university of groningen

faculty of mathematics and natural sciences

In by Out again

Arbitrarily-deep zooming on Iterated Function Systems by ‘self-similarity jumping’

Bachelor’s thesis

August 17, 2020

Student: Wiebe-Marten Wijnja

Primary supervisor: dr. J. Kosinka

Secondary supervisor: G. J. Hettinga

Contents

1 Abstract **2**

2 Introduction **2**

2.1 Overview . . . 3

3 Background **3**

3.1 Informal description of an Iterated Function System . . . 3

3.2 Formal definition of an Iterated Function System . . . 4

3.2.1 Restriction to affine transformations on the two-dimensional euclidean plane . . . 4

3.2.2 The viewport transformation . . . 4

3.3 Rendering an Iterated Function System . . . 5

3.3.1 The deterministic method . . . 5

3.3.2 The chaos game . . . 6

3.4 Paralellizing IFS rendering by using a Graphical Processor Unit . . . 7

3.4.1 The chaos game on the GPU . . . 8

3.4.2 The deterministic method on the GPU . . . 8

4 Research Question **8**

5 Approach **8**

5.1 Design . . . 8

5.1.1 Point clouds . . . 9

5.1.2 Potential point cloud-based optimizations . . . 9

5.1.3 Self-similarity jumping: 'zooming in by zooming out' . . . 9

5.1.4 Coloring the rendering . . . 11

5.2 Implementation . . . 11

5.2.1 Simplicity . . . 12

5.2.2 Command-line options . . . 12

5.2.3 '.ifs' file format . . . 13

5.2.4 Rendering . . . 13

5.2.5 Changing the camera viewport . . . 14

5.2.6 Manually performing 'self-similarity jumping' . . . 14

5.2.7 Rendering 'guides' . . . 14

6 Findings **14**

6.1 Restrictions on 'self-similarity jumping' 14
6.2 Memory Usage 17

7 Conclusion **17**

8 Further Work **17**

A IFSs used **20**

Todo list

Some examples of IFSs? (that are seen again later on in the thesis) 4

1 Abstract

Iterated Function Systems (IFSs) are a mathematical approach to rendering fractals that sees wide usage in the modeling of physical phenomena, image compression, and the creation of abstract art. When exploring an IFS interactively, current IFS rendering techniques require a full re-approximation of the IFS' attractor at every frame.

This thesis proposes a combination of two techniques to enable the faster exploring of IFSs. First, a point cloud is used as intermediate attractor approximation, that can be re-used between animation frames. Secondly, a technique coined 'self-similarity jumping' is employed to keep the attractor representation detailed, even when zooming in very far.

A proof-of-concept computer program has been implemented which shows that the employed techniques, while promising, are somewhat restricted in their usefullness because self-similarity jumping cannot be used in all situations.

2 Introduction

Iterated Function Systems (IFSs) are a method to generate infinitely detailed fractal images by repeatedly applying simple mathematical functions until a fixed point is reached. [Bar88] IFSs see use in the modeling and rendering of physical phenomena, image compression [Har96] and representing gene structures [Jef90]. Sometimes they also are used plainly for the aesthetic beauty of their graphical representations [DR03].

Various computer algorithms to visualize IFSs exist [HPS91]. However, these all take a single still image as final result. If they want to render an animation, this is treated as a sequence of completely separate still images.

This leaves a venue for potential optimization: if there is information that remains the same between animation frames, then we could compute it only once and re-use it for all frames.

For instance, many kinds of animations consist of transformations of the camera viewport w.r.t the viewed fractal over time like translation, rotation and scaling. These transformations do not require alterations to

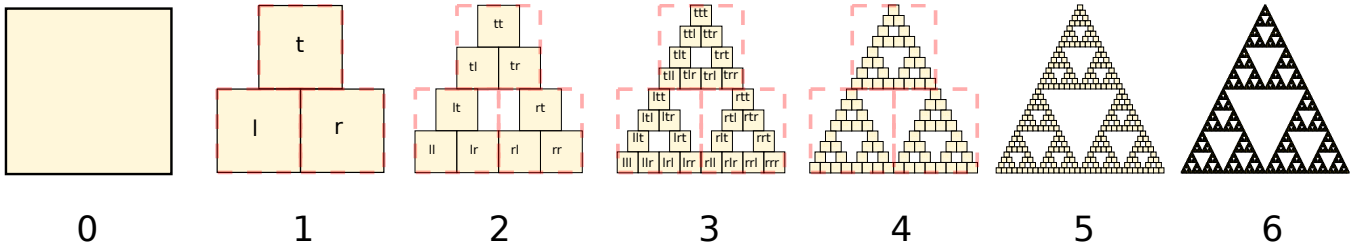


Figure 1: *The first six iterations of the Sierpińsky triangle IFS (IFS A.1). The initial image is just the unit square. We then iteratively combine the results of transforming the current image using one of the three mappings. The letters indicate which (sequence of) transformation(s) resulted in this part of the image. Dashed red lines are used for the first four iterations to indicate the self-similarity between the previous iteration and the current one extra clearly. Even after a couple of iterations it can be seen that the shape of the original image has no influence on the shape of the attractor.*

the fractal itself. This means that (an approximation of) the fractal might be computed once and then be used for all frames.

Furthermore, because of the self-similar nature of the rendered fractals, it might it be possible to simulate zooming in to an arbitrary depth by 'jumping up' to a more shallow camera viewport that shares the same self-similarity as the original one.

This thesis is an in-depth investigation of these two ideas.

2.1 Overview

In the next section, § 3, Iterated Function Systems will be introduced and pre-existing methods to render their attractors in single-threaded and heavily parallel environments described. This then leads to a clear definition of the research question in § 4. The approach taken to test this question is described in § 5, followed by a qualitative discussion of the results in § 6. We conclude in § 7 and finally hint at some approaches for further work in § 8.

3 Background

This section will describe the different building blocks necessary to formalate the research question. First, IFSs are formalized, followed by a description of the different ways in which an IFS' attractor can be rendered, and how these techniques might be paralellized.

3.1 Informal description of an Iterated Function System

Informally, an Iterated Function System is a set of mappings (transformation functions) that, given any input image, can create a new image by

1. transforming the input image with each of the transformations;
2. combining all transformed images together. This is the new image.

This process is then repeated an arbitrary number of times, until changes between the input image and new image are no longer visible to the human eye.

The image you end up with is a visual representation of the IFS's attractor. A simple example of this process can be seen in Figure 1.

