The Fractal Flame Algorithm

The *Fractal Flame* algorithm is a member of the Iterated Function System (IFS) class of fractal algorithms. A two-dimensional IFS creates images by plotting the output of a chaotic system directly on the image plane. The fractal flame algorithm is distinguished by three innovations over text-book IFS: non-linear functions, log-density display, and structural coloring. In combination with standard techniques of anti-aliasing and motion blur the result is striking image variety and quality. Some examples appear in Figure 1.

The guiding principle of the design of the algorithm is to expose and preserve as much of the information content of the attractor as possible. I found that preserving information maximizes aesthetics.

The paper begins by defining classic, linear iterated function systems and hence grounding our notation and terminology. The classic formulation is then extended with non-linear variations in Section 2. Section 3 describes how log-density display works and why it's important, and Section 4 covers the coloring algorithm. These three sections cover the core innovations. Sections 5 and 7 explain other important properties of the implementation, and Section 6 shows how to create symmetric flames. We conclude with Section 8 on the history and availability of the implementation.

1. Classic Iterated Function Systems

A two-dimensional Iterated Function System (IFS) is a finite collection of n functions F_i from R^2 to R^2 . The solution of the system is the set S in R^2 (and hence an image) that is the fixed point of the recursive set equation:

$$S = \bigcup_{0 \le i < n} F_i(S)$$

In the formulation defined by Hutchinson [XXX] and popularized by Barnsley [XXX] the functions F_i are linear (technically they are affine as each is a two by three matrix capable of expressing scale, rotate, translate, and shear):

$$F_i(x,\ y) = (a_i\, x + b_i\, y + c_i,\ d_i\, x + e_i\, y + f_i)$$

In order to facilitate the proofs and guarantee the algorithms, the functions are constained to be contractive, that is, to bring points closer together. In fact the normal algorithm works under the much weaker condition that the whole system is contractive *on average*, though we have not formalized this notion. In my practice, no guarantee is given or extracted. Some parameter sets result in better images than others, and some result in degenerate images. The random parameter set generator has a heuristic that avoids the worst, and the rest is left to human discrimination.

The normal algorithm for approximating S is called the *chaos game*. In pseudocode it is:

```
(x, y) = a random point in the biunit square
iterate {
  i = a random integer from 0 to n − 1 inclusive
  (x, y) = Fi(x, y)
  plot (x, y) except during the first 20 iterations
}
```

The biunit square are those points where x and y are in [-1, 1].

This works because if $(x,\ y) \in S$ then $F_i(x,\ y) \in S$ too. Though we start out with a random point, because the functions are on average contractive the distance between the solution set and the point decreases exponentially. After 20 iterations with a typical contraction factor of

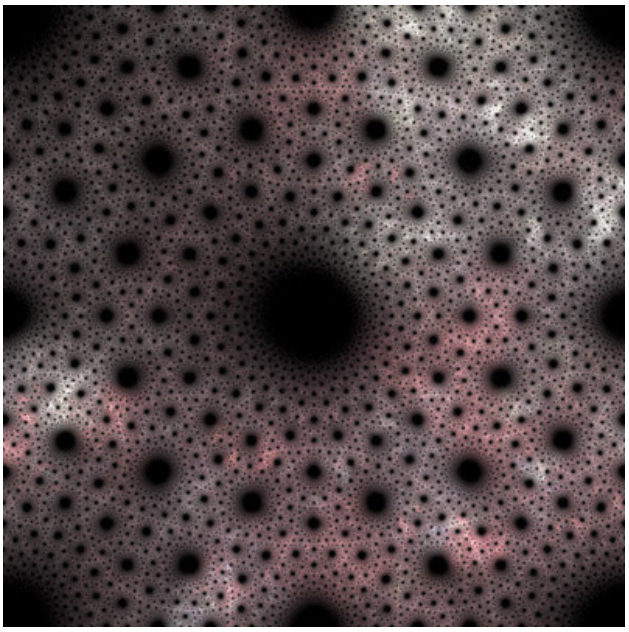
0.5 the point will be within 10^{-6} of the solution, much less than a pixel's width. Every point of the solution set (or one arbitrarily close to it) will be generated eventually because an infinite string of random symbols contains every finite substring of symbols. This proof is given in more complete form in Section XXX.YYY of [ZZZ].

No fixed number of iterations is given by the algorithm. Because the chaos game operates by stochastic sampling, the more iterations one makes the closer the result will be to the exact solution. The judgement of how close is close enough remains for the user. In my implementation, the number of samples is specified with the more abstract parameter *quality*, so that image quality as measured by decibels of noise remains constant in face of changes in resolution, zooming, or oversampling.

