University of
St Andrews

School of Computer Science

CS5001

Mandelbrot Set Explorer

Parts completed:

- All basic requirements
- Different Colour Mappings extension
- Full Undo, Redo, Reset extension
- The user is given full capacity to change any of the Mandelbrot calculation inputs
- Save and Load the Mandelbrot set from a file extension

ID: 170018405

Table of Contents

Table of Figures

1.0 Program requirements

The purpose of this practical was to develop a Mandelbrot set explorer graphical user interface (GUI) to allow users to both visualise the Mandelbrot set, and explore it through interactions with the rendered image displayed by the GUI. As per the specification, users should be given ways to:

- pan/zoom the rendered image
- change the number of maximum iterations of the complex formula $z_{n+1} = {z_n}^2 + c$, which in turn allows us to check if Z remains bounded within a certain distance from the origin of the complex plane - allowing for more detailed/accurate images to be rendered
- toggle a superimposed magnification estimate to be rendered on the Mandelbrot Set image
- Undo and redo the last zoom or pan operations.

The first step to developing this application involved thinking about how such an application could be implemented in a way so that it is as future proof as possible.

Clearly, there needed to be a separation between the core logic (the model) that determines how a Mandelbrot set can be calculated and manipulated, and the code required to actually generate the user interface that displays the results of the calculations. The reasons for this separation are plentiful, however, the most important reasons were:

1. To allow many different GUIs (on the same or different platforms) to be developed in a way that they can all share the same logic code without needing to rewrite the logic code in a specific way for each specific application.
2. This means that there is increased efficiency in terms of the amount of code written, but also means that if a type of calculation needed to be updated (to improve efficiency for example), or a new calculation was implemented, the code need only be updated in a single place for the result to take effect in all GUIs developed using the model.
3. A GUI could be easily adapted to use a different model since only a loose coupling needs to be implemented.

Yet, this application is not so complex as to require a separate controller to perform complex GUI-based logic before the model is manipulated. In fact, it was easy to imagine that arguably the most complex functions the "controller" would need to perform is adjust the bounds that the model uses to calculate the Mandelbrot Set to be displayed. Therefore, the Model-Delegate design pattern was deemed to be an appropriate design strategy to peruse in the development of this application.

2.0 Model design and implementation

As discussed, the model needed to be designed in a way that provides flexibility in how the Mandelbrot Set can be generated to work with both GUI and none GUI type applications. The first step was thus to determine how flexibility can be ensured.

The design process began with an analysis of the Mandelbrot Calculator class provided. This class contains both some initial static fields used to calculate a default Mandelbrot set, and a method that calculates and returns the set as a 2D array of integer values - given some parameters. Of particular interest were the parameters the calcMandelbrotSet method accepts as an input. Those parameters in effect represent the state of any Mandelbrot set that can be generated. Thus any operation that requires the model to calculate a Mandelbrot set, or make a change to an existing set would hence entail a change to one or more of the eight inputs.

It was thus reasoned that the model essentially needs to give users the flexibility to change those parameter separately, as well as provide methods that changed those parameters in ways that required specific computations (like zooming and panning). Furthermore, thinking back to the

requirements, in order to support undo and redo operations, it was determined that the state of previous Mandelbrot Sets needed to be kept in memory. For this reason, the MandelbrotState class was created.

The MandelbrotState class can be found in the model package, and it is essentially an extended MandelbrotCalculator class that stores the eight extra parameters (the state) used to calculate a particular Mandelbrot set. The reasons why it was determined that the calculator class should be extended by the state object class, as opposed to keeping both classes separate, were to ensure that the state class provided the extended functionality of the Calculator class without requiring users to make calls to the Calculator class methods every time they needed to calculate the set corresponding to the state values stored in the state objects or the default values in the Calculator class itself. Furthermore, given that the fields in the Calculator class are static anyway, each state object won't be storing its own copies of the default input parameters, which ensures that the state objects remain relatively small and easy to store [1]. Choosing not to inherit the class on the other hand would mean having to edit the class so that either the fields are made public and the methods are made static, or static getter methods are added to get the default fields. Of course, another alternative option is to add constructors to the class and change the static final fields to none static variables, but what benefit does this really give if the class can simply be extended. All the options described would constitute forms of bad object oriented practice.

On the other hand, by inheriting the default parameters through inheritance, more flexibility can be given to the user in how state objects can be generated. The MandelbrotState class in fact provides three constructors to ensure flexibility of use. The first constructor takes only two parameters to set the the resolution of the Mandelbrot set to be generated, and uses the default parameters of the Calculator class to instantiate the initial state. The second constructor takes all eight parameters as inputs to initialise a completely custom state object, and thirdly, the last constructor takes another MandelbrotState object as a parameter to construct a state object that is identical to the one passed in. The latter may seem like an odd thing to do, but, it makes sense if a user wanted to construct an object that has identical fields to another state object with the exception of a few fields that are different. The user can then use one of the provided setter methods to make the appropriate changes that differentiate the new state object from the old. This is especially useful if a panning or zooming action was performed, and only the min and max real or imaginary parameters were changed. The old object can then be copied, and stored in a list to keep track of the changes made.

In the spirit of maintaining flexibility, the state objects provide little checking in terms of the inputs that can be passed to the setter methods. This means that users can flip a Mandelbrot set by passing a max Real value that is smaller than a min Real value if they wished. The only exception is that the resolution parameters are checked to ensure that a valid Mandelbrot set array can be generated. These checks are found in both the constructors and setter methods used to set the resolution. If invalid inputs are found, illegal argument exceptions are thrown.