Some examples of IFSs? (that are seen again later on in the thesis)

3.2 Formal definition of an Iterated Function System

Formally, an Iterated Function System consists of a finite set of contraction mappings that map a complete metric space $(\mathcal{M}, d)$ to itself [Bar88] :

$$= \{f_i : \mathcal{M} \to \mathcal{M} | i = 1, 2, \ldots, N\}, N \in \mathbb{N}$$

All mappings are required to be contractive. This means that for each mapping f_i, the distance between every two arbitrary points a and b in $(\mathcal{M}, d)$ needs to be larger than the distance of these points after transforming them:

$$d(f_i(a), f_i(b)) < d(a, b)$$

We can then take the union of performing all of these mappings on any compact set of points $\mathcal{S}_0 \subset \mathcal{M}$. This procedure is called the *Hutchkinson Operator* (H). It can be iterated as many times as desired:

$$\mathcal{S}_{n+1} = H(\mathcal{S}_n) = \bigcup_{i=1}^{N} f_i(\mathcal{S}_n)$$

When performed an arbitrary number of times, the fixed-point or attractor, $\mathcal{A}$, of the Iterated Function System is approached:

$$\mathcal{A} = \lim_{n \to \infty} \mathcal{S}_n$$

Curiously, which set of points $\mathcal{S}_0$ we started with makes no difference. (We might even start with a single point.) [Men03].

3.2.1 Restriction to affine transformations on the two-dimensional euclidean plane

Most research of IFSs restricts itself to using $\mathbb{R}^2$ as metric space[1] which can easily be rendered to screen or paper. Furthermore, most commonly-used IFSs only use *affine transformations* as mappings.

It is very practical to work in this restricted domain and potentially generalize obtained results to a wider domain later. Therefore, these are also the restrictions that will be used in this thesis.

3.2.2 The viewport transformation

When rendering graphics, we view the world through a (virtual) *camera* which has a particular frustrum that limits what parts of the world (in this case: the IFS' attractor) ends up on the *viewport*.

[1] More formally, the two-dimensional Euclidean space: $\left(\mathbb{R}^2, d(p, q) = \sqrt{(p-q)^2}\right)$.

1. Scaling vs zooming

   Because of the presence of a camera, the part of an object that will be visible in the camera viewport will change when scaling said object. We use the term 'zooming' to disambugate this type of scaling where a camera is present.

2. Freedom of choosing an initial camera position and frustrum

   For any IFS with mappings we can transform its attractor by any invertible function t by adjusting each of the mappings according to the transform theorem [Bar88] $f_i' = t \cdot f_i \cdot t^{-1}$, essentially transforming points from the new space to the old space, then applying the mapping, and finally transforming them back to the new space. This allows us to give users freedom to choose any desired mappings together with an 'initial camera transformation', while still allowing all calculations to happen with regard to the unit square, keeping them simple.

3.3 Rendering an Iterated Function System

A couple of algoritms exist to render (visualize) the attarctor of an Iterated Function System. While it is impossible to render the attractor exactly, as this would require an infinite number of transformation steps, we can approximate it until we are certain that the difference between our approximation and the attractor is smaller than the smallest thing we can visually represent (e.g. smaller than the size of a pixel).

Because we apply H many times and each time consists of taking the union of N different transformations, the result can be seen as traversing an (infinitely deep) tree of transformations, where each sub-tree is self-similar to the tree as a whole.

Different algorithms take different approaches to evaluating this tree (up to a chosen finite depth).

More in-depth information about the rendering of Iterated Function Systems can be found [HPS91]. Short summaries of the two most common techniques will now follow.

3.3.1 The deterministic method

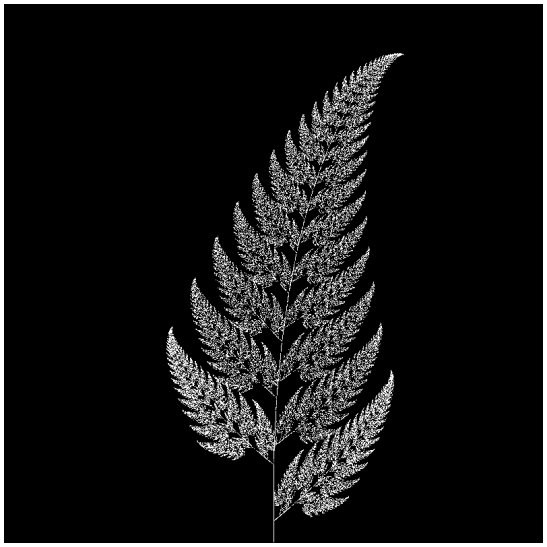
In this approach we evaluate the whole tree up to a chosen depth. The algorithm works as follows:

1. Pick a starting point z_0;
2. Traverse the tree down to the chosen depth k, building up a sequence of transformations [2] $f_{i_k} \circ \ldots \circ f_{i_1}$;
3. For each node at this depth, evaluate and render $z_k = (f_{i_k} \circ \ldots \circ f_{i_1})(z_0) = f_{i_k - 1}(z_{k-1})$;

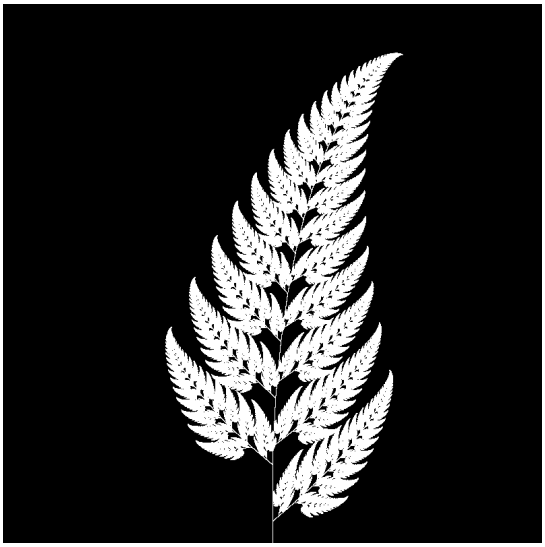
Since $z_k = f_{i_k - 1}(z_{k-1})$ this procedure takes, for an approximation that consists of N points, depending on the tree traversal chosen:

- a linear amount ($\mathcal{O}(N)$) of memory for a breadth-first tree-traversal.
- a logarithmic amount ($\mathcal{O}(\log N)$) of memory for a depth-first tree-traversal.

[2] $\circ$ stands for function composition: $(f \circ g)(x) = f(g(x))$. Be aware that when affine transformation functions are represented as matrices (e.g. F and G), matrix multiplication is in the opposite order ($f \circ g \equiv G \cdot F$)



(a) 1,000,000 points



(b) 10,000,000

Figure 2: *The Barnsley Fern, rendered using the chaos game with different amounts of points.*

The advantage of the breadth-first traversal is that generation could be stopped interactively, while the depth-first traversal requires the stopping criterion to be known beforehand. [HPS91]

While the deterministic method is easy to understand (and indeed is a direct translation of the informal process described at the start of § 3), it is usually less efficient and more complex to implement on a computer than the algorithm that will be described next.

