university of groningen

faculty of science and engineering

In by Out Again

Arbitrarily-Deep Zooming on Iterated Function Systems by 'Self-Similarity Jumping'

Bachelor's thesis

August 24, 2020

Student: Wiebe-Marten Wijnja

Primary supervisor: dr. J. Kosinka

Secondary supervisor: G. J. Hettinga

Contents

Todo list

Turn off todo list

Abstract

Consider improving formulation of IFSs

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's attractor at every frame.

This thesis proposes a combination of two techniques to enable faster exploring of IFSs. First, a point cloud is used as an intermediate attractor approximation, that can be re-used between animation frames. Secondly, a technique coined 'self-similarity jumping' is proposed 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.

1 Introduction

Consider improving formulation of IFSs

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 [DR04].

Various computer algorithms to visualize IFSs exist [HPS91]. However, these all take a single still image as final result. If these algorithms are employed to render an animation, this animation 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 this information 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 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 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 camera viewport.

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

1.1 Overview

In the next section, § 2, Iterated Function Systems are 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 § 3. The approach taken to test this question is described in § 4, followed by a qualitative discussion of the results in § 5. We conclude in § 6 and finally hint at some approaches for further work in § 7.

2 Background

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

2.1 Informal description of an Iterated Function System

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

1. transforming the input image with each of the transformations;

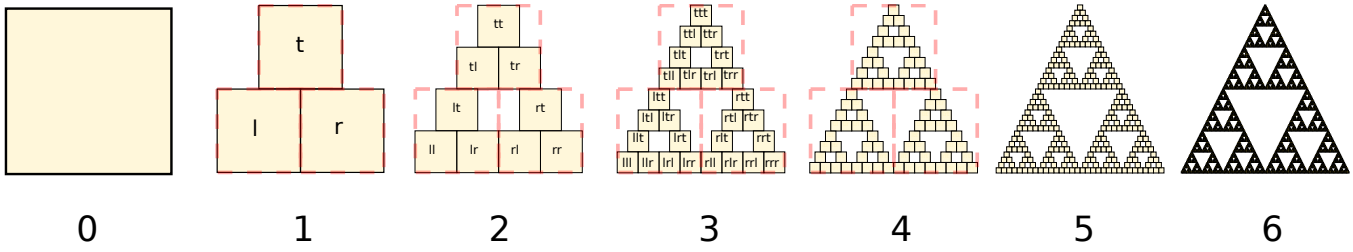


Figure 1: *The first six iterations of the Sierpińsky triangle IFS (IFS A.1). The initial image is just the unit square. We then iteratively combine the results of transforming the current image using one of the three mappings. The letters indicate which (sequence of) transformation(s) resulted in this 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. Already 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.*

2. overlaying all transformed images, forming one new image that is the combination (union) of the transformed images. 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.

2.2 Formal definition of an Iterated Function System

Formally, an Iterated Function System F consists of a finite set of contraction mappings that map a complete metric space[1] $(\mathcal{M}, d)$ to itself [Bar88]:

$$\mathcal{F} = \{f_i : \mathcal{M} \to \mathcal{M}\}_{i=1}^{N}, N \in \mathbb{N}.$$

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:

$$\forall i \big(d(f_i(a), f_i(b)) < d(a, b)\big).$$

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* and denoted H. It can be iterated as many times as desired:

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

[1] A metric space is a set $\mathcal{M}$ together with a *metric* $d(x, y)$ on that set . The metric is a function that for any two elements (or 'points') in $\mathcal{M}$ returns the 'distance' between them, for any notion of distance adhering to the 'identity of indiscernibles' $(d(x, y) = 0 \Leftrightarrow x = y)$, 'symmetry' $(d(x, y) = d(y, x))$ and 'triangle inequality' $(d(x, z) \leq d(x, y) + d(y, z))$ properties. Often, d is elided and just $\mathcal{M}$ is used to refer to the metric space when it is clear from context which metric is used.

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

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

Curiously, which set of points $\mathcal{S}_0$ we picked does not influence the shape of $\mathcal{A}$ [Men03]. We might even start with a single point (denoted z_0).

2.2.1 Restriction to affine transformations on the two-dimensional Euclidean plane

Improve info about restrictions (c.f. feedback)

Most research of IFSs restricts itself to using $\mathbb{R}^2$ as metric space[2] 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.