Finally, the MandelbrotState class implements the Serializable interface to allow users to save state objects to file. This gives users of this class an ability to allow their users to save progress made while manipulating Mandelbrot sets.

Now that a state class which allows us to work with instances of Mandelbrot set states was fully designed and implemented, a class that acts upon the state objects to provide the advanced

[1] State objects needed to be small and easily storable for two reasons: 1. Redo and Undo functions require a way for the changes made during the manipulation of the Mandelbrot set to be stored. State objects provide a very efficient way to store the changes simply by storing states in a list. It therefore follows that the smaller the state object, the more previous states that can be stored in the list. This is why the state objects do not store the Mandelbrot set, but, only store the parameters used in its calculation. 2. The objects can be written and read to/from a file relatively quickly, and the file generated will require little space.

functionalities to be utilised by both GUI and none GUI applications was developed. This class was named the MandelbrotSetGenerator, and can also be found in the model package.

The MandelbrotSetGenerator is essentially a MandelbrotState collection class that provides extended functionalities (in addition to the basic setter methods provided by state objects) to change the render bounds, pan, zoom, and even undo/redo/reset the generated Mandelbrot sets. The class can achieve the latter trio in particular through its implementation of two stacks (prevStates and nextStates). The stacks are used to store MandelbrotState objects that contain the states of the previous (or next) displayed sets. This is where the third constructor of the MandelbrotState class becomes particularly useful.

Let's say a user wants to shift the calculated MandelbrotSet by a given number of pixels in the real and imaginary axis. The user would call the shiftBounds method and pass the number of pixels to shift the calculated set by. This method will then make a calculation as to how much shift each pixel translates into real and imaginary numbers. Following this, the method will construct a state object that is identical to the current state, change the bounds in the new state object to represent the pixel shift, then add the now "old" state object in the prevStates stack before finally calling the updateMandelbrotSet method to calculate a new Mandelbrot set using the new state object. The updateMandelbrotSet object will then also fire a property change event, and this is exactly how the model can be loosely tied to a GUI application, yet, not include any GUI specific code. In fact, it can be observed that all methods that cause a change in the calculated Mandelbrot set essentially end up calling the updateMandelbrotSet method to both calculate the new set, and fire the property change event.

This class ensures flexibility even though it provides a single constructor because this constructor takes in a MandelbrotState object, which, as discussed can itself be instantiated using default values if needed. It was of course possible to provide an additional no parameter constructor that creates a default state object using some resolution default parameters, however, it was determined that this is unnecessary. Extra flexibility is given by the fact that the user can also get the current state object, and thus access all the getter and setter methods of this object. However, it should be noted that the updateMandelbrotSet method is private, so even if a user alters the fields of the current state object, the effect will not be calculated. The only way the effect could be visualised is if the user calls the undo followed by redo methods. However, there shouldn't be any need to change the state of a set in this way as sufficient methods are provided to change any combination of state fields including setting a completely new state object. The get state method is thus solely intended to allow users to use the get methods if they need to use those values to display them for example.

Finally, the class also provides a toString method that returns a string in the form of a Matrix containing the Mandelbrot set values. This method can be useful for none-GUI programs as well as testing. In fact, it was used to test the class performed as expected.

3.0 Model testing

To ensure the functionality of the model (i.e. the combination of the MandelbrotState and MandelbrotGenerator classes), some preliminary testing was carried out in a main file found within the test package.

This test while preliminary helped to establish mainly that the setResolution, undoState, redoState, reset, setBounds, setSqRadius, setMaxIterations, and shiftBounds methods all performed as expected. The test also includes checks for several edge cases. For example, undo and redo calls are repeatedly made even when the stacks holding the undo and redo states no longer have any more states within them. Moreover, the bounds are changed so that the min and max values for both the real and imaginary axis are flipped. Negative sq radius values are then tested, as well as negative max iterations, and negative resolution changes.

As it can be seen, the majority of the tests were visual. However, some of the tests are checked automatically. None the less, given that all functions would essentially be testing by the GUI, further automation of this test class was no perused (to save time).

4.0 GUI design and implementation

4.1 Design

The GUI was designed using the Swing package provided by Java. The first step taken to design the UI was to list all the functionalities required by the specification, and any additional functionalities that I wanted to include in the GUI. By listing the functionalities, the task became to separate them into “easily accessible”, and “can be hidden” from immediate sight.

Following a minimalistic design strategy, which sticks to existing mental models for how desktop GUI applications (specifically those that handle the manipulation of images) should operate, it was decided that a combination of a tool bar and menu bars were the best options to produce a minimalist UI that is intuitive to the user and makes the rendered image the centre of attention.

Figure 1 shows the completed design of the GUI.