3.3.2 The chaos game

The *stochastic method* [HPS91], also known as the *random iteration algorithm* [Bar88] or more frequently the *chaos game*, works as seen in **Algorithm** 1

Algorithm 1: the chaos game

n: the number of transformations the IFS consists of.
z: a random point on the screen
while less than N points plotted **do**
    i: a random integer between 0 and n.
    $z = f_i(z)$
    render(z) except during the first x iterations
end

This method converges to a correct result because of the following two facts:

- because the precision of the canvas we render on is finite, and because all transformations are contracting, two points a and b are indistinguishable after only x transformations. In other words, only the latest x transformations determine at what location on the canvas a point will end up (with the latest transformation having the largest effect on the point's final location).[3]

[3]Methods for precisely determining the lower and upper bounds of IFS contraction for a particular IFS (and therefore the exact value of x) exist [HPS91], but are not relevant for this thesis.

- at each depth in the tree the subtree remains the same, so every sequence of transformations approaches the attractor.

Therefore, all intermediate points after the first x iterations are visually indistinguishable from the a point that is part of the attractor. By running this non-deterministic approach for enough iterations we approach a diverse enough set of 'transformation sequences of length x' that we end up covering the whole attractor.

The nice thing about the chaos game is that it does not require any extra memory (besides the point z). Also, because it is so simple and little auxiliary memory is needed, it runs very efficiently on modern CPU architectures.

A disadvantage of the chaos game is that the result is by its very nature *non-deterministic*. If not enough points are used, the result might end up 'grainy' and it is not predictable what part of the attractor will be covered (see Figure 2).

One further disadvantage the chaos game has, is that in its simplest form, all transformations have an equally likely chance to be used. However, because some transformations might be (much) more contracting than others, this means that coverage of the attractor is not even, which means that we need to use much more iterations than would be the case if we balance it out.

Therefore, most implementations of the chaos game allow (or require) the user to specify a *probability* for each transformation. All these probabilities together ought to sum up to 1.[4]

❋

Because of its simplicity and computational efficiency, the chaos game is used more frequently than the deterministic method for practical implementations. The chaos game is also easier to paralellize for Graphical Processor Units (GPUs), as will be outlined in the next subsection.

3.4 Paralellizing IFS rendering by using a Graphical Processor Unit

It is enticing to port IFS rendering to run on GPU-architecture because to produce a smooth image, often hundreds of millions of points are needed.

However, optimizing IFS rendering to run well on GPU-architectures is a bit of a challenge.

GPU shaders usually operate by running a check for every pixel on the final canvas, to determine its color. For other fractals like the Mandelbrot- and Julia-sets, this is a natural fit since the construction of those fractals works exactly in that way.

For an IFS this does not work, as an IFS is created in the other direction. Points end up at some location on the canvas *only after transforming* many times. Attempts to go the other way fall flat, for instance because this would require to invert the IFS' mappings, but they are not guaranteed to be invertible.

Instead, General-Purpose GPU-techniques are used that are able to use the top-down approach.

[4] These probabilities are often fine-tuned by hand, although algorithms to determine balanced probabilities exist as well [HPS91].

3.4.1 The chaos game on the GPU

The deterministic method is difficult to paralellize on the GPU because of the extra memory that is required to keep track of the current position in the tree. Coordinating which GPU thread would calculate which part of the tree and sharing results would be a hassle.

Instead, the chaos game is more frequently used because of its simplicity. It is paralellized in a straightforward way, by running the iteration process many times side-by-side (one per GPU thread), and then combine the final results of all of these on a single canvas. [Gre05]

3.4.2 The deterministic method on the GPU

An exciting approach taken in [Law12] uses the deterministic method instead: by using the fast inverse square root operation, even unbounded (noncontracting) and nonlinear IFSs can be efficiently evaluated using the deterministic method, programmed in normal GPU shaders that manipulate a couple of GPU textures.

4 Research Question

In the last section, the construction of an IFS's attractor was formally defined, and different approaches of rendering it were outlined.

While many different approaches to IFS rendering exist, some of them quite efficient, none of them re-uses information from rendering one image of the IFS for rendering another.

This leads us to the research question of this thesis:

Is it possible, by re-using information between animation frames, to render animations of an Iterated Function System's attractor in which the camera zooms in, in real-time?

5 Approach

To put this to the test, a simple software program was created which calculates the IFS' attractor only once, and then allows a user to interactively zoom and pan the camera around to investigate different parts of the attractor.

5.1 Design

The inspiration of the design is two-fold:

First, we use the insight that the (parallel) chaos game can be used to generate a *point cloud*, allowing us to re-use parts of the computation between animation frames and thus render each frames faster.

Second, while a point cloud only allows *zooming in* up to a particular depth before losing considerable detail, it is possible to detect when we are looking at a self-similar part of the attractor. This allows us, in many situations, to replace the current camera viewport with a more *shallow* one, keeping the amount of detail high.

5.1.1 Point clouds

The main inspiration for the re-usability approach is that we can modify the GPU-variant of the chaos game algorithm outlined in §§§ 3.4.1 to render to a *point cloud* instead of immediately to a canvas. When we then move around the camera, we are able to re-use the points in the point cloud; only where the points in the point cloud end up on screen exactly needs to be re-calculated, by transforming all of the points exactly once with the 'view transformation' (and culling all points outside of the viewport).

This is faster than re-evaluating the whole attractor using the chaos game at every frame which would require transforming all points *many* times.

Formally, to render an attractor approximation consisting of N points, running the whole chaos game each frame takes $2(N + x)$ transformations per frame. [5] Running this on p parallel threads has a time complexity of $\mathcal{O}(\frac{2(N+px)}{p})$.

Unoptimized, it takes N transformations to render a precomputed point cloud to screen each frame (in parallel this corresponds to a time complexity of $\mathcal{O}(\frac{N}{p})$). This does not seem very impressive since $\mathcal{O}(\frac{2(N+x)}{p}) \approx \mathcal{O}(\frac{2N}{p}) \approx \mathcal{O}(\frac{N}{p})$, placing the two approaches in the same order of efficiency. However, it is possible to optimize point cloud-based rendering using the techniques outlined in the next section to run in $\mathcal{O}(\frac{\log N}{p})$ instead, which is a big improvement.

5.1.2 Potential point cloud-based optimizations

The generation and rendering of point clouds is a quite well-understood problem [WS06]. point clouds see widespread use, most commonly in 3D-graphics that originates from a '3D scanner' .

point clouds can be rendered in a reasonably efficient manner by storing them in a 'Bounding Volume Hierarchy', for instance in a binary search tree that is ordered using the Morton space filling curve. [Lau+09] Storing the points of a point cloud in this way allows us to efficiently cull most uninteresting points (i.e. points that would end up outside of the current camera viewport), which speeds up the rendering procedure tremendously.