2.2.2 The viewport transformation

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

1. Scaling vs zooming

   Because of the presence of a camera, the part of an object that will be visible in the camera viewport may 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 frustum

   For any IFS we can transform its attractor by any invertible map t by adjusting each of the IFS's mappings according to the transform theorem, defined as $f_i' = t \circ f_i \circ t^{-1}$ [Bar88]. Essentially points are transformed from the new (program-desired) space to the old (user-supplied) space, then the mapping is applied, and finally the points are transformed back to the new space. This allows users the freedom to choose any desired mappings together with an 'initial camera transformation' (i.e. the camera's initial position + frustum), while still allowing all calculations to happen with regard to the unit square ('unit space'), keeping them simple.

2.3 Rendering an Iterated Function System

A couple of algoritms ([Bar88], [HPS91], [Law12]) exist to render the attractor of an Iterated Function System. It is impossible to render the attractor exactly, as this would require an infinite number of transformation steps. However, we can approximate it until the difference between our approximation and the attractor is smaller than the smallest detail we can visually represent (e.g. when rendering to a screen, smaller than the size of a pixel).

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

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.

Define trees (in a footnote?)

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 in [HPS91]. Short summaries of the two most common techniques now follow.

2.3.1 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, keeping track of the traversed sequence of transformations [3] $f_{i_k} \circ \ldots \circ f_{i_1}$;
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 P points, depending on the tree traversal chosen:

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

Explain why. Or refer to external resource?

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

Both kinds of traversals take a linear amount of time ($\mathcal{O}(N \cdot P) \approx \mathcal{O}(P)$, where N is the number of mappings the IFS consists of).

While the deterministic method is easy to understand (and indeed is a direct implementation of the informal process described in §§ 2.1), it is usually less efficient and more complex to implement on a computer than the algorithm that is described next.

Double-check pre/postmultiplication problems

2.3.2 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.

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

[3] $\circ$ 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), the matrix premultiplication resulting in the same transformation is in the opposite order ($f \circ g \Leftrightarrow G \cdot F$). Matrix postmultiplication cannot be (easily) used in all of the presented algorithms, c.f. [HPS91].

(a) *1,000,000* ***(b)*** *10,000,000*

Figure 2: *The Barnsley Fern (IFS A.3), rendered using the chaos game with different numbers of points.*

Algorithm 1: the chaos game

N: the number of mappings of the IFS;
z: a single arbitrary starting point;
v: the camera's view transformation;
$m = 0$;
for $m \in [0..n + P)$ **do**
    i: a random integer between 0 and N;
    if $m \geq n$ **then**
        render($v(z)$) cumulatively;
    end
    $z = f_i(z)$;
end

- 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 n 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).[4]
- 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 n iterations are visually indistinguishable from a point that is part of the attractor. By running this non-deterministic approach for sufficiently many iterations we approach a diverse enough set of 'transformation sequences of length n' that we end up covering the whole attractor.

The nice thing about the chaos game is that it requires only a constant amount of auxiliary memory, so its memory complexity is $\mathcal{O}(1)$. Furthermore, its time complexity is similar to the deterministic method but with a smaller constant factor, at $\mathcal{O}(2(P+n)) \approx \mathcal{O}(P+n)$. When $n << P$, which is often the case, this is $\approx \mathcal{O}(P)$.

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 is that in its simplest form, all mappings have an equally likely chance to be used. However, because some mappings might be (much) more contracting than others, this means that coverage of the attractor is not even, which means that we need to use many more iterations.

Therefore, most implementations of the chaos game allow the user to specify for each mapping a *probability* that it is used. When highly contracting mappings are chosen less frequently, coverage of the attractor will be even[5].

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 parallellize for Graphical Processor Units (GPUs), as is outlined in the next subsection.

2.4 Parallellizing IFS rendering by using a Graphical Processor Unit

It is enticing to port IFS rendering to run on a GPU because to produce a smooth image, hundreds of millions of points are often 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 texture (i.e. 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's mappings, but they are not guaranteed to be invertible.

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

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

But what if they are invertible?

Instead, General-Purpose GPU-programming (GPGPU) techniques have to be employed, as these are able to use the top-down approach.

2.4.1 The chaos game on the GPU

The (classical) deterministic method is difficult to parallellize 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 very difficult.

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

2.4.2 The deterministic method on the GPU

An exciting approach taken in [Law12] *does* use the deterministic method instead: by using the fast inverse square root operation together with a few other tricks, 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.

3 Research Question

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

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

4 Approach

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

4.1 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 frame faster.

Second, while zooming in on a point cloud only works 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.