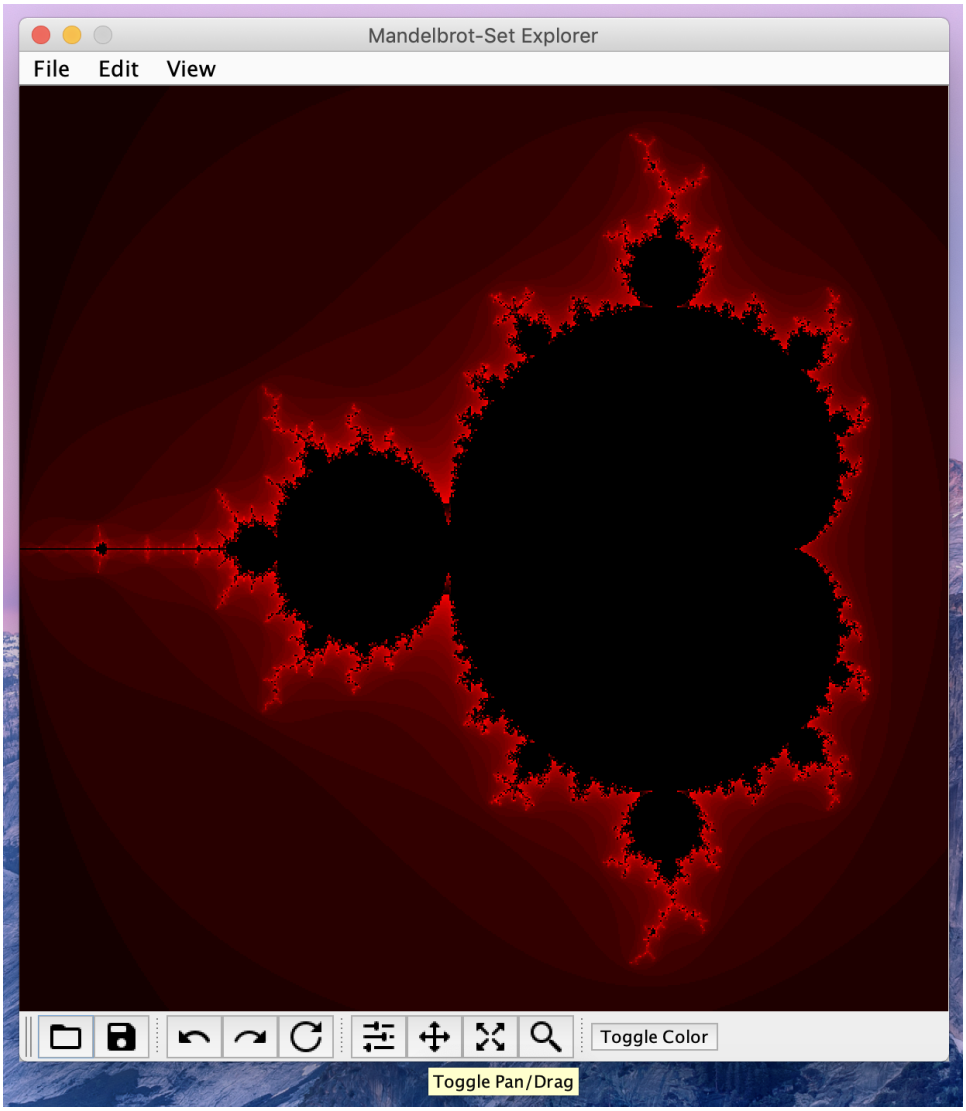

Figure 1 - Completed design of v.1.0 of the Mandelbrot-Set Explorer

As the figure shows, the controls for the UI are placed on the edges, making the render itself the centre of attention. There is a menu bar with the menus: file, edit, and view that should be familiar

to any desktop user, and a toolbar which can be repositioned by the user both within the program window and even outside the window.

A decision was made to avoid using text labels to describe what each button on the tool bar does. Instead the toolbar uses common icons to convey meaning. This is useful for the following reasons:

1. Users who do not read or understand English could still learn how to use the tool bar relatively quickly (since the icons use common symbols found on desktop and mobile applications).
2. Text would make the buttons larger than they need to be, and the buttons themselves need to be reread every time the user wanted to remember which button achieved which function.
3. Therefore icons are much more rapidly distinguishable.

In the figure above, from left to right, the icons represent open (to open a saved Mandelbrot set), save (to save the current set to a file), undo, redo, reset, change maximum iterations, toggle pan mode, toggle zoom mode, display magnification estimate, and toggle colour. In any case, the icons will all display a custom tip indicating what function they do if the user was to hover the cursor over them.

While re-evaluating the design (by asking people to guess what each icon does), it was determined that a better icon needed to be made to replace the reset button icon. This is because a large minority indicated that they thought the icon meant “refresh”. Future work would also include changing the icon for toggling the colour. However, more on this is discussed in the implementation section.

The menu bar on the other hand provides all the functionalities offered by the tool bar to provide an alternative methods of use. Figure 2 shows the contents of the edit menu. What should be noted is that each option contains a key combination shortcut for power users to make use of. The key combinations also make use of existing mental models. For example, undo follows the convention of being Ctrl+Z.

