GPU-Accelerated Rendering of Unbounded Nonlinear Iterated Function System Fixed Points

Orion Sky Lawlor

Abstract

Nonlinear functions, including nonlinear iterated function systems, have interesting fixed points. We present a non-Lipschitz theoretical approach to nonlinear function system fixed points which generalizes to non-contractive functions, compare several methods for evaluating such fixed points on modern graphics hardware, and present a nonlinear generalization of Barnsley's Deterministic Iteration Algorithm. Unlike the many existing randomized rendering algorithms, this deterministic method avoids noncoherent branching and memory access, and takes advantage of programmable texture mapping hardware. Together with the performance potential of modern graphics hardware, this allows us to animate high quality and high definition fixed points in real time.

Keywords: Nonlinear Fixed Points, IFS Rendering, Graphics Hardware, Flame Fractal

1. Introduction

Iterated Function Systems are a method to generate easily controlled, infinitely detailed fractal images such as Figure 1 from the repeated application of simple mathematical functions.

1.1. Mathematical Background

This paper shows how to compute fixed points of image-to-image functions. We define an *image* $I : X \to C$ as a function mapping some domain X (for example, 2D space) into some range C (a color space). Then an image-to-image transform $F : I \to I$ has a fixed point image $a \in I$ when $F(a) = a$: that is, the fixed point image remains unchanged under the image transform.

It is reasonable to ask if such a fixed point always exists, and the answer is no. For example, in a binary color space $C = \{0, 1\}$ the color inversion transform $F(I(\vec{p})) = 1 - I(\vec{p})$ does not have a fixed point. Yet a number of theorems establish sufficient conditions for the existence of such a fixed point. The majority of iterated function system work uses the well known Banach fixed point theorem, which gives both the existence and uniqueness of the fixed point, and merely requires X and C to be complete, but requires the image transform F to be Lipschitz contractive. This theorem has been used for much iterated function system work [1], but it does require contractivity. Since many interesting nonlinear functions are not contractive everywhere, including the functions shown in Figure 1, we will not use the Banach theorem.

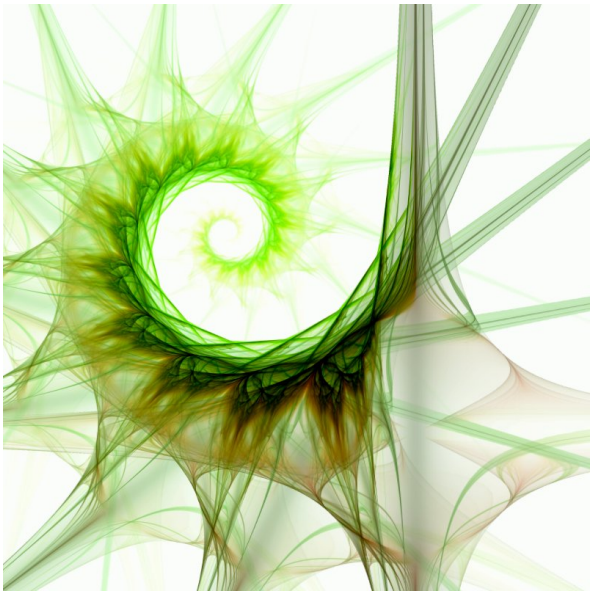


Figure 1. A simple two-map nonlinear iterated function system, as rendered on the GPU at 30fps using these techniques.

Instead, the Schauder-Tychonoff [2] fixed point theorem, an extension of the Brouwer fixed point theorem to infinite dimensional spaces, establishes sufficient conditions for the existence of a fixed point.

Schauder-Tychonoff Fixed Point Theorem 1. *Let K be a non-empty, compact, convex subset of a locally convex topological vector space. Given any continuous mapping $F : K \to K$, there exists a fixed point $a \in K$ such that $F(a) = a$.*

The first difficulty is that ordinary Euclidean space is non-empty and convex but not compact: the expand-

Email address: lawlor@alaska.edu (Orion Sky Lawlor)

ing 1D function $F(\vec{p}) = 2\,\vec{p}$ sends points off to infinity and hence has no fixed point in Euclidean space. We instead use projective space $\mathbb{RP}^d$, which is compact because it includes points at infinity—such as the fixed points of $F(\vec{p}) = 2\,\vec{p}$. We survey several ways to implement projective space in Section 3. We thus define an image as a function from points in some d-dimensional real projective space $\mathbb{RP}^d$ (typically $\mathbb{RP}^2$: 2D space plus the circle at infinity) and returning a c-dimensional *color* $C \in [0,1]^c$ (typically simply $c = 3$ channels of red, green, and blue), so an *image* is a function $I : \mathbb{RP}^d \to [0,1]^c$.

Note that without contractive maps, we are not guaranteed unique fixed points. For example, any point symmetric image is a fixed point of the image-to-image function $F(I(\vec{p})) = I(-\vec{p})$.

1.2. Nonlinear Iterated Function Systems

We define an *Iterated Function System* as a set of n separate *geometric distortion functions* $w_i : \mathbb{RP}^d \to \mathbb{RP}^d$. For example, the 2D plane-filling IFS shown in Figure 1 consists of these two geometric distortion functions, which are a rotation plus translation, and a simple nonlinear distortion.

$$w_0(x,y) = \begin{bmatrix} 0.8 & -0.4 \\ 0.4 & 0.8 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} -1 \\ 0.3 \end{bmatrix}$$

$$w_1(x,y) = \begin{bmatrix} x/(x^2+y^2) \\ y \end{bmatrix}$$

We apply a geometric distortion function to an image using an *image distortion function* $W_i : I \to I$ defined as follows:

$$W_i(I(\vec{p})) = I(w_i^{-1}(\vec{p}))\,|J_{w_i^{-1}}(\vec{p})| \tag{1}$$

Note that the function inverse in the geometric portion of this transform $I(w_i^{-1}(\vec{p}))$, as described in Section 5, is equivalent to forward-transforming each point in I by the geometric distortion function w_i, as in Section 4. The Jacobian determinant $|J_{w_i^{-1}}(\vec{p})|$, as discussed in Section 6, is present in order to preserve the integral of the transformed image, at least for well-behaved map functions. This can be seen via the substitution method for multiple integrals, where $\vec{q} = w_i^{-1}(\vec{p})$.

$$\iint W_i(I(\vec{p}))\,d\vec{p} = \iint I(w_i^{-1}(\vec{p}))\,|J_{w_i^{-1}}(\vec{p})|\,d\vec{p} =$$

$$\iint I(\vec{q})\,d\vec{q} = \iint I(\vec{p})\,d\vec{p}$$

Finally, the image-to-image function $F : I \to I$ is defined as a combination of the image distortion functions W_i and a set of constant color weights $c_i \in [0,1]^c$.

$$F(I) = \sum_{i=0}^{n-1} c_i W_i(I)$$

If our overall image-to-image transform F is continuous, in the sense that the colors of the output image depend continuously on the colors in the input image, then in the convex nonempty compact domain $\mathbb{RP}^d \times [0,1]^c$ the Schauder-Tychonoff theorem guarantees F has a fixed point. If the distortion function w_i^{-1} is continuous, then the geometric distortions in F will be continuous; and if the Jacobians $|J_{w_i^{-1}}|$ change continuously, the color intensity changes in F will be continuous; thus by Schauder-Tychonoff F has a fixed point.

2. Calculating IFS Fixed Points

There are a variety of ways to calculate an IFS fixed point, but the simplest is the *random iteration algorithm* [3], also known as the *chaos game*. We begin with a randomly chosen point $\vec{x}_0 \in \mathbb{RP}^d$, and repeatedly apply randomly chosen geometric distortion functions $\vec{x}_j = w_i(\vec{x}_{j-1})$. A histogram of the resulting point locations converges to the IFS fixed point, usually called an *attractor* in this context. For our example, if we repeatedly apply the rotation w_0, our points form a simple spiral. If we repeatedly apply the nonlinear transform w_1, our points rapidly approach the X and Y axes. Yet if we randomly alternate between these functions, the random iteration algorithm points plot out the much more complex Figure 1. The underlying motivation behind this paper is that the random iteration algorithm typically requires billions of points to produce a smooth image, which is too slow to produce high quality animations in real time.

There are many other interesting methods to render IFS—for a survey, see Nikiel's recent book [4]. Modern practical applications of IFS range from fractal image compression to artistic and rendering applications. The particular functional form for our nonlinear IFS map functions comes from Draves' [5] *fractal flames* nonlinear IFS, most commonly seen in his distributed-rendering screensaver Electric Sheep [6, 7]. The nonlinear function w_1 above is actually Draves' function "hyperbolic."

Beyond IFS, recursive Lindenmayer or *L-systems* [8] include the ability to pass parameters between recursive instances. This makes them much more useful than IFS to represent imperfectly self-similar shapes such as plants, as extended by Prusinkiewicz [9]. In fact, affine iterated function systems are actually equivalent [10] to a restricted form of L-systems known as turtle graphics. L-systems are often described as string rewriting systems, which is a useful definition but an extraordinarily inefficient implementation; L-systems can be efficiently incrementally instantiated, for example during ray tracing [11]. The context dependence and parameter passing of L-systems makes them more complex and difficult to analyze than IFS, so we will confine our attention to IFS in this paper.

2.1. Bounding Iterated Function Systems

Our overall goal is to render nonlinear iterated function systems using graphics hardware textures, which are uniform-resolution rectangular grids of pixels. That is, textures are bounded. But an IFS attractor may span the unbounded plane, so we must somehow deal with this mismatch.

A simple bounding method is to define each IFS such that the attractor is known to lie within some bound, such as the unit square. For example, if each map function takes points inside the unit square to a subset of the square, then by the recursive bounding theorem [12] the unit square is guaranteed to bound the IFS attractor. This is the approach taken by van Wijk [13], Gröller [14], Raynal et al. [15], and several others. The difficulty with this bounding method is that manipulating the IFS maps near the artificial boundary becomes cumbersome; sometimes in order to make room to manipulate one map, the user needs to adjust all the other maps, which seems unnecessarily difficult.