However, while this problem is well-understood, the implementation is far from trivial [Lau+09].

5.1.3 Self-similarity jumping: 'zooming in by zooming out'

When using a point-cloud, we retain detail when zooming in up to a certain depth. In this way, a point cloud is more flexible than a static pixel canvas, which will already show rendering artefacts when zooming in slightly beyond its intended size.

Nonetheless, beyond a certain depth, the number of points of the point cloud that fall outside of the current camera viewport (and thus are 'useless' for the quality of the rendered attractor) grows larger and larger.

However, it follows from the self-similar nature of the IFS that we might, in certain situations, 'secretly' zoom out to a shallower camera viewport of the point cloud that shows the same information of the attractor as the original viewport, but containing more points of the point cloud.

[5] Using the definitions of §§§ 3.3.2: N is the total number of points we want to render in our attractor approximation, and x is the minimum number of transformations we need to apply to any arbitrary point to make it visually indistinguishable from a point exactly on the attractor.

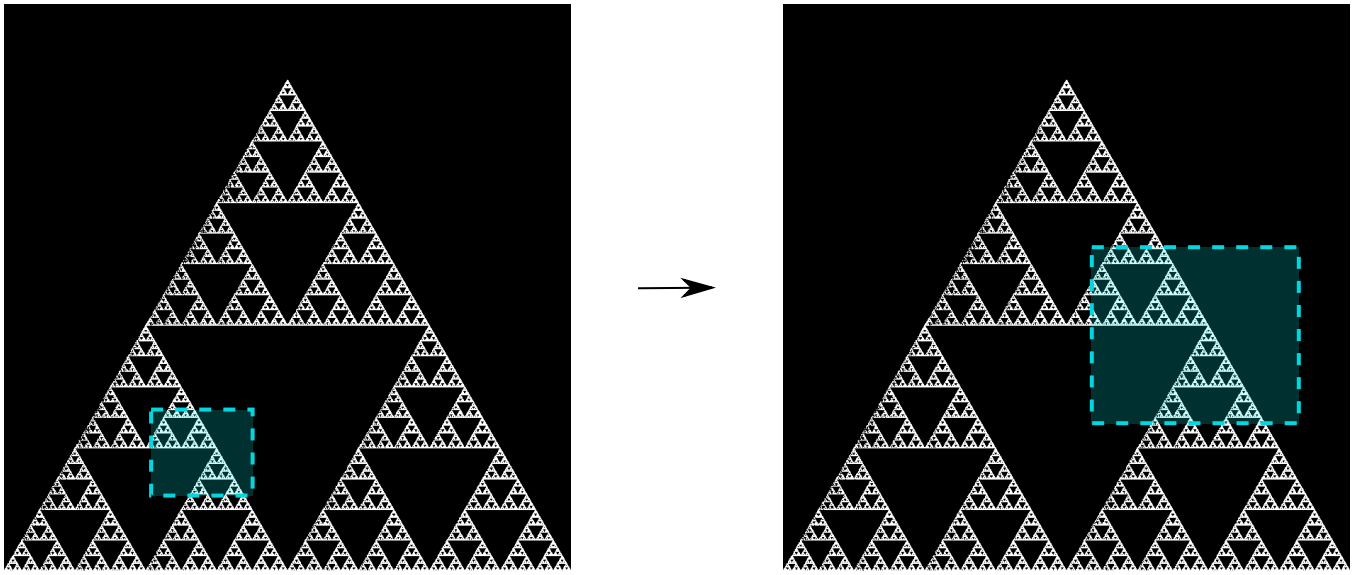


Figure 3: *An example of the self-similarity jumping technique. Pictured is the Sierpińsky triangle IFS (autoref:ifs:sierpinsky). The viewport (pictured in cyan) on the left can be transformed to the one on the right by applying the inverse mapping f_1^{-1} to it. The resulting viewport looks the same but contains more points.*

This can be done by identifying a mapping that fully encompasses the current camera viewport, and then applying its inverse to the viewport. 'Fully encompasses' here means that the unit square transformed by the mapping fully encompasses the unit square transformed by the inverse of the camera viewport transformation [6].

See Figure 3 for an example.

[6] A simple way to do this is to treat the unit square as a simple polygon, and then transform all of its corner points. For the resulting two polygons, the 'even-odd rule' algorithm [Hai94] can be used to check whether all points of one polygon are inside the other.

The algorithm is specified in **Algorithm** 2 and **Algorithm** 3.

Algorithm 2: self-similarity jump-up

n: the number of transformations the IFS consists of.
v: the current camera's viewport transformation.
s: a stack of jumps made until now.
for $i \leftarrow [0..n)$ **do**
    if is_invertible(f_i) and inside(v^{-1}, f_i) **then**
        push(s, f_i)
        $v = f_i^{-1}(v)$
        break
    end
end

Algorithm 3: self-similarity jump-down

u: the identity transformation
v: the current camera's viewport transformation.
s: a stack of jumps made until now.
if not_empty(s) and outside(v^{-1}, u) **then**
    f_i = pop(s)
    $v = f_i(v)$
end

5.1.4 Coloring the rendering

The simplest way of rendering an IFS attractor simply renders points that are on the attractor a different colour than the points that are not.

However, more visually pleasing methods use a *color map* to e.g. indicate the density (the number of points ending up at a particular canvas location) of the attractor. Yet more advanced methods [DR03] keep track of a per-point colour, based on the sequence of transformations it has undergone.

It seems possible to combine these techinques with the 'self-similarity jumping', since we keep track of which mappings we've (inversely) applied to the camera viewport: to determine the final colors of all points, all visible points need to be multiplied all points's colors need to be altered by the color-mutations that each of the inversely-applied mappings would apply. Essentially, say we are viewing the lower left leaf of a fractal fern, whose mapping would make the contained points reddish. If we now 'jump up' (so we use points from virtually the whole fern), we have to alter all points so they get the same reddish hues.

5.2 Implementation

The program was implemented using the general-purpose programming language Haskell, in combination with the GPGPU library Accelerate [Cha+11]. This programming stack was chosen because Accelerate offers a statically-typed EDSL[7] to array-based GPGPU programming, which is more high-level and less error-prone than writing shader code in e.g. CUDA or OpenCL directly.[8]

[7] Embedded Domain-Specific Language.

