Fraskell: Exploring Fractals in Haskell[*]
Final Report

Wessel Custers 5993334
w.r.custers@students.uu.nl
Utrecht University

Xinliang Lu 0822760
x.lu3@students.uu.nl
Utrecht University

Jan Willem de Ruig 6369502
j.w.deruig@uu.nl
Utrecht University

April 21, 2023

1 Introduction

Self-similarity is a widely observed phenomenon in our world; objects or patterns that show self-similar structures can often be described by fractals: recursively self-similar geometric shapes. Even at a small scale - arbitrarily small for infinite fractals - fractals show a detailed structure that is similar to the fractal itself, or a part of it. With their intricate shapes, fractals inspire artists and mathematicians alike; they can be more than just pleasing to the eye. Fractal theory has provided a way to describe and model self-similar patterns and chaotic processes.[1] Complex networks can, for example, exhibit fractal behaviour to certain degrees[2], and fractals are particularly relevant in the field of chaos theory.

Before one can use fractal patterns to model systems or explain certain properties, a good understanding of the underlying concepts is necessary. This requires the practitioner to answer questions like 'what is a fractal?', 'which kinds do exist?' and 'how do they work?'. As we believe that experiencing and experimenting with concepts is a great way to grow understanding, we have created a fractal exploration tool which the user can use to create different kinds of fractals; moreover, the user can scale and translate the rendered fractals to further explore their shapes at different levels. The code base is written in Haskell as we anticipated that the mathematics behind fractals would translate well to Haskell's functional paradigm and syntax. The groundwork of our implementation was laid by a paper on rendering the Mandelbrot and Julia set fractals in Haskell by Mark P. Jones.[3]

2 On Escape-time Fractals

From the many fractal patterns that have been studied, we have decided to limit our explorer to only render escape-time fractals. These all have a characteristic recurrent relation that is applied to each point in space. This makes them fairly straightforward to model via recursion: define a space of points, for example a 2D-grid, and iterate on each coordinate with the characteristic function of a certain escape-time fractal. This simple yet elegant approach to rendering these fractals was provided by Jones.[4] The most common fractals falling into this category are the Mandelbrot, Julia, Newton, Nova, Burning Ship and Lyapunov fractals.

[*] Available at: https://github.com/largeword/UU-INFOAFP2023-Fractals

[1] What are Fractals? – Fractal Foundation. 2023. What are Fractals? – Fractal Foundation. [ONLINE] Available at: https://fractalfoundation.org/resources/what-are-fractals/. [Accessed 20 April 2023].

[2] Wikipedia. 2023. Fractal dimension on networks - Wikipedia. [ONLINE] Available at: https://en.wikipedia.org/wiki/Fractal_dimension_on_networks. [Accessed 20 April 2023].

[3] Mark P. Jones. 2004. Composing fractals. J. Funct. Program. 14, 6 (November 2004), 715–725. https://doi.org/10.1017/S0956796804005167

[4] Mark P. Jones. 2004. Composing fractals. J. Funct. Program. 14, 6 (November 2004), 715–725. https://doi.org/10.1017/S0956796804005167

Escape-time fractals are quasi self-similar, meaning that the patterns are not exactly the same on different scales. This can be observed when zooming in on the Mandelbrot fractal: the smaller satellites resemble the bigger pattern yet they have distinct shapes. The main fractal pattern may occur in different and distorted forms, meaning that subsets of points may not be equal to the entire fractal set.

Below are the recurrent relations of the fractals that we have been able to render with our explorer. The origin of the functions is denoted by z, the translation (called the offset in the following) of the functions by c.

- ***Mandelbrot***: $z_{a+1} = z_a^2 + c$, where z is fixed on $z = 0$ at the start of each iteration sequence.
- ***Julia***: $z_{a+1} = z_a^n + c$, where c is fixed at the start of each iteration sequence.
- ***Burning Ship***: $z_{a+1} = (|Re(z_a)| + |Im(z_a)|)^2 + c$, where either z or c is fixed at the start.

