Brock University

Department of Computer Science

Report on the final project of COSC 3P98

Mandelbrot and Julia Sets

Torben Krüger, Maike Rees

January 2016

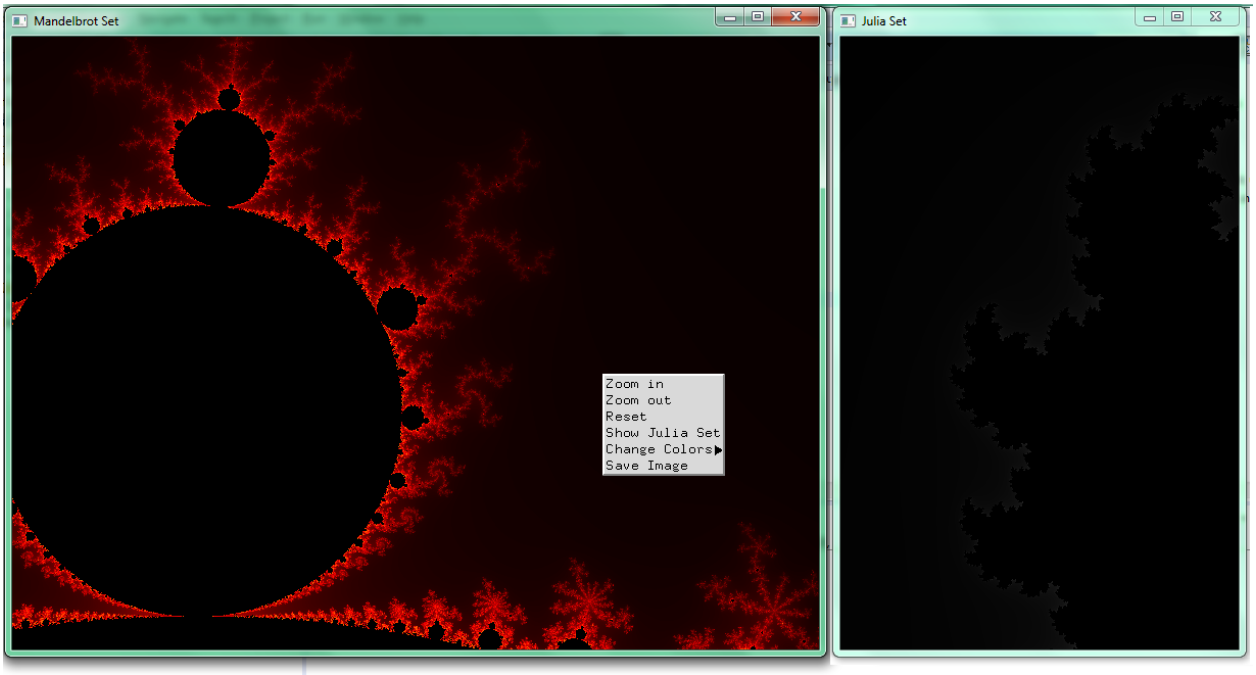
1 Introduction

During the process of the course, fractals catched our interest so we desided to make an application about fractals for our final project of the course. The Mandelbrot Set is one of the most popular fractals and it is closely related to the Julia Set so we chose it as our main focus. There are a lot of android apps, online applications and pretty pictures about and of Mandelbrot Sets that inspired us further more.

We developed the Application in c/c++ with openGl, freeglut and freeimage. We created it on Windows 7 in Eclipse Mars and migrated it later into a visual studio(2013) project.

2 Functionality

The Application demonstrates the relation between the Mandelbrot set and Julia sets. Furthermore you get interesting observations through the coloring of the sets. Additionally, images can be saved. Here is a screenshot of our program.



3 User Manual

The Application comes with two windows. One where the Mandelbrot set is shown and in the other one you can see a Julia set.

You have multiple options here. With left-click of your mouse you can zoom into the chosen set. The point of your mouse click is going to be the center of the zoomed image.

With a right click a context menu opens. You can chose to zoom in and out, reset the set, change the color or save an image of the set.

If you open this menu in the Mandelbrot set you additionally have the option to show the Julia set of the chosen Point. It will then show the according Julia Set in the second Window.