4.1.1 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 §§§ 2.4.1 to store the resulting points in a *point cloud* instead of immediately drawing them on a canvas. When we then move the camera around, 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 camera's 'view transformation'.

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 P points, running the whole chaos game each frame takes $2(P + n)$ transformations per frame (c.f. §§§ 2.3.2). Running this on p parallel threads has a time complexity of $\mathcal{O}\big(\frac{2(P+pn)}{p}\big)$.

Unoptimized, it takes P transformations to render a precomputed point cloud to screen each frame (paralellized this corresponds to a time complexity of $\mathcal{O}\big(\frac{P}{p}\big)$). This does not seem very impressive since $\mathcal{O}\big(\frac{2(P+pn)}{p}\big) \approx \mathcal{O}\big(\frac{2P}{p}\big) \approx \mathcal{O}\big(\frac{P}{p}\big)$, 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 $\mathcal{O}\big(\frac{\log P}{p}\big)$ instead, which is a big improvement.

4.1.2 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 also allows us to efficiently cull most points that would end up outside of the current camera viewport, which speeds up the rendering procedure tremendously.

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

4.1.3 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, while zooming in, more and more points of the point cloud fall outside of the current camera viewport (and thus are 'useless' for the quality of the rendered image), degrading quality beyond a certain depth more than is acceptable.

However, it follows from the self-similar nature of the IFS that we might, in certain situations, 'unnoticeably' zoom out to a shallower camera viewport of the point cloud that shows the same information of

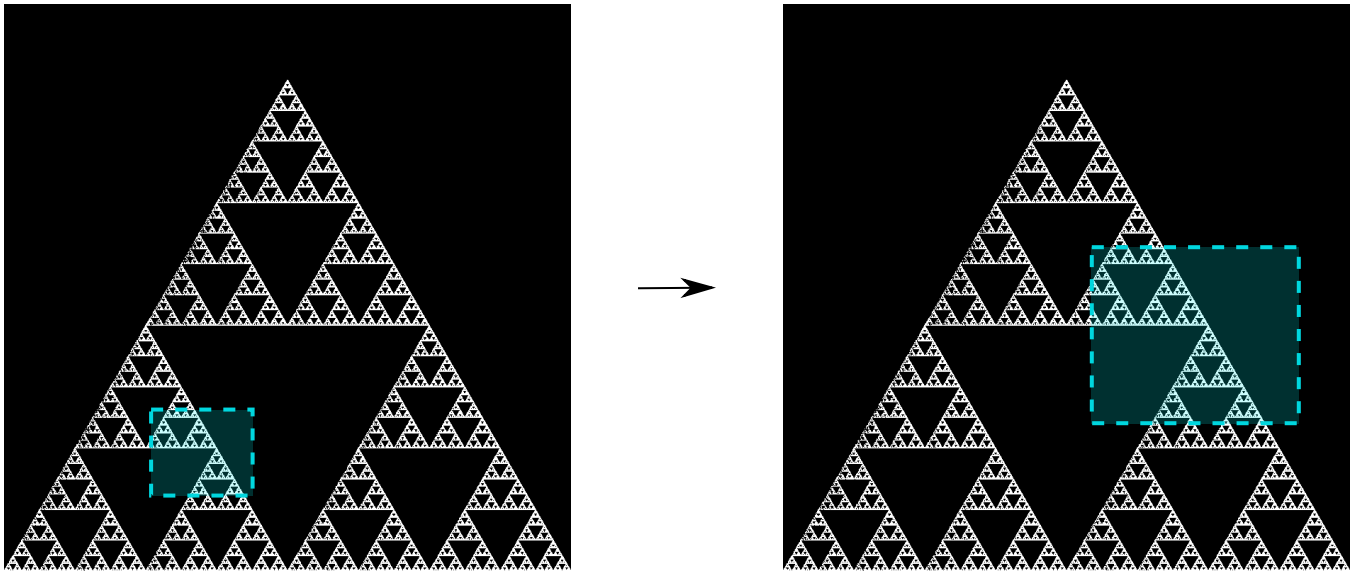


Figure 3: *An example of the self-similarity jumping technique. Pictured is the Sierpińsky triangle IFS (IFS A.1). The viewport (pictured in cyan) on the left can be transformed to the one on the right by applying the inverse mapping f_1^{-1} to it. The resulting viewport looks the same but contains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

the attractor as the original viewport, but containing more points of the point cloud.

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 all corners of the unit square transformed by the inverse of the camera viewport transformation lie inside of the unit square transformed by the mapping [6].