3 Application

The application itself is a stack-based project, broadly following the MVC framework. There are six modules in total, each with their respective responsibilities:

- `Main` provides the starting point for the application
- `Model` provides definitions for various data types, magic numbers, as well as general helper-functions pertaining to those types
- `View` provides functions for generating a `Picture` from our grid
- `Controller` provides functions for the main step loop, as well as in-app user interaction
- `Console` provides functions for various IO functions enabling console interaction to define parameters
- `Generator` provides functions regarding the generation of the fractal itself

Some of these modules contain strictly more than one responsibility, such as the inclusion of helper-functions in `Model` and grouping both the main step loop and in-app user interaction in `Controller`. However, given the relatedness of the functionalities, as well as the scope of the code, this was deemed acceptable - if not favourable over having various other modules with only a few functions in each.

3.1 Rendering Algorithm

Central to our fractal exploration tool is the system responsible for rendering these fractals. This is done by mapping a sequence of functions over each pixel for our screen. Starting out, we generate a `Grid` of `Points` representing the screen at the start of the application. The dimensions of this grid depend on a specified width and height, and is shifted on creation to have $(0, 0)$ at its center. This grid is created once at the start of the application, and threaded through the various computation steps.

Main pipeline

The first step in the generation process is to transform this grid of integer coordinates into floating point coordinates on the imaginary plane. By default we scale such that the screen is mapped to a range of $[-2, 2]$, along the shortest axis in the case that the x and y dimensions are not the same. This ensures that the salient areas of the fractal around $(0, 0)$ are always plainly visible from the start. The user is then free to zoom further in, out, or pan.

After scaling, the grid is mapped into a sequenced grid: for each pixel, we generate an infinite list populated with the recursively applied fractal function. This function takes the base form $f(z) = z^n + c$, where z is our starting coordinate and c provides an offset. The sequenced list takes the form $[z, f(z), f(f(z)), ...]$, and starting from head of the list, a number of elements equal to the user-defined escape-step[5] count is subsequently taken.

[5] The escape-step variable represents a measure of fractal resolution.

Next, we check for each of these points if they are close enough to the fractal interior or instead have crossed this escape-step treshold; this produces a list of booleans. By counting the number of points that fall within the exterior before escaping outwards, we obtain the escape-step value for this coordinate. This value can then be mapped to a colour by interpolation: we scale the range of escape-step values to our colour range, and find a corresponding colour for every point. If a point falls between two colours, we mix these colours, with proportions relative to where the point is mapped between them. This results in a smooth gradient of colours, without needing to define more specific colours.

Finally, we zip this grid of colours with our original grid of screen space coordinates, to allow us to map every colour to the right position and create a picture. By making the picture generation itself a part of the rendering pipeline, we can store the result in our world state. This allows us to repeatedly display this same picture every step, as long as no changes to the world (i.e. zooming and panning) have been made, which significantly reduces computation time. As soon as the user does pan or zoom, a flag is raised, and the fractal picture is recalculated.

User interaction

As mentioned above, the user is able to interact in various ways with the application while it is running. The arrow keys and WASD allow the user to pan the image, and the 'Q' and 'E' keys enable the user to zoom out and in, respectively. We keep track of this internally using a tuple containing the `ZoomScale` and `Translation`: a Float defining the scale of zoom and a tuple defining a translation in respectively x and y direction.

The zoom scale is 1 by default and multiplied with our default scale factor. It is changed by a static factor of 0.1 with every press of the button, and is applied to both our coordinates via simple scalar multiplication.

The translation is applied by simply adding the x-component to our x-coordinate and the y-component to the y-coordinate. However, these panning factors do not change statically with every press of the button: Rather, they change depending on both the default scale factor *and* the zoom scale. This ensures that the number of pixels panned every time will be more-or-less consistent, invariant of zoom level.

3.2 Abstraction

To abstract over different kinds of escape-time fractals, we first looked at what parameters they had in common. The Mandelbrot, Julia and Burning Ship fractals are all generated via a polynomial of the form $z^n + c$, where z is the origin of the function and c the offset; both values are complex numbers. For the Burning Ship fractal, the absolute value should be computed instead of the regular complex value. In our model, we have parametrized the polynomial degree n and the choices whether to iterate on the absolute value of z, and whether to vary the origin or the offset values. These choices define the shape of the fractal, the information about the (current) fractal, i.e. the users decisions, are stored in the GeneratorData data type.