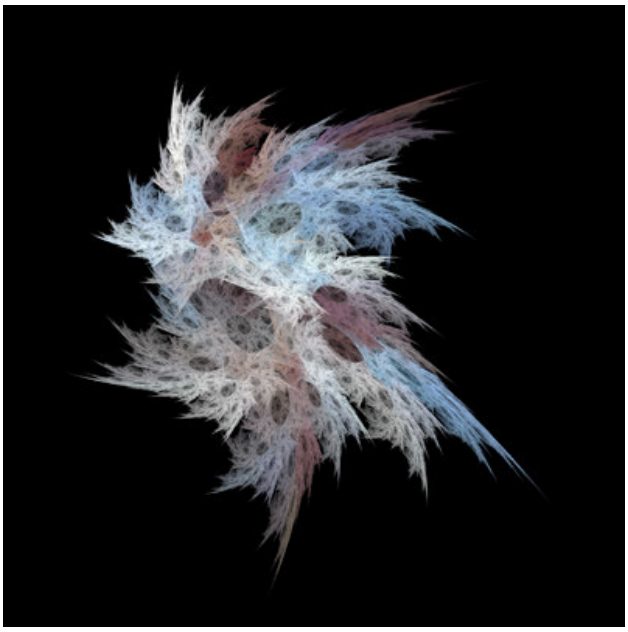
For example, if the functions are

$$F_0(x, y) = \left(\frac{x}{2}, \frac{y}{2}\right) \qquad F_1(x, y) = \left(\frac{x+1}{2}, \frac{y}{2}\right) \qquad F_2(x, y) = \left(\frac{x}{2}, \frac{y+1}{2}\right)$$

the result is Sierpinski's Gasket, as seen in Figure 2.



206, spherical



149, sinusoidal

102, swirl

191, horseshoe

Figure 1. Example fractal flame images. The numbers are just for identification and indicate the order in which they were discovered. The name is of the variation as described in Section 2. These images were selected for their esthetic properties.

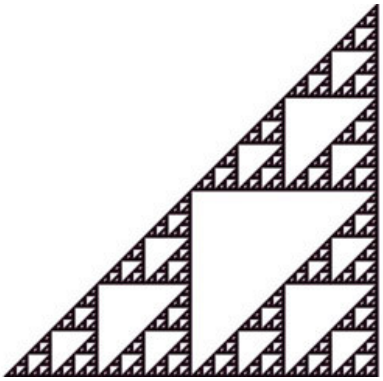


Figure 2. Sierpinski's Gasket, a prototypical linear IFS. The axes increase towards the lower-right of the image.

It is useful to be able to weight the functions so they are not chosen with equal frequency in line 3 of the chaos game. We assign a weight, or relative probability w_i to each function F_i . This is useful for interpolation between function systems with different numbers of functions: the additional functions can be phased in by starting their weights at zero. Differently weighted functions are also necessary for symmetric flames as shown in Section 6.

Some implementations of the chaos game make each function's weight proportional to its contraction factor. When drawing one-bit per pixel images this has the advantage of converging faster than equal weighting because it avoids redrawing pixels. But in our context with mutiple bits per pixel and non-linear functions (where the contraction factor is not constant) this device is best avoided.

2. Non-Linear Variations

We now generalize this algorithm. The first step is to use a larger class of functions than just affine functions. Composing a non-linear function V_j from R^2 to R^2 with the affine functions changes the shape and character of the solution:

$$F_i(x,\ y) = V_j(a_i\, x + b_i\, y + c_i,\ d_i\, x + e_i\, y + f_i)$$

We call each such function a *variation*. The current flame implementation has 14 variations. See Appendix A for a complete description of all the variations. Variation 0 is the identity function. Four more examples are:

$$V_1(x, y) = (\sin x, \sin y) \qquad \textit{sinusoidal}$$

$$V_2(x, y) = \left(x / r^2, y / r^2\right) \qquad \textit{spherical}$$

$$V_3(x, y) = (r\cos(\theta + r), r\sin(\theta + r)) \qquad \textit{swirl}$$

$$V_4(x, y) = (r\cos(2\theta), r\sin(2\theta)) \qquad \textit{horseshoe}$$

where

$$r = \sqrt{x^2 + y^2}$$
$$\theta = \text{atan}(y / x)$$

It is very useful to be able to animate one fractal transforming into another. Such continuous interpolation between arbitrary parameter sets requires another generalization: replacing the integer parameter j with a blending vector of length j. Then

$$F_i(x, y) = \sum_{0 \le j < 14} v_{i\,j} V_j(a_i x + b_i y + c_i, d_i x + e_i y + f_i)$$

where

$$\sum_j v_{i\,j} = 1$$

3. Log-Density Display

The chaos game produces a series of (x, y) points which are plotted on the image plane. The collection of these points approximates the solution S to the iterated function system. S is a subset of the plane, and hence membership is a binary function, and the image is therefore black and white, lacking even shades of gray. See Figure 3a for an example.

