Image(Rendering Environment)
- Handle Window Resize()
- Handle Click Navigation()
- Handle Interface Button Press()
- Handle Keyboard Key Press()
- Draw Interface()
- Get Image From(Rendering Environment)
- Show Equation Preset Dialog()
- Close Equation Preset Dialog(Selected Preset)
- Show Colour Palette Preset Dialog()
- Close Colour Palette Preset Dialog(Selected Preset)
- Show Text Window Dialog(Text)
- Close Text Window Dialog()
- Zoom Fractal In()
- Zoom Fractal Out()
- Reset Fractal Zoom()
- Save Image()
- Move Fractal Focus Up()
- Move Fractal Focus Down()
- Move Fractal Focus Left()
- Move Fractal Focus Right()
- Toggle Coordinates()
- Increase Fractal Iteration Limit()
- Decrease Fractal Iteration Limit()
- Show Help()
- Clear Button State Buffer()
- Unload Image()
- Unload Texture()
- Show Save Render State Dialog()
- Show Load Render State Dialog()
- Save Render State To Database()
- Load Render State From Database()
- Close Database Dialog()
- Scroll Up Render State List()
- Scroll Down Render State List()
+ GUI Entry Point()

Program Flow Diagram

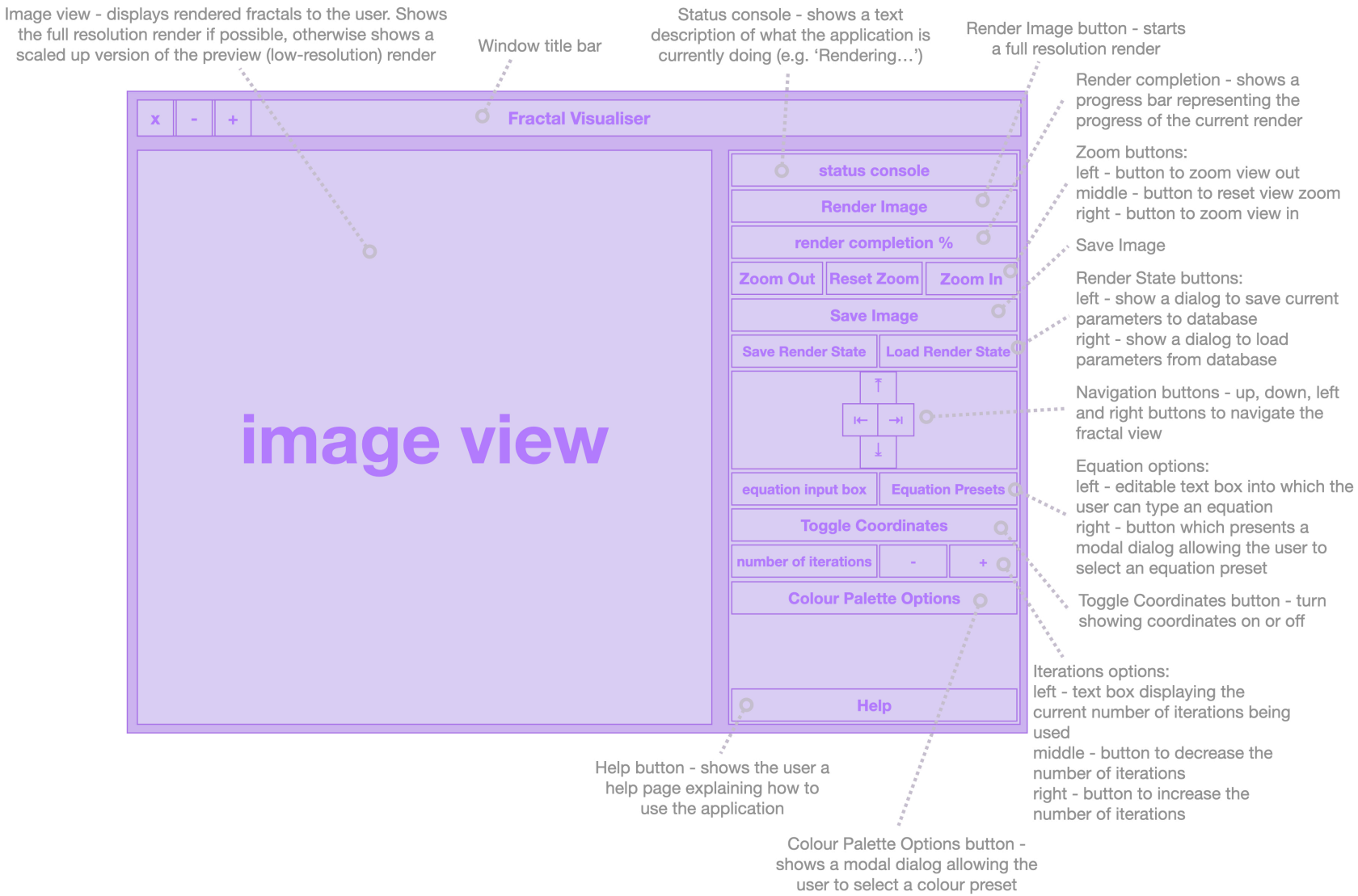



GUI Main
Perform GUI setup
Build the preview render
Should the application close?
Yes
No
Unload resources and return back to caller
Is the GUI rendering?
No
Handle window resize events
Yes
Is the GUI in normal mode?
Yes
Has the user clicked on the image?
Yes
Handle the user clicking to navigate
No
No
Handle the user pressing a keyboard key
Deselect any selected textbox
Is the GUI rendering?
No
GUI mode
Load dialog mode
Save dialog mode
Text dialog mode
Handle 'OK' button press
Handle save dialog buttons
Yes
Update the image buffer from the renderer
Normal mode
Handle all other buttons
Handle load dialog buttons
Is the render finished?
No
Did the GUI handle a button press?
Yes
Yes
Deselect any selected textbox
Flag the GUI as no longer rendering
Update the displayed render percentage
Draw the interface
No
Did the rendering parameters change?
Build the preview render
Yes