All concrete values of the parameters come together in our *FractalFunction* data type. At the start of our application, the user chooses whether to take the absolute value of z or not and what n, the degree of the fractal function, should be. The user can change n while the program is running by reprompting the GenerationData initialization in the console with the 'G' button. One needs to be careful though with inserting large values for n as larger values will increase the computational cost of rendering the fractals.

3.3 Packages

Gloss

The main program loop is defined using Gloss' interactive 'game' preset. The `play` function provides us with a framework that allows us to define a world state, drawHandler, inputHandler and stepHandler. It is important for us to use this preset rather than for instance the 'simulation' preset, as 'simulation' does not provide the inputHandler necessary for user interaction.

We decided to use Gloss for this, as we were already fairly familiar with the framework, and it included everything we needed. However, our code does deviate from the structure Gloss provides in some areas. Mainly, the drawing: as mentioned earlier, drawing is a part of our pipeline, as this allows us to cache the generated picture and display it at every step if no changes are made. However, this makes the drawHandler only responsible for drawing the existing picture to the screen, not for generating the picture. And likewise, the functionality which renders the picture is called from the stepHandler function.

There exist ways to mitigate this discrepancy. For instance, we could check for flags in both the stepHandler and drawHandler to decide whether we want to re-generate and re-render the image, or show the original one. This would return the responsibility of image rendering back to the drawHandler. However, this would also require us to keep track of additional data in our world state, in the form of our escape-step values, where currently this grid is simply threaded through to the next function and discarded. Ultimately, we decided that it would be better to store less data in the world state and refrain from keeping track of this grid between ticks, even if gives a discrepancy in our code.

Accelerate

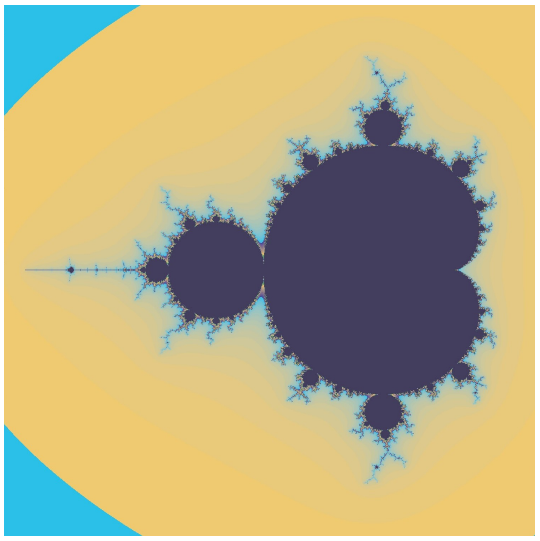
The main reason for using the Accelerate library is that mapping functions to 1000 by 1000 pixels is very time-consuming running for a single core. Making the mapping function run on multiple CPU cores or even on the GPU - with potentially thousands of cores - would be a substantial improvement. Accelerate is a Haskell library containing an embedded domain-specific language for array acceleration, which utilizes both fusion - combining multiple operations into one - and parallelism - running mapping functions and other operations on multi-cores.

In our code, the main changes we have made are in the *Generator.hs* file. Logical functions are rewritten with `ifThenElse` to avoid too frequent lifting and unlifting between the embedded language of Accelerate and the normal Haskell syntax, and this is the same with numeric calculation operations. More particular, because the `iterate` function behaves differently in Haskell and Accelerate, we cannot simply generate an infinite function list and use the `take` function to get the fractal sequence. Instead we create the list (with the same value) first, then we map the `FractalFunction` to every item in that list. The index along this axis determines how often we iterate. We also changed the `getColors` function in *View.hs*, to map the escaping step into a color within Accelerate. Moreover, we decided to perform CPU acceleration instead of using the GPU. This is because not all operations in Accelerate are supported for GPU. This would lead to operations being spread across both CPU and GPU, resulting in overhead due to data transfer. Ultimately, our project can run at both the CPU and the GPU, and use multiple cores to boost the process.

