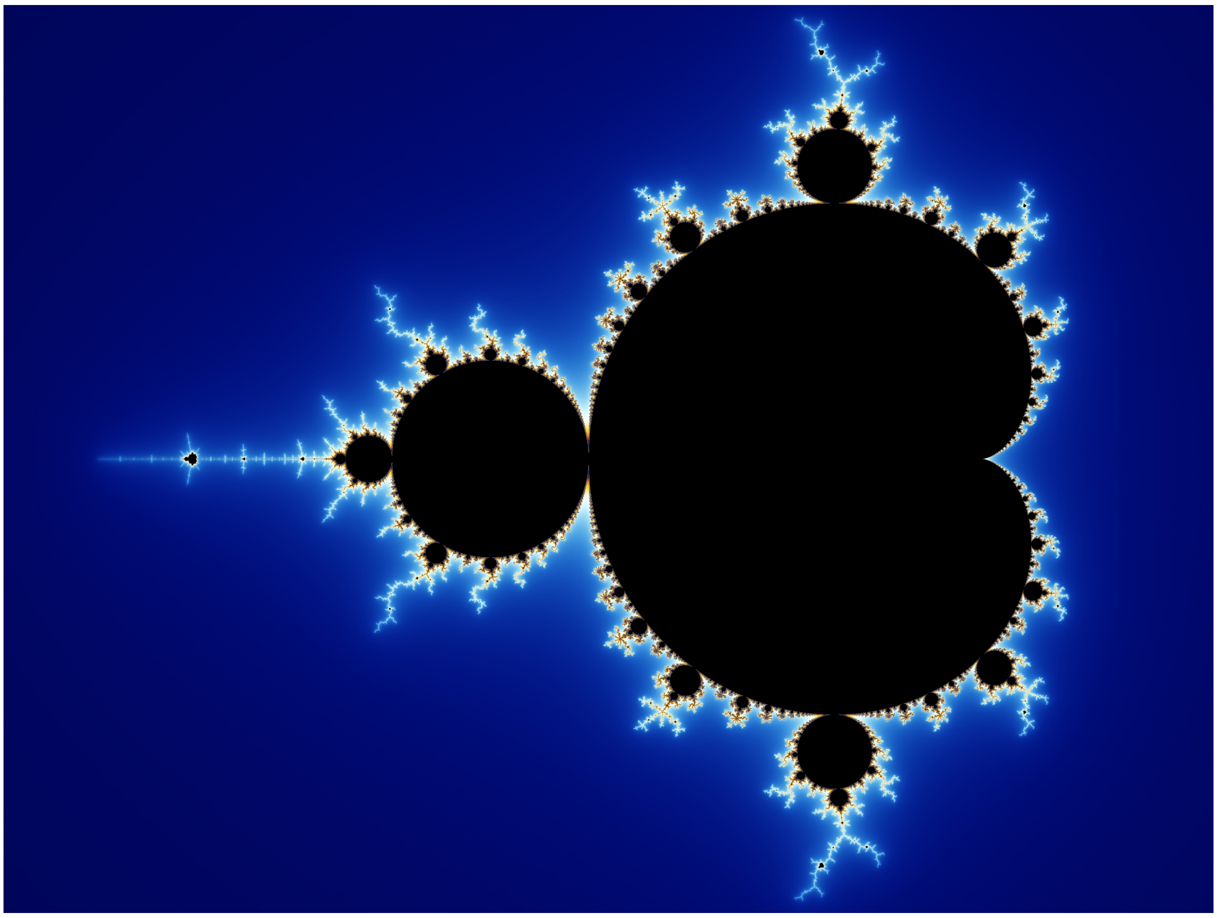
Mandelbrot Set Using the GPU
UCSB CMPSC 290B
Spring 2015



Samson Svendsen
Simen Aakhus
Anders Kristiansen
Eivind Kristoffersen

Abstract

The objective of our project is to use the GPU instead of the CPU as the processing unit for calculating the Mandelbrot set. We will show that this results in a significant increase in the performance, due to the GPU's huge advantage in computing parallel tasks, like the Mandelbrot set. We have also included some features like zoom and movement capabilities, color smoothing, and adjustable iteration limit and resolution. Using JOGL to reach the GPU, we experienced that using the GPU would be 173 times faster than the CPU. We conclude that the Mandelbrot set is a problem that maps well to the GPU, and that JOGL is a great tool for doing so.