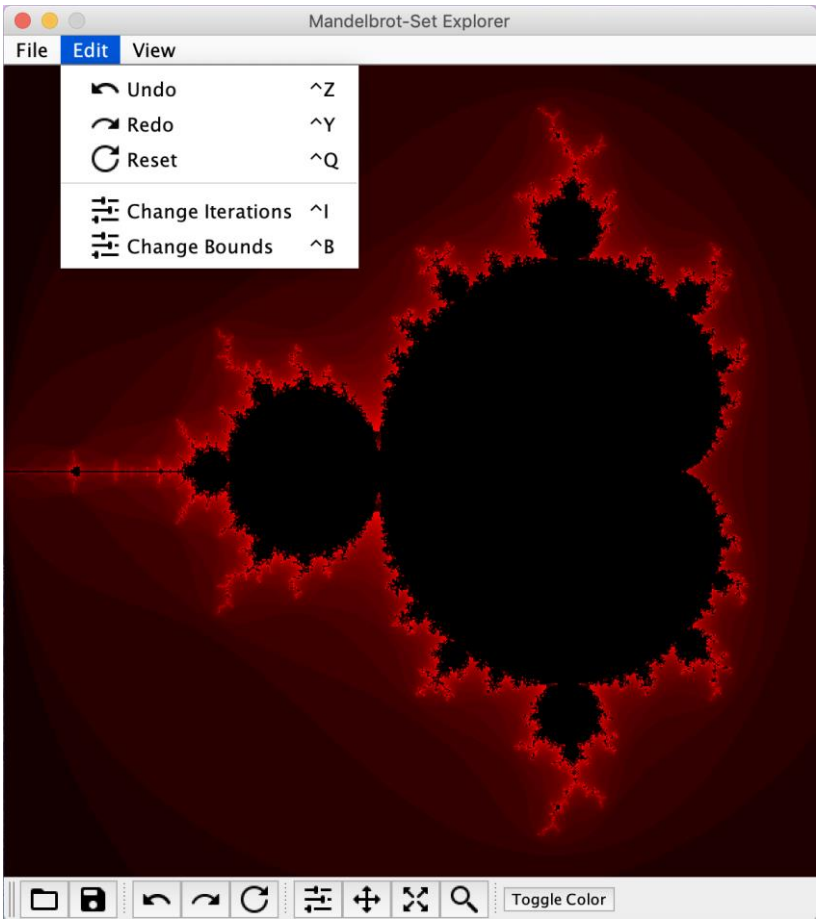

Figure 2 - menu bar edit menu

4.2 Implementation

Given that fact that the GUI planned consisted of three main parts (the graphical panel, toolbar, and menu bar), four classes were initially implemented, where each class was intended to handle a particular functional requirement. For example, the toolbar code was implemented in a separate class in the guiDelegate package named MyToolBar. This class extends JToolBar, and implements all the code that makes MyToolBar specific to the Mandelbrot-Set Explorer program. Similarly, the menu bar follows a similar strategy as well as the JPanel on which the image is rendered. Separating the code in this way is helpful so that the different GUI components can be easily edit separately form the other components. In fact, the GUI components (like the toolbar) can be replaced all together without even having to delve into the code that details how the menu bar should look. This design strategy makes the code easily modifiable, and reusable. For example, this same toolbar could be used in another application.

Acting as a spine connecting all the GUI components into one JFrame, is a class named MandelbrotGuiDelegate. This can be considered the "main" class on the GUI aspect of this application. In this class, there are methods to setup and add the menu bar, toolbar, and graphical display panel to the main JFrame. Moreover, the class contains all the control functions that affect the model and the GUI. This means that instead of implementing a control function to save the current Mandelbrot set to a file in both the menubar and toolbar classes, this method needs to only be implemented once in the delegate class, and the menubar and toolbar action listeners can then call this shared method within the delegate. This is helpful to reduce repeated code, and helps separate the responsibilities of the different classes – making the GUI easily extendable.

The delegate class also implements the PropertyChangeListener interface, and registers itself as an observer of the model. This is how changes in the model are observed and updated throughout the GUI.

The MyGraphicalDisplayPanel class is where the code used to render the Mandelbrot set is contained. The class currently has a single constructor which sets the size of the JPanel based on width and height values contained in a configurations file (Config.java) found in the guiDelegate package. The class also implements the MouseListener and MouseMotionListener interfaces to detect interactions with the images displayed.

Initially in the paint method, the Mandelbrot set was rendered in black by drawing a 0 length line (a pixel) at each point where the number of iterations undertaken in every location equalled the number of maximum iterations. Later colour was added through the implementation of a ColorMixer class which provides a method to change the colour at each location based on the number of iterations performed, and, a colour choice from one of the statically defined public colour scheme fields. This colour class was implemented in a way so that it may be extended in such a way as to allow more colour schemes to be implemented. None the less, one problem with the way this colour class was implemented is that it does not allow users to define their own colour mappings. In a future implementation, this class could be changed so that it allows users to choose colours from a colour wheel (for both the background and the Mandelbrot set), and change the relative effect the number of iterations made has on the colour shift.

5.0 Demonstration of system operation

The following figures demonstrate the system operation.

To launch the GUI application, navigate to the src directory, and type the following commands in the terminal:

$$javac\ model/*.java$$

$$javac\ guiDelegate/*.java$$

$$javac\ main/*.java$$

Now that the source code has been compiled, the application can be launched with the command:

$$java\ main/MandelbrotMain$$

This will launch a GUI which contains a Mandelbrot set image with the width and height dimensions defined in the Config class inside the guiDelegate package.

To perform a pan operation, the mouse left button is held down, and the mouse is dragged towards the direction the image is to be panned. Upon release, the image will be panned in the direction indicated by the grey line drawn and to that magnitude.