[8] Instead of being presented with a black screen when a programming mistake is made, Accelerate presents errors at compile-time in many cases. Furthermore, Accelerate features a single-threaded reference implementation that runs on the CPU that

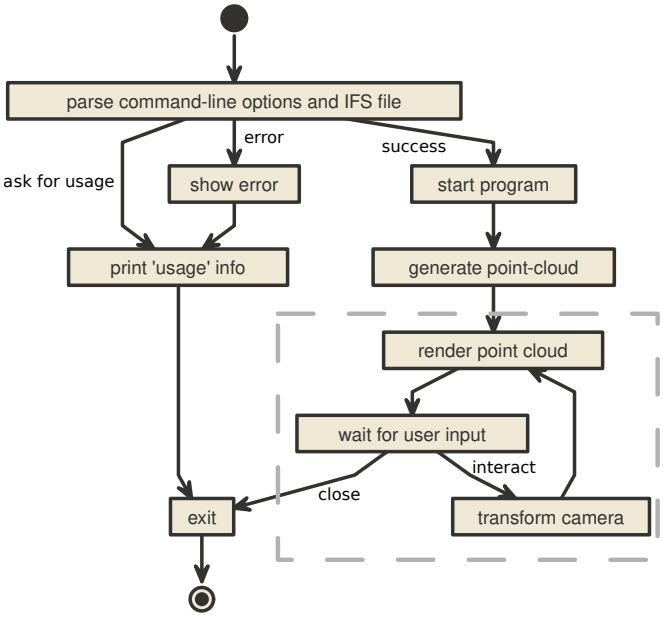


Figure 4: *Overview of the proof-of-concept program's execution flow.*

The usage of Haskell as implementation language also allowed the easy construction of different sub-components making up the program, and testing each of these independently, being a pure functional language.

A general overviwe of the flow of the program can be seen in Figure 4.

5.2.1 Simplicity

To be able to complete the implementation within the time allotted for the thesis project, the decision was made to keep the implementation as simple as possible.

This means that

- the optimizations mentioned in §§§ 5.1.2 were not implemented.
- points are rendered on screen using a simple binary mapping (if a pixel contains one or more points, it is white; otherwise black.) the more fancy coloring techniques outlined in §§§ 5.1.4 were not used.

This means that while the program cannot on its own fully answer the question of whether this technique is fast enough for real-time usage in all circumstances, it is able to answer the simpler question of whether the technique of using a point cloud in combination with 'self-similarity jumping' is at all feasible.

5.2.2 Command-line options

The proof-of-concept program allows the customization of the following options

can be used to sanity-check the behaviour of code.

```
{ initialCamera =
  { a = 9.090909090909091e-2
  , b = 0.0
  , c = 0.0
  , d = -9.090909090909091e-2
  , e = 0.5
  , f = 1.0
  }
, transformations =
  [ { transformation = { a = 0.0, b = 0.0, c = 0.0, d = 0.16, e = 0.0, f = 0.0 }
    , probability = 1.0e-2
    }
  , { transformation = { a = 0.85, b = 4.0e-2, c = -4.0e-2, d = 0.85, e = 0.0, f = 1.6 }
    , probability = 0.85
    }
  , { transformation = { a = 0.2, b = -0.26, c = 0.23, d = 0.22, e = 0.0, f = 1.6 }
    , probability = 7.0e-2
    }
  , { transformation = { a = -0.15, b = 0.28, c = 0.26, d = 0.24, e = 0.0, f = 0.44 }
    , probability = 7.0e-2
    }
  ]
}
```

Listing 1: *barnsley_fern.ifs, representing IFS A.3*

- 'samples': the number of points to use for the chaos game (defaults to 100,000,000)
- 'paralellism': the number of GPU-threads to split the number of samples across. (defaults to 2048)
- 'seed': a number to seed the random number generator with. If not provided, a different arbitrary seed will be used each time.
- 'render_width' and 'render_height' set the resolution of the program window that is displayed (defaults to 800×800).

5.2.3 '.ifs' file format

The configuration language 'Dhall' [Gc19] was used to easily faciltate the specification of different IFSs.

The file structure allows one to indicate a list of affine transformations with associated chaos game probabilities, as well as an 'initial camera viewport transformation'. §§§ 3.2.2

Dhall allows the definition and re-use of variables, which can be useful for numerical constants that are used in multiple transformations.[9]

An example file can be seen in Listing 1 .

5.2.4 Rendering

The program computes the point cloud once, on startup, and then re-renders the image that is shown in the program window every time the user changes the camera viewport.

[9]Unfortunately, Dhall explicitly does not allow floating-point arithmetic. As such, one still needs to write e.g. 1/3 as 0.3333333333333333.

Rendering is done by iterating (in parallel) over all points in the point cloud and filling a two-dimensional histogram with the same dimensions as the canvas with numbers. This histogram is then used to draw the attractor (any non-empty pixel is colored white and the rest black).

More sophisticated rendering techniques are possible (see §§§ 5.1.4), but not implemented in the program.

5.2.5 Changing the camera viewport

Changing the camera viewport can be done by either zooming in or out using the scrollwheel, or dragging with the mouse to translate the camera viewport.

The camera's view transformation itself is stored as a transformation matrix relative to the unit square.

5.2.6 Manually performing 'self-similarity jumping'

While the program is running, a user can go back to a more shallow view by pressing '+'[10], and then when inside one or multiple shallower views, '-' can be pressed to undo the last jump.

Care is taken to only allow the jump up if the current camera viewport is fully contained within one mapping's region.

That this process is kept manual was intentional, because it allows the user to more easily compare how the representation looks with and without the jumping, and allows for a full exploration of the circumstances in which a jump up is and is not actually correct (see §§ 6.1).

5.2.7 Rendering 'guides'

To make it easier to see how an IFS is constructed, as well as easier for a user to orient themselves when testing the 'self-similarity jumping', it is possible to toggle the rendering of 'guides' by pressing the 'g' key. Similarly, the rendering of points can be toggled by pressing the 'p' key (allowing one to see the guides more clearly, when desired).

These 'guides' are the unit squre, after undergoing a sequence of zero, one, two etc. mappings of the IFS. Different colours are used for guides at different sequence-depths.