Contents

1 Introduction

Over the past decades GPUs and CPUs have become increasingly more powerful. An important enhancement has been the inclusion of more and more cores, allowing the processing units to operate in parallel. Some algorithms can be divided into independent (but identical) sub tasks, the performance of these highly parallel algorithms can be boosted significantly by utilizing the GPU. Computing the Mandelbrot Set is one such problem. The Mandelbrot is a fractal that can be calculated by a repeating calculation. Each pixel is assigned a color value based on the number of iterations before reaching a critical value (or the iteration limit is reached), and each pixel value is independent of all the other values.

For our project we have implemented a solution to the Mandelbrot set by using Java Binding for the OpenGL (hereby JOGL), to calculate the Mandelbrot set in order to find out how much of an improvement we get by utilizing the GPU for a highly parallel problem. We also want to make a pleasing GUI, where the user can zoom, pan the camera, select between several color schemes and set the iteration limit and resolution in real time.

2 Functional Requirements

- Mandelbrot Set running on the GPU
- Real time zoom and panning capabilities
- Adjustable iteration limit and resolution
- Color smoothing
- Adjustable color schemes

All of these functional requirements manifest themselves visually. As a consequence, all the performance measures are tested by the look, feel and smoothness of the application. More specifically the Mandelbrot set should be rendered by the GPU with a smooth frame rate, and functionality such as color smoothing and adjustable color schemes should be pleasing visually.

3 Additional Functionality

In addition to the originally planned functionality of our program, we decided to add recoding functionality, giving the user the ability to play back and make a movie of the Mandelbrot Set. This is a useful functionality, as it lets the user produce an image sequence of the Mandelbrot Set in real time, allowing the user to share interesting parts of the Mandelbrot set with other people. Since this is done in real time, a lot of time is saved compared to rendering each frame on the CPU.

Currently the recording only takes pictures and saves them in sequence to the snap folder of the project. The user has to put the images together to a movie him/her self, by using a third party program. We have included two sample movies taken with our application, one GIF and one MP4, which are put together using www.gifmaker.me.

The performance of the application is significantly compromised when recording, so we added functionality to toggle recording on/off, leaving it off by default. Even with the performance hiccups, we feel that this is a very useful feature. When recording we recommend finding an interesting location in the Mandelbrot set, and zoom straight out with a very slow zoom speed, as this will produce the most natural result.

4 Performance Requirements

- Number of frames per second

As the main objective of our project is to use the GPU to calculate the Mandelbrot set, the only performance requirement of interest is the speed at which the Mandelbrot set is rendered to the screen. The program should be running smoothly, and at a steady frame rate, with as few hiccups as possible. We expect to have a significant boost in performance when using the GPU vs the CPU, somewhere in the order of 10-100x faster [1].

5 Key Technical Issues

When converting a serial application to a parallel application (that can run on the GPU) the problem has to be partitioned into smaller, more manageable tasks. For each function, the following has to be done [2]:

1. Serialize state to GPU memory
2. Define the kernel code that the GPU will execute
3. Control the kernel launch
4. (De)serialize the state back to CPU memory

The main technical issue we faced in our project was to get the code to run on the GPU. GPUs are specialized for compute-intensive, highly parallel computations, and as a consequence 80% of the transistors are devoted to data processing. This leaves fewer transistors for data caching and flow control compared to CPUs. This along with the large number of cores makes it hard to debug GPU code. However, there are significant advantages to compute certain problems on the GPU, especially when it comes to speed. The GPU has certain strengths over the CPU that makes it ideal for highly parallel problems, such as a lot more cores, raw computing power (FLOPs) and memory bandwidth. In order to experience any improvements in rendering speed, the problem has to map well to the GPU. Problems that generally don't map very well are either too small (so they lack the parallelism needed to use all the threads) or have an unpredictable memory usage.