4 Architecture and Implementation

The architecture of the application is not too complicated. It is object-oriented and structured into header and cpp - files.

4.1 Complex Numbers

Since we're calculating with complex numbers most of the time in those sets there is a class called Complex.h that describes those complex numbers as a struct. Every complex number comes with a real and imaginary part which are represented by two integers.

This struct also overloads methods for the + and * operator. It also contains a method that squares the number and one that calculates the absolute value of the complex number. Those methods are all needed to calculate the sets.

```
1  struct complex {
2     double r, i;
3     double getAbsolute() {...}
4     complex pow2() {...}
5     inline complex operator+(complex a) {...}
6     inline complex operator*(complex a) {...}
7  };
```

4.2 Mandelbrot Set

For a Mandelbrot set we set up a header and the cpp-file that implements the header methods. The Mandelbrot set has public methods to calculate the set, zoom in and out of it, reset it as well as some Getters and Setters.

```
1  class MandelbrotSet {
2  public:
3     MandelbrotSet(unsigned int, unsigned int);
4     virtual ~MandelbrotSet();
5     void calculate();
6     void setWidth(unsigned int);
7     void setHeight(unsigned int);
8     void setIterations(unsigned int);
9     void reset();
10    void zoom(unsigned int, unsigned int);
11    void zoomOut(unsigned int, unsigned int);
12    void setColorMode(int);
13 };
```

4.3 Julia Set

Looking at the formular of both sets, one can see that they are very close. That is why we decided to let the Julia set be a child of the Mandelbrot set. Since there are only a few changes that we had to make in the Julia classes, we save a lot of code and our implementation gets a lot easier with this design decision.

The Julia set only overrides the cnstructor and the calculate(), draw() and reset() methods. Additionally it has a complex number k, that is needed for the calculation.

```
1  class JuliaSet: public MandelbrotSet {
2  public:
3     JuliaSet(unsigned int, unsigned int);
4     void draw(unsigned int, unsigned int, complex);
5     void calculate();
6     void reset();
7     complex k;
8  };
```

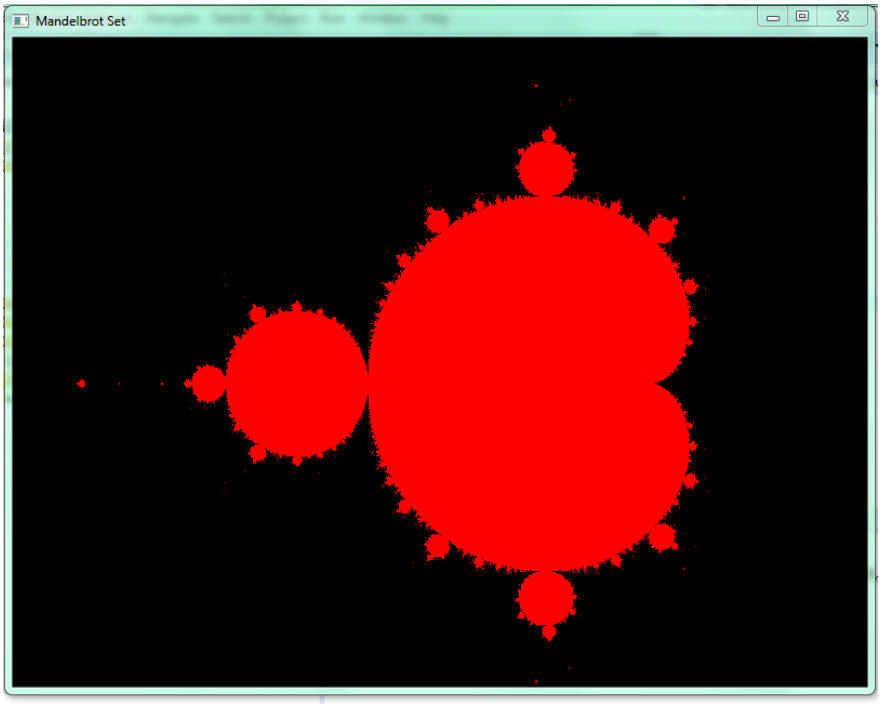
4.4 The Main Class

The Main.cpp puts it all together. It sets up the OpenGl / freeglut environment, handles mouse event and the menus and starts the drawing of the sets.

