Assignment 1 - Mandelbrot Set Renderer

Rory Stephenson 300160212 stepherory

20 March 2014

What is the Mandelbrot Set

The Mandelbrot set is a set of numbers which do not tend towards infinity when the following equation is repeatedly applied to them: $Z_n + 1 = Z_n^2 + c$

The significance of the Mandelbrot set is that it is infinitely complex (there is no limit on how far in you can zoom on the set and still discover new detail). It is one of the first examples of a fractal (something which contains self-similar patterns), which some believe to be the mechanism with which nature creates complex structures.

Another peculiarity of the Mandelbrot set is the amount of time it took humans to discover such a set (and realise its complexity). This is due to the vast amount of calculations required to rendered such a set which was only made possible with the advent of computers.

There are multiple ways to render the Mandelbrot set. The implementation below represents the pixels on the screen as coordinates on the complex number plane. Each pixel's coordinate is iterated over the Mandelbrot formula, colouring a pixel based on the number of iterations until either it is determined to escape to infinity or the maximum number of iterations is reached. The escape condition used is whether the real or imaginary part of coordinate is greater than two. If a coordinate takes more than 300 iterations to escape it is assumed that is does not escape to infinity. Whilst not totally accurate this allows for much faster rendering and only presents an issue when very fine levels of detail are examined.

How to use

Simply run mandelbrot from the command line and follow the instructions. A generated mandelbrot.pnm file will appear in the same directory. If you wish to save one of the mandelbrot images you should move it before re-running the program or it will be overwritten.

Implementation

Firstly the Pnm module (which provides rendering functionality for a list of `Rgb` values) is loaded. A pixel list is then generated which contains a tuple (x, y) for each coordinate in a 700x400 size. The tuples are in ascending order by the y value and then the x value (i.e. (300, 1) comes before (1, 300). Lastly the helper method getDouble is declared which is used to read user input as a double.

```haskell
import Pnm

pixels :: [(Int, Int)]
pixels =
  concat $ map (\ y -> (map (\ x -> (x, y)) [0..699])) [0..399]

getDouble :: IO Double
getDouble = readLn
```

The main function begins by requesting the x and y coordinate of the Mandelbrot set which will be central on the image. A zoom value is also used to determine how far the set should be zoomed in to (to allow viewing of fine detail). Lastly it converts the pixel list into an Rgb list (by mapping the `findColour` function on to the pixel list) and draws the pixel list to a file called ”mandelbrot.pnm” using the `writePnmColour` function of the Pnm module.

```haskell
main :: IO()
main = do
         putStrLn "Centre x coordinate (-2.5 to 2.5)?"
         xCentre <- getDouble
         putStrLn "Centre y coordinate (range is -1.5 to 1.5)?"
         yCentre <- getDouble
         putStrLn "Zoom (1 is completely zoomed out)?"
         zoom <- getDouble
         putStrLn "Generating PNM..."
         writePnmColour (map (\ (x ,y) -> findColour
          (mbrotScale
            (fromIntegral x, fromIntegral y)
```

```haskell
              (xCentre, yCentre, zoom))
              ) pixels) 255 (700, 400) "mandelbrot.pnm"
          putStrLn "...Done"
```

`findColour` takes a coordinate and returns the Rgb value for that coordinate. It applies the `incrementMandelbrot` function to the provided set of coordinates until the `escapes` function evaluates to True. At this point the triple that evaluated to true is passed to `colourOf` which returns the colour for the number of iterations used.

```haskell
findColour :: (Double, Double) -> Rgb
findColour (x0, y0) = colourOf (
  until (escapes) (incrementMandelbrot) (0, (x0, y0), (0, 0)))
```

`incrementalMandelbrot` accepts a triple containing the depth (number of iterations), the initial coordinate and the current coordinate. It applies the Mandelbrot function and returns a triple of the same type containing the input depth + 1, the original coordinate and the new coordinate as a result of applying the Mandelbrot function.

```haskell
incrementMandelbrot :: (Int, (Double, Double), (Double, Double)) ->
                       (Int, (Double, Double), (Double, Double))
incrementMandelbrot (depth, (x0, y0), (xn, yn)) =
  ((depth + 1), (x0, y0), (((xn*xn) - (yn*yn) + x0), (((2*xn)*yn) + y0)))
```

`escapes` accepts the current depth of iteration, the initial coordinate and the current coordinate. It evaluates to true (the coordinate escapes) if either 300 iterations has been reached or the real/imaginary part of the current coordinate is ¿= 2.

```haskell
escapes :: (Int, (Double, Double), (Double, Double)) -> Bool
escapes (depth, _, _)     | depth >= 300 = True
escapes (_, _, (xn, yn)) | (xn*xn) + (yn*yn) >= 4 = True
                          | otherwise = False
```

`colourOf` accepts a triple of the same type as `incrementMandelbrot`, despite the fact that only the depth is used, in order to simplify the code in findColour. It returns an `Rgb` based on the depth such that a lower number of iterations results in a darker colour. The exception is in the case where the maximum number of iterations is reached which results in black. Black is used to indicate that the number is inside the mandelbrot set (because it could not be determined that it was trending to infinity).

```haskell
colourOf :: (Int, (Double, Double), (Double, Double)) -> Rgb
colourOf (depth, _, _) | depth < 25  = Rgb 0 0 (fromIntegral ((4*depth) + 1))
                       | depth < 40  = Rgb 0 0 119
                       | depth < 100 = Rgb 0 0 189
                       | depth < 150 = Rgb 99 99 255
                       | depth < 300 = Rgb 159 159 255
                       | otherwise   = Rgb 0 0 0
```

`mbrotScale` converts a pixel coordinate into its equivalent coordinate on the Mandelbrot Set scale using the specified x/y centre and zoom (xCtr, yCtr, zm).

```haskell
mbrotScale :: (Double, Double) -> (Double, Double, Double) ->
              (Double, Double)
mbrotScale (x,y) (xCtr, yCtr, zm) =
  ((((5*(x/700)) - 2.5)/zoom)+xC, -((((3*(y/400)) - 1.5)/zoom)+yC))
                                where
                                  zoom = max zm 1
                                  yC = max (min 1.5 yCtr) (-1.5)
                                  xC = max (min 2.5 xCtr) (-2.5)
```