For any IFS we can transform the fixed point a by any invertible function T, simply by adjusting the individual map functions w_i according to the well-known [3] transform theorem. The new IFS has map functions $w_i = T \circ w_i \circ T^{-1}$. Essentially, the maps to represent the IFS attractor in the new space are found by first transforming points into the old space, applying the old map, and finally transforming back to the new space.

This implies that we can allow users complete freedom in defining their maps, yet still do our IFS processing on the convenient unit square, simply by finding a suitable transformation function F. For example, if we can find a bounding volume for the IFS attractor, we can then trivially compute an affine transformation of that volume to the unit square. Many authors have taken this approach, including the simple bounding sphere of Hart and DeFanti [16], the tighter sphere of Rice [17] and the anisotropic sphere of Martyn [18]. Convex bounds include the linear programming method of Lawlor and Hart [19], the bound refinement technique of Chu and Chen [20], and a heap-based convex bound refinement method [21, Appendix C]. Bounding volumes are also useful for raytracing 3D shapes, including affine IFS [16], Gröller's grid-deformation nonlinear IFS [14], and nonlinear CSG-pL systems [11].

Yet bounding volumes are less useful for many nonlinear IFS, because in practice these systems are often unbounded.

2.2. The Unbounded Non-Contractive IFS

An IFS attractor is bounded whenever the IFS map functions satisfy the Lipschitz contractivity condition [1]. Occasionally, a non-contractive IFS will nonetheless have a bounded attractor—contractivity is a sufficient condition, but not necessary. Yet there are actually a variety of interesting iterated function systems, including many nonlinear systems, that are unbounded and non-contractive, yet are still visually interesting.

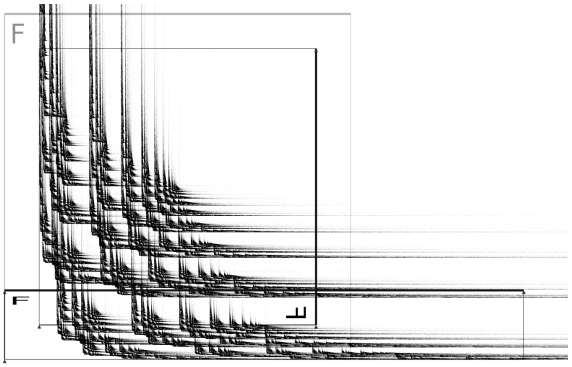


Figure 2. An unbounded non-contractive affine IFS. Large grey box is the reference unit square, black boxes the two maps.

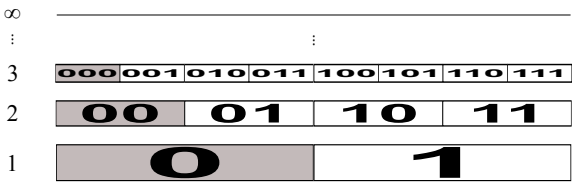


Figure 3. As additional maps are applied to the points of the IFS, fewer and fewer codespace points have never had w_1 applied.

For example, the two map affine IFS of Figure 2 does not satisfy the contractivity condition, and indeed its attractor extends to infinity along the thin horizontal and vertical spikes. The map functions are:

$$w_0(\vec{x}) = \begin{bmatrix} 1.5 & 0 \\ 0 & 0.2 \end{bmatrix} \vec{x} + \begin{bmatrix} 0.25 \\ -0.4 \end{bmatrix}$$

$$w_1(\vec{x}) = \begin{bmatrix} 0 & 0.8 \\ 0.8 & 0 \end{bmatrix} \vec{x}$$

This IFS is not contractive, because w_0 stretches points horizontally. Repeated compositions of w_0 stretch out points arbitrarily far. However, the attractor is still well defined, and the random iteration algorithm very rarely explodes points to infinity, because any application of w_1 will quickly bring distant points closer to the origin.

We agree with Saupe [22, pg. 310] that "There should be interesting theory of non-contractive IFS." Notice that for rendering, it is acceptable if a few of our points escape to infinity. We can even quantify this threshold, for example by taking the Lebesgue measure of these points at infinity. We can get reasonable Lebesgue measures by lining up all the computed IFS points on a 1D line, for example using Barnsley's map-based "codespace" coordinate system as illustrated in Figure 3. For our example non-contractive IFS, the 1D Lebesgue measure is zero for the set of points that escape to infinity.

Thus we can classify IFS as:

1. Contractive and hence bounded, the usual case for affine IFS.

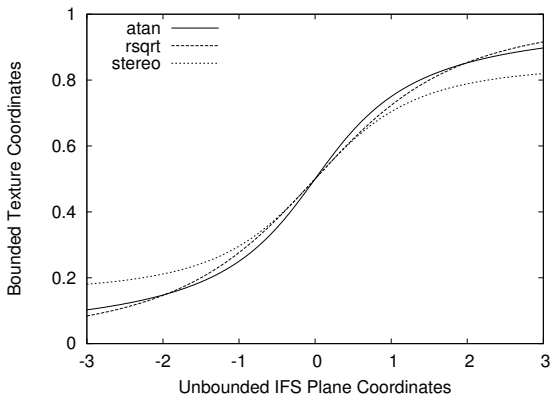


Figure 4. Comparing 1D plane-to-square functions used to compress an infinite plane IFS onto a finite texture.

2. Non-contractive yet bounded, which exists but is fairly rare.
3. Unbounded yet most[1] points are finite, such as the IFS above.
4. Unbounded with most points at infinity, a classic non-convergent IFS.

Only IFS in that last class cause serious problems during rendering, so in this paper we embrace unbounded iterated function systems.

We have noticed that some map functions, including that of Figure 1, cause the naive random iteration algorithm to become stuck in one local area, rather than properly sampling the entire attractor. This cannot happen for contractive affine maps, which have a single attractive fixed point, but for non-contractive nonlinear maps with multiple fixed points, it actually seems to be fairly common. This can be remedied by periodically restarting the random iteration algorithm at a new random plane point. For a similar reason, the output range of the random iteration algorithm's initial random number generator is important, and generally a larger range will be more likely to correctly sample a larger attractor. The deterministic algorithms we present seem to be more robust against these effects, especially with a large initial attractor estimate image.

3. Rendering Unbounded IFS on the GPU

To fit an IFS attractor into a GPU hardware texture, we must have a bounded IFS. But many of the IFS we would like to render are unbounded, and the attractor even includes a few points at infinity. Our solution is to *compress* the unbounded IFS into an equivalent bounded IFS, using an invertible but nonlinear compression transform function. At display time, if desired we can *uncompress* the attractor back into the original plane.

In particular, we would like to convert plane coordinates, in the interval $(-\infty, \infty)$ on both axes, into a square in graphics card texture coordinates, within $(0, 1)$ on both axes. The plane-to-square mapping we choose must be smooth, because it will be used to write discrete samples from an IFS attractor into a texture; any discontinuities in the plane-to-square function will manifest themselves as sampling artifacts in the texture. The function also must be invertible, so that we can read samples of the IFS attractor from the texture. Finally, both the function and its inverse must be efficient to compute on the graphics card, because every access to the texture will require at least one execution of the function.

The quite similar general shapes of several such functions are compared in Figure 4, and their equations and performance are compared in Table 1. Of these, the fastest functions on current GPU hardware[2] are the 2D polar stereographic projection `stereo`, and the per-axis equivalent projection `rsqrt`. We attribute the excellent speed of these two functions to the fact that reciprocal-square-root is a GPU hardware instruction.

Comparing these two, there is slightly more shape distortion in the `rsqrt` version. While `rsqrt` compresses an infinite plane to the unit square, `stereo` compresses the plane into a unit disk, as illustrated in Figure 5(b). These two functions are identical precisely along the x and y coordinate axes, but since GPU textures are square, `rsqrt` produces fewer sampling artifacts along the diagonal lines. This is especially true near the points at infinity, which `rsqrt` cleanly maps to the texture's outer boundary pixels, while `stereo` maps to a pixel-discretized approximation of a circle.

For this reason, we use `rsqrt` as our compression and decompression scheme; it seems to work well to transform an unbounded IFS into an equivalent bounded IFS. Due to sampling issues, some IFS may benefit from an additional pre-`rsqrt` affine plane transformation, typically just a translation and scaling. But because plane-to-square functions are necessarily nonlinear, we may need to render a (bounded) nonlinear IFS even if the original IFS was linear (but unbounded). We explore several approaches to render a bounded nonlinear IFS in the next two sections.

4. Inverting Functions per Vertex by Rasterization

Barnsley's Deterministic Iteration Algorithm [3] determines an IFS fixed point A, by iteratively following

[1] "Most" here in a Lebesgue sense, comparing measures.

[2] GPU performance measured on an NVIDIA GeForce 280 GTX.

Name	Description	Plane to Texture Function	Texture to Plane Function	Timing
`erfc`	Gaussian error function	$T(p) = \text{erfc}(p)/2$	$P(t) = \text{erfc}^{-1}(2\,t)$	0.40 ns
`atan`	Trigonometric tangent	$T(p) = \arctan(p)/\pi + \frac{1}{2}$	$P(t) = \tan(\pi\,t - \frac{\pi}{2})$	0.16 ns
`rsqrt`	Reciprocal square root	$T(p) = p/(2\sqrt{1+p^2}) + \frac{1}{2}$	$u = 2\,t - 1;\ P(t) = u/\sqrt{1-u^2}$	0.08 ns
`stereo`	Polar stereographic	$T(\vec{p}) = \vec{p}/(2\sqrt{1+\vec{p}\cdot\vec{p}}) + \frac{1}{2}$	$\vec{u} = 2\,\vec{t} - 1;\ P(\vec{t}) = \vec{u}/\sqrt{1-\vec{u}\cdot\vec{u}}$	0.08 ns

Table 1. Compression and decompression functions, and corresponding GPU per-pixel performance. `erfc`, `atan`, and `rsqrt` are evaluated independently along each axis; `stereo` is a 2D vector function. `erfc` is not built into GLSL, though it exists in CUDA.