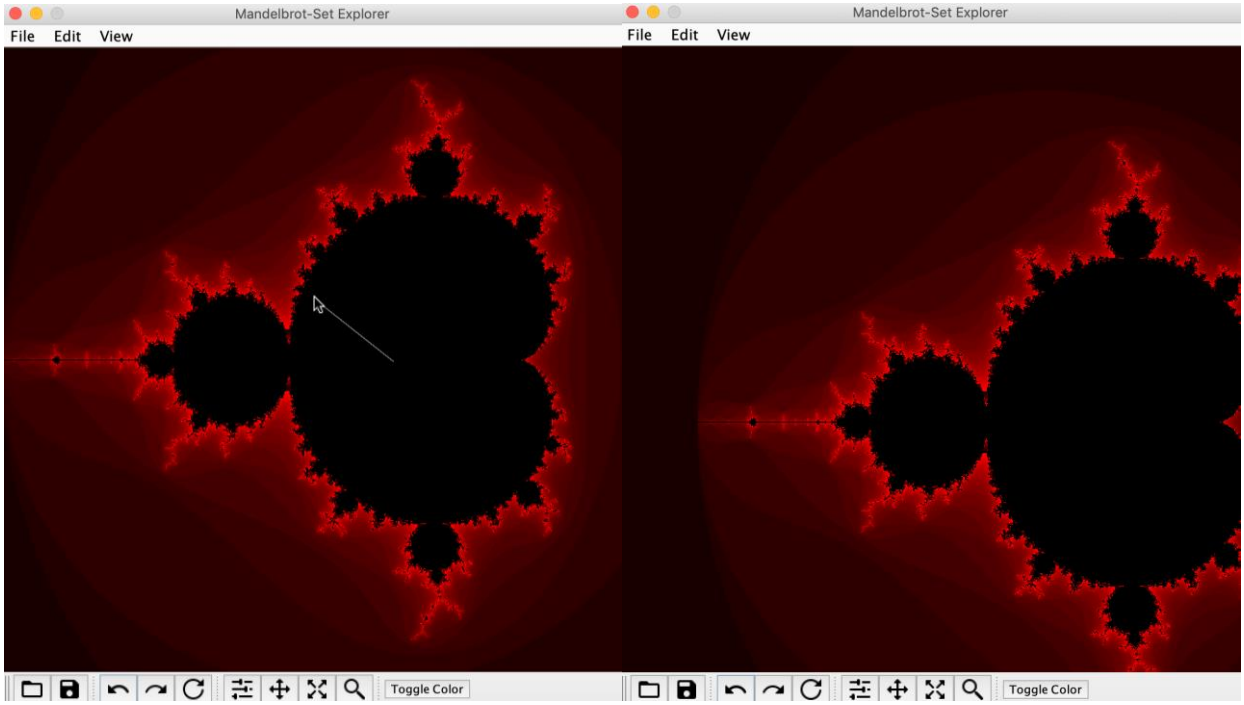

Figure 3 - pan operation

To perform a zoom operation, the zoom Toggle zoom button (3rd from the right on the toolbar) needs to be click. Then, in a similar manner to panning, the left mouse button is held down, this will initiate the render of a square region defining the bounds where zooming will take place. An example in shown below.

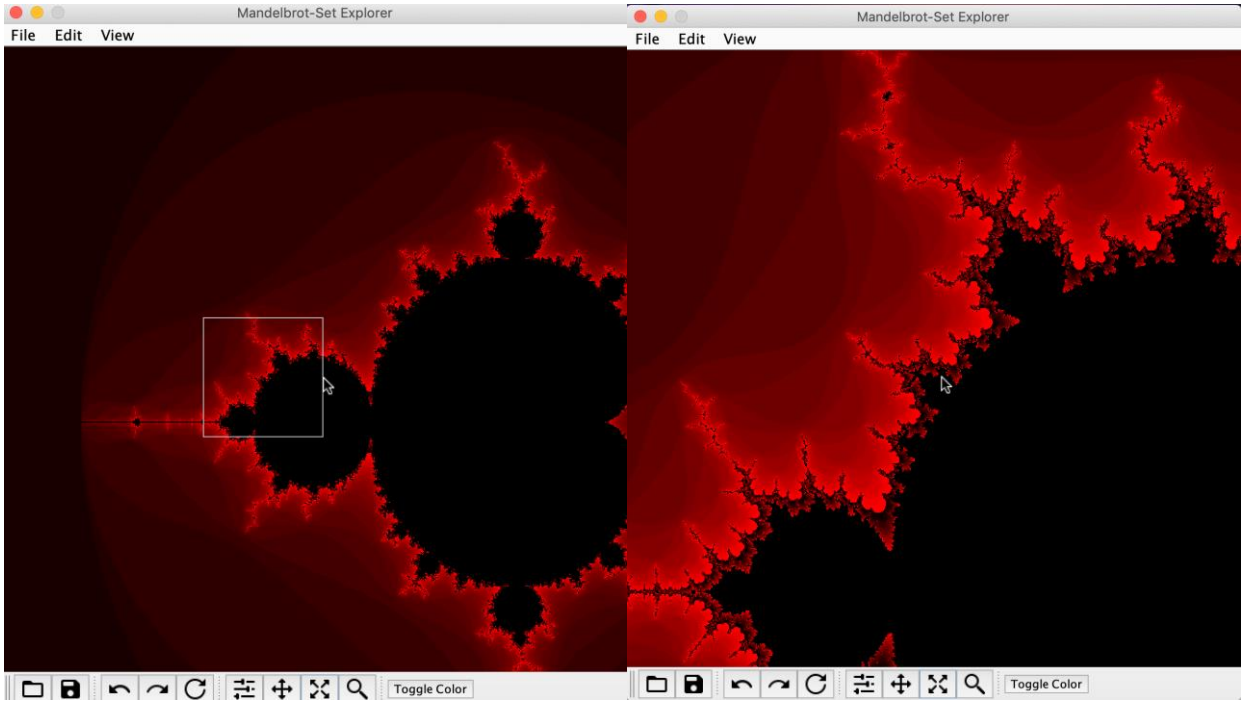

Figure 4 - zoom operation

The number of maximum iterations can be changed by clicking on the change max iterations button on the toolbar, using the edit menu, or by hitting the Ctrl+I buttons. This will bring a popup dialog box asking the user to enter a new iterations count value. Note, the old value will be displayed in the text field. An extension to provide the user the ability to also change the Mandelbrot set bounds manually was also undertaken. This option can be accessed through the edit menu or by hitting Ctrl+B.