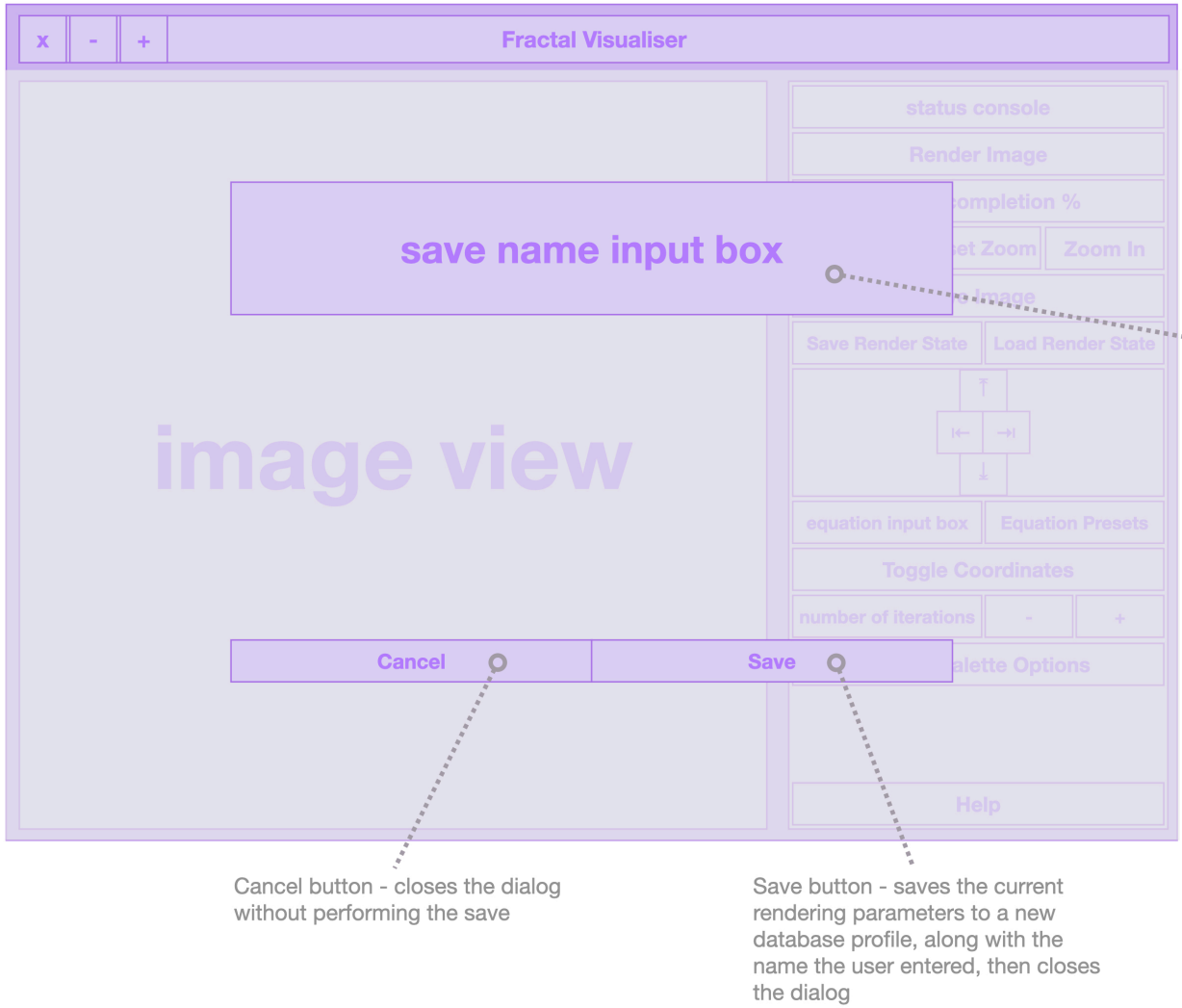

Generate image/build preview render
Is this renderer already rendering?
Yes
Return
Is the equation valid?
No
Return
Flag the renderer as now rendering
Clear the thread pool
Instantiate the specified number of rendering threads
Jump each one to...
Initialise constants (scale factors and offsets)
Get an uncompleted pixel from the image
Initialise Z and C values with coordinate
Calculate coordinates in the complex space
Initialise counter to 1
Evaluate equation substituting in Z and C values
Assign result back to Z variable
Is modulus of Z > 2?
No
Increment counter
Yes
Assign counter value to pixel in image data
Are there more pixels left to compute?
Yes
Have all other threads already finished executing?
No
Yes
Flag the renderer as no longer rendering
Has the caller requested to wait for the render to finish?
No
Wait for all threads to complete and exit
No
Return