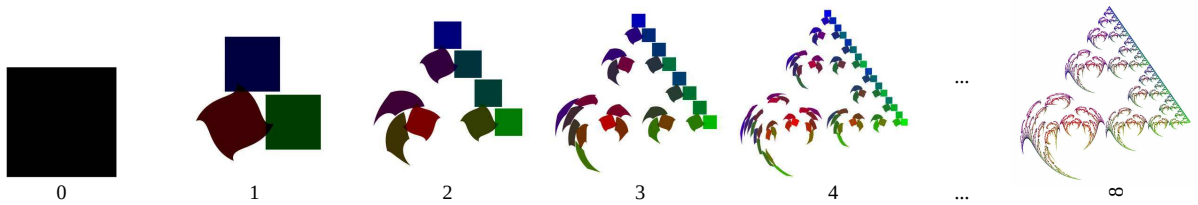


Figure 6. As we repeatedly distort our texture by the IFS map functions, the texture iteratively approaches the attractor (see Figure 16).

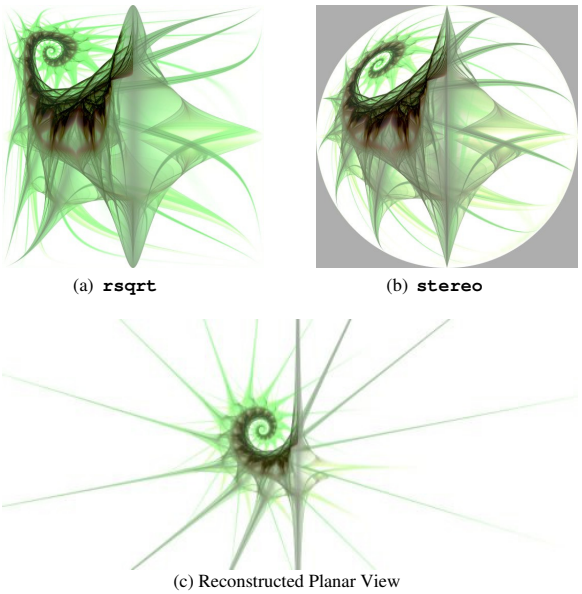


Figure 5. Comparing 2D `rsqrt` and `stereo` textures.

its definition as the union of its images under the IFS maps W_i:

$$A = \cup_{i=0}^{n-1}\ W_i(A)$$

In this method, we approximate the IFS attractor A using a rectangle of $p \times p$ pixels in a GPU texture—hence the attractor must be bounded, for example using one of the techniques discussed in Section 3. At each step k, we create a better approximation of the attractor A_k by applying the IFS maps to the previous approximation A_{k-1} (for now, ignoring map probabilities, colors, and density effects) with:

$$A_k = \sum_{i=0} W_i(A_{k-1}) \qquad (2)$$

Repeatedly applying this process eventually converges on the attractor, as illustrated in Figure 6. In practice, many IFSs produce a good attractor estimate in as few as a dozen such map iterations. In theory we can begin the iteration with any arbitrary approximation A_0, typically either a unit square or entire plane, or the solution from the previous frame or multigrid level (see Section 7.4).

On the GPU, we can implement Equation 2 by simply rasterizing each mapped attractor image $W_i(A_{k-1})$ into the framebuffer A_k. For affine maps, even ancient graphics interfaces such as OpenGL 1.1 directly support this rasterization; van Wijk and Saupe [13] found excellent GPU performance for affine IFS back in 2004. However, nonlinear maps are substantially more difficult to rasterize with high quality.

One obvious approach to render nonlinear map images on graphics hardware is to discretize our old attractor estimate A_{k-1} using a grid of $v \times v$ vertices V_{jk}. Then we can transform each vertex V_{jk} by each map function w_i, and then draw the attractor texture interpolated between the mapped vertices in the usual fashion,

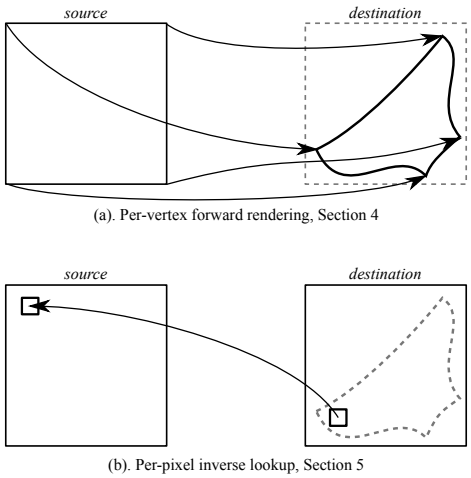


Figure 7. Two approaches for distorting a texture by an IFS map: per-vertex forward application of maps, and per-pixel inverse mapping.

as illustrated in Figure 7(a) and the following CPU-side pseudocode.

```
texture A=init_attractor_estimate();
texture D; // destination texture
do { // iterative attractor refinement
  set_framebuffer(D); // render to D
  clear_framebuffer();
  for (int i=...) { // loop over IFS maps
    mesh m=new mesh(v,v); // v×v vertices
    for (int j=...) { // loop over vertices
      vec2 texcoord=V_j; // fixed source
      vec2 geometry=w_i(V_j); // destination
      m.add_vertex(texcoord,geometry);
    }
    // texture the map's geometry using A
    draw_textured_mesh(A,m);
  }
  A=D; // copy out our new estimate
} until (... attractor converged ...);
```

In practice, the `A=D` step is typically implemented by swapping the two texture handles, a "ping-pong," rather than actually copying any data.

This per-vertex algorithm works well for smooth map functions, when linearly interpolating the mapped geometry between vertices would be reasonably accurate. Gröller's [14] nonlinear IFS maps are actually defined as a linearly interpolated vertex grid. Clearly, an adaptive mesh refinement scheme [11] or higher-order vertex interpolation would improve accuracy. However, neither adaptivity nor high-order schemes will work if the map function jumps discontinuously.

In any case high quality images or less-smooth map functions require a dense set of vertices, which causes several increasingly unfortunate effects. First, the vertex work per map per iteration soon overwhelms the CPU's arithmetic capacity. This problem is fairly easy to address by simply moving the $w_i(V_j)$ computation into a vertex shader, which yields about a tenfold performance improvement in our experiments. One could also prepare a vertex buffer object for each map, since the mesh does not change across iterations, only the texture shown on it. But we then reach a far more serious bottleneck, which is the GPU's triangle setup rate.

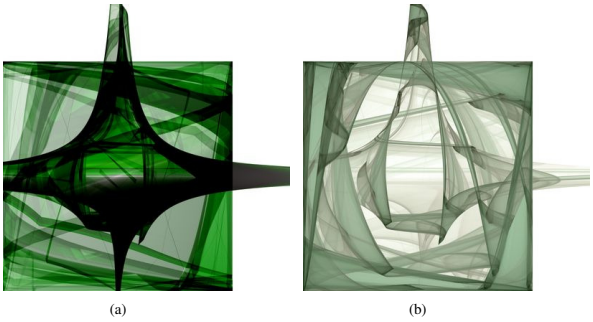


Figure 8. Comparing per-vertex and per-pixel nonlinear maps.

Theoretically, to achieve a framerate of f frames per second, when repeating r iterative attractor expansions per frame, each of which draws m maps with a grid of $v \times v$ vertices, requires the graphics card to draw $2frmv^2$ triangles per second. For example, at $f = 20$ frames per second, with just $r = 10$ iterations per frame, $m = 3$ maps, and $v = 1000$ vertices per side, would require 1.2 billion triangles per second. Though a typical modern GPU can process tens of billions of pixels per second (known as "fill rate"), even the best cards process less than a billion triangles per second, so even this moderate end-to-end performance is not achievable via vertex shaders. See the performance experiments in Section 7.3 for details.

Also, a dense grid of $v = 1000$ vertices per side may not be enough vertices. For example, consider an IFS using the "sinusoidal" map function, which collapses the entire plane into a cube via the sine function; clearly Nyquist-level vertex sampling over an infinite plane is impossible. In Figure 8(a), even at $v = 1000$ the vertex geometry used to discretize this map function is still visible. By contrast, Figure 8(b) shows the much cleaner results obtained via the per-pixel analytic map inversion method described in the next section.

5. Analytic Per-Pixel Function Inversion

Arbitrary nonlinear functions may combine sharp discontinuities, smooth curves, and high-frequency regions that are all difficult to sample accurately. But we are trying to render an attractor estimate texture A_k, so sampling in the destination space is trivially simple, a regular grid of pixels. That is, instead of trying to sample the function $W_i(A_{k-1})$ so that the resulting geometry covers A_k with sub-pixel accuracy, we instead follow Equation 1 directly and start at a pixel $A_k(\vec{t})$

with 2D destination texture coordinates $\vec{t}$, and sample the source texture A_{k-1} at source texture coordinates $w_i^{-1}(\vec{t})$.

In equations, we start from Barnsley's whole-image set-theoretic deterministic iteration method, where we must apply a nonlinear image distortion W_i to an entire image A_{k-1}:

$$A_k = \cup_{i=0}^{n-1} W_i(A_{k-1})$$

Hence we switch to a pixel-by-pixel function inversion, starting in the destination space:

$$\forall \vec{t} \in \mathbb{RP}^d \quad A_k(\vec{t}) = \sum_{i=0}^{n-1} A_{k-1}(w_i^{-1}(\vec{t}))\ |J_{w_i^{-1}}(\vec{t})| \qquad (3)$$

This approach maps perfectly to graphics hardware pixel shaders: A_k is the framebuffer, A_{k-1} is bound as a source texture, and we compute the source texture coordinates $w_i^{-1}(\vec{t})$ and Jacobian via a programmable shader.

But by contrast with affine maps, which only lack an inverse function for degenerate cases, there are a number of practical and theoretical difficulties with inverting general nonlinear functions.