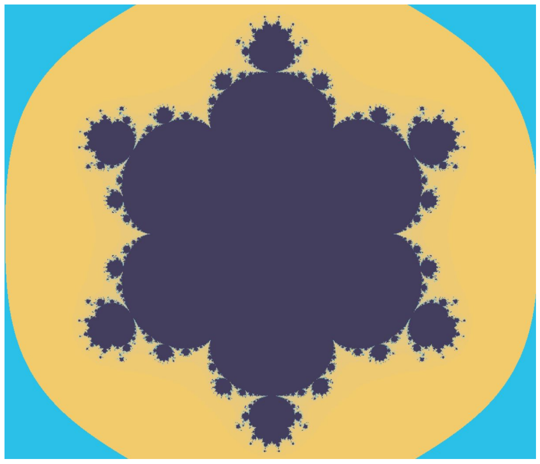
4 Results

The final result of the project is a fractal exploration tool which allows the user to check out various flavours of the Mandelbrot, Julia and Burning Ship fractals. It has both a universal version which works on most infrastructures that support Haskell, and an accelerated version, which works only on linux infrastructures with CUDA-enabled GPUs but features a speed increase. The features of these versions, aside from the aforementioned, are identical: The user can pan, zoom, and tweak parameters.

To demonstrate the capabilities of the tool, we have generated various intricate and non-standard fractals, attached below.

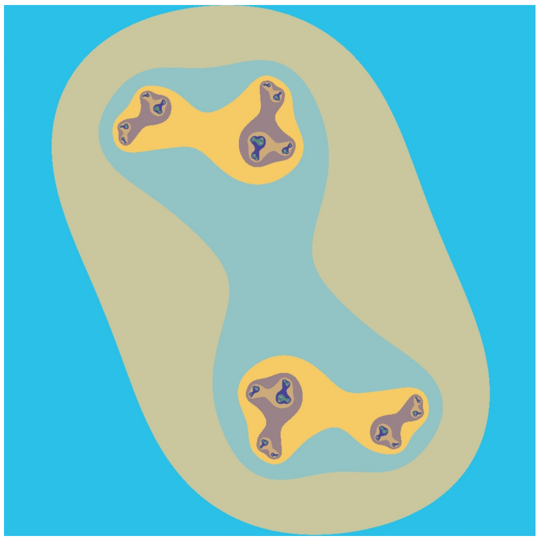


(a) second polynomial degree

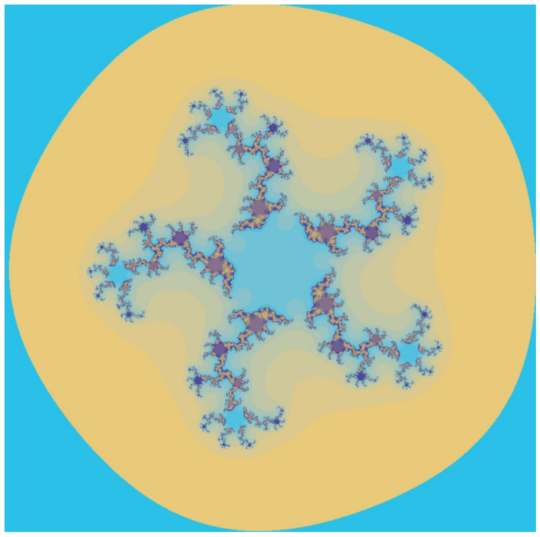


(b) seventh polynomial degree

Figure 1: Mandelbrot Set.



(a) second polynomial degree



(b) fifth polynomial degree

Figure 2: Julia Set.