6 Findings

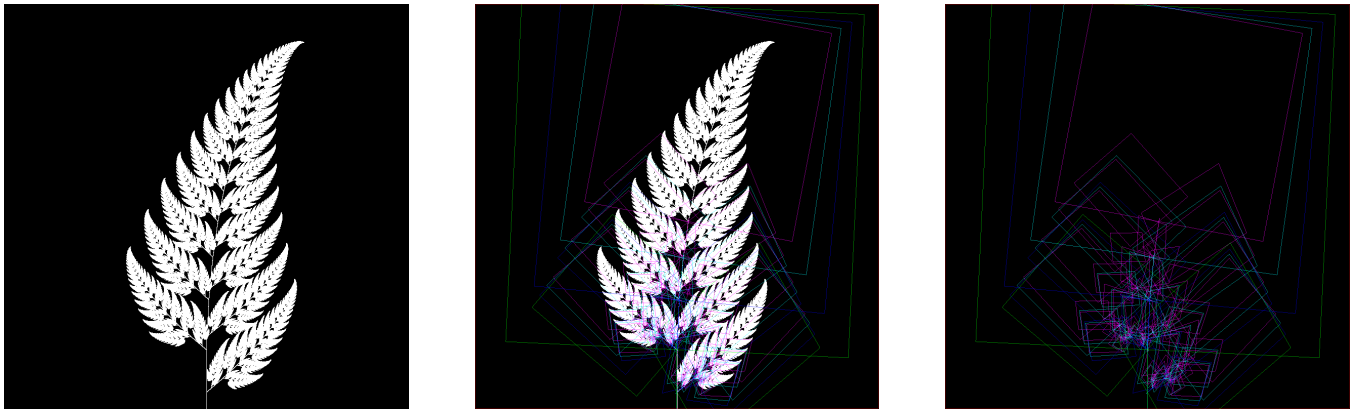
6.1 Restrictions on 'self-similarity jumping'

From experimentation with the program it turns out that there are two common situations in which the technique of replacing the camera viewport with a more shallow camera viewport that is outlined in §§§ 5.1.3 cannot be used.

1. Borders between transformations

   It is rather common to zoom in on the borders between transformations, as this is often where interesting visual details of the IFS might appear.

[10] Strictly speaking, by pressing the '=' key; pressing SHIFT is not necessary.



(a) *only points* ***(b)*** *guides and points* ***(c)*** *only guides*

Figure 5: *The Barnsley Fern (IFS A.3) rendered by the program in different ways.*

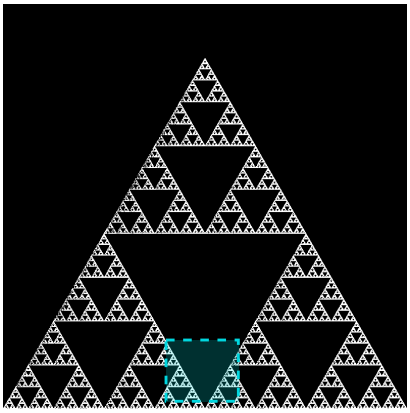


Figure 6: *In this example the camera viewport (indicated in cyan) overlaps both f_1 and f_2 of IFS A.1 partially. This case is not handled by* ***Algorithm*** *2.*

However, **Algorithm** 2 is not able to handle borders between transformations, thus making it useless in these scenarios.

An example can be seen in figure Figure 6

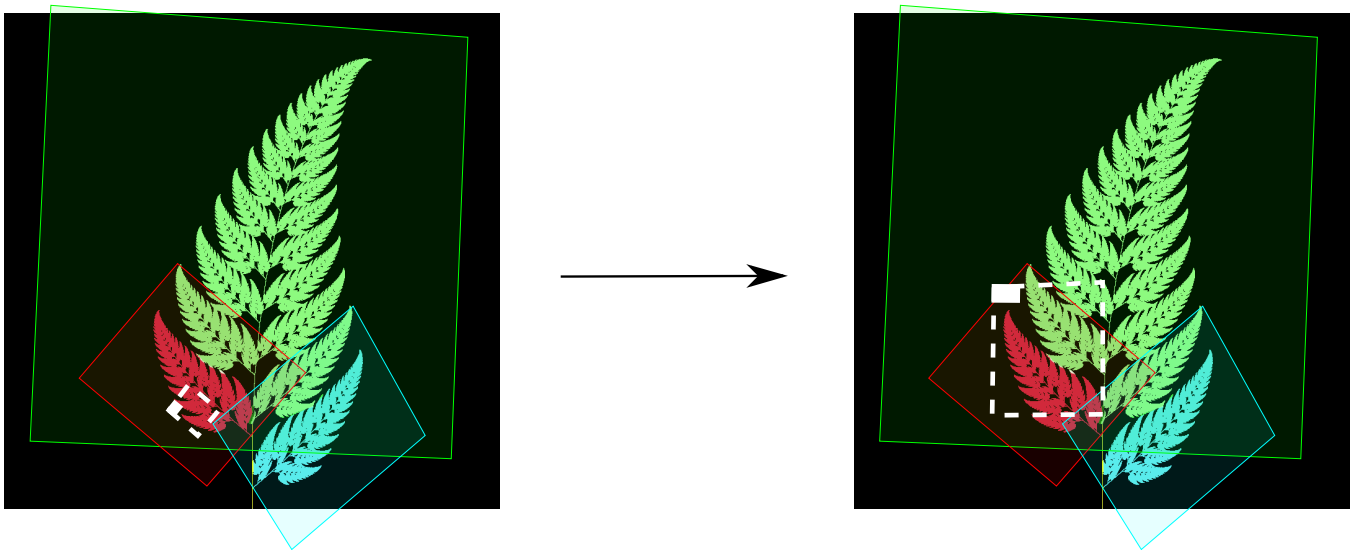
2. Overlapping subtransformations

   A more shallow view of the attractor only shows the same as a deeper view when there are no points transformed by another mapping that end up in the deeper view.

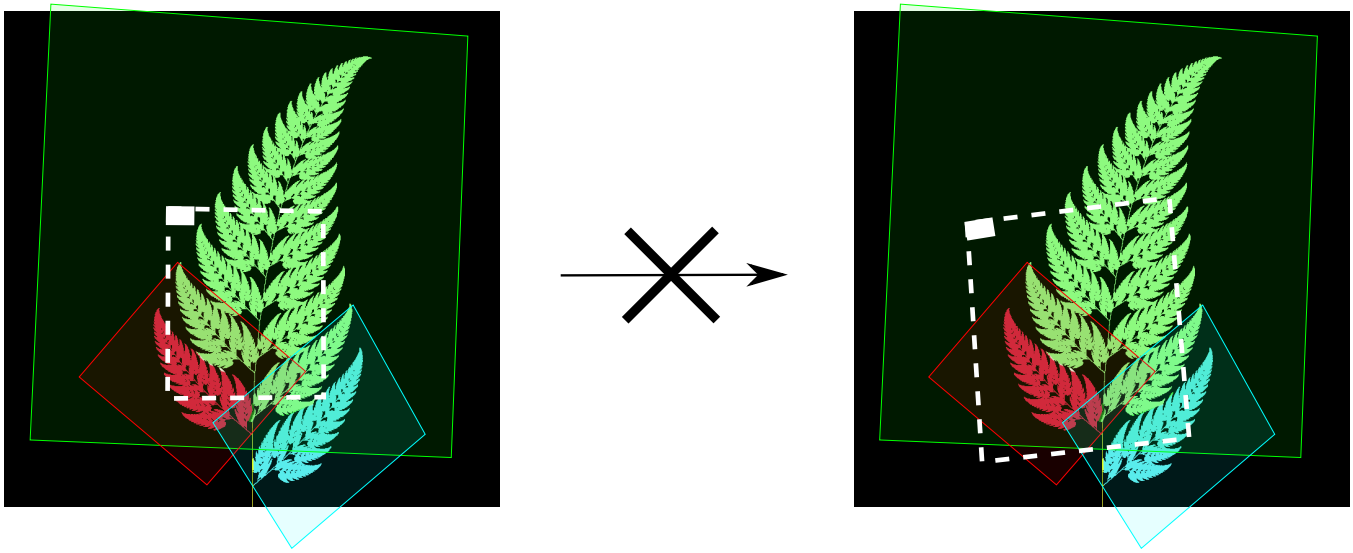
   When there are points from another mapping in the current view, going to a more shallow view will make points 'disappear' from the perspective of the user. In practice, this means that for many IFSs there are large regions in which the technique cannot be used at all.