The Mandelbrot set is well suited for the GPU. First off it is compute bound and has almost no inputs, so the memory usage is quite stable. Secondly, all the threads are independent (each pixel value is independent of all other pixel values, similar to a typical graphics-rendering problem). Another important aspect is that the core algorithm in the task on the GPU is small enough to fit into the register of each multiprocessing unit of the GPU.

[1]CPU/GPU Multiprecision Mandelbrot Set

[2]Rootbeer: Seamlessly using GPUs from Java

There are several different ways to get java code to compile on the GPU. We tested several ways during the project. First we tried using Rootbeer [2], a library for running java code on the GPU. Unfortunately we couldn't even get their own demonstration project to compile. In general we found Rootbeer to be unreliable, with little to no support, and there seemed to be very little activity on their library. From there on we moved on to try OpenCL [3], which we also had problems with.

We ended up using JOGL - Java Bindings for OpenGL [4]. It was much easier to get JOGL up and running, and there is a lot more support behind the library, as OpenGL is the industry standard for high performance graphics. As a consequence there is also a lot more documentation for it.

6 Architecture

Our architecture consists of five classes: **MainClass**, **Mandelbrot**, **Setting**, **Camera** and **ColorSelector**. In addition to these classes we have two GLSL shaders, one fragment shader **mandelbrot.fs** and one vertex shader **mandelbrot.vs**. See figure 1.

6.1 Classes

MainClass - The entry point to our program, all it does is instantiate a Mandelbrot object.

Mandelbrot - This is the class that handles all the OpenGL code. It implements the GLEventListener interface, that comes with four methods: init, display, reshape and dispose. These classes help control the OpenGL rendering. The init method is used to initiate necessary variables for the rendering, in this case it loads the shaders. Display is called for each rendering frame, and is where the quad is drawn and the uniform variables for the shaders are updated. The reshape method is called during the repaint right after the component has been resized. In this case it reshapes the viewport after the canvas changes size.

To create a user friendly and responsive GUI the class makes use of the java swing library by extending JFrame. The class also implements the KeyListener interface, in order to handle keyboard inputs. By having Setting, Camera and ColorSelector objects, everything can be controlled from this class.

Both the fragment shader, and the vertex shader are applied in the init method. To make the program loop we make use of an FPSAnimator capped at 60 frames per second. In the display method the fragment shader uniform variables are updated on each frame to match the variables found in the Settings class. Next a quad is drawn to the screen and the draw operations are flushed to the graphics card.

It is worth mentioning that we do not compensate for the time between each frame render. This might lead the program to slow down on slow computers that spend a lot of time rendering each frame.

[2]Rootbeer: Seamlessly using GPUs from Java
[3]The OpenCL Library
[4]Java Binding for the OpenGL API

Setting - stores positional information as well as the width and height of the Mandelbrot set, the iteration limit and a pseudo boolean value for toggling color smoothing on/off.

Camera - handles the panning and zooming in the Mandelbrot set. It is initiated with an instance of the **Setting** class, which it uses to modify the positional information, to make the camera zoom and/or move.

ColorSelector - contains a method for swapping between different color schemes for the Mandelbrot set. Each color scheme consists of 4 colors represented as three dimensional float vectors where the entries represents red, green and blue respectively. These colors are uniform variables, and can therefore be reached by the fragment shader.

