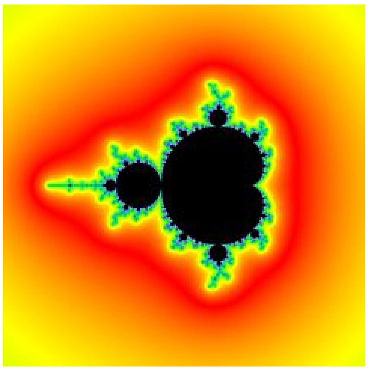
Imperial College London – Department of Computing

MSc in Computing Science

536: Introduction to Java
Tutorial 3 — Fractals



A rendering of the Mandelbrot fractal set

Introduction

Fractals are patterns that display self-similarity at any level of magnification. One example of a fractal pattern is the Mandelbrot set, which can be seen in the image above. If you were able to zoom into this image endlessly, you would keep seeing the main pattern repeated, in many different and beautiful variations.

In this exercise, you will write a simple Java GUI application to display a rendering of the Mandelbrot set. The application will consist of a window (`JFrame`) with a single component which renders a portion of the Mandelbrot set at a particular level of magnification. Your application will respond to mouse clicks by centring the image on the point clicked, and you will implement zooming by making your application respond to particular key presses. An extension of this exercise will involve improving the rendering speed by making use of *concurrency* via `Thread` objects.

The Mandelbrot Set

The Mandelbrot set is basically a set of complex numbers. A complex number c is in the Mandelbrot set if the sequence $z_0, z_1, z_2, \ldots$ given by the following equations:

$$z_0 = c$$
$$z_{n+1} = z_n^2 + c$$

converges (i.e. tends towards a finite number). The Mandelbrot set can be visualised in two dimensions by taking each pixel on the plane to represent a complex number, and then computing whether that number is (probably) in the Mandelbrot set: if it is, then that pixel is coloured black. If it is not, then the pixel can be coloured according to how fast the sequence $z_0, z_1, z_2, \ldots$ diverges.

Complex Numbers

Complex numbers are made up of a *real* component, and an *imaginary* component. A complex number (usually denoted z) can be represented as

$$z = a + bi$$

where a and b are real numbers, and i is the square root of -1. The number a is called the real component of the number and bi is called the imaginary component since it is a multiple of i, which is not a real number (there is no real number r that satisfies the equation $r^2 = -1$).

If $z_1 = a_1 + b_1 i$ and $z_2 = a_2 + b_2 i$, then the product $z_3 = z_1 * z_2$ is given by the following formula:

$$z_3 = ((a_1 * a_2) - (b_1 * b_2)) + ((a_1 * b_2) + (a_2 * b_1))i$$

Complex numbers can be represented by points on a two dimensional plane, by taking the x-coordinate to be the real component a, and the y-coordinate to be the imaginary coefficient b. We can then talk about the *modulus*, or *magnitude*, of a complex number $z = a + bi$ as being the distance from the origin, which can be calculated using Pythagoras' Theorem: $|z| = \sqrt{a^2 + b^2}$.

Calculating Convergence

It is not known if the Mandelbrot set is computable, that is whether there is an analytical way to decide if an arbitrary complex number is in the set or not. Thus, to calculate whether any given point is in the set or not, we must use numerical methods which tell us if a point either is definitely *not* in the set, or if it is probably in the set.

One of the simplest ways of calculating whether a point is in the Mandelbrot set is the Escape Time Algorithm. In this method, we calculate the sequence $z_0, z_1, z_2, \ldots$, at each stage checking whether the value z_i is greater than some threshold value that we know is part of a diverging sequence. We then say that the point has escaped and the value i is called the *escape value*. We also pick some number n, and if the value z_n has not escaped, then we stop our calculations and assume that the point is probably in the set. By taking larger and larger values of n we can make our calculation more accurate, but at ever increasing computational cost.

A basic threshold value is 2 — that is, when either the real or imaginary coefficient of z_i has reached 2, then we can be sure that the sequence will diverge. We can detect escape slightly quicker if we calculate whether the *magnitude* of z_i is greater than 2, although this is ever so slightly more computationally intensive.

Writing The Application

The Mandelbrot Viewer application will have several components:

- Classes to represent complex numbers and the Mandelbrot set.
- A subclass of `JComponent` to draw a portion of the Mandelbrot set.
- A main application class
- Helper classes to provide event listeners, GUI actions etc.

A Complex Number Class

Write a class `Complex` that will allow you to represent and manipulate complex numbers. This class should therefore have two (private) fields of type `double` to store the real and imaginary coefficients. Make sure you write appropriate constructors. It will also be useful to write methods that calculate the sum and the product of two complex numbers. In order to practice using packages, place this class in its own package called `complex`.

A Mandelbrot Set Class

The `MandelbrotSet` class will represent the Mandelbrot set. Objects of this class will have a maximum escape value, which can be set in the constructor. The class should also have a method that returns the escape value for a complex number `c`. The method should return the lesser of the actual escape value for `c` and the maximum escape value of the set.