When clicking on an option of the menu the program often needs to know where the mouse is right now to calculate with the right coordinates. For this purpose the application is using the passive motion function that glut offers. It always saves the current mouse coordinates in the global struct.

5 Coloring

The Mandelbrot set itself only tells us if the pixel is in the set or not. So basically only two colors are needed to display the Mandelbrot set. Like in this screenshot (chose colormode 'pure' in the application).



A lot of fun comes with different coloring techniques so we want to take a closer look to it.

The Mandelbrot calculates its pixel with a preset maximum number of iterations (because it can calculate infinitely, but thats obviously not helping us). The bigger the maximum number of iterations, the more precise is the image. But this also means that with a big number of max. iterations the calculating time is way longer that with a small number. So the idea behind the iterations is that every iterations looks a little bit closer into the Mandelbrot set. So at some point, the algorithm will determine that a pixel is not included in the mandelbrot set and will then break the for-loop. If the pixel is inlcuded in the Mandelbrot set the for-loop will terminate regularly after all iterations have been run through.

To make this a little more clear, here is the code snippet:

```
1  void MandelbrotSet::draw(unsigned int x, unsigned int y, complex c) {
2      complex z = c;
3      unsigned int n = 0;
4
5      for (n = 0; n < iterations; n++) {
6
7          if (z.getAbsolute() > 4) {
8              break; // the pixel is not in the Madelbrot set
9          }
10         z = z.pow2() + c;
11     }
12     data[x + y * width] = getColor(n); // get the color based on the iteration
13 }
```

This means that every pixel has a number n that is the iteration at wich the for-loop terminated. For the pixels that are in the Mandelbrot set

$$n = number\ of\ max\ iterations$$

and for all the others

$$0 \leq n < number\ of\ max\ iterations$$

According to this number n the different coloring methods calculate the color.

5.1 Linear Coloring

E.g. the greyscale colormode uses a linear coloring method. First of all the step between each color is calculated. That means if the number of iterations is big the steps are small and the other way round. Then the red value is calculated. The pixels within the Mandelbrot set get the color black.

```
1 float step = 1.0f / iterations;
2 ...
3 // greyscale
4 red = 2 * n * step * 255;
5 if (n > iterations - 2) {
6    red = 0;
7 }
8 blue = red;
9 green = red;
```

In the red and white colormode, the application uses a gradient from black to red for the first half of the iterations and a gradient from red to white for the second half of the iterations.

```
1  red = 2 * n * step * 255;
2  if (n < iterations / 2 - 1) { // black to red
3     green = 0;
4     blue = 0;
5  } else if (n < iterations - 2) { // red to white
6     green = 255;
7     blue = 255;
8  } else { // pixel in the set are black
9     red = 0;
10    green = 0;
11    blue = 0;
12 }
```