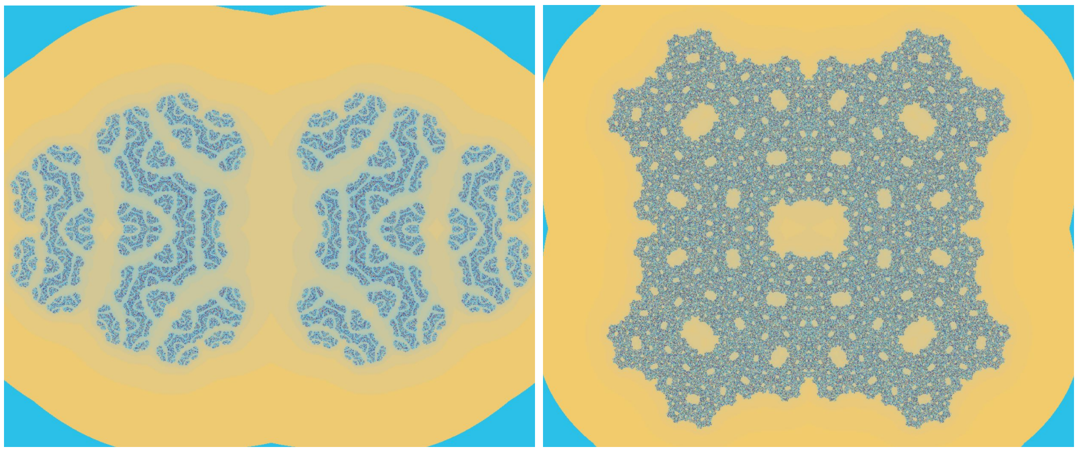


Figure 3: Burning Julia fractals.

5 Reflection

While developing the application, we encountered several roadblocks which we will discuss in detail below.

Perhaps the biggest issue we had was with the Accelerate library. Our initial attempts at using the library did not pan out, due to various factors within and outside of our control. For starters, the Hackage package provided on the website is out of date and does not support for LLVM-12 – the oldest LLVM supported by the current system. With the help of Ivo Gabe de Wolff, we managed to pull the latest source code from GitHub, and compile this with our system. A further limitation of using Accelerate, however, is the fact that Accelerate requires a CUDA-enabled GPU, as well as certain packages which are not supported on Windows. This resulted in only one member from our group being able to work on the accelerated project. In hindsight, it may have been preferable to look in to VM-technology that might have allowed more of us to contribute to this accelerated branch, and might have enabled us to spread the work to be done more evenly.

Another issue we encountered during the earlier development stages, was that the `getColors` function produced NaN errors. In the situation that all pixels escaped at the very first step, mapping the escape-time value into the color range would result in a $\frac{0}{0}$ division. Using the `ifThenElse` operation to set the mapped value to zero when the numerator would be zero turned out to be the solution.

Lastly, the generalization underlying our fractal function model is the observation that the recurrent relations of the Mandelbrot, Julia and Burning Ship sets have the same polynomial form, $z^n + c$. Other escape-time fractals, such as the Nova and Lyapunov fractals, cannot be described by this model as their recurrent relations have a different form and contain additional parameters[6] [7]. How to extend or adjust the current fractal model to include these fractals as well is left for future research.

6 Future Work

Mentioned below are some suggestions and ideas that we have considered but did not have the time to (fully) implement.

- Extending our model to support Nova (including Newton fractals) and Lyapunov fractals. This will probably require a different fractal model than we currently use.

- Further optimization of the code. We believe parallelization has the biggest potential to speed up our program as for each point in our grid, all calculations happen independently from other grid points. We have parallelized most of our functions with Accelerate but some, like `draw` and

[6] Wikipedia. 2023. Newton fractal - Wikipedia. [ONLINE] Available at: https://en.wikipedia.org/wiki/Newton_fractal. [Accessed 21 April 2023].

[7] Wikipedia. 2023. Lyapunov fractal - Wikipedia. [ONLINE] Available at: https://en.wikipedia.org/wiki/Lyapunov_fractal. [Accessed 21 April 2023].

`pointToPicture`, are still using the single-threaded mapping, since we need to convert the pixel list to the internal data type of Gloss. Using the Gloss-Accelerate library, which extracts APIs compatible with Accelerate array, may save the overhead of several data conversions.

- Supporting smooth panning or zooming when keeping down the controls for zooming and panning. This heavily depends on the time needed to compute a screen frame. If fractal rendering is smooth every frame, the user will be able to see the impact of their actions live, and can react to these changes in real-time.
- Only recompute points on the grid that need to be adjusted may reduce the number of computations significantly. For each point, one could check whether it changes in the next frame, and only calculate new grid and colour values for those points. The challenge is how to implement this check. Even if one could think of one, the implementation should also be more efficient than the current one.
- Publishing our fractal explorer as an interactive web-app. Elm could be used to provide the front-end of the app while our code base in Haskell forms the back-end. The console interface could be transformed into a GUI with sliders or input fields for the different fractal parameters like origin, offset and polynomial degree.