Interface Block-out

Base GUI State



Database Save Dialog GUI State

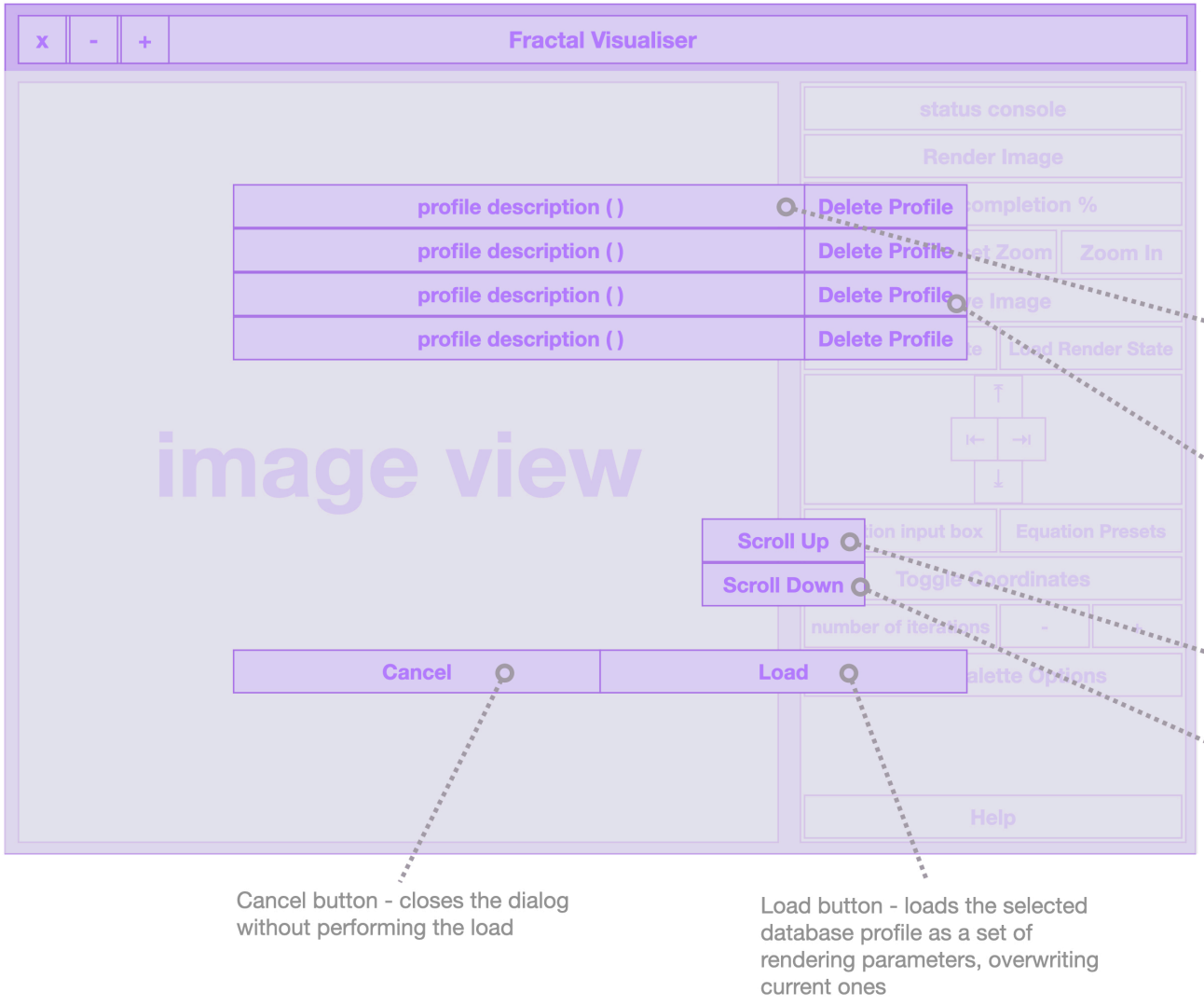


Save name input box - an editable text box which the user can type into to name their render state configuration profile

Cancel button - closes the dialog without performing the save

Save button - saves the current rendering parameters to a new database profile, along with the name the user entered, then closes the dialog

Database Load Dialog GUI State



List of profiles showing their descriptions, and which one is selected