A `JComponent` Viewing Class

The main work of the application will be to render the representation of the Mandelbrot set. We will achieve this by writing a subclass of `JComponent`, on which we will set the value of individual pixels based on the Escape Time algorithm. Since we will be drawing on this component, let's call it `MandelbrotCanvas`.

The `MandelbrotCanvas` will need a `MandelbrotSet` to display. This should be passed into the constructor. We will keep track of the point on the complex plane at which the image is centred. We will also keep track of the scale at which we are viewing the set. This will allow us to calculate which complex number each pixel represents. Declare two fields of type `double` called `centreX` and `centreY` to keep track of the centre of the image. A sensible default for the centre of the image is $(-0.75, 0)$. In addition, declare another `double` field called `pxScale` which will keep track of the scale at which the set is being rendered. Set its default value to be 0.005.

Rendering The Image

We will render the Mandelbrot set by overriding the `paintComponent` method of `JComponent`. This method is called whenever the component must be (re)drawn on the screen; we can also call the `repaint()` method if we want the component to redraw itself. The `paintComponent` method is passed a `Graphics` object as a parameter, and by manipulating this object we can make things appear on the screen. In this exercise, we will be creating a `BufferedImage` on which we will set the value of pixels, and then use the `drawImage` method of the `Graphics` object to draw the image on the screen. Define a method in your class with the signature `public void paintComponent(Graphics g)`, then carry out the following:

Creating the image to draw on. The first thing to do is create a new `BufferedImage` object. We will need to specify a height and a width (in pixels) to the constructor. We should create a `BufferedImage` that is the same size as the `MandelbrotCanvas` object we will be drawing on (i.e. the receiver `this`) - we can get the height and width by calling the `getHeight` and `getWidth` methods inherited from `JComponent`. We also need to specify how colours should be specified in the image - use `BufferedImage.TYPE_INT_RGB`.

Setting the pixels' colours. We now need to calculate which colour to make each pixel. Write two nested `for` loops to process each pixel. Within this loop, the first step is to calculate which complex number the pixel corresponds to. Remember that the x coordinate corresponds to the real component, and the y coordinate to the imaginary component. Remember, also, that pixel $(0, 0)$ is at the *top left* corner of the canvas. The complex number for this pixel can be calculated as follows:

$$a_{00} = (\texttt{centreX} - ((1 - (\texttt{getWidth()}\%2)) * \texttt{pxScale} * 0.5)) - ((\texttt{getWidth()}/2) * \texttt{pxScale})$$
$$b_{00} = (\texttt{centreY} + ((1 - (\texttt{getHeight()}\%2)) * \texttt{pxScale} * 0.5)) + ((\texttt{getHeight()}/2) * \texttt{pxScale})$$

Then pixel (x, y) corresponds to the complex number:

$$(a_{00} + (x * \texttt{pxScale}) + (b_{00} - (y * \texttt{pxScale}))i$$

Once you have calculated the coefficients of the complex number for the pixel, use them to create a new `Complex` object, and use this to calculate its escape value.

Having calculated the escape value, we need to use this to calculate a colour. To begin with, the simplest approach is to choose either black or white, to produce a monochrome image. If the escape value is equal to the maximum value you chose, this means the point is (probably) in the Mandelbrot set, and so use `Color.BLACK`, otherwise use `Color.WHITE`. The pixel's colour can be set by using the `setRGB` method of the `BufferedImage` object.

Once you have the basic application working, you can experiment with producing a more colourful image. To produce a suitable integer you could use the ratio of escape value to maximum escape value, and scale it to produce a integer between 0 (black) and $2^{24} - 1$ (white). To code this number, note that its binary representation is 24 `1`s.

Drawing the image. Once you have calculated and set the value of each of the pixels in your `BufferedImage` object, you can draw it on the screen using the `drawImage` method of the `Graphics` parameter `g` passed to `paintComponent`. There are several overloaded versions of this method — use the one with the signature `boolean drawImage(Image img, int x, int y, ImageObserver o)`. Pass in `null` as the final parameter[1].

The Main Application Class

At this point, you can write the main application class and produce a basic runnable application. The `HelloWorldSwing` class from the Java Tutorial is a good reference to use here: `https://docs.oracle.com/javase/tutorial/uiswing/examples/start/`.

Create a class called `MandelbrotViewer`. Give it a static method that creates a new `JFrame` with a suitable title. The method should also create a `MandelbrotSet` with a suitable max escape value: 100 might be a good value to try initially. Then give this set to a new `MandelbrotCanvas` object, add the canvas to the `JFrame` using its `add` method, call the `pack` method of the `JFrame`, and then display it using the `setVisible` method. The `JFrame` should also be set to `EXIT_ON_CLOSE`. The `pack` method tell the `JFrame` to calculate its size based on the sizes of its component. So far, we haven't told the `MandelbrotCanvas` component how big it should be, so do that now by defining the `getPreferredSize` method in the `MandelbrotCanvas` class. This method overrides the one from `JComponent` and needs to return a `Dimension` object. A good size to make the component might be 512 pixels wide by 384 high.