5.1. Handling Multivalued Nonlinear Inverses

Many nonlinear functions have no well-defined inverse function, but instead have a multi-valued inverse relation. For example, the inverse of the 1D function $V(x) = x^2$ has both positive and negative branches $V^{-1}(s) = \pm\sqrt{s}$; the inverse of the 2D function $V(x, y) = (x^2, y^2)$ has four branches $V^{-1}(s, t) = (\pm\sqrt{s}, \pm\sqrt{t})$. In practice, we can often simply sum up the quantity of interest, such as an attractor density estimate, over each of the inverse values. Of course, periodic functions such as $V(x) = \sin(x)$ have infinite families of inverses, so in practice we must eventually truncate this summation; typically we find summing up a few periods is sufficient.

A further complicating factor is that the current generation of GPU hardware supports neither dynamically sized arrays, nor virtual functions, nor function pointers, nor even simple recursion. This complicates any design supporting multi-valued inverses—we cannot dynamically allocate a list of inverse values, we cannot call a virtual method or function pointer for each inverse we find, and we cannot recursively search for values. However, with some graphics interfaces, such as GLSL or OpenCL, we do generate the GPU functions at runtime; this means each time we find an inverse, at function generation time we can simply paste in a call to the function needing the inverse value.

In particular, our output pixel $A_k(\vec{t})$ will require texture samples from the input texture A_{k-1} at *each* inverse value found for the inverse map $w_i^{-1}(\vec{t})$, as illustrated in Figure 9. In the forward direction, a typical

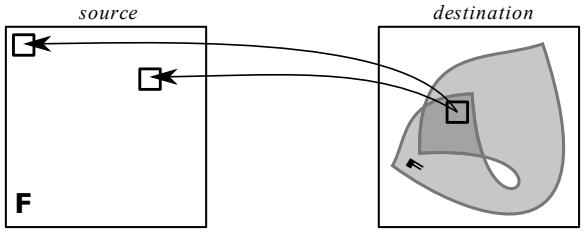


Figure 9. Highly nonlinear maps may have multiple inverses.

IFS map w consists of a texture-to-plane decompression function P (from Section 3), then an affine matrix transformation M, then a nonlinear "variation" function V, and finally a plane-to-texture re-compression function T:

```
vec2 destloc=T(V(M(P(srcloc))));
```

To look up texture values for each inverse, we simply apply the inverse of each transformation, $w^{-1} = P^{-1} \circ M^{-1} \circ V^{-1} \circ T^{-1}$, and then look up the resulting points in the source texture. Due to the above GPU hardware limitations, we split the call sequence after the nonlinear inverse step $V^{-1} \circ T^{-1}$, and put $P^{-1} \circ M^{-1}$ into a separate function, here called `f`. This lets our nonlinear inverse relation simply call `f` at each inverse value, and sum up the densities it returns:

```
vec4 nonlinear_inverse(vec2 destloc)
{ // sum up IFS's density at destloc
  vec2 st=P(destloc); // (s,t)= T^-1
  vec2 srcloc=...; // V^-1(s,t) see Table 2
  return f(+srcloc)  // positive branch
       + f(-srcloc); // negative branch
}
vec4 f(vec2 srcloc) // post-V coordinates
{ // return IFS's density at srcloc
  vec2 srctex=T(Minv(srcloc)); // P^-1(M^-1)
  return density(texture2D(src,srctex));
}
```

Because we can generate the exact code to invert only the maps of the IFS currently being rendered, this approach seems to perform quite well. A more static code structure, that compiled in code for every supported nonlinear IFS map function, would require a large switch statement or nested series of comparisons to select the appropriate nonlinear function, neither of which would perform well on current GPU hardware. Although we dynamically generate the code above at runtime, the GLSL driver takes a few milliseconds to recompile the code, so for animation purposes rather than recompiling every frame, we pass in function parameters via uniform variables. This means we only need to recompile when the functional form of the IFS changes, not just its parameters.

In practice, the sampling properties of this per-pixel inversion method seem to be excellent. Further, the

texture sampling hardware provides both linear texture filtering when a map sample lands between source texture pixels, and anisotropic mipmapping when the mapped sample should read from a larger area. Compared to the conventional random iteration algorithm, per-pixel inversion gives extremely smooth images such as Figure 1, especially in low-density regions where the random iteration algorithm's discrete points are far apart. Compared to the per-vertex deterministic iteration algorithm described in the previous section, this per-pixel algorithm follows mapped curves more accurately and handles map function discontinuities more cleanly.

5.2. Nonlinear Inverses May Not Exist

Simple nonlinear functions sometimes have very complicated inverses, both in terms of execution time and code complexity. For example, Cardano's solution is much more complex than a cubic polynomial. Computer algebra systems can help to find and simplify inverse relations, although substantial human effort is often still required to create and test working code.

Table 2 summarizes the first twenty nonlinear "variation" functions proposed by Draves [5]. We found easy-to-compute inverse relations for thirteen of these functions; two additional functions do have inverses, but they are too complex to write here, and likely too complex to actually use at runtime.

The remaining five functions appear to have non-elementary inverses, but it is rare one can prove the non-existence of an elementary inverse relation. We attempted to find inverses using both the computer algebra system Mathematica 7.0 and manual effort, but neither found an inverse in a reasonable time.

Many nonlinear functions simply do not possess an elementary inverse. For example, $V(x) = \sin(x) + x$ does not have an elementary inverse, nor does a general fifth-degree polynomial. Though function inverse reverse-distributes over composition $(F \circ G)^{-1} = G^{-1} \circ F^{-1}$, few other relations hold; for example, knowing F^{-1} and G^{-1} tells you nothing about $(F + G)^{-1}$ or $(F * G)^{-1}$.

In these cases inverse values could still be numerically approximated using a power series (for example, via the Lagrange inversion theorem), or using a generic nonlinear root-finding approach such as bisection or Newton's method. These approximations could be precomputed and stored in a texture, or evaluated at runtime per pixel, depending on the hardware's ratio of memory and arithmetic bandwidth.

An entirely different solution to the difficulty of function inversion is to simply invert the definitions: choose easy to evaluate functions as the "inverses" w_i^{-1} and apply these functions directly in the deterministic iteration algorithm. The "forward" functions w_i are then equally difficult to compute, but our algorithm never needs to compute them. Typically an automated image compressor or artist chooses from a fixed set of pre-defined basic functions anyway, so defining the map functions this way is less restricting than it may seem.

6. Attractor Density Estimation

The Jacobian determinant $|J_w(\vec{x})|$ of the function $w : \mathbb{RP}^d \to \mathbb{RP}^d$ at a point $\vec{x} \in \mathbb{RP}^d$ is defined in 2D as the determinant of the function's 2×2 Jacobian matrix of partial derivatives, evaluated at $\vec{x}$:

$$J_w(\vec{x}) = \begin{bmatrix} \frac{\partial w.x}{\partial x} & \frac{\partial w.x}{\partial y} \\ \frac{\partial w.y}{\partial x} & \frac{\partial w.y}{\partial y} \end{bmatrix} (\vec{x})$$

Expanding out the determinant, we get:

$$|J_w(\vec{x})| = \left(\frac{\partial w.x}{\partial x} \frac{\partial w.y}{\partial y} - \frac{\partial w.x}{\partial y} \frac{\partial w.y}{\partial x} \right) (\vec{x})$$

Substantial cancellation often occurs in this expression, so computing Jacobian determinants in practice is normally quite efficient.

For affine map functions, the Jacobian is a constant, so it is often folded together with the arbitrary map probability. But for our nonlinear maps, the Jacobian determinant may vary with location, so we cannot simply fold it into the constant map probability.

Note that we must in general evaluate the Jacobian determinant after inverting the function, because our nonlinear function inverses can take multiple values, yet not all these values will necessarily have the same Jacobian.

Sometimes the Jacobian of the forward function J_{w_i} is simpler to evaluate than $J_{w_i^{-1}}$, in which case we use the fact that the inverse of the Jacobian matrix is the Jacobian of the inverse function, and applying the determinant transforms a matrix inverse into a multiplicative inverse.

$$|J_{w_i^{-1}}(\vec{p})| = |[J_{w_i}(w_i^{-1}(\vec{p}))]^{-1}| = 1/|J_{w_i}(w_i^{-1}(\vec{p}))|$$

We show the Jacobian determinants for a variety of nonlinear functions in Table 2. Some extremely nonlinear functions such as "swirl" nonetheless have a constant Jacobian determinant, indicating swirl is area-preserving everywhere. It is noteworthy that each of the nonlinear functions we were able to invert has a simple Jacobian determinant, indicating some underlying simplicity. The non-invertible functions we examined, even those with superficially similar functional form, all had substantially more complex Jacobian determinants. Despite this still-moderate complexity, unlike function inverses, every elementary function also has an elementary Jacobian.

It is also possible to evaluate Jacobian determinants numerically. Some graphics programming languages even include builtin primitives for this, such as GLSL's

