In by Out again

Faking arbitrarily-deep zooming on Iterated Function Systems

Bachelors's thesis

August 13, 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].

<div align="center">✻</div>

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] $f_{i_k} \circ \ldots \circ f_{i_1}$;

---

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

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.

3.2.2   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:

- 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.

---

[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.

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.

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]

<p style="text-align:center">✳</p>

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.

3.3 IFS rendering on a GPU

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.

3.3.1 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]

3.3.2 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

---

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

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?

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 main inspiration for the re-usability approach is that we can modify the GPU-variant of the chaos game algorithm outlined in §§§ 3.3.1 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, running the whole chaos game takes $(2(N + x))$ transformations to render N points. [5]

Unoptimized, it takes N transformations to render a precomputed point cloud to screen. 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.

5.1.1 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' .

---

[5]Using the definitions of §§§ 3.2.2: 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.

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 not trivial at all [CITE]. As such, the program developed for this thesis does not implement these techniques. Instead, the choice was made for simplicity, and the use of a bounded volume hierarchy is treated as a potential future optimization.

5.1.2 Simplicity

As already mentioned in the previous subsection, the decision was made to keep the program as simple as possible. This allows a qualitative (human) measurement of whether the technique of using point clouds as intermediate re-usable information is at all feasible.

It can thus be considered a first step in answering the research question.

5.2 Implementation

6 Findings

7 Discussion

8 Conclusion

9 Further Work