5.2 Logarithmic Coloring

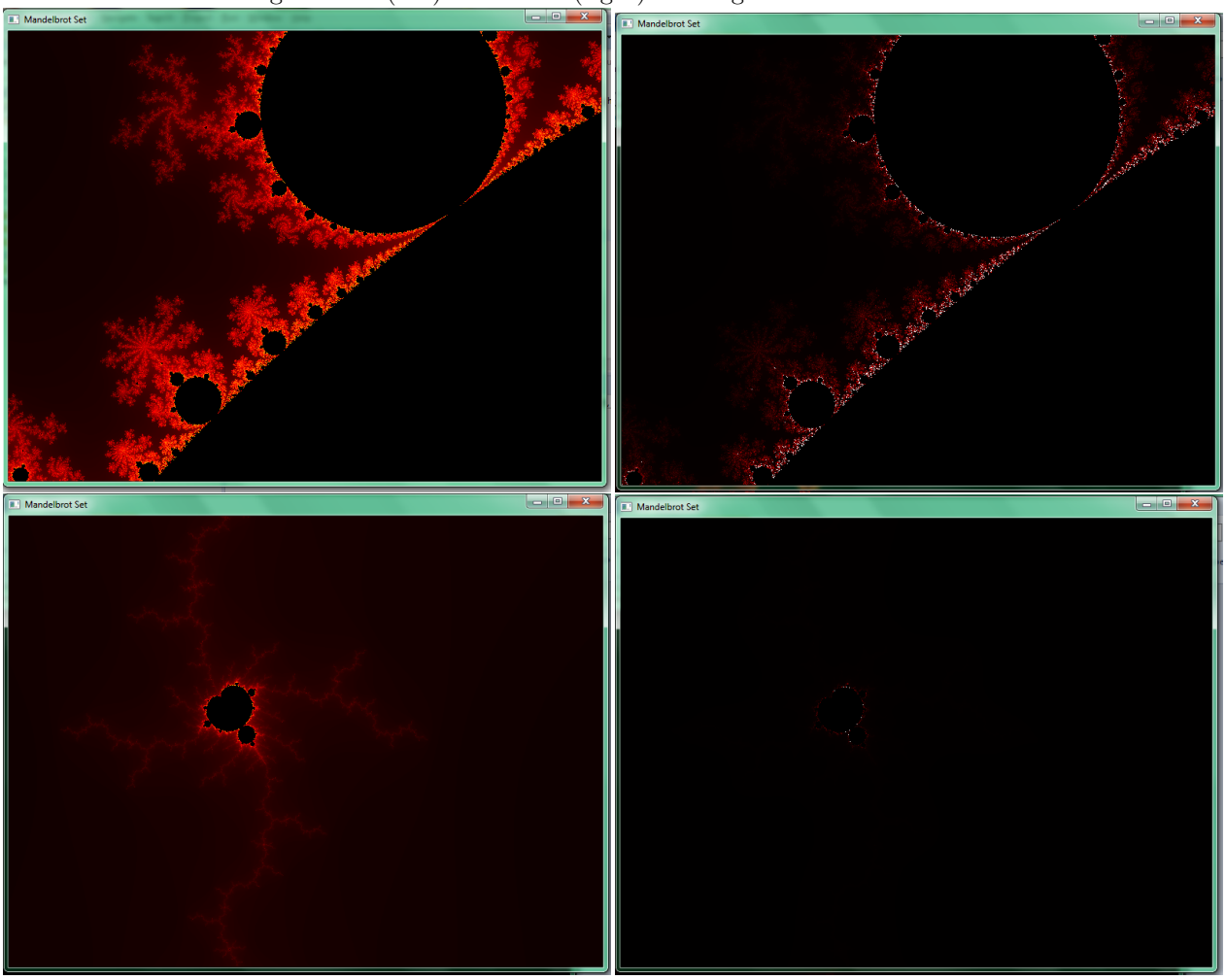
The Logarithmic coloring method provides more detailed coloring for the edges of the Mandelbrot set. The linear coloring uses one color gradient wheras the logarithmic coloring uses 7 different color gradients. Furthermore it seperates the iterations in unequal parts to apply the gradient.

```
1  if (n == iterations) {
2  } else if (n < 64) {
3     red = n * 2.0f;
4  } else if (n < 128) {
5     red = ((((n - 64.0f) * 128.0f) / 126.0f) + 128.0f);
6  } else if (n < 265) {
7     red = ((((n - 128.0f) * 62.0f) / 127.0f) + 193.0f);
8  } else if (n < 512) {
9     red = 1.0f;
10    green = ((((n - 256.0f) * 62.0f) / 255.0f) + 1.0f);
11 } else if (n < 1024) {
12    red = 1.0f;
13    green = ((((n - 512.0f) * 63.0f) / 511.0f) + 64.0f);
14 } else if (n < 2048) {
15    red = 1.0f;
16    green = ((((n - 1024.0f) * 63.0f) / 1023.0f) + 128.0f);
17 } else /*if (n < 4096)*/ {
18    red = 255;
19    green = ((((n - 2048.0f) * 63.0f) / 2047.0f) + 192.0f);
20 }
```

The result is that the edges are more nicely shaded in comparison to the linear gradient. As you see in the two images below, the logarithmic gradient has a smooth gradient from red to yellow and the linear gradient

'jumps' between red and white.