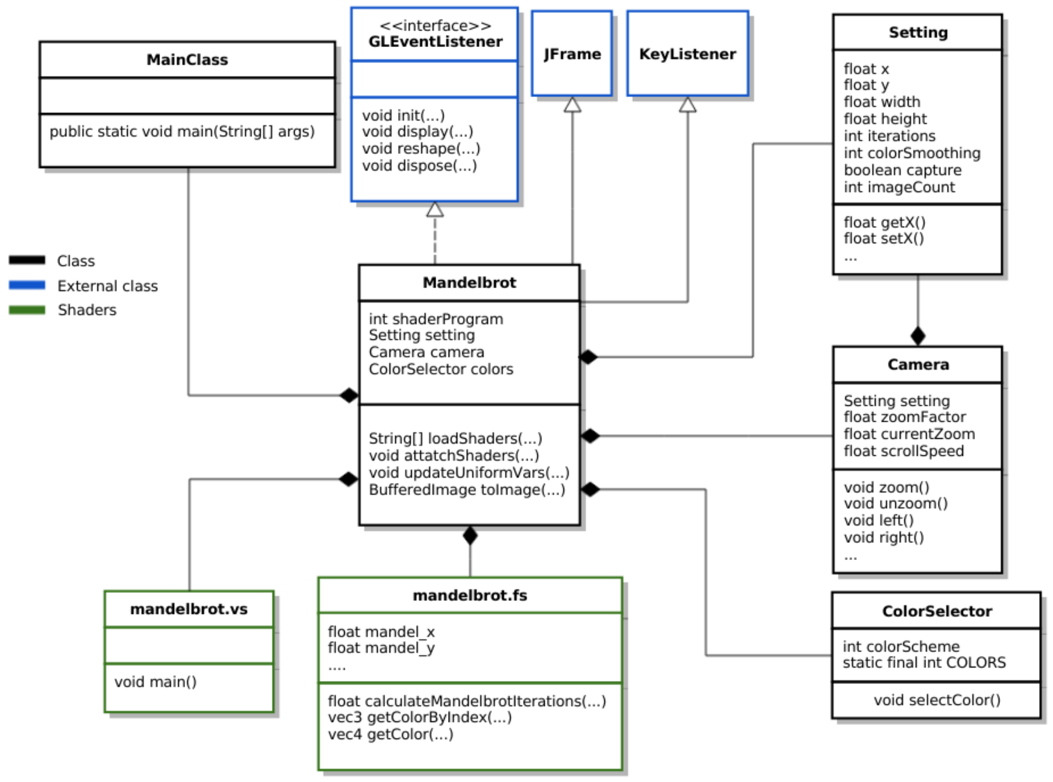


Figure 1: Diagram showing the relationship between classes.

6.2 Shaders

Shaders - When rendering pixels, we are using the OpenGL vertex and fragment shaders. The shaders are written in GLSL (a C like language) and is based on Morten Nobel's blog post [5], with some added functionality like color smoothing (using the Normalized Iteration Count algorithm [6]), as well as allowing the user to change the color scheme. To get access to the shaders, we use JOGL. JOGL is a wrapper library that allows Java to access OpenGL through

[5]Real time mandelbrot in Java
[6]Color Smoothing

the use of the Java Native Interface. Using the shaders, we can create a Quad of vertices, and then use the vertex shader to define a reference coordinate system. Then, the fragment shader computes the color of each pixel, and as a consequence this is where the main Mandelbrot escape time algorithm and color smoothing is executed. Figure 2 shows this entire process, which is called the graphics pipeline.

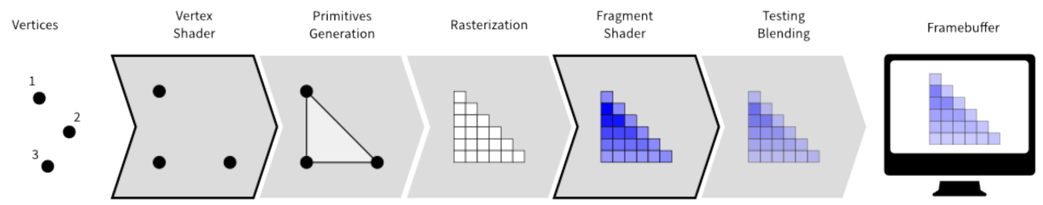


Figure 2: Sequential drawing of the pipeline showing how the shaders work.