Function	**Forward** $V(x,y)$, with $r = \sqrt{x^2+y^2}$	**Inverse** $V^{-1}(s,t)$, with $r = \sqrt{s^2+t^2}$	**Jacobian**
1. Sinusoidal	$\begin{bmatrix} \sin x \\ \sin y \end{bmatrix}$	$\begin{bmatrix} \sin^{-1} s \\ \sin^{-1} t \end{bmatrix}$ plus multiples of π	$\cos x \cos y$
2. Spherical	$\begin{bmatrix} x \\ y \end{bmatrix} / r^2$	$\begin{bmatrix} s \\ t \end{bmatrix} / r^2$	$1/r^4$
3. Swirl	$\begin{bmatrix} x \sin r^2 - y \cos r^2 \\ x \cos r^2 + y \sin r^2 \end{bmatrix}$	$\begin{bmatrix} s \sin r^2 + t \cos r^2 \\ -s \cos r^2 + t \sin r^2 \end{bmatrix}$	1
4. Horseshoe	$\begin{bmatrix} (x-y)(x+y) \\ 2xy \end{bmatrix} / r$	$y = \pm\sqrt{(r^2 - sr)/2}$; $x = y(s+r)/t$	2
5. Polar	$\begin{bmatrix} \theta/\pi \\ r - 1 \end{bmatrix}$	$(t+1)\begin{bmatrix} \sin(\pi s) \\ \cos(\pi s) \end{bmatrix}$	$1/(\pi r)$
7. Heart	$r\begin{bmatrix} \sin(\theta r) \\ -\cos(\theta r) \end{bmatrix}$	$r\begin{bmatrix} \sin((\text{atan2}(s,-t) + 2\pi k)/r) \\ \cos((\text{atan2}(s,-t) + 2\pi k)/r) \end{bmatrix}$	r
8. Disc	$\theta/\pi \begin{bmatrix} \sin \pi r \\ \cos \pi r \end{bmatrix}$	$(\text{atan2}(s,t)/\pi + 2k)\begin{bmatrix} \sin \pi r \\ \cos \pi r \end{bmatrix}$	$\theta/(\pi r)$
10. Hyperbolic	$\begin{bmatrix} (\sin\theta)/r \\ r\cos\theta \end{bmatrix} = \begin{bmatrix} x/r^2 \\ y \end{bmatrix}$	$\begin{bmatrix} (1 \pm \sqrt{1 - 4s^2t^2})/2s \\ t \end{bmatrix}$	$\cos(2\theta)/r^2$
13. Julia	$\pm\sqrt{r}\begin{bmatrix} \cos(\theta/2) \\ \sin(\theta/2) \end{bmatrix} = \pm\begin{bmatrix} \sqrt{(r+y)/2} \\ x/\lvert x\rvert\sqrt{(r-y)/2} \end{bmatrix}$	$r^2\begin{bmatrix} \sin(2\theta) \\ \cos(2\theta) \end{bmatrix} = \begin{bmatrix} 2st \\ (s-t)(s+t) \end{bmatrix}$	$1/(4r)$
14. Bent	$\begin{cases} [x\ \ y]^T & \text{if } x \ge 0, y \ge 0 \\ [2x\ \ y]^T & \text{if } x < 0, y \ge 0 \\ [x\ \ y/2]^T & \text{if } x \ge 0, y < 0 \\ [2x\ \ y/2]^T & \text{if } x < 0, y < 0 \end{cases}$	$\begin{cases} [s\ \ t]^T & \text{if } s \ge 0, t \ge 0 \\ [s/2\ \ t]^T & \text{if } s < 0, t \ge 0 \\ [s\ \ 2t]^T & \text{if } s \ge 0, t < 0 \\ [s/2\ \ 2t]^T & \text{if } s < 0, t < 0 \end{cases}$	$\begin{cases} 1 \\ 2 \\ 1/2 \\ 1 \end{cases}$
16. Fisheye	$2/(r+1)\begin{bmatrix} y \\ x \end{bmatrix}$	$1/(2-r)\begin{bmatrix} t \\ s \end{bmatrix}$	$4/(1+r)^3$
18. Exponential	$e^{x-1}\begin{bmatrix} \cos \pi y \\ \sin \pi y \end{bmatrix}$	$\begin{bmatrix} \log(r) + 1.0 \\ \text{atan2}(t,s)/\pi \end{bmatrix}$	πe^{2x-2}
19. Power	$r^{x/r-1}\begin{bmatrix} y \\ x \end{bmatrix}$	$r^{r/s-1}\begin{bmatrix} s \\ t \end{bmatrix}$	$r^{2x/r-2}x/r$

Function	**Forward** $V(x,y)$	**Inverse**	**Jacobian Determinant**
6. Handkerchief	$r\begin{bmatrix} \sin(\theta+r) \\ \cos(\theta-r) \end{bmatrix}$	Non-elementary	$\cos 2r + \frac{2xy}{r} - r\sin 2r$
9. Spiral	$\begin{bmatrix} \cos\theta + \sin r \\ \sin\theta - \cos r \end{bmatrix} / r$	Non-elementary	$(1 - r\cos(r-\theta) + \sin(r-\theta))/r^2$
11. Diamond	$\begin{bmatrix} \sin\theta\cos r \\ \cos\theta\sin r \end{bmatrix}$	32 root families	$(\cos(2r) + 2y^2/r^2 - 1)/2r$
12. Ex	$r\begin{bmatrix} \sin^3(\theta+r) + \cos^3(\theta-r)^3 \\ \sin^3(\theta+r) - \cos^3(\theta-r)^3 \end{bmatrix}$	Non-elementary	$(6xy + r\cos 2r - 3r^2\sin 2r)$ $*(-3/(2r)) * (\sin 2r + xy/r^2)^2$
15. Waves	$\begin{bmatrix} x + a_1\sin(a_2 y) \\ y + a_3\sin(a_4 x) \end{bmatrix}$	Non-elementary	$1 - a_1a_2a_3a_4\cos(a_4x)\cos(a_2y)$
17. Popcorn	$\begin{bmatrix} x + a_1\sin(\tan 3y) \\ y + a_2\sin(\tan 3x) \end{bmatrix}$	Non-elementary	$1 - 9a_1a_2\cos(\tan 3x)\cos(\tan 3y)$ $+ \sec^2(3x)\sec^2(3y)$
20. Cosine	$\begin{bmatrix} \cos(\pi x)\cosh y \\ -\sin(\pi x)\sinh(y) \end{bmatrix}$	16 root families	$\pi/2(-\cos 2\pi x + \cosh 2y)$

Table 2. Functions and inverses are discussed in Section 5.2 and used for per-vertex and per-pixel rendering respectively. The Jacobian determinants are used in Section 6 for attractor density; only the forward direction is shown for the Jacobians. θ stands for $\arctan(x/y)$ or $\arctan(s/t)$, the reciprocal of their normal usage for compatibility with Draves' large existing library of nonlinear fractals.

dFdx and dFdy keywords, which compute a finite difference by examining neighboring pixels. However, we find that analytically evaluated Jacobians are better behaved near discontinuities, and are less susceptible to numerical and sampling artifacts.

6.1. Properties of the Density Jacobian

In general, because the Jacobian is assembled from partial derivatives, a straightforward application of the chain rule can determine the Jacobian of the composition of two functions A and B:

$$J_{A\circ B}(\vec{x}) = J_A(B\vec{x})J_B(\vec{x})$$

So, for example, a variation function V applied after a matrix M will have Jacobian

$$J_{V\circ M}(\vec{x}) = J_V(M\vec{x})\|M\|$$

That is to say we evaluate V's Jacobian after first transforming $\vec{x}$, then multiply by the matrix's Jacobian.

We can ignore the Jacobian contributions from the plane-to-texture compression and decompression functions T and P from Section 3, because they will cancel each other out. That is, we can compute the attractor density for the original infinite-plane IFS, although we sample that density geometrically only at the texture pixels. This results in an easier-to-interpret density—with T and P included, the choice of these plane compression functions affects the resulting attractor density values, not just their sampling geometry. Avoiding the plane-to-texture and texture-to-plane Jacobian contributions this way also saves a little arithmetic work during rendering.

6.2. Jacobian Density Estimation on the GPU

The dynamic range of the Jacobian determinant term can be quite large. Draves [5] applies a nonlinear log-exposure function after accumulating counts into a high-range framebuffer. We find that the modern GPU exponential and log functions are fast enough that we can actually store the $\log_2$ of the density in our texture pixels, and "unpack" this density using an exponential operation after every texture fetch. Arithmetic can then performed in high range linear-density space, while all storage happens in the range-limited log space.

These nonlinear density unpack and pack functions are implemented in GLSL as follows. Our texture color channels store numbers in the interval $[0, 1]$ using 8 bits of precision. The factor of 20 below lets us store a density dynamic range of 2^{20}, about a millionfold, which seems to be enough for most IFS to create smooth attractive images. On modern GPU hardware, it is actually several times faster to use 8-bit fixed-point memory storage and expand the range in software using these exponential and logarithm operations, than it is to simply use 32-bit floating-point storage without any additional arithmetic.

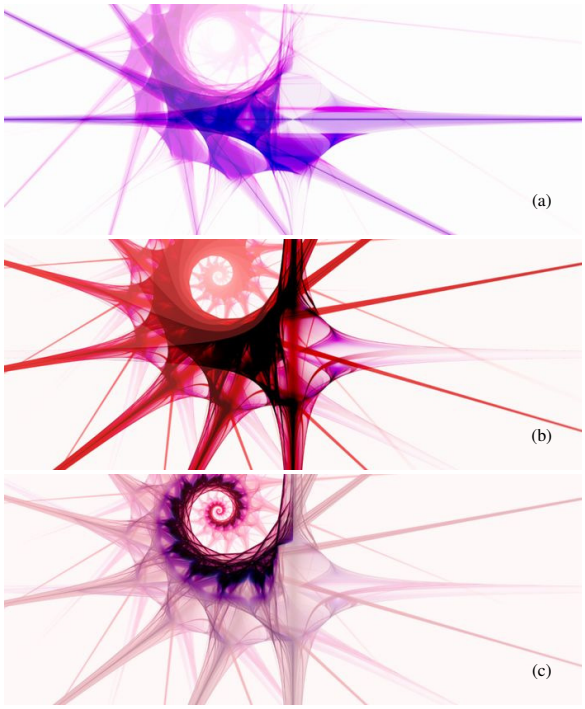


Figure 10. The IFS of Figure 1 rendered (a) with linear output, no Jacobian; (b) with Jacobian; (c) log output with Jacobian.

```
vec4 density=exp2(tex*20)-1; // unpack
vec4 tex=log2(density+1)/20; // pack
```

For per-vertex rendering, this unpack-sum-pack operation is not possible as a standard framebuffer blend operation, so one must resort to rendering the map into a separate texture, then performing the blending manually in a second pass. For per-pixel rendering, we can actually fetch, unpack, and sum up the densities for all the IFS maps in a single pass, and then apply the log transformation before writing the pixels out to the framebuffer. We illustrate the effect of the Jacobian and log-density output in Figure 10, and show the details in the complete code example in Figure 14.