Each list item is a button which selects the profile it represents when clicked

Delete Profile button - each profile is accompanied by a button which deletes it from the database when clicked

Scroll Up button - scrolls the profile list view one item further towards the top

Scroll Down button - scrolls the profile list view one item further towards the bottom

Cancel button - closes the dialog without performing the load

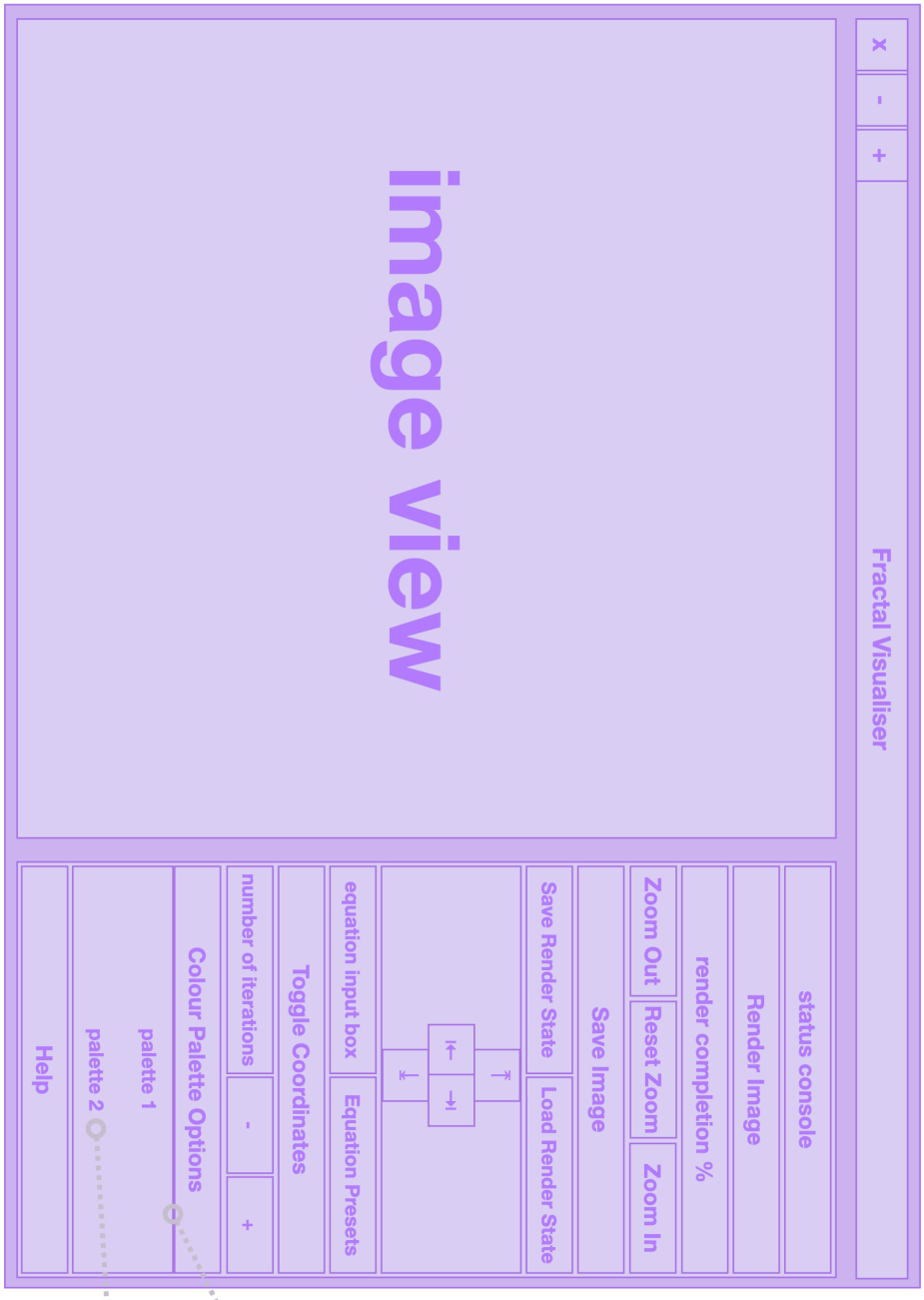
Load button - loads the selected database profile as a set of rendering parameters, overwriting current ones