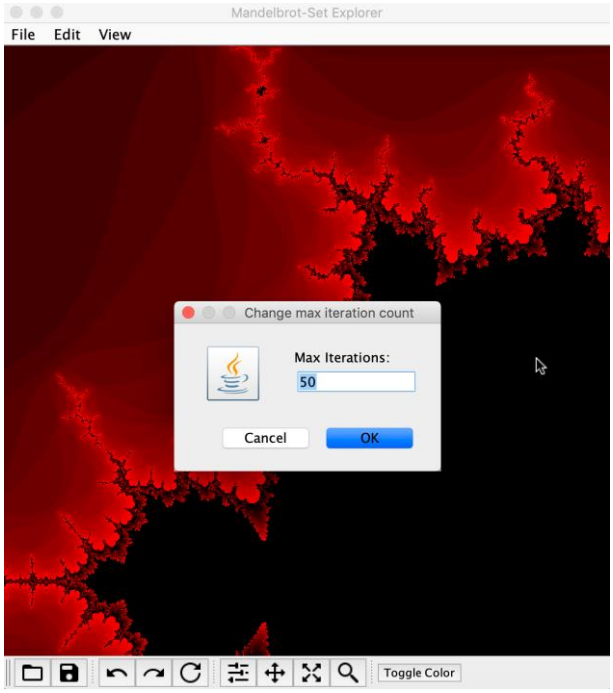

Figure 5 - figure showing popup dialog to change the Max Iterations used to calculate the Mandelbrot set

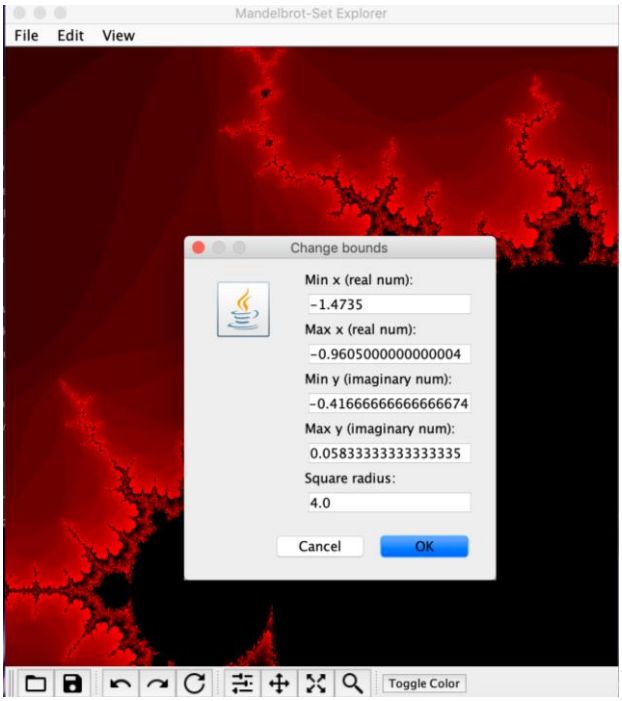

Figure 6 - figure showing popup dialog to change the Bounds used to calculate the Mandelbrot set

The current Mandelbrot set can be saved by either clicking on the save button on the toolbar, clicking save in the file menu, or hitting Ctrl+S. This will bring up the following menu.

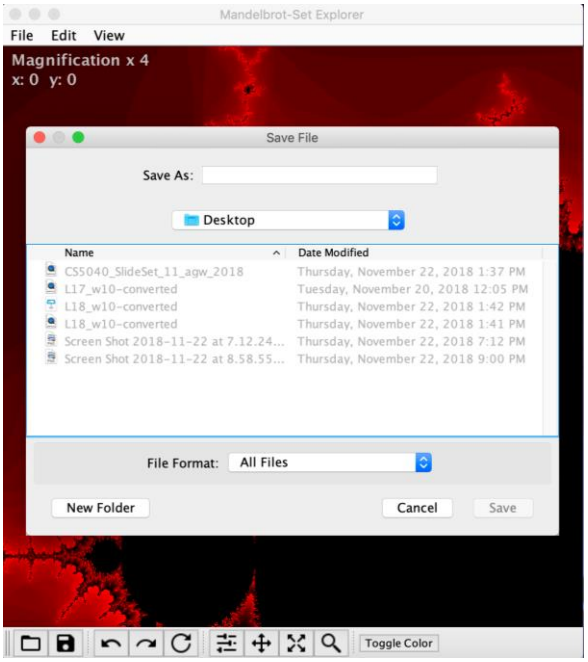

Figure 7 - Save menu

A similar menu is shown for opening saved files.

Note Figure 7 also show the results of clicking the toggle magnification button in the top left corner of the rendered image.

Finally, the colours used to render the Mandelbrot set can be changed at the click of a button using the “toggle color” button in the toolbar.

Figures showing the different colours can be found in the appendix.

6.0 Future work

The application as it stands is complete and meets the requirements described in the brief. However, some small additions would make the GUI far more user friendly. Those additions would include:

- Giving users a colour wheel to change both the fore-ground and background colours as well as the gradients of colour shift based on iteration count.
- Users should be given the freedom to re-arrange, add, or remove buttons from the toolbar.
- An export feature could be implemented to allow users to save the rendered image as a Jpeg or PNG.
- The application should be resizable, and the user should be given the option to change the resolution of the rendered image.