In a similar sense, transformations a and b 'overlap' if the unit square transformed by a overlaps the unit square transformed by b.

See Figure 3 for an example.

The algorithm and its inverse are specified in **Algorithm** 2 and **Algorithm** 3, respectively.

Algorithm 2: self-similarity jump-up

n: the number of mappings the IFS consists of;
v: the current camera's view transformation;
s: a stack of jumps made so far;
for $i \leftarrow [1, \ldots, n]$ **do**
  if isInvertible(f_i) and isInside(v^{-1}, f_i) **then**
    push(s, f_i);
    $v = f_i^{-1} \circ v$;
    break;
  end
end

[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.

Algorithm 3: self-similarity jump-down

v: the current camera's view transformation;
s: a stack of jumps made until now;
if notEmpty(s) and isOutsideUnitSquare(v^{-1}) **then**
    f = pop(s);
    $v = f \circ v$;
end

4.1.4 Coloring the rendering

The simplest way of rendering an IFS attractor simply renders points that are on the attractor a different color 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 [DR04] keep track of a per-point color, based on the sequence of transformations each point has undergone.

It seems possible to combine these techinques with the 'self-similarity jumping', since we keep track of which mappings we have (inversely) applied to the camera viewport: to determine the final colors of all points that will be rendered this frame, all visible points' colors need to be altered by the color-mutations that each of the mappings in the stack s specified by **Algorithm 2** and **Algorithm 3** would apply.

As an example, say we are viewing the lower left leaf of a fractal fern (IFS A.3) and that mapping creating the lower left leaf would make the contained points red. If we now 'jump up' we use points from virtually the whole fern. To make these points still look visually identical from the lower left leaf, we have to alter the points' colors so they get the same reddish hues.

Picture of example fern with fancy coloring?

4.2 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] for array-based GPGPU programming, which is more high-level and less error-prone than writing code in lower-level alternatives like CUDA or OpenCL directly.[8]

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

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

4.2.1 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.

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

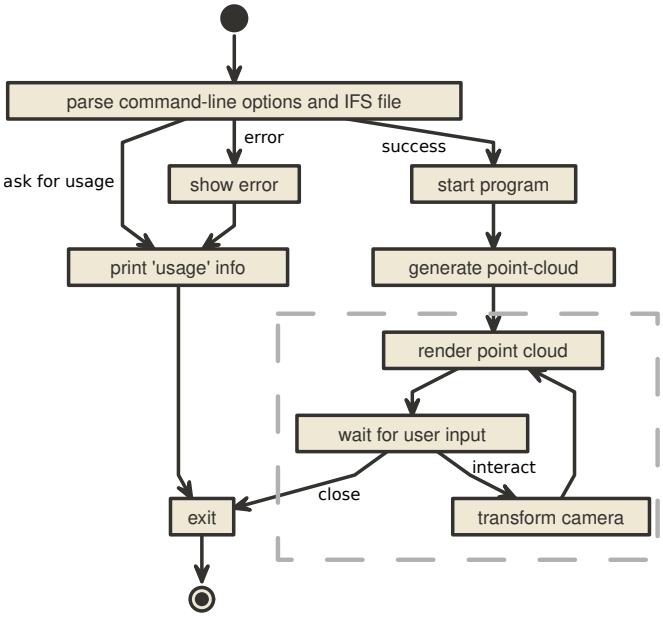


Figure 4: *Overview of the proof-of-concept program's execution flow. The dashed box indicates the main program loop.*

This means that:

- The optimizations mentioned in §§§ 4.1.2 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 §§§ 4.1.4 were not used.

While the program on its own might therefore not be enough to fully answer the research question, it is able to answer the simpler question of whether the technique is at all feasible.

