Fractal Computation and Visualization

an INFOAFP project proposal

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

February 27, 2023

1 Domain Description

Fractals are found everywhere in nature: leaves, branches, flower petals, snowflakes and coastlines are just a few examples of fractal occurrences. In a more abstract sense, they are representations of chaotic dynamical systems. Some more abstract phenomena can even contain fractal patterns, such as complex networks. Fractals can therefore be used to model self-similar recursive properties of objects and phenomena.

1.1 Main Goal

We want to abstract over different fractal patterns. Can we find common structural behavior that can, for example, be modeled through a type class in Haskell? First, we will try to model one particular fractal pattern and build a renderer for it. If that goes well, we can extend our project, working towards our main goal, in the following ways:

1. Adding new fractal patterns and trying to abstract over them.
2. Adding basic interaction functionality (zooming, panning, adjust colours)
3. Implementing the renderer with ray-tracing technologies.

1.2 A brief mathematical description

Although the word 'fractal' has several connotations, for the purpose of our application, we consider a fractal to be a geometric shape that is recursively self-similar. No matter how far one would zoom in or out, parts of the fractal that are in view will still resemble its original form. In general, fractals contain the following features:

1. Self-similarity, which can occur in different fashions.
2. Multifractal scaling: ruled by more than one fractal dimension or scaling rule.

In fact, a fractal is mathematically represented by the formula $z_{n=1} = f(z_n)$, which is interesting in that we can use the lazy evaluation of Haskell to create an infinite algebraic type to represent this pattern, and this list will only be evaluated as the diagram evolves. We can use this Haskell feature to efficiently show how the image changes and how fractals come to be.

1.3 Rendering and interactivity

There exist various libraries and packages to facilitate rendering images in Haskell. For this project, we have decided to make use of Gloss, as this package hides many of the difficulties surrounding the domain. Since we will mainly be working with flat 2D fractals, we do not have to concern ourselves with trying to rotate or otherwise permute our renderings. Given the simple nature of the package, and our familiarity therein, it seemed like the natural option.

Interactivity may prove to be more intricate to implement, even if kept to just panning and zooming. It would require us to be able to read keyboard (and potentially mouse) controls, and have a constant update function. For this reason, Gloss' Game template seems more useful than just the Display or Animate templates.

Reading the controls using Gloss will be simple, though handling them may prove more complex. Our main focus on interactivity will be on the ability to pan or zoom, initially using just the keyboard though mouse support may be included eventually. One potential concern about the zoom functionality is that many parts of the fractal that are off-screen (and thus not rendered) may end up taking up a lot of memory, but we hope that Haskell's lazy evaluation may be able to help us out here.

Giving the fractal a unique colour is also a part of the rendering, though we expect this to be less of a challenge. By using an enumeration type for colours, we may be able to use a function such as `succ` to ensure every next iteration of the fractal is rendered in a different colour. If time permits, we may be able to give the user some control over the sequence of these colours as well.

2 Problem Description

Being able to visualize fractals can be a great way to familiarize with the concept, and what could be better than being able to play around with it? Our application aims to be a tool for others to explore what fractals are, how they are made, and therefore how they work. For the fractal fanatics out there, one might even be able to create their own fractal art through a simple interface for interacting with the fractal renderer.

3 Schedule Estimation

Week 9	Setting up the working environment (Github, Cabal), basic reading
Week 10	Style guide definition, First fractal implementation
Week 11	First render engine implementation, fractal rendering pipeline
Week 12	Abstract fractal definition, implement more fractals
Week 13	Implement basic movement (pan, zoom), prepare presentation
Week 14	Time for unforeseen challenges, Code cleanup, Start with final report
Week 15	Report, Code cleanup

Table 1: estimate schedule

See table 1 for a simple overview of our estimate schedule. We expect to get less work done during the first few weeks due to other assignments and plan on using these mainly to get our environment set up. Thereafter, we will focus on expanding the application by abstracting over fractals, implementing more different fractals, and improving the rendering engine. If time allows, we can explore improving our renderer, for instance by implementing raytracing techniques, adding an 'adjust color' functionality to the GUI, and parallelization. The later weeks are reserved primarily for studying for the upcoming exam, working on the report, and polishing our existing code.