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 12, 2020

Student: Wiebe-Marten Wijnja

Primary supervisor: dr. J. Kosinka

Secondary supervisor: G. J. Hettinga

Contents

1 Abstract

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 viewport that shares the same self-similarity as the original one.

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

2.1 Overview

3 Background

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

3.1 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.

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

PICTURE OF THIS TREE

3.2.1 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.

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

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

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]

3.2.2 The chaos game

The stochastic method, more frequently called 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
```

REF ALG

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

- at each depth in the tree the subtree remains the same.
- because the precision of the canvas we render on is finite, and because all transformations are contracting, only the latest x transformations determine at what location on the canvas a point will end up.[3]

Therefore, because we render all intermediate points, and these points correspond to different 'x-long sequences of transformations', 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, it can be optimized very well by modern CPU architecture.

One 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],

[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.

3.2.3 GPU

4 Problem Description

5 Implementation

6 Findings

7 Discussion

8 Conclusion

9 Further Work