Viewing the image - We also have to define how to view the 2D image. Since OpenGL is used to render 3D elements, we have to place the vertices in a single plane, and use an orthogonal camera without perspective to get the right view [7]. Figure 3 illustrates this concept: we create a virtual two-dimensional image in a 3D space (at the far clip plane). Then, this image is projected at the near clip plane, which is what the camera "sees". Since the only thing rendered in the 3D space is our virtual image, this is also exactly what the camera sees. To avoid any distortions in the output image, the camera has to be orthogonal. Finally, the OpenGL libraries cannot make window-system related calls through Java, which means that we have to overlay Java Swing components on top of the OpenGL rendering, to make the GUI correct.

[7]Modern OpenGL

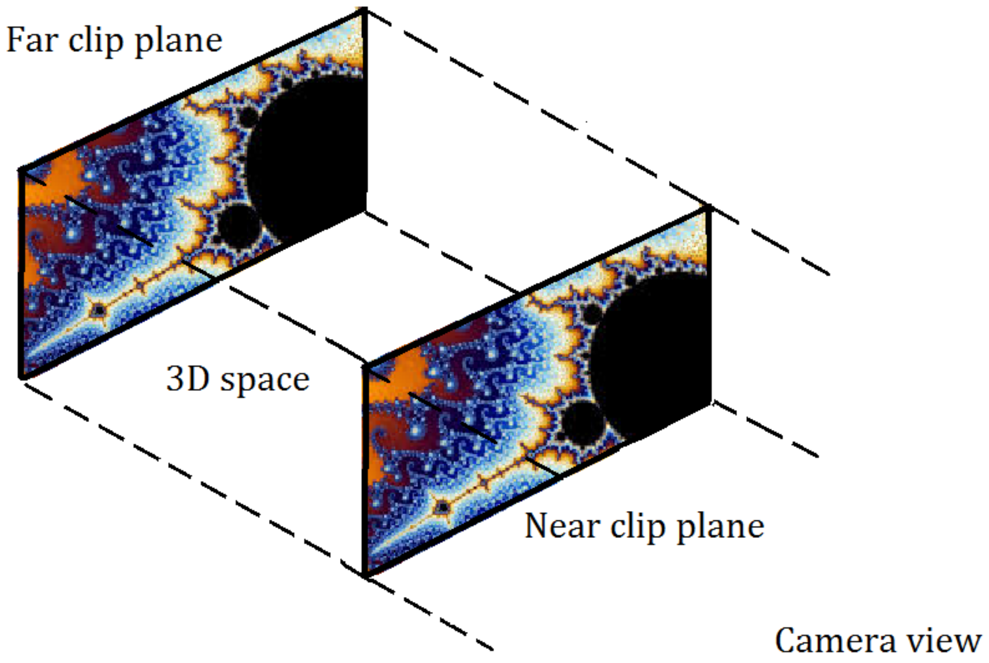
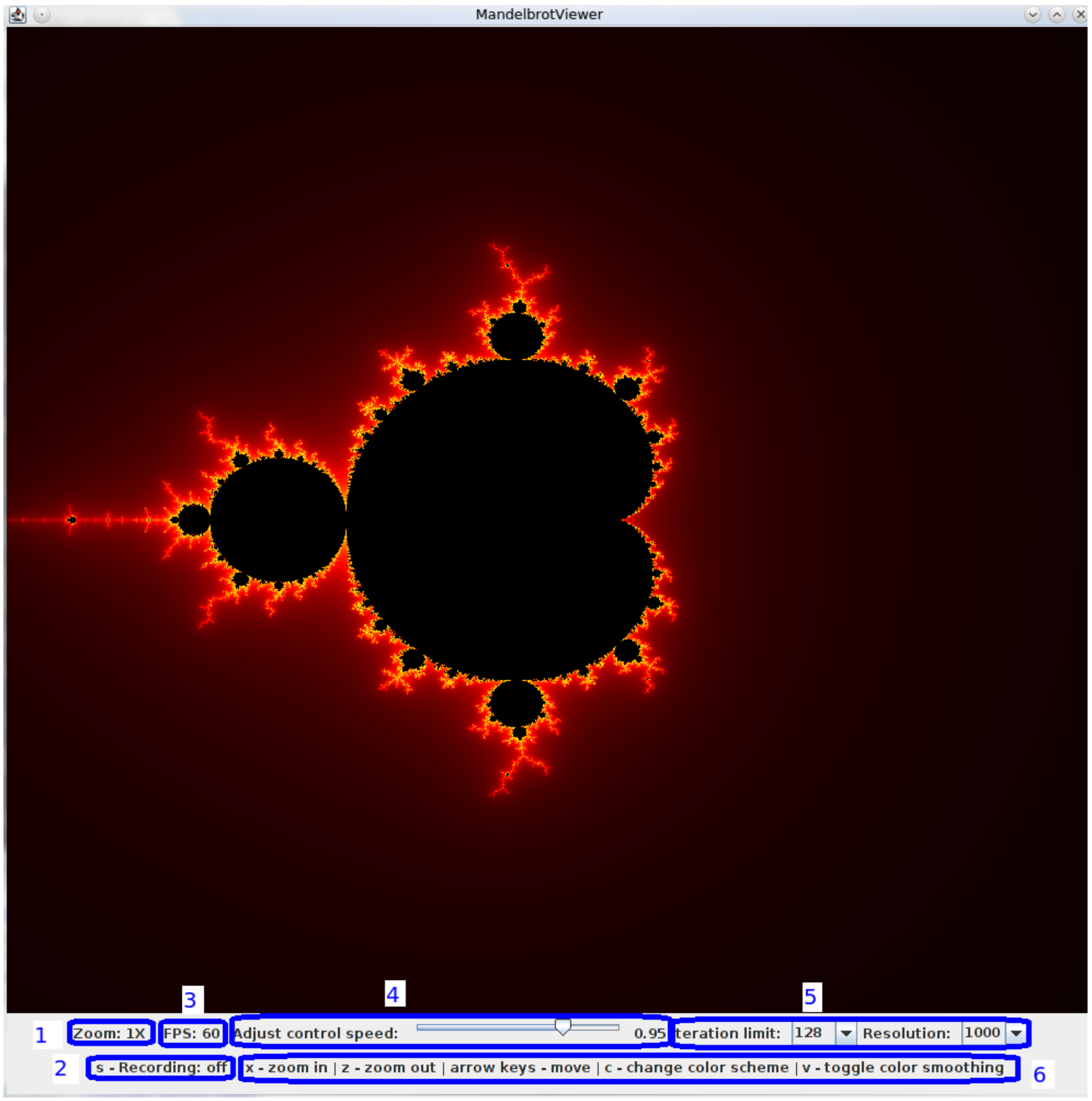