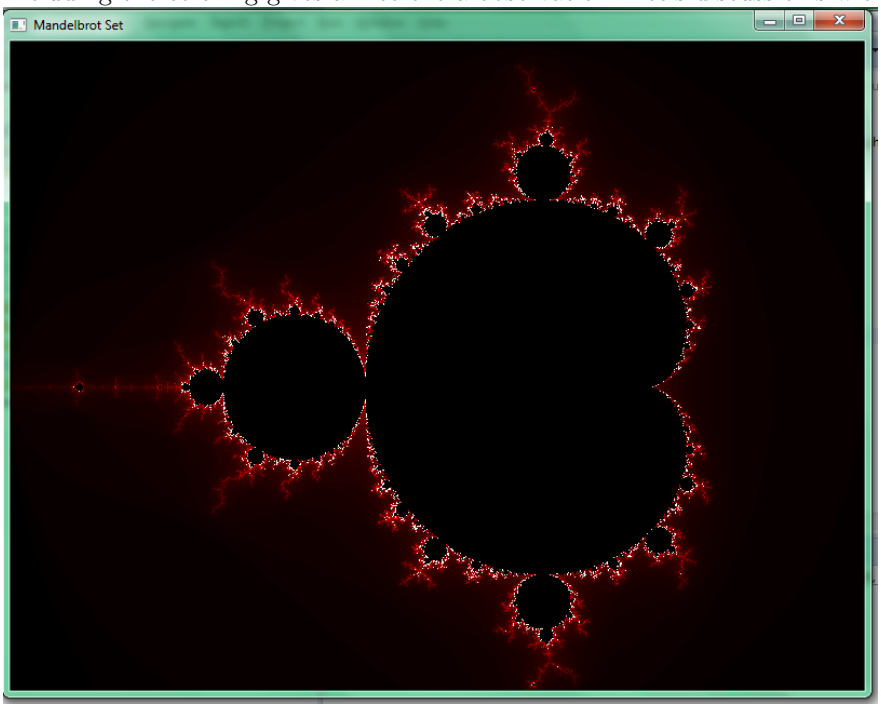
Difference between logarithmic (left) and linear (right) coloring.



6 Mathematic Background

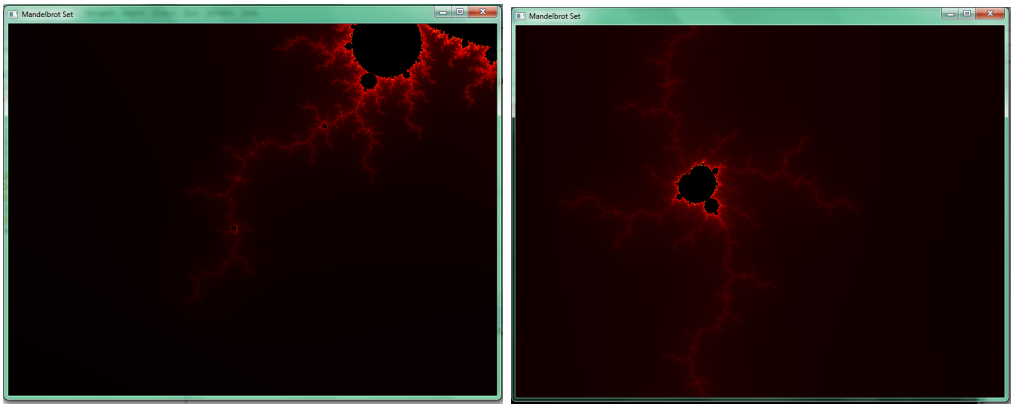
7 Discussion

Including the coloring gives a nice extra observation. Let's discuss this with the help of an example.



The Mandelbrot set has the property that the pixels that are closer to the boarder have higher iteration numbers. For this coloring method this means that white pixels are very close to the boarder of the set and red pixel are closer than black pixels but not as close as the white pixels to the boarder.

This means that the red pixels that look like roots of a tree tell us that there is a boarder some where close to them. So if we zoom in deep enough we will find a part of the Mandelbrot set there as well.



Looking at these zoomed pictures we also see that the small part of the Mandelbrot set has the same shape as the big one where we started. This is another interesting property of the Mandelbrot set.

8 References