4.2.2 Command-line options

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

- ‘samples‘: the number of points to use for the chaos game (defaults to 100,000,000)
- ‘parallellism‘: 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).

```
{ 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*

4.2.3 The '.ifs' file format

The configuration language 'Dhall' [Gc19] was used to easily allow a user to specify 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 view transformation.

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.

The fields $a \dots f$ used for each of the transformations allow one to specify an affine transformation matrix of the shape

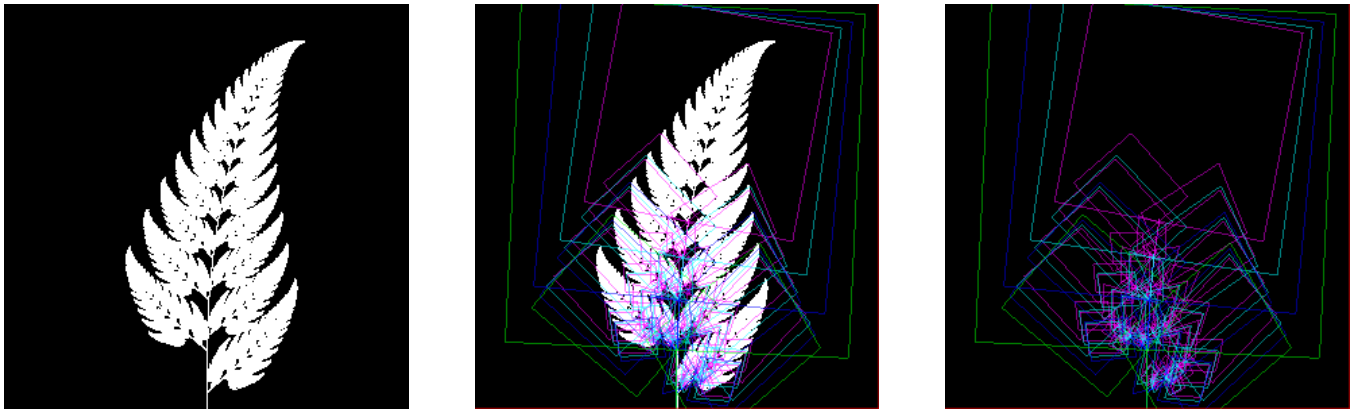
$$\begin{bmatrix} a & b & e \\ c & d & f \\ 0 & 0 & 1 \end{bmatrix}.$$

4.2.4 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 moves the camera.

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).

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



(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.*

4.2.5 Moving the camera

The camera can be moved by either zooming in or out using the scrollwheel, or translating the camera by dragging with the mouse.

These operations alter the camera's current view transformation, which is stored as a transformation matrix relative to unit space.

4.2.6 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.

This process was intentionally kept manual, because it allows the user to more easily compare how the visualization 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 §§ 5.1).

4.2.7 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 square, after undergoing a sequence of zero, one, two etc. mappings of the IFS. Different colors are used for guides at different sequence-depths.

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

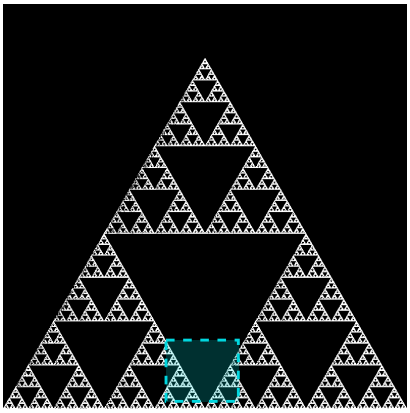


Figure 6: *In this example the camera viewport (indicated in cyan) overlaps (as defined in §§§ 4.1.3) both f_1 and f_2 of IFS A.1 partially. This case is not handled by* ***Algorithm*** *2.*

5 Findings

5.1 Restrictions on 'self-similarity jumping'

From experimentation with the program it turns out that there are two common situations in which the technique outlined in §§§ 4.1.3 cannot be used.

1. Borders between transformations

   Explain what these are

   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.