Finally, we can add RGBA color to our attractor by multiplying the output of each W_i with a color c_i—this is equivalent to adjusting the color of each function's output pixels. This is why we use a four float "vec4" above, and it is equivalent to the method of iterating a color through the IFS maps along with position. Draves' fractal flames use a different method, where a single "color" index is iterated along with position, and then looked up in a 256-entry color lookup table before blending to the framebuffer; his approach makes it easier to highlight substructures in the attractor, and could be emulated by simply using more color channels in the texture, but we find the classic IFS coloring method to be sufficient. Plane-to-texture compression, per-pixel inversion, Jacobian density estimation, and log density output all combine well to render nonlinear IFS in color on the GPU.

7. Performance Comparisons

The *Graphics Processing Unit* (GPU) has evolved quickly over the last decade, so current GPUs can now run sophisticated C-like code fragments at every pixel. Because GPU languages such as GLSL, OpenCL, or CUDA do not allow dependencies between pixels, the GPU hardware can execute these programs in parallel across pixels. A GPU typically executes hundreds of pixels per clock, with thousands of pixels in flight, and so delivers orders of magnitude higher performance than multicore [23]. GPUs have hardware support to both read and write textures, which are 2D or 3D arrays of pixels or voxels stored in a variety of formats. The main advantage of using C++ and OpenGL is that the same code can be run on Windows, Macintosh, and UNIX computers.

Many researchers have rendered iterated function systems at interactive rates. In 1995, Monro and Dudbridge [24] achieved nearly 1M pixels per second using a fixed point SIMD within a register software implementation. In 2004, for affine IFS a deterministic GPU texture-based implementation completed about 13M pixels per second [13] (50fps for a 512x512 output image). In 2005, a GPU-based nonlinear IFS render to vertex buffer implementation of the random iteration algorithm completed 20M finished output points per second [25] (20fps for a 1M point buffer). The state of the art as of late 2011 appears to be 1 billion point-iterations per second, achieved using a highly optimized random iteration algorithm implementation in CUDA [26].

7.1. Theoretical Performance Comparison

We define an IFS image as an estimate of the true proportion p of random iteration algorithm trials which hit that pixel.

The random iteration algorithm computes those pixels stochastically, where each trial either hits a pixel or does not, and hence a pixel's hit count obeys the well known binomial distribution with variance $p(1-p)$. Thus assuming the central limit theorem, after n trials our estimate's variance is $p(1-p)/n$. An Agresti-Coull 95% binomial confidence interval has a width of approximately $4\sqrt{p(1-p)/n}$. Thus to halve the image noise, we must compute four times as many samples. Especially with gamma correction, which has a steeper response for darker pixels, this sub-linear convergence rate is a problem in darker areas of the image.

In the deterministic iteration algorithm, by contrast, every pixel receives a density estimate during every image pass. This makes deterministic iteration images smoother, especially in dark regions. However, it may take several iterations before the images begin to converge to the attractor, and for pathological cases may never converge at all. This happens rarely in practice, because repeated geometric scaling drives points to their destinations exponentially fast: points undergoing a scaling factor of s move by s^n after n passes. If s is below unity, a contraction, points converge to the attractor at a well-known exponential rate; even if s is over unity, an expansion, points similarly approach their fixed point of infinity at an exponential rate. In fact, to begin the random iteration algorithm, typically a few dozen "fuse" iterations are performed to converge points to the attractor before beginning to accumulate pixel counts. Experimentally it takes a similar number of image-to-image iterations, typically about a dozen, for the deterministic iteration algorithm to converge, at which point the algorithm is finished.

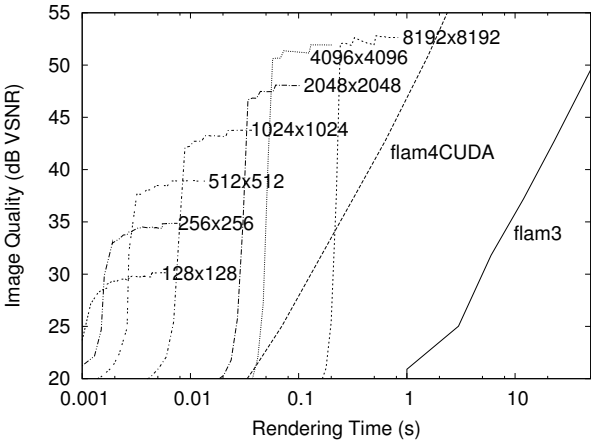


Figure 11. Comparing our algorithm at various texture resolutions with existing GPU and CPU based random iteration approaches. (GeForce 580)

In addition to the exponential difference in convergence rate, the random iteration algorithm produces points stochastically, at unpredictable locations; even ignoring efficiency these random writes are difficult to parallelize correctly. By contrast, the deterministic iteration algorithm produces image samples at every image pixel, which allows the image rendering work to be divided among many parallel processors in a straightforward fashion.

7.2. Quantitative Performance Comparison

Figure 11 compares the performance of our algorithm with two existing random iteration algorithm nonlinear IFS renderers. The vertical axis in this comparison is Chandler and Hemami's Visual Signal to Noise Ratio (VSNR) [27], a perceptually-based image comparison method computed using wavelets and measured in decibels. The IFS used for this comparison is the "swirlpinski" IFS from Figure 16, rendered at 1024×1024 resolution, but other display sizes and rendered IFS appear to display similar relative performance.

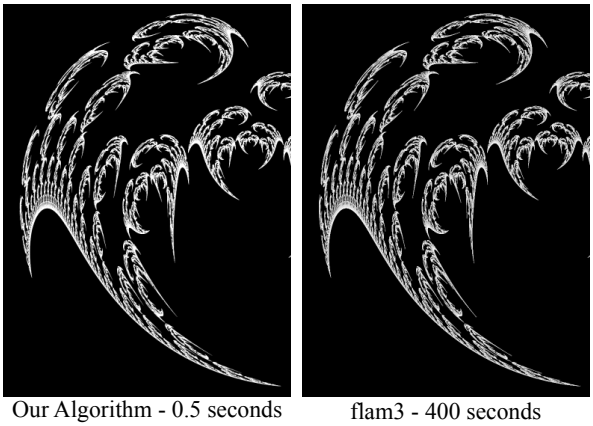
flam3 [7] version 2.8 is a well tuned multicore aware CPU based library for rendering nonlinear IFS using the random iteration algorithm. The CPU is a 3.1GHz

quad-core Intel Core i5 2400, using all four cores. The performance versus accuracy trade-off can be adjusted using the "quality" parameter, which is the number of random iteration algorithm trials per pixel: a quality of 1 executes in under one second, but produces a very noisy stochastic image; while a quality of 1,000,000 takes most of a day to execute but produces an excellent image. Since this is the oldest and most widely used existing nonlinear IFS renderer, we use this renderer with 1,000,000 trials per pixel as the reference implementation.

flam4CUDA [26] is a well tuned CUDA implementation of the random iteration algorithm. The GPU is an NVIDIA GeForce 580 desktop card, the same used for our algorithm below. flam4CUDA uses several quite clever optimizations, such as per-warp random number generation, to synthesize random points while maintaining good branch coherence. One serious inherent shortcoming of the random iteration algorithm approach on highly parallel architectures is a floating point read-modify-update race condition while writing points to the framebuffer. flam4CUDA currently ignores this race condition, because it is not clear how to resolve it efficiently on the GPU. flam4CUDA and flam3 also use slightly different spatial antialiasing (flam4CUDA uses per-pixel jitter) and gamma correction schemes. Finally, the density estimation filter radius is limited by the size of GPU shared memory, so it is difficult to directly compare image outputs. For these reasons, in the figure we generously assume flam4CUDA's output image was identical to the flam3 output image given the same number of trials per pixel. flam4CUDA finishes these same trials approximately 40× faster than flam3, and is performance-competitive with our algorithm for large image sizes, but does not scale down as well to interactive rates.

Our algorithm displays several separate regimes depending on parameters. Small output texture sizes cannot represent the sharp features in the IFS, and hence are limited to low VSNR regardless of run time. Very large images, over 4096×4096, produce only minor improvements in the finished image quality, mostly because we must scale down to compare against the 1024× 1024 reference image. At the crucial 10ms to 50ms interactive animation range, our algorithm can comfortably compute screen-sized textures accurately. Generally, convergence to the final output begins slowly and ends extremely rapidly, with most of the VSNR gains happening in one or two crucial iterations before converging. This rapid convergence, as predicted in the previous subsection, compares favorably against the exponentially slower accumulation process of the random iteration algorithm.

However, to compute extremely high quality images exactly, our algorithm may require very high resolution textures. Note the slight blurring at the left edge of Figure 12, where the swirl map repeatedly shears



Our Algorithm - 0.5 seconds flam3 - 400 seconds

Figure 12. A zoomed-in portion of the swirlpinski fractal, computed using the deterministic and random iteration algorithms.

the texture. To exactly duplicate the results of the 2D random iteration algorithm using IEEE 32-bit floating point arithmetic would require an image of size $2^{32} \times 2^{32}$, using exabytes of storage. For this reason, in some situations very high quality images may be rendered more accurately by the random iteration algorithm, while our algorithm dominates below one second of compute time, the region most useful for image compression or animation applications.

7.3. Vertex vs. Fragment Rendering

Our deterministic iteration algorithm consists of an iterative series of passes where we apply Equation 1 to our textured attractor approximation, slowly improving the approximation. The performance of each pass, using our per-pixel and per-vertex methods at various resolutions is summarized in Table 3. This subsection's performance numbers are presented for the two-map nonlinear IFS shown in Figure 1, on an NVIDIA GeForce GTX 280, a midrange desktop graphics card.

The per-vertex rendering algorithm is substantially slower for small output textures, even for a low vertex resolution of $v = 100 \times 100$ vertices. Both algorithms scale poorly to very small meshes, because the framebuffer object setup time and post-render mipmap building dominate the actual rendering work. The per-vertex algorithm is slightly faster than the per-pixel method at low vertex and high pixel resolutions, because the per-pixel method must consider and discard a large number of pixels that are not touched by the map function output. But at a higher vertex resolution of $v = 1000 \times 1000$, the per-vertex method becomes vertex-rate dominated, so it takes almost exactly the same amount of time to output to a tiny texture as a huge one, and much more time than the per-pixel method in any case. It takes 46.8ms to draw both maps using a mesh of 1 million vertices each, which is 2 million triangles per map or 4 million triangles per pass, a net triangle rate of 85 million triangles per second.