   Simple IFSs like the Sierpi\'nsky Triangle (IFS A.1), in which transformations do not overlap, do not exhibit this problem. Slightly more complicated IFSs like the Dragon Curve[REF] or the Barnsly Fern[REF] however do. See Figure 8 for an graphical explanation.

   This case is annoyingly common and there is no clear solution to alleviate this restriction. What is more, it is not simple to check whether we are currently in a region that exhibits the problem, as this would require evaluating the IFS itself.



(a) *Since the camera viewport only sees points of f_2, the jump up is proper.*



(b) *Since the camera viewport sees both points of f_2 and f_3, the jump is incorrect. Note the leaf in the lower left missing after the jump.*

Figure 7: *Problems when jumping up on IFS A.3. Camera viewport indicated as white dashed polygon. Points colored based on their latest mapping.*

It is possible to take a rough 'upper bound' estimate of the disallowed regions by keeping track, per mapping, where the unit square would end up after a couple of mappings with this mapping as last (i.e. most significant) one.

6.2 Memory Usage

Point clouds take up a lot of data on the GPU. To render a fractal at reasonable detail, hundreds of millions if not billions of points are necessary (depending on the particular IFS).

A reasonable way to store a point cloud is by using for each 2D-point, 32 bits for each coordinate, thus fitting the pair in exactly one machine word of 64-bit systems. Stored this way, a point cloud of 100,000,000 points requires 0.596 GiB of GPU memory, and 1,000,000,000 points requires 5.96 GiB. For current generation GPUs[11], this often is more memory than available.

7 Conclusion

A program was implemented which has shown that there is *some* merit to rendering an IFS' attractor using a point-cloud as re-usable intermediate structure. However, the self-similarity detection method that was proposed turns out to be unusable in common cases. Furthermore, self-similarity 'jumps' make more sophisticated rendering techniques difficult if not impossible to use.

As long as these two problems remain unsolved, the proposed technique can only be considered impractical.

8 Further Work

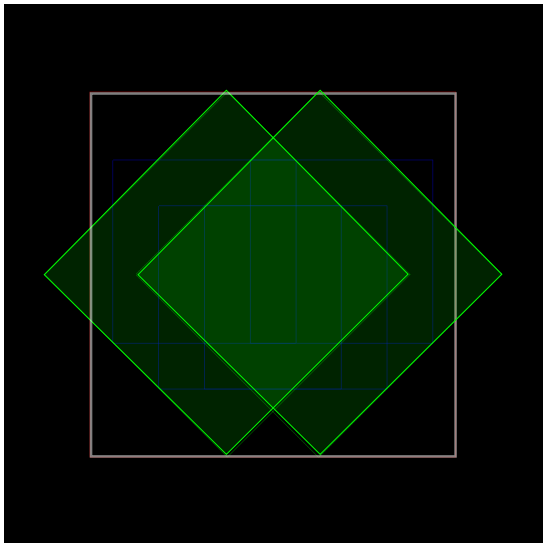
It is our hope that a more sophisticated way of detecting self-similarity might be found, which would make 'self-similarity jumping' more practical.

Besides this, while we have shown in a proof-of-concept program that it is possible to render an IFS using a point cloud with a reasonable speed, there are many optimizations that could be made to make the program run faster (potentially even in real-time), most notably the rendering optimizations listed in §§§ 5.1.2

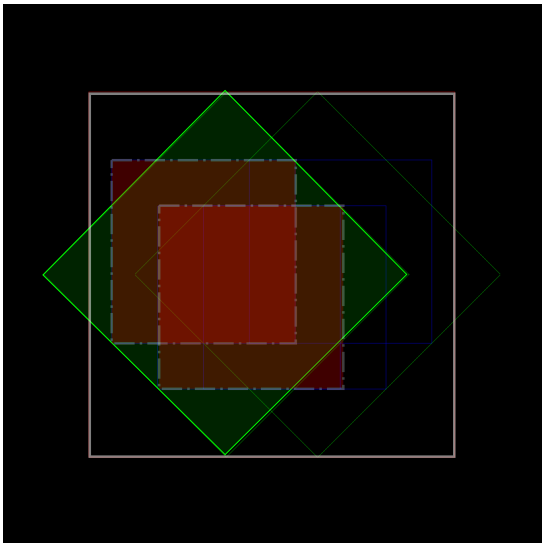
Another venue that could be explored is the rendering of an IFS' attractor at multiple 'levels of detail': It might be possible to create more detailed local versions of the point cloud (based on the points of the less detailed point cloud) when the user zooms in on a particular region, on demand.

Finally it is worth noting that as mentioned in §§§ 3.4.2, [Law12] already presents an efficient way to render a large set of IFSs using a very different approach, which might be worthwhile to explore further.

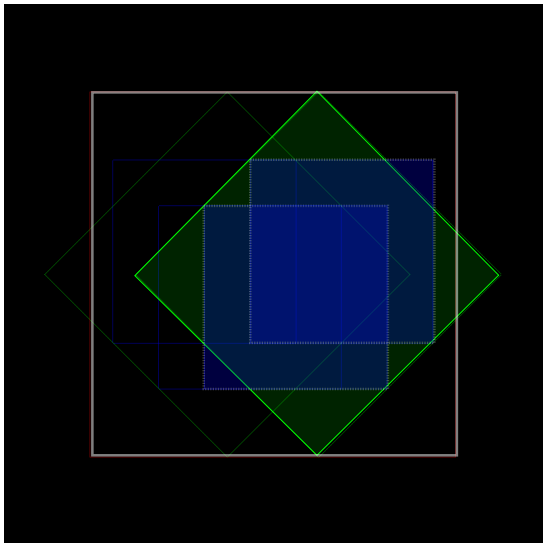
[11] At the time of writing, high-end consumer GPUs contain somewhere between 2 and 24 GiB of available memory. [LK20]