   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 actually is self-similar to the current view when there are no points transformed by another mapping that end up in the current view.

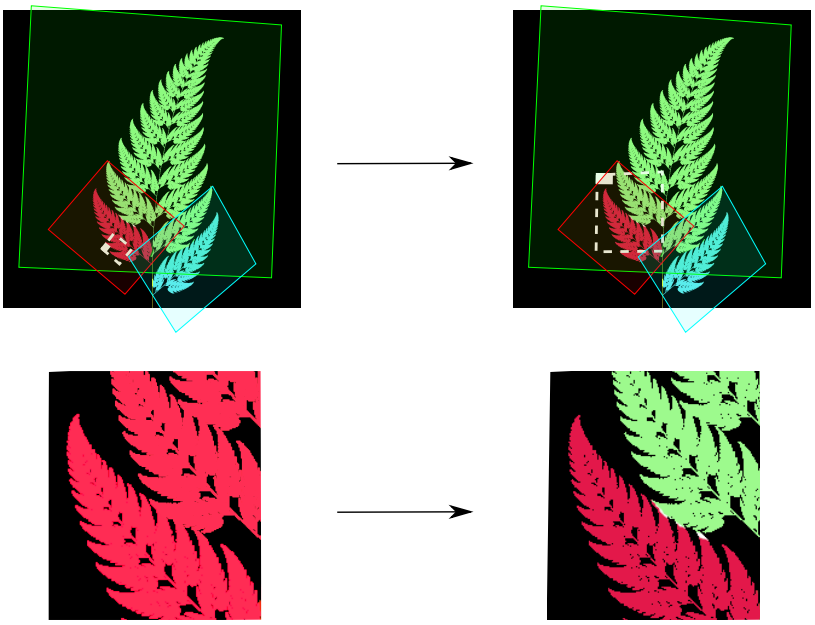
   Better explain paragraph above

   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.

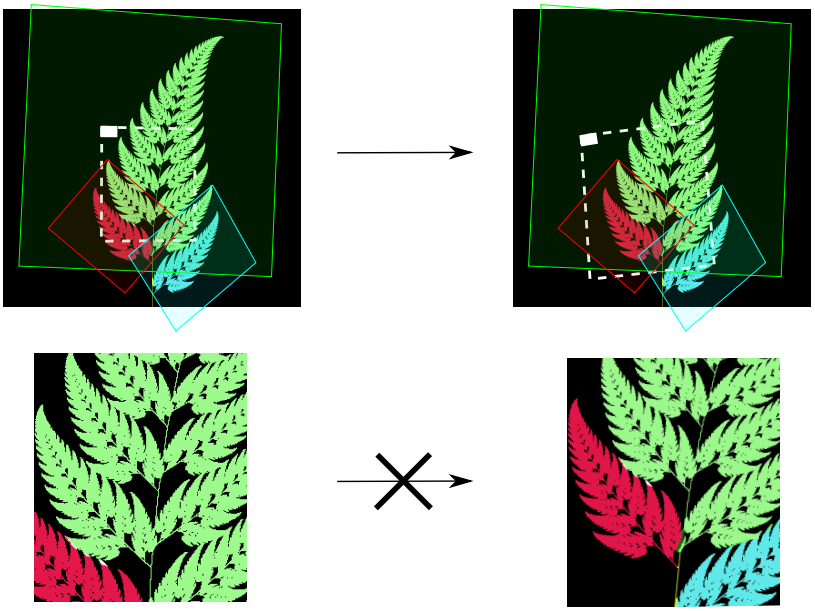
   Simple IFSs like the Sierpińsky Triangle (IFS A.1) in which transformations do not overlap[11], do not exhibit this problem. Slightly more complex IFSs like the Dragon Curve (IFS A.2) or the Barnsley Fern (IFS A.3) however do.

   See Figure 7 for an example and Figure 8 for a graphical explanation of what regions overlap.

[11] As defined in §§§ 4.1.3.



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



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

Figure 7: *Problems when jumping up on IFS A.3. The mappings f_1, f_2 and f_3 refer to the mappings of IFS A.3. The top row of a) and b) is 'world space' with the camera viewport indicated as white dashed polygon. The bottom rows show the camera viewport. Points are colored based on their latest mapping.*

This case is annoyingly common and there is no clear solution to alleviate this restriction. Furthermore, 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.

It is possible to take a rough 'upper bound' estimate of the disallowed regions by keeping track, per mapping f_i, what region would be covered by taking the union of transforming the unit square by all k-long sequences of mappings that start with f_i. This estimate increases in precision as k grows.

where the unit square would end up after a few transformation steps with this mapping as last (i.e. most significant) one.

Increase whitespace between subfigures

5.2 Memory Usage

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

A reasonable way to store a point cloud is by using 32 bits for each of the two coordinates of a point. This means that one point takes up exactly one machine word of a 64-bit computer system. 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[12], this often is more memory than available.

Elaborate claim. 'The latter only'? (see feedback)

6 Conclusion

A program was implemented which has shown that there is *some* merit to rendering an IFS's 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.