Resolution	Per-Pixel	$v = 100$	$v = 1000$
32×32	0.07ms	0.55ms	46.7ms
64×64	0.08ms	0.56ms	46.7ms
128×128	0.09ms	0.57ms	46.7ms
256×256	0.11ms	0.57ms	46.8ms
512×512	0.17ms	0.58ms	46.8ms
1024×1024	0.41ms	0.59ms	46.8ms
2048×2048	1.31ms	0.99ms	46.8ms
4096×4096	4.97ms	3.44ms	46.8ms

Table 3. GPU time per pass for various texture sizes, as rendered with the per-pixel and per-vertex methods. (GeForce 280)

By contrast, the per-pixel algorithm can make one complete pass through a 4096×4096 texture, applying the plane-to-texture transform, both map functions, the Jacobian density compensation, and our log-density output packing, in under 5ms. This is a total of 3.3 billion color pixels written per second, 6.7 billion map function applications per second, or over 0.3 *trillion* floating-point operations per second—many of which are divides, square roots, exponentials, and logarithms.

7.4. Multigrid Rendering

Table 3 shows that our iterative per-pixel rendering algorithm is much faster per pass when working on small output textures, such as 128×128 pixels. Although we can still discretize the entire plane, such a tiny texture doesn't provide much detail, so it is not useful as a final output. However, depending on the initial attractor estimate, the first few passes will be seriously inaccurate until we begin to converge on the IFS attractor's general shape.

This suggests a multigrid-type approach, where instead of performing all the passes at the highest output resolution, we perform early passes on much smaller textures until we are near convergence, and then we can incrementally increase the texture resolution up to the final size. Table 4 summarizes the performance results from this, where we compare ten passes at each of eight power-of-two multigrid levels, versus eighty passes all at the highest resolution. Multigrid provides the biggest performance improvement, over five fold, for larger output images, but it makes a significant difference even at lower resolutions. For example, at a final output resolution of 2048×2048, the multigrid implementation still runs at well over 30fps, while the fixed-size implementation is under 10fps. As shown in Figure 13, multigrid provides equivalent visual quality much faster than any existing implementation.

Multigrid is surprisingly easy to implement when using OpenGL's default texture coordinates, which run from 0.0 on one edge of the texture to 1.0 on the opposite edge. Because the entire rendering coordinate system is independent of the number of pixels used in either the source or destination texture, we can simply substitute a higher-resolution destination texture to switch from one grid level to the next. Compared to array indices or pixel numbers, which have different values at different resolutions, and yet different values when switching resolutions, resolution-independent coordinates significantly reduce the complexity of a multigrid implementation.

Resolution	Multigrid	Fixed-Size
512×512	10.9ms	13.7ms
1024×1024	14.4ms	32.4ms
2048×2048	26.8ms	105.0ms
4096×4096	76.0ms	398.4ms

Table 4. End-to-end time to generate Figure 1 using multigrid and fixed-size per-pixel rendering, for a constant 80 passes total. (GeForce 280)

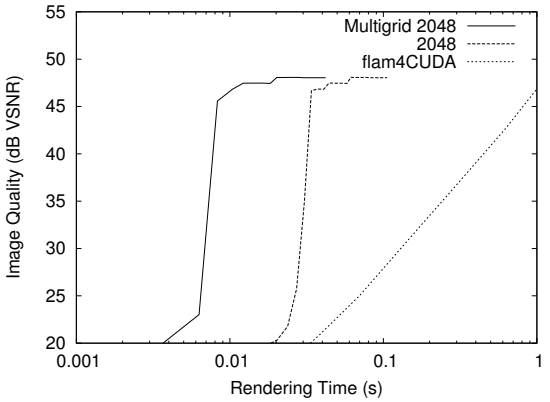


Figure 13. Comparing performance versus visual quality for multigrid, non-multigrid, and random iteration. (GeForce 580)

8. Conclusions and Future Work

We have presented a set of techniques that allow nonlinear function fixed points and iterated function systems to be computed on graphics hardware with extremely high performance. In particular, we have shown how to compress an unbounded IFS into the unit square, so an approximation to the IFS attractor can be stored in a texture. We have shown how analytic map inversion can be used per pixel to iteratively improve this approximation. We have explored how to use a map function's Jacobian determinant to track our discretized attractor density, and how to store that density in a range-limited fixed point texture. Finally, we showed how to use multigrid to accelerate the convergence of our approach. Together, these techniques allow a single GPU to interactively render nonlinear IFS that previously could only be rendered offline on a large distributed cluster.

There is much work remaining to do. Currently we do not exploit frame-to-frame coherence, which could

cut the number of rendering passes required, especially on low-end machines. Multigrid has the advantage that it allows arbitrary animations, including smash cuts and strobe-type effects, but many animations do have significant coherence.

We also currently perform no precomputation, and compute the map functions using only arithmetic. More complex nonlinear functions, or functions whose inverse can only be approximated numerically, would benefit from a discretized version of the map inverse relation, such as a "source coordinate texture" lookup table. A lookup table could have various features folded in, such as an attractor-dependent texture-to-plane function. In addition, Draves has now collected nearly a hundred nonlinear variation functions, of which we only analyzed the first twenty, so it would take significant effort to find analytic inverses for the remaining functions. Thus some simple procedure to approximate or store inverse relations would be useful for backward compatibility.

As with any finite approximation, our attractor texture occasionally displays sampling artifacts. In the center of spirals, where attractor pixels are repeatedly resampled using the hardware's bilinear texture interpolation, the geometry can become somewhat blurred. A resource-intensive solution is to increase the texture resolution, but it seems likely that a higher-order texture interpolant, such as cubic sampling, could produce better results.

Some attractors have important features stored far away in the plane, where our texture resolution is low. This effect is visible on asymptotic spikes, which become blurry near the tips, especially at low texture resolutions. Some sort of intelligence added to the plane-to-texture mapping should be able to reduce this effect. For example, instead of a fixed uniform texture resolution, the attractor could be approximated using adaptive resolution texture tiles.

Our basic per-pixel algorithm generalizes almost trivially to higher dimensions, and 3D GPU textures consisting of voxels are well supported. However, unlike the random iteration method, for our method the storage and computational requirements of higher dimensions quickly become prohibitive; for example, a 2048^2 pixel color image takes only 16 megabytes of storage, while a 2048^3 voxel color volume requires 34 gigabytes. However, even today's graphics cards are capable of comfortably volume-rendering 512^3 voxel volumes at interactive rates, so per-voxel 3D nonlinear iterated function system rendering will eventually become affordable, especially if using adaptive resolution tiled 3D textures.

In this work we have only addressed plain IFS, the simplest form of nonlinear recursive geometry. Yet there are many other more complex procedural models, including recurrent iterated function systems [28], superfractals, escape-time fractals [4], and programmable L-systems. It seems likely that a modified version of the deterministic iteration algorithm could have excellent GPU performance in rendering these other methods to represent fractal geometry. In particular, we would like to explore varying the map functions at each level, for example to mirror the growth cycle of plants.

Finally, we welcome the interested reader to download,[3] extend, and contribute to our open source implementation of this technique.

References

[1] R. F. Williams, Composition of contractions, Bol. Soc. Brasil Mat. 2 (1971) 55–59.

[2] A. N. Tychonoff, Ein fixpunktsatz, Mathematische Annalen 111 (1935) 767–776.

[3] M. F. Barnsley, Fractals everywhere, Morgan Kaufmann, 1993.

[4] S. Nikiel, Iterated Function Systems for Real-Time Image Synthesis, Springer London, 2007. doi:10.1007/1-84628-686-7.

[5] S. Draves, E. Reckase, The fractal flame algorithm, `http://flam3.com/flame.pdf` (2003).

[6] S. Draves, The electric sheep screen-saver: A case study in aesthetic evolution, in: Applications of Evolutionary Computing, Proceedings of EvoMusArt05, 2005.

[7] S. Draves, The electric sheep and their dreams in high fidelity, in: ACM NPAR, 2006, pp. 7–9.

[8] A. Lindenmeyer, Mathematical models for cellular interactions in development, Journal of Theoretical Biology 18 (1968) 280–315.

[9] P. Prusinkiewicz, A. Lindenmayer, The Algorithmic Beauty of Plants, Springer-Verlag, New York, 1990.

[10] T. Ju, S. Schaefer, R. Goldman, Recursive turtle programs and iterated affine transformations, Computers & Graphics 28 (6) (2004) 991–1004.

[11] M. G. Peter Wonka, Raytracing of nonlinear fractals, in: WSCG Plzen 1998 Proceedings, 1998, pp. 424–431.

[12] J. Hutchinson, Fractals and self-similarity, Indiana University Mathematics Journal 30 (5) (1981) 713–747.

[13] J. van Wijk, D. Saupe, Image-based rendering of iterated function systems, Computers & Graphics 28 (6) (2004) 937–943.

[14] E. Gröller, Modeling and rendering of nonlinear iterated function systems, Computers & Graphics 18 (5).

[15] F. Raynal, E. Lutton, P. Collet, M. Schoenauer, Manipulation of non-linear IFS attractors using genetic programming, in: Congress on Evolutionary Computation (CEC'99), IEEE Press, 1999, pp. 1171–1177.

[16] J. C. Hart, T. A. DeFanti, Efficient anti-aliased rendering of 3D linear fractals, Computer Graphics (SIGGRAPH 1991 Proceedings) 25 (1991) 91–100.

[17] J. Rice, Spatial bounding of self-affine iterated function system attractor sets, in: Graphics Interface, 1996, pp. 107–115.

[18] T. Martyn, Tight bounding ball for affine IFS attractor, Computers & Graphics 27 (2003) 535–552.

[19] O. S. Lawlor, J. Hart, Bounding iterated function systems using convex optimization, in: Proceedings of Pacific Graphics 2003, 2003, pp. 283–292.

