university of groningen

faculty of mathematics and natural sciences

In by Out again

Faking arbitrarily-deep zooming on Iterated Function Systems

Bachelors's thesis

August 14, 2020

Student: Wiebe-Marten Wijnja

Primary supervisor: dr. J. Kosinka

Secondary supervisor: G. J. Hettinga

Contents

Todo list

1 Abstract

Write

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. [CITE] IFSs see use in rendering/modeling of physical phenomena[CITE], image compression [CITE] and representing gene structures [CITE]. Sometimes they also see use simply for the aesthetic beauty of their graphical representations[CITE].

Various computer algorithms to visualize IFSs exist [CITE], [CITE]. However, these all take either a still image as final result, or, if they want to render an animation, view this as a sequence of separate still images to generate.

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

Investigating this claim in detail is the essence of this thesis.

Odd ending of section that needs to be fixed

2.1 Overview

Write Overview

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 transformations 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 output image are no longer visible to the human eye.

What you end up with is a visual representation of the IFS's attractor.

reference picture

Some examples of IFSs? (that are used 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:

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

That each mapping needs to be contractive 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 the 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). We can iterate it as often as we'd like:

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

If we perform this operation an arbitrary number of times, we approach the fixed-point or attractor, $\mathcal{A}$, of the Iterated Function System:

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

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

❇

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

Because of their prevalence, these are also the restrictions that will be used in this thesis.

Mention transformations and viewports w.r.t the unit square.

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.

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

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 [CITE]. Short summaries of the two most common techniques will now follow.

PICTURE OF THIS TREE

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.

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. [CITE]

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*[CITE], also known as the *random iteration algorithm*[CITE] 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:

[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 ($f \circ g \equiv G \cdot F$)

- 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]
- 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.

Picture outlining the importance of the number of points?

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 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.

[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 [CITE], but are not relevant for this thesis.

[4] These probabilities are often fine-tuned by hand, although algorithms to determine balanced probabilities exist as well [CITE] section 2.4.

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.

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. [CITE]

3.4.2 The deterministic method on the GPU

An exciting approach taken in [CITE] 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 Problem Description

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?

Maybe rephrase question?

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]

Unoptimized, it takes N transformations to render a precomputed point cloud to screen each frame. This does not seem very impressive since $\mathcal{O}(2(N + x)) \approx \mathcal{O}(2N) \approx \mathcal{O}(N)$, 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 $\mathcal{O}(\log N)$ 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[CITE]. 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. [CITE] 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 [CITE].

5.1.3 Detecting self-similarity: '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.

More info

[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.

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

Collage theorem?

that we might, in certain situations, 'secretly' zoom out to a more shallow camera viewport of the point-cloud that shows the same information of the attractor.

Example picture

5.2 Implementation

Picture of program running as state diagram

The program was implemented using the general-purpose programming language Haskell, in combination with the GPGPU library Accelerate [CITE]. This programming stack was chosen because Accelerate offers a statically-typed EDSL[6] 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.[7]

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.

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 mainly means that the optimizations mentioned in §§§ 5.1.2 were *not* implemented.

Mention how this still allows us to (dis)prove hypothesis

5.2.2 Rendering

5.2.3 Command-line options

5.2.4 IFS file format

The configuration language 'Dhall'[CITE] 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'.

[6] Embedded Domain-Specific Language.

[7] 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.

Dhall allows the definition and re-use of variables, which can be useful for numerical constants that are used in multiple transformations.[8]

refer to (to-be created) earlier definition of the initial camera viewport transformation

5.2.5 Manually performing 'self-similarity jumping'

While the program is running, a user can go back to a more shallow view by pressing '+', 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.

5.2.6 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.

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.

picture

6 Findings

6.1 Restrictions on replacing the view with a more shallow view

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.

   However, the algortihm as outlined in [REF EARLIER ALGORITHM] is not able to handle borders between transformations, thus making it useless in these scenarios.

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.

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

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 Sierpinský Triangle[REF], 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 [PICTURE OF DRAGON CURVE] 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.

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.

pseudocode

6.2 Difficulty of embellishing 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 [CITE fractal flame] keep track of a per-point colour, based on the sequence of transformations it has undergone.

These techniques are difficult, if not impossible, to combine with the concept of 'secretly zooming out' because this would create a noticeable jump in colours. While techniques to 'smooth out' the jump exist (such as overlaying mulltiple cameras at different zoom-levels at once and gradually fading between them), they are more akin a 'treatment of the symptoms' rather than a full solution of the cause.

6.3 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[9], this often is more memory than available.

[9] At the time of writing, high-end consumer GPUs contain somewhere between 2 and 24 GiB of available memory. [CITE] CITE: https://www.ign.com/articles/the-best-graphics-cards-3

7 Discussion

8 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.

9 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.

The same is true for if a technique for coloring the result that is usable even when jumping between zoom-levels on a point cloud. This would also make the proposed technique more practical to use.

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, [CITE] already presents an efficient way to render a large set of IFSs using a very different approach, which might be worthwhile to explore further.