Figure 3: Illustration of the orthogonal camera view.

7 Graphical User Interface



Zoom: 1X | FPS: 60 | Adjust control speed: 0.95 | Iteration limit: 128 | Resolution: 1000

s - Recording: off | x - zoom in | z - zoom out | arrow keys - move | c - change color scheme | v - toggle color smoothing

Figure 4: Illustration of the GUI with enumeration of the functions.

1. States the current zoom level.
2. States the recording status (off/on), and the key press needed to start/stop recording.
3. Shows the number of frames per second (this is capped at 60).
4. This slider allows the user to adjust the control speed (the rate at which you zoom and pan).
5. Combo boxes that lets you select the iteration limit and the resolution of the Mandelbrot program.

6. Instructions telling the user which keyboard button controls what function.

8 Experiments

To show that computing the Mandelbrot set on the GPU is actually worthwhile, we have to compare our application with results from running on the CPU. As we have not implemented it for CPU in this project, we will compare results with our solution for homework number 2. Unlike our project, this is a compute space, so to get comparable results, we use a single JVM to run the task. The parameters for the testing are as follows:

- Pixels: 1024x1024
- Iteration limit: 512

For the CPU, we only test for the following configuration (like in the homework):

- x: -0.7510975859375
- y: 0.1315680625
- Edge length: 0.01611

When doing the CPU tests, we only time the sub tasks computed, and sum these together. This way overhead like latency etc., is completely neglected, and we only focus on the computations themselves. After several tests we found that the average time for computing all the Mandelbrot values on the CPU is 865 ms (ran on a 4 core Intel i7 processor).

When testing the GPU we printed out the time it took for each frame to render, and averaged them. The GPU used for testing was an nVidia GeForce GT750M. We tested around the same area in the Mandelbrot set, and we got an average rendering time of 5 ms for each frame.

$$Speedup: \frac{865ms}{5ms} = \underline{173}$$

As shown, it is around 173 times faster computing the values on the tested GPU than the CPU (these results may vary depending on how fast the GPU and/ or CPU is).

9 Conclusion

This paper has presented our way of utilizing the advantages of calculating the Mandelbrot set on the GPU. We experienced that OpenGL, through JOGL, was a great tool for executing code on the GPU, like described in Key Technical Issues. As mentioned, we achieved a speedup of 173 when comparing the GPU with our previous CPU application from homework two. We conclude that this indeed is a successful result, well within the limit of what we had considered a good speedup. We have also implemented every feature we set out to, by making a self-explainable GUI, allowing the user to change the iteration limit, the resolution, etc.

10 Future Work

There are several things we could look further into, if we had more time.

- A major issue we could try to fix is to use some sort of high precision data structure like **BigDecimal** in the fragment shader instead of using floats. Floats are great for graphics processing, as operating on floats is very fast. This, however, comes at a cost, namely that the floats have fairly low precision. Because of this, the rendered Mandelbrot set will start to get pixelated when zoomed too far in. The problem is that the shading language GLSL is not optimized towards precision, so there is no implementation of a precise data structure.
- The way the recording function works now, is that it takes pictures sequentially and then stores all the pictures under the snap folder, and the user has to put them together to a video themselves. If we had some more time to look into this, we would have liked for this to be automated, with no user interaction.
- To aid with the video recording of the Mandelbrot set we would also like to implement some sort of animator, letting the user save several coordinates of interesting points in the Mandelbrot set, and have the camera move automatically between these points. This would make the recording process much smoother when panning sideways/up/down, resulting in a better end result.
- We would like to have an option to click where we want to zoom, instead of always zooming in at the center of the image.

11 References

[1] Eric Bainville (2009, Dec), *CPU/GPU Multiprecision Mandelbrot Set* - Retrieved from `http://www.bealto.com/mp-mandelbrot_benchmarks.html`

[2] Pratt-Szeliga, Waxcett, Welch, *Rootbeer: Seamlessly using GPUs from Java* - Retrieved from `https://github.com/pcpratts/rootbeer1/blob/master/doc/rootbeer1_paper.pdf`

[3] OpenCL, *The OpenCL Library* - Retrieved from `https://www.khronos.org/opencl/`

[4] OpenGL, *Java Binding for the OpenGL API* - Retrieved from `http://jogamp.org/jogl/www/`

[5] Morten Nobel (2010, Feb 23), *Real time mandelbrot in Java* [Blog post] - Retrieved from `http://blog.nobel-joergensen.com/2010/02/23/real-time-mandelbrot-in-java-%E2%80%93-part-2-jogl/`

[6] Wikipedia, *Color Smoothing* - Retrieved from `http://en.wikipedia.org/wiki/Mandelbrot_set#Continuous_.28smooth.29_coloring`

[7] Nicolas P. Rougier *Modern OpenGL* - Retrieved from `https://glumpy.github.io/modern-gl.html`