Equation Preset Dialog GUI State

When the Equation Presets button is clicked, a modal dropdown will appear, containing a button for each equation preset available

Clicking any one of these will close the dialog and make this preset the currently active equation being used by the renderer

Colour Palette Dialog GUI State



When the Colour Palette Options button is clicked, a modal dropdown will appear, containing a button for each colour palette available to choose from

Clicking any one of these will close the dialog and make this palette the currently active palette being used to interpret rendered images

Algorithms & Data Structures

Parsing A Mathematical Expression with Recursion

This is one of the most difficult elements of this project, simply due to the variability of possible cases because this is a complex user-specified parameter.

Equations will be stored using **Reverse Polish Notation**, also known as **postfix notation**. This can be easily stored in an array, and can then be evaluated using a **stack** method.

Extracting the equation from a string is much harder however. I decided to do this in a step-by-step process, as can be seen in the program flow diagram. The string is first cleaned to remove whitespace, then checked to detect errors and invalid equations which would otherwise cause the parser to encounter problems. The parser can then step through the string and convert the sequence of characters into a sequence of intermediate tokens, which are much easier to process afterwards.

This is done by tracking the type of the character currently under inspection (i.e. whether it is a digit, letter, '.', operation, etc) and using changes in the current character type to decide when a new token begins (e.g. when the last character was a digit and the next is a '+', the parser can determine that a numerical constant has finished being read and that an operation token has just begun).

Handling brackets is the ideal situation to use a **recursive function**; this can be done by simply extracting the contents of each set of brackets, and feeding it back into the same tokenising function. The result can then be inserted into the resultant array of tokens of the outer function (and so on as far as necessary until the base case, the outermost function call being performed over the whole equation, is reached again), so that the entire equation is processed including all sets of brackets.

The next step of this is to turn the sequence of 'intermediate' tokens into a finalised array of tokens formatted as postfix notation. Postfix notation is laid out with the operator after the operands for each binary operation.

Before this finalising step, mathematical processing must be performed, which means obeying mathematical rules, specifically the BIDMAS rule for order of operations. This is done simply by iterating over the sequence of tokens and pairing them up over an operation, and wrapping this in brackets (making post-processing of the equation very easy). This is done in multiple passes, one per operation type, so first indices, then division, and so on, following the order of operations in mathematics.

However, another issue must be fixed as well. A common practice in mathematical notation is to use 'ab' instead of writing 'a * b'. This is known as implicit multiplication, and for the purposes of the equation parser it needs to be translated from the shortened form into the explicit form, by replacing instances of the form 'ab' with the form '(a*b)'. Order of operations still needs to be taken into account, so fixing implicit multiplication must be done in between the 'division' and 'multiplication' passes of the BIDMAS simplification phase.