7.0 Appendix

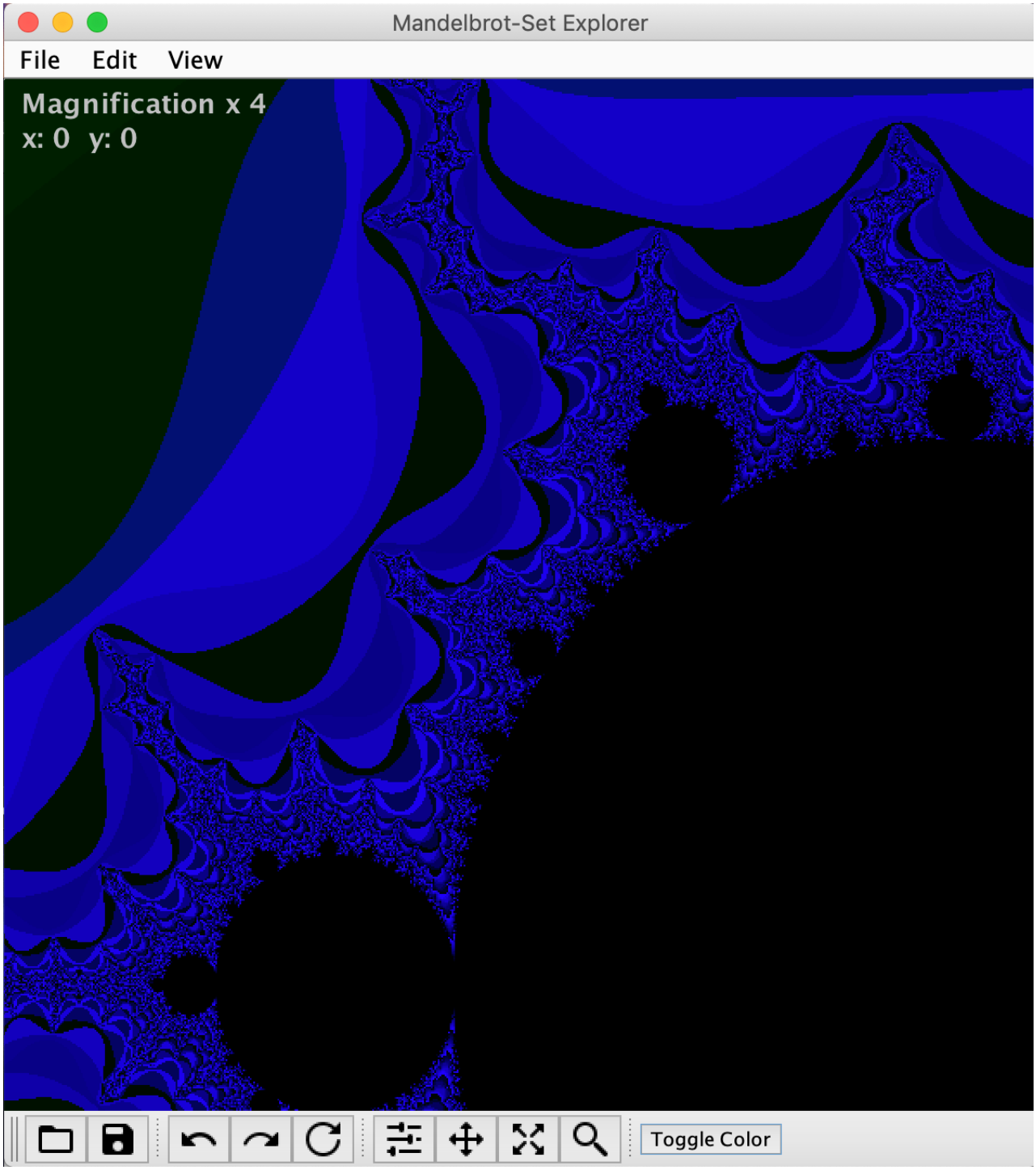

Figure 8 - Blue/Green colour scheme

Figure 9 - Banded green colour scheme