Therefore, the proposed technique can be considered of limited practicality, at least until a more sophisticated self-similarity detection method is found.

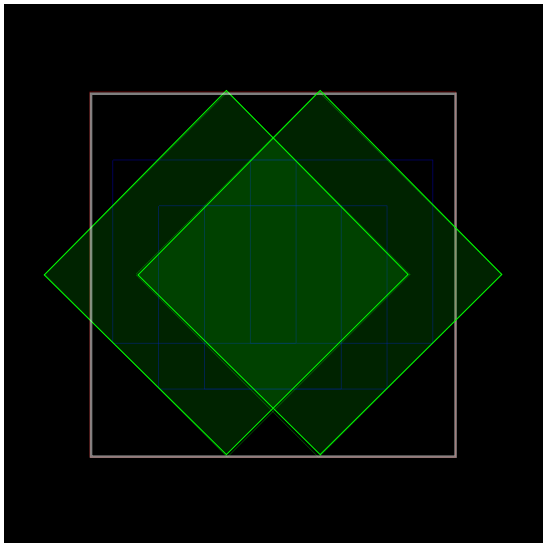
7 Future Work

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

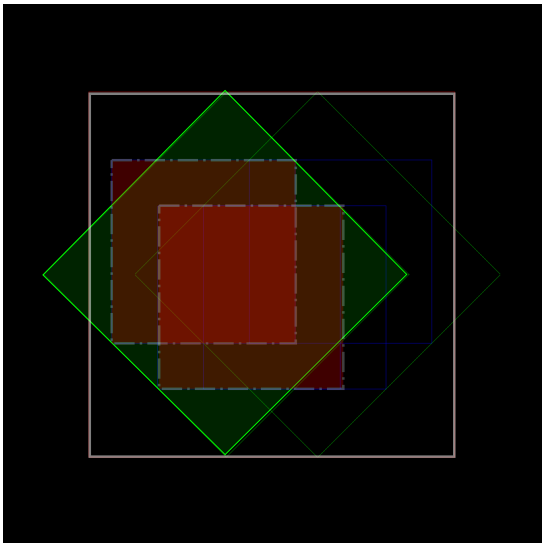
Besides this, while we have shown in a proof-of-concept program that it is possible to render an IFS using a point cloud, there are many optimizations that could be made to make the program run faster (potentially even in real-time), most notably the rendering optimizations listed in §§§ 4.1.2.

Another venue that could be explored is the rendering of an IFS's 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.

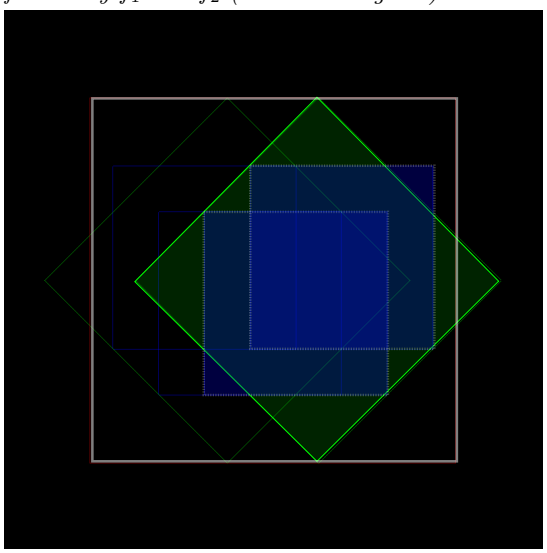
[12] At the time of writing, high-end consumer GPUs contain somewhere between 2 and 24 GiB of available memory [LK20].



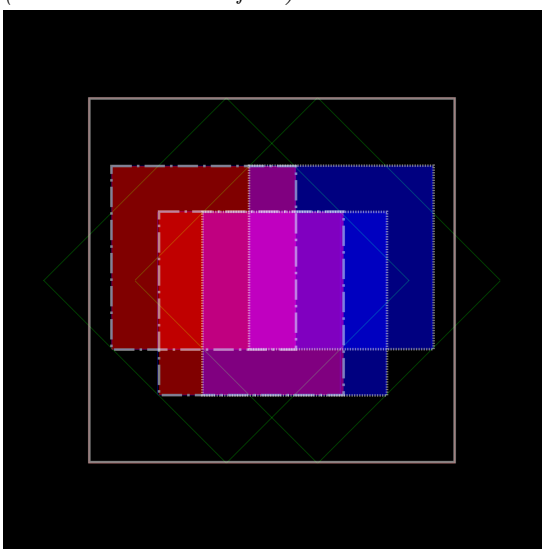
(a) *The unit square (indicated in white) transformed by f_1 and f_2 (indicated in green)*



(b) *Subtransformations of f_1: $f_1 \circ f_1$ and $f_1 \circ f_2$ (indicated in shades of red)*



(c) *Subtransformations of f_2: $f_2 \circ f_1$ and $f_2 \circ f_2$. (indicated in shades of blue)*



(d) *The region in which (b) and (c) overlap, indicated in shades of 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 (As defined in §§§ 4.1.3).*

Finally it is worth noting that [Law12] already presents an efficient way to render a large set of IFSs using a very different approach (c.f. §§§ 2.4.2), which might be worthwhile to be explored further.

References

[Bar88] Michael F Barnsley. *Fractals everywhere*. Academic press, 1988 (cit. on pp. 4–7).

[Cha+11] Manuel MT Chakravarty, Gabriele Keller, Sean Lee, Trevor L McDonell, and Vinod Grover. "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. 13).

[DR04] Scott Draves and Erik Reckase. *The fractal flame algorithm*. http://flam3.com/flame.pdf. 2004 (cit. on pp. 4, 13).

[Gc19] Gabriel Gonzalez and contributors. *Dhall Configuration Language*. https://github.com/dhall-lang/dhall-lang. 2019 (cit. on p. 15).

[Gre05] Simon G Green. "GPU-accelerated iterated function systems". In: *ACM SIGGRAPH 2005 Sketches*. 2005, 15–es (cit. on p. 10).

[Hai94] Eric Haines. "Point in Polygon Strategies." In: *Graphics Gems* 4 (1994), pp. 24–46 (cit. on p. 12).

[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. 4).