Another mathematical practice which must be observed is the use of '-' as a unary operator. Binary operators are operators (like '+', '-', '*', and so on) which take two arguments and return a single output. Unary operators however only take a single argument. As a result, instances of '-a', where 'a' is any other value (e.g. a bracket, a number, etc), with '0-a'. This transforms the '-' into a binary operator, making it easier to process.

However, once all this is done, the finalising/post-processing of the equation is relatively simple. Because of the simplification phase, we can be sure that the sequence of 'intermediate' tokens will be formed only of a single <token><operation><token>, where each token may be a bracket (which contains another set of <token><operation><token>), a number, or a letter (out of z, x, y, c, a, b, i). In this way, tokens can be easily reordered into postfix notation; if they are simple tokens such as an operation or a number they can simply be copied out, otherwise they will be post-processed **recursively**, once again by calling the post-process function and inserting the result into the postfix token sequence from the outer function (again, this process of recursion is called down each set of brackets, until each sub-call has returned its results and the outermost function has been reached again).

This will generate a simple, linear, no-parsing-required sequence of instructions effectively which can be easily evaluated many times over by the fractal evaluator module.

Evaluating Mathematical Expressions using a Stack

After the majority of the work has been done by the parser, the fractal evaluator has minimal processing to do. This is deliberate, as the module must evaluate the equation many times (a maximum of $ImageSize * ImageSize * IterationLimit$ times, which on an average screen with average settings is very large). This is done by reading along the array of tokens produced by equation parser and treating each as an instruction.

A **stack** will be used to keep track of the computation state. Different tokens are treated differently:

- Number tokens will push a constant onto the stack with the value specified
- Letter tokens will be replaced with the relevant value (e.g. the letter token 'z' will be replaced with the current value for the z variable, see analysis for explanation of fractals) and pushed onto the stack
- Operation tokens will be executed over the top two items on the stack. These top items will be popped and the result of the operation will be pushed
- At the end of the token sequence, there will be only one value left on the stack, which can be copied off and returned.

This is done many times over for each pixel, until the computed value z is greater than the threshold, which is typically set a 2. Again, see analysis for explanation.

A major optimisation which can be made for this system is for built-in equation presets. The speed of evaluating equations in this way is limited by the overhead of performing the stack operations. However, if the equation being used is already known (i.e. it is one of the application's presets) evaluating equations in this way is unnecessary, and so instead for the equation presets (such as Mandelbrot and Julia sets) the evaluation function can be hard-coded and switched to at runtime, providing a major performance boost. Such hard-coded calculations avoid the overheads associated with stack operations and iterating over an array, and can also be optimised in terms of machine code heavily at compile-time by the compiler.

Dividing Up Workload with Multithreading

Generating the entire fractal image requires a lot of processing, and would take an extremely long time on a single execution thread. Nearly all modern computers have more than one logical processor, meaning they have the capacity to run multiple operations simultaneously, allowing multiple pixels to be computed at the same time. Excluding overheads, this would effectively divide required computation time for the whole image by the number of logical processors, vastly increasing performance (especially on more powerful machines).

Multi-threading is key to achieving as realtime a result as possible. The command line arguments will allow the user to specify the number of worker threads to use, whilst the GUI mode will automatically detect the optimal number of threads to use for the given host system and configure the rendering environments accordingly.

From here, the workload of rendering the image must be divided up between the worker threads. The mechanism to be used is very simple: a custom image management class keeps track of not only the image data containing computed pixel values (i.e. the outputs for each pixel from the evaluator module), but also of whether or nor a pixel needs to be calculated or not. This way, each worker thread simply requests a new pixel coordinate from the image manager, which would then mark that pixel as 'in-progress'. The image class should keep track of which pixel should be assigned next to minimise wait times when threads request new pixels to compute. After each pixel is computed, it can be assigned back to the image data of the image class, which will flag the pixel as completed at the same time.

Some locking is required on the image manager class to prevent collisions, but this will be minimised in order to maintain performance without encountering collisions.

When the image is complete, the threads can be automatically halted as computation is complete.

Technical Solution

On the following pages are inserted the source files for the implementation, including the Makefile. A configured, buildable version of the project is available on https://github.com/JkyProgrammer/HyperFractal.

Each of the files below has been formatted with a comment at the top which states the name of the file it originates from.

Mini-Contents for Implementation

// Also TODO

Testing

Test	Requirement	Test Description	Test Data/ Setup	Expected Output/Result	Actual Output/Result	Test Result