In the `main` function of your application class, fire up your GUI and display the `JFrame` by instantiating an anonymous inner class that subclasses `Runnable` and passing it to the `invokeLater` method of the `SwingUtilities`. Compile your main application class and try running your application. This should display a window which renders the Mandelbrot set. Try resizing the window - the image should be re-rendered to fill the whole window.

Responding to Mouse and Keyboard Input

At this point, your application simply displays the Mandelbrot set at a fixed level of magnification. We now want to enable the user to interact with the application by centring it on points specified by mouse clicks, and by zooming in or out when the user presses an appropriate key on the keyboard.

[1] This parameter allows your code to be *notified* when the image has finished being drawn, but we are not interested in being notified in this exercise.

Responding To Mouse Clicks

We can make the MandelbrotCanvas respond to mouse clicks by adding a MouseListener to it using the inherited addMouseListener method. Do this in the constructor by defining an anonymous inner class that subclasses MouseAdapter and defines a mouseClicked method, and passing this as an argument to the addMouseListener method. Your code should look something like this:

```
addMouseListener(new MouseAdaptor() {
     public void mouseClicked(MouseEvent e) {
         ...
     }
  });
```

You have now added an object to the component which will respond when the mouse is clicked by executing the mouseClicked method. This method takes a MouseEvent object as a parameter, which contains information about the click event. In particular, it contains the relative coordinates in the component where the mouse was clicked. These can be retrieved using the getX and getY methods of the MouseEvent.

We want our application to respond to the mouse click by re-centring the image. This will involve two things: firstly calculating where the new centre is and storing these coordinates in the centreX and centreY fields of the MandelbrotCanvas, and secondly calling the repaint method on the MandelbrotCanvas object to re-render the image. The new centre coordinates can be calculated by

$$\begin{aligned}\texttt{centreX} &= a_{00} + (\texttt{e.getX()} * \texttt{pxScale})\\ \texttt{centreY} &= b_{00} - (\texttt{e.getY()} * \texttt{pxScale})\end{aligned}$$

Responding To Keyboard Input

We can make our MandelbrotCanvas component respond to keyboard input by manipulating its *input map* to map keystokes to action names, and manipulating its *action map* to associate those action names with Action objects. We can access these maps using the getInputMap and getActionMap methods inherited from JComponent. When getting the input map, we want the component to respond to the input whenever the main application window is in focus, so pass the JComponent.WHEN_IN_FOCUSED_WINDOW constant as a parameter. As for processing mouse input, this can all be done in the MandelbrotCanvas constructor.

Zooming In. Let's make the application zoom in when the "+" key is pressed. We can register an association between a keystroke object and an action name in the input map by using its put method. We can get a keystroke object corresponding to the "+" key by passing a string descriptor to the getKeyStroke static method of the KeyStroke class. The string descriptor we need to use is "typed +". Associate this keystroke with the action name "zoomIn".

We now need to associate the action name with an Action object in the action map. Again, we can use its put method. We will create an Action object as an anonymous inner class that extends `AbstractAction`. We will need to override its `actionPerformed` method, so your code should look something like this:

```
getActionMap().put("zoomIn", new AbstractAction() {
      public void actionPerformed(ActionEvent e) {
         ...
      }
   });
```

The action object will respond to the `"zoomIn"` action by executing the `actionPerformed` method, and that particular action will be fired when the user presses the "+" key. The `actionPerformed` method is passed an `ActionEvent` object as a parameter which, like the `MouseEvent` object, contains information about the event in question. In this case, however, we do not need to use any of this information. All we need to do is modify the `pxScale` field of the `MandelbrotCanvas` object. We could, for example, divide it by 2. We should then call the `repaint` method of the `MandelbrotCanvas`. Although unnecessary, since the Java compiler can figure out that this method belongs to the containing class, to avoid ambiguity we can explicitly refer to the containing class as the receiver using the syntax `MandelbrotCanvas.this.repaint()` (similarly when referring to its `pxScale` field).

Zooming Out. Now implement a zoom out action when the "–" key is pressed.

After enabling your application to respond to user input, play around with it and discover the intricacy and beauty of the Mandelbrot set!

Bonus Challenge: Enhancing Rendering Using Threads

In the specification above, it was suggested that you use a certain maximum value for the escape value. Your image can be made more detailed and accurate by increasing this maximum value. Play around with a few different values to see what effect this has. What you will notice however is that as you increase this value, the longer it will take to render the image as you re-centre, zoom in or out, or resize the window. As a bonus challenge, see if you can enhance the speed of rendering by using *threads* - if you create a number of different threads, these can be run *concurrently* and can each calculate the pixel values for different portions of the image.

The first step in completing this will be for the `paintComponent` method to split the image to be rendered into a number of distinct portions. As a hint, consider using the `Rectangle` class, which can be used to encapsulate a width and a height, and also an x and y offset. After dividing the image, create the threads to each work on a different area. The `run` method of each thread will essentially contain the same code as described above. After starting all the threads, the `paintComponent` method should wait for them all to finish using the `join` method for each thread. When all the threads have finished executing, the image will be fully calculated and it can then be drawn on the component's `Graphic` object.