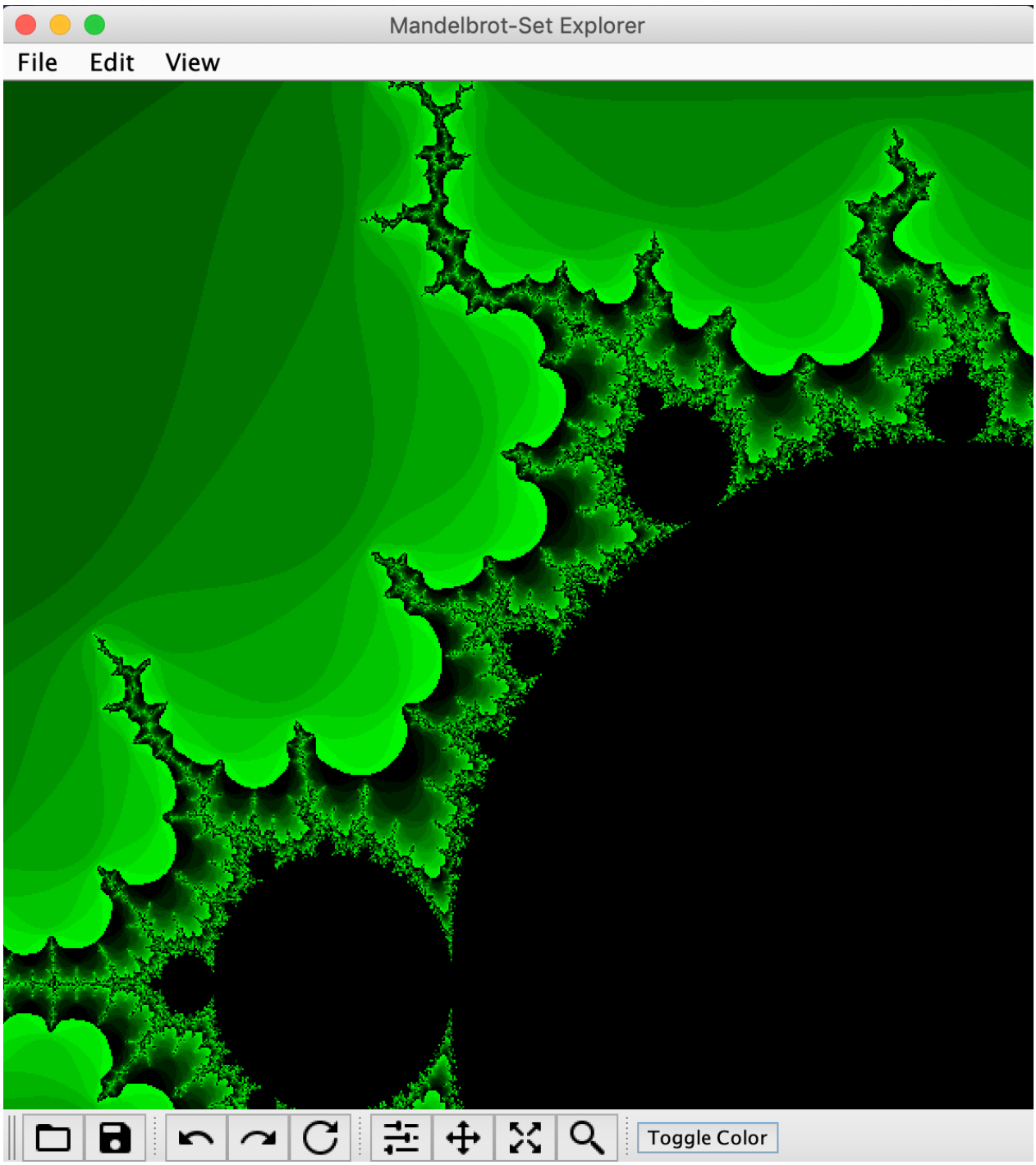

Figure 10 - flare green colour scheme

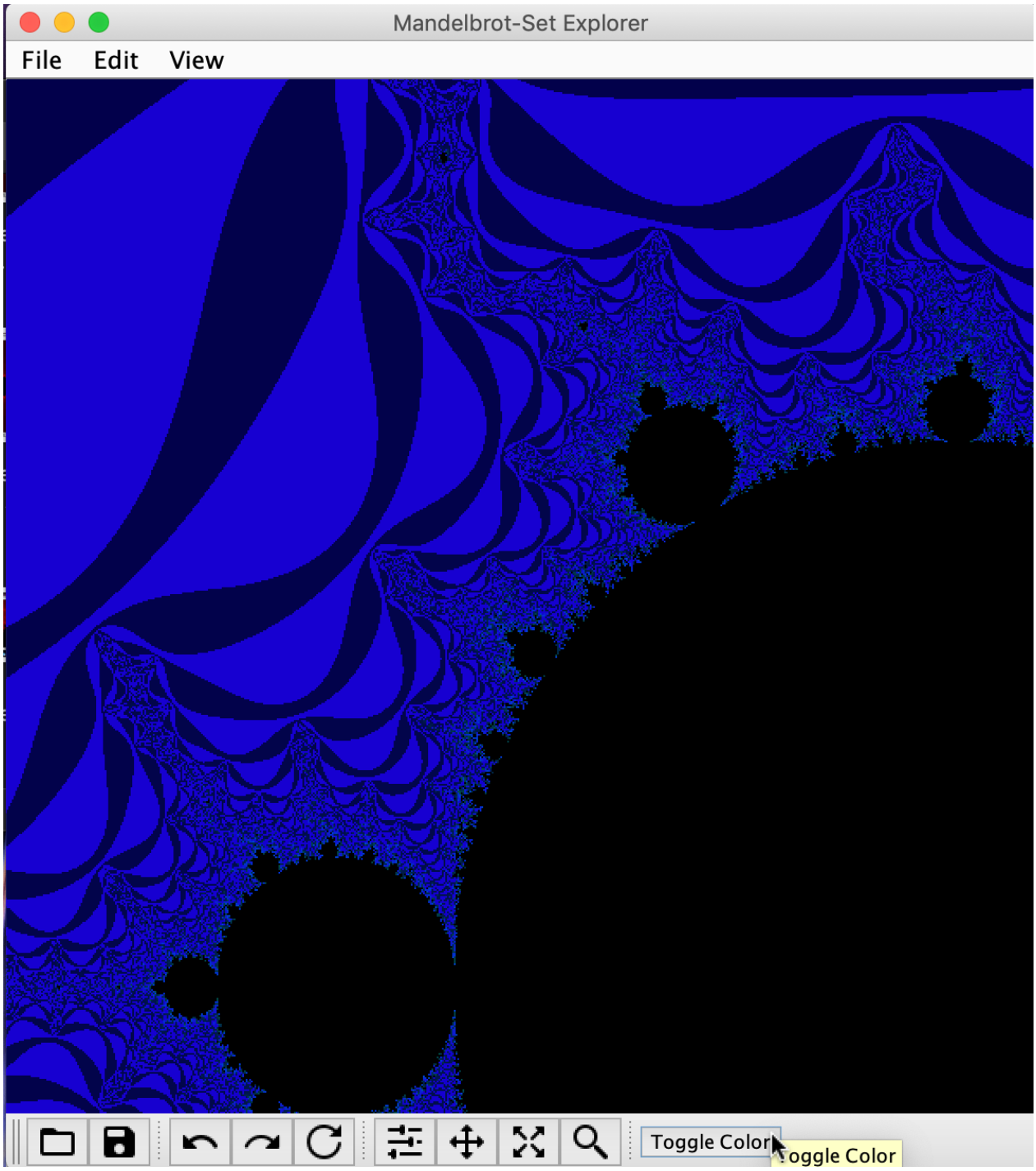

Figure 11 - Deep blue colour scheme