[HPS91] Daryl Hepting, Przemyslaw Prusinkiewicz, and Dietmar Saupe. "Rendering methods for iterated function systems". In: North-Holland, 1991 (cit. on pp. 4, 6, 7, 9).

[Jef90] H Joel Jeffrey. "Chaos game representation of gene structure". In: *Nucleic acids research* 18.8 (1990), pp. 2163–2170 (cit. on p. 4).

[Lau+09] C Lauterbach, M Garland, S Sengupta, D Luebke, and D Manocha. "fast BVH construction on GPUs". In: *Proceedings of the Eurographics Symposium on Rendering, Eurographics and ACM/SIGGRAPH*. 2009 (cit. on p. 11).

[Law12] Orion Sky Lawlor. "GPU-accelerated rendering of unbounded nonlinear iterated function system fixed points". In: *ISRN Computer Graphics* 2012 (2012) (cit. on pp. 6, 10, 21).

[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. 19).

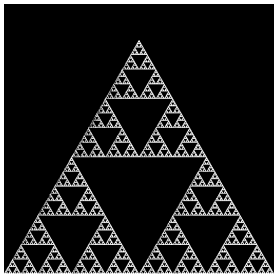
[Men03] Franklin Mendivil. "Fractals, graphs, and fields". In: *The American mathematical monthly* 110.6 (2003), pp. 503–515 (cit. on p. 6).

[WS06] Michael Wimmer and Claus Scheiblauer. "Instant Points: Fast Rendering of Unprocessed Point Clouds." In: *SPBG*. Citeseer. 2006, pp. 129–136 (cit. on p. 11).

A Iterated Function Systems used

This appendix lists the mapping functions of the IFSs that were used throughout this thesis. The p-values next to each of the mappings references the probability that this mapping ought to be chosen, if a chaos game-based rendering method is used (c.f. §§§ 2.3.2).

Refer to all 3 examples here.

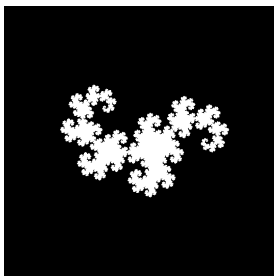


$$f_1(x,y) = \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}, \qquad p_1 = \frac{1}{3}$$

$$f_2(x,y) = \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \frac{1}{2} \\ 0 \end{bmatrix}, \qquad p_2 = \frac{1}{3}$$

$$f_3(x,y) = \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \frac{1}{4} \\ \frac{\sqrt{3}}{4} \end{bmatrix}, \qquad p_3 = \frac{1}{3}$$

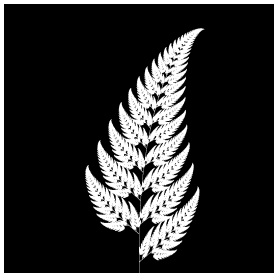
IFS A.1: *the Sierpińsky triangle*



$$f_1(x,y) = \frac{1}{\sqrt{2}} \begin{bmatrix} \cos 45° & -\sin 45° \\ \sin 45° & \cos 45° \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}, \qquad p_1 = \frac{1}{2}$$

$$f_2(x,y) = \frac{1}{\sqrt{2}} \begin{bmatrix} \cos 135° & -\sin 135° \\ \sin 135° & \cos 135° \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \qquad p_2 = \frac{1}{2}$$

IFS A.2: *the Heighway Dragon Curve*



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

IFS A.3: *the Barnsley Fern*