[20] H.-T. Chu, C.-C. Chen, On bounding boxes of iterated function system attractors, Computers & Graphics 27 (2003) 407–414.

[21] O. S. Lawlor, Impostors for parallel interactive computer graphics, Ph.D. thesis, University of Illinois at Urbana-Champaign (December 2004).

[22] D. Saupe, From classification to multi-dimensional keys, in: Fractal Image Compression - Theory and Applications to Digital Images, Springer-Verlag, 1994, pp. 302–310.

[3] Download the implementation at http://tinyurl.com/gpuifs

[23] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, K. Yelick, Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures, in: Supercomputing '08, IEEE Press, Piscataway, NJ, USA, 2008, pp. 1–12.
[24] D. Monro, F. Dudbridge, Rendering algorithms for deterministic fractals, IEEE Computer Graphics and Applications 15 (1995) 32 – 41.
[25] S. G. Green, GPU-accelerated iterated function systems, in: ACM SIGGRAPH 2005 Sketches, ACM, New York, NY, USA, 2005, p. 15.
[26] S. Brodhead, flam4cuda: Gpu flame fractal renderer, `http://flam4.sourceforge.net/` (2011).
[27] D. M. Chandler, S. S. Hemami, VSNR: A wavelet-based visual signal-to-noise ratio for natural images, IEEE Transactions on Image Processing.
[28] M. F. Barnsley, J. H. Elton, D. P. Hardin, Recurrent iterated function systems, Constructive Approximation 5 (1989) 3–31.

Glossary of Symbols

$\mathbb{RP}^d$	d-dimensional projective space; here $d = 2$.
$[0,1]^c$	The space of c-channel colors; here $c = 3$.
I	The space of all images. $\mathbb{RP}^d \to [0,1]^c$
w_i	A geometric distortion function. $\mathbb{RP}^d \to \mathbb{RP}^d$.
J_{w_i}	The $d \times d$ Jacobian matrix of w_i.
W_i	An image distortion function. $I \to I$.
F	The sum of the W_i. $I \to I$.
A	IFS attractor image, the fixed point of F. In I.
A_k	The kth texture approximation of A.
T	Plane-to-texture compression function.
P	Texture-to-plane function, T^{-1}.
M	An affine transformation: a $d+1 \times d+1$ matrix.

Appendix

Figure 14 shows a slightly reformatted version of the per-pixel GLSL code generated to render Figure 1. On the next page, Figure 15 shows the "fountains" IFS, which has these maps.

$$w_0(x,y)=\begin{bmatrix} x \\ y \end{bmatrix}+\begin{bmatrix} -2.0 \\ 0.0 \end{bmatrix}$$

$$w_1(x,y)=\text{polar}\left(\begin{bmatrix} 1 & 0.5 \\ -1 & 0.5 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix}+\begin{bmatrix} 0.6 \\ -3.0 \end{bmatrix}\right)$$

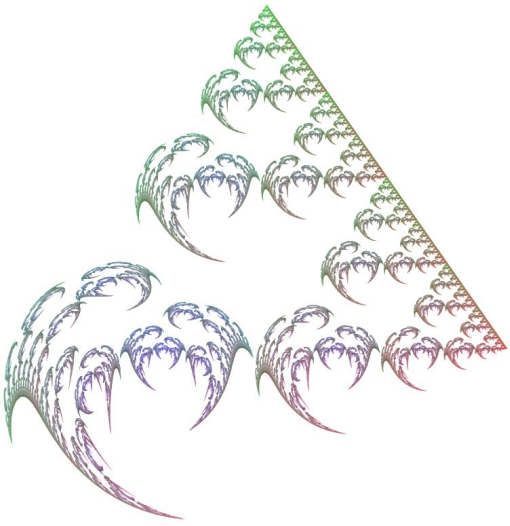
$$w_2(x,y)=1.8 * \text{spherical}\left(\begin{bmatrix} -0.2 & -0.4 \\ 0.4 & -0.3 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix}+\begin{bmatrix} -1.0 \\ -1.3 \end{bmatrix}\right)$$

The spherical and other nonlinear functions used in our IFS maps are defined in Table 2. Figures 16 and 17 show other interesting nonlinear IFS. Figure 18 shows one of Draves' "electric sheep" fractals, one of the few interesting existing sheep that uses invertible map functions. Each of these IFS animate beautifully, an effect that simply cannot be conveyed in print.

```
varying vec2 destcoords; // destination location
uniform sampler2D src; // previous attractor

/*---- convert plane-to-texture -------*/
uniform float texscale,texscalei;
float T(float x) { // plane to texture
  x*=texscale;
  return x/sqrt(1.0+x*x)*0.5+0.5;
}
vec2 T(vec2 p) { return vec2(T(p.x),T(p.y)); }
float P(float s) { // texture to plane
  float u=2.0*s-1.0;
  return texscalei*u/sqrt(1.0-u*u);
}
vec2 P(vec2 s) { return vec2(P(s.x),P(s.y)); }

/*---------------- w0 -----------------*/
uniform vec4 color0;
uniform vec2 w0x,w0y,w0o; // w0's parameters
uniform float w0j,w0v;
float jacobian0(vec2 t) {return 1;}
vec4 f0(vec2 inv) { // runs at each inverse
  float area=1e-2+abs(w0j*jacobian0(inv));
  vec2 t=T(w0x*inv.x+w0y*inv.y+w0o);
  return (exp2(texture2D(src,t)*20)-1)/area;
}
vec4 nonlinear_inverse0(vec2 p) {
  p=p*w0v; // Draves' variation coefficient
  return f0(p);
}

/*---------------- w1 -----------------*/
uniform vec4 color1;
uniform vec2 w1x,w1y,w1o; // w1's parameters
uniform float w1j,w1v;
float jacobian1(vec2 t) {
  float r2=dot(t,t);
  return (1-2*t.y*t.y/r2)/r2;
}
vec4 f1(vec2 inv) { // runs at each inverse
  float area=1e-2+abs(w1j*jacobian1(inv));
  vec2 t=T(w1x*inv.x+w1y*inv.y+w1o);
  return (exp2(texture2D(src,t)*20)-1)/area;
}
vec4 nonlinear_inverse1(vec2 p) {
  p=p*w1v; // Draves' variation coefficient
  float ix = 0.5/p.x;
  float det=1 - 4*p.x*p.x*p.y*p.y;
  if (det>=0) {
    float sq = sqrt(det);
    return f1(vec2(ix*(1 - sq),p.y))
         +f1(vec2(ix*(1 + sq),p.y));
  } else { return vec4(0); }
}

/* Combined inverse-sampling function */
vec4 sum_inverses(vec2 p) {
  vec4 sum=vec4(0);
  sum+=color0*nonlinear_inverse0(p);
  sum+=color1*nonlinear_inverse1(p);
  return log2(sum+1)*(1.0/20);
}

void main(void) {
  gl_FragColor = sum_inverses(P(destcoords));
}
```

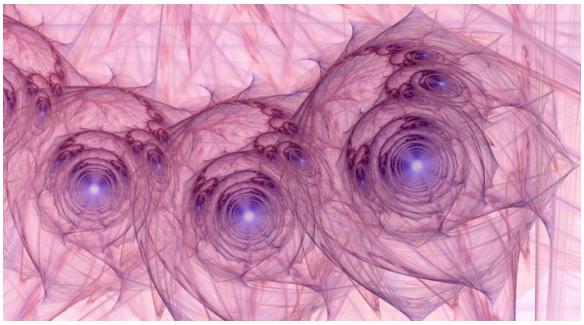
Figure 14. GLSL source code generated to render the Figure 1 IFS using the per-pixel inversion method. An IFS with more or different map functions will generate a different listing. Affine parameters are uniforms, to allow animation.

Figure 15. A three-map “fountains” IFS. All these IFS are rendered on the GPU at interactive rates with the per-pixel method.



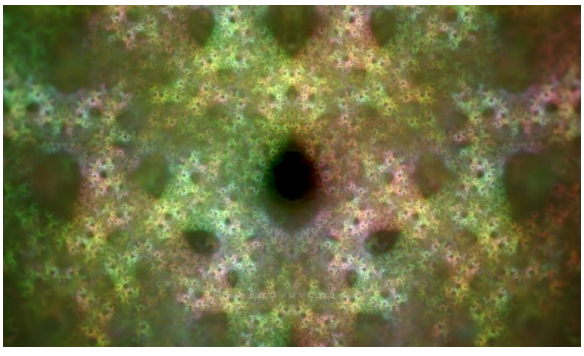
$$
\begin{aligned}
w_0(x,y) &= \text{swirl}\left(0.5 * \begin{bmatrix} x \\ y \end{bmatrix}\right) \\
w_1(x,y) &= 0.5 * \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 1.3 \\ 0 \end{bmatrix} \\
w_2(x,y) &= 0.5 * \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0.4 \\ -1.3 \end{bmatrix}
\end{aligned}
$$

Figure 16. The three-map “swirlpinski” IFS, the same one shown in Figure 6. This is the only IFS in this paper whose attractor is actually bounded–every other IFS we present extends to infinity in at least one direction.



$$
\begin{aligned}
w_0(x,y) &= \begin{bmatrix} 0.8 & -0.4 \\ 0.4 & 0.8 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} -1 \\ 0.3 \end{bmatrix} \\
w_1(x,y) &= 0.5 * \text{fisheye}(x,y) \\
w_2(x,y) &= 3.2 * \text{hyperbolic}(x-y, x+y)
\end{aligned}
$$

Figure 17. A three-map “eyes” IFS.



$$
\begin{aligned}
w_0(x,y) &= 1/3 * \text{spherical}\left(0.5 * \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} -0.566 \\ 0.4 \end{bmatrix}\right) \\
w_1(x,y) &= 1/3 * \text{spherical}\left(0.5 * \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} +0.566 \\ 0.4 \end{bmatrix}\right) \\
w_2(x,y) &= 1/3 * \text{spherical}\left(0.5 * \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ -0.551 \end{bmatrix}\right)
\end{aligned}
$$

Figure 18. Draves’ fractal generation 165, sheep 48.