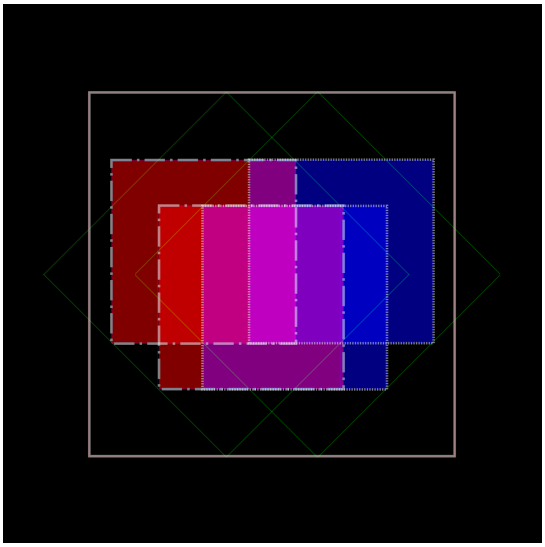
(a) *The unit square (indicated in white) transformed by the two mappings of D (indicated in green)*



(b) *The first of the two mappings, transformed one more time by the mappings of D (indicated in red)*



(c) *The second of the two mappings, transformed one more time by the mappings of D (indicated in blue)*



(d) *The region in which (b) and (c) overlap (indicated in magenta)*

Figure 8: *Showing the first couple of iterations of rendering the attractor of the dragon curve IFS D (IFS A.2), and the regions in which (sequences of) transformations overlap.*

References

[Bar88] Michael F Barnsley. *Fractals everywhere*. Academic press, 1988 (cit. on pp. 2, 4–6).

[Cha+11] Manuel MT Chakravarty et al. "Accelerating Haskell array codes with multicore GPUs". In: *Proceedings of the sixth workshop on Declarative aspects of multicore programming*. 2011, pp. 3–14 (cit. on p. 11).

[DR03] Scott Draves and Erik Reckase. "The fractal flame algorithm". In: *Forthcoming, available from http: I Itlam3. com/flame. pdf* (2003) (cit. on pp. 2, 11).

[Gc19] Gabriel Gonzalez and contributors. *Dhall Configuration Language*. https://github.com/dhall-lang/dhall-lang. 2019 (cit. on p. 13).

[Gre05] Simon G Green. "GPU-accelerated iterated function systems". In: *ACM SIGGRAPH 2005 Sketches*. 2005, 15–es (cit. on p. 8).

[Hai94] Eric Haines. "Point in Polygon Strategies." In: *Graphics Gems* 4 (1994), pp. 24–46 (cit. on p. 10).

[Har96] John C Hart. "Fractal image compression and recurrent iterated function systems". In: *IEEE Computer Graphics and Applications* 16.4 (1996), pp. 25–33 (cit. on p. 2).

[HPS91] Daryl Hepting, Przemyslaw Prusinkiewicz, and Dietmar Saupe. "Rendering methods for iterated function systems". In: North-Holland, 1991 (cit. on pp. 2, 5–7).

[Jef90] H Joel Jeffrey. "Chaos game representation of gene structure". In: *Nucleic acids research* 18.8 (1990), pp. 2163–2170 (cit. on p. 2).

[Lau+09] C Lauterbach et al. "fast BVH construction on GPUs". In: *Proceedings of the Eurographics Symposium on Rendering, Eurographics and ACM/SIGGRAPH*. 2009 (cit. on p. 9).

[Law12] Orion Sky Lawlor. "GPU-accelerated rendering of unbounded nonlinear iterated function system fixed points". In: *ISRN Computer Graphics* 2012 (2012) (cit. on pp. 8, 17).

[LK20] Kevin Lee and Mark Knap. *Best Graphics Cards 2020: Top GPUs for Every Budget*. https://www.ign.com/articles/the-best-graphics-cards-3. Aug. 2020 (cit. on p. 17).

[Men03] Franklin Mendivil. "Fractals, graphs, and fields". In: *The American mathematical monthly* 110.6 (2003), pp. 503–515 (cit. on p. 4).

[WS06] Michael Wimmer and Claus Scheiblauer. "Instant Points: Fast Rendering of Unprocessed Point Clouds." In: *SPBG*. Citeseer. 2006, pp. 129–136 (cit. on p. 9).

A IFSs used

This appendix lists the mapping functions of the IFSs that were used throughout this thesis.

$$
\begin{aligned}
f_1(x,y) &= \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \quad \begin{bmatrix} x \\ y \end{bmatrix}, && p_1 = \frac{1}{3} \\
f_2(x,y) &= \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \quad \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \frac{1}{2} \\ 0 \end{bmatrix}, && p_2 = \frac{1}{3} \\
f_3(x,y) &= \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \quad \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \frac{1}{4} \\ \frac{\sqrt{3}}{4} \end{bmatrix}, && p_3 = \frac{1}{3}
\end{aligned}
$$

IFS A.1: *the Sierpińsky triangle*

$$
\begin{aligned}
f_1(x,y) &= \frac{1}{\sqrt{2}} \begin{bmatrix} \cos 45^\circ & -\sin 45^\circ \\ \sin 45^\circ & \cos 45^\circ \end{bmatrix} \quad \begin{bmatrix} x \\ y \end{bmatrix}, && p_1 = \frac{1}{2} \\
f_2(x,y) &= \frac{1}{\sqrt{2}} \begin{bmatrix} \cos 135^\circ & -\sin 135^\circ \\ \sin 135^\circ & \cos 135^\circ \end{bmatrix} \quad \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}, && p_2 = \frac{1}{2}
\end{aligned}
$$

IFS A.2: *the Heighway Dragon Curve*

$$
\begin{aligned}
f_1(x,y) &= \begin{bmatrix} 0.00 & 0.00 \\ 0.00 & 0.16 \end{bmatrix} \quad \begin{bmatrix} x \\ y \end{bmatrix}, && p_1 = 0.01 \\
f_2(x,y) &= \begin{bmatrix} 0.85 & 0.04 \\ -0.04 & 0.85 \end{bmatrix} \quad \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.00 \\ 1.60 \end{bmatrix}, && p_2 = 0.85 \\
f_3(x,y) &= \begin{bmatrix} 0.20 & -0.26 \\ 0.23 & 0.22 \end{bmatrix} \quad \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.00 \\ 1.60 \end{bmatrix}, && p_3 = 0.07 \\
f_4(x,y) &= \begin{bmatrix} -0.15 & 0.28 \\ 0.26 & 0.24 \end{bmatrix} \quad \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.00 \\ 0.44 \end{bmatrix}, && p_4 = 0.07 \\
camera(x,y) &= \begin{bmatrix} 0.09 & 0.00 \\ 0.00 & -0.09 \end{bmatrix} \quad \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.50 \\ 1.00 \end{bmatrix}
\end{aligned}
$$

IFS A.3: *the Barnsley Fern*