Information is lost every time we plot a point that has already been plotted. A more interesting image can be produced if we render a histogram of the chaotic process, that is, increment a counter at each pixel instead of merely plotting. These counters can be visualized by mapping them to shades of gray or by using a color-map that represents different densities (larger count values are more dense) with different colors. A linear mapping of counters to gray values results in Figure 3b.

The result is unsatisfying because of the large dynamic range of densities. Like many natural systems, the densities are often distributed according to a power law (frequency is inversely proportional to value). Figure 4 has two histograms demonstrating this. The densest points are much denser than the average density, hence with a linear map most of the image is very dark, and information is lost.

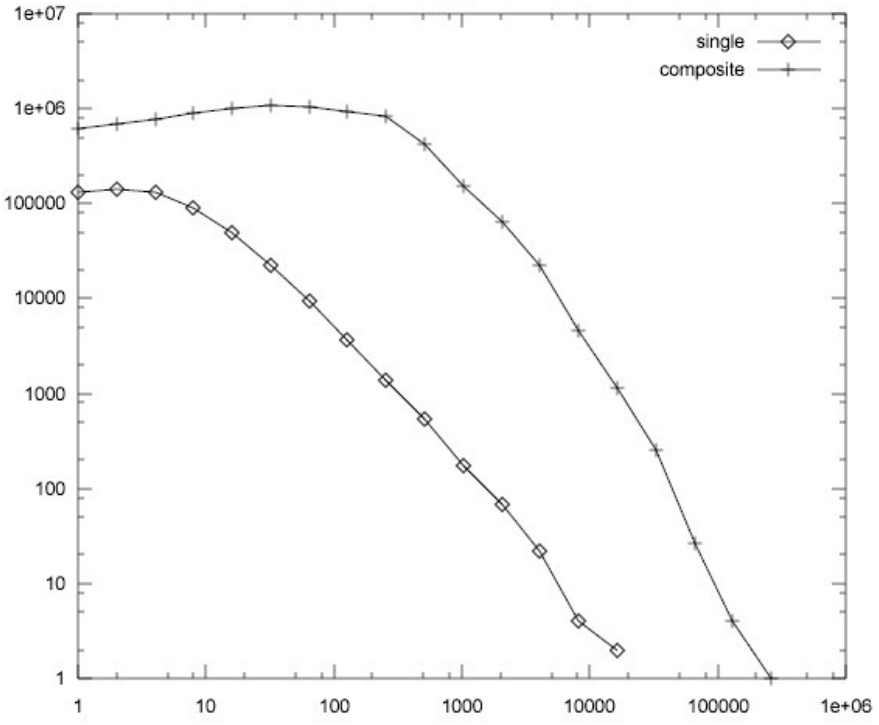


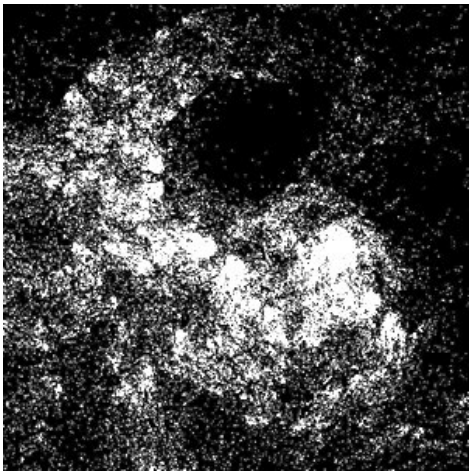
Figure 4. Plots showing that the distribution of densities in an IFS follows the power law. The density (on the horizontal axis) is the number of hits by the system in a pixel, the frequency (on the vertical axis) is the number of pixels with that density (or up to the next power of two). The line is from the image in Figure 3, the other is the composite of 19 favorite systems including Figure 3. In each case, after a plateau the graph is nearly a straight line in log-log space. This is the definitive characteristic of a power law distribution.

The flame algorithm solves the problem by using a logarithmic map from density to brightness. See Figure 3c for the result. The logarithm allows one to visually distinguish between densities of 3,000 and 5,000 in one part of the image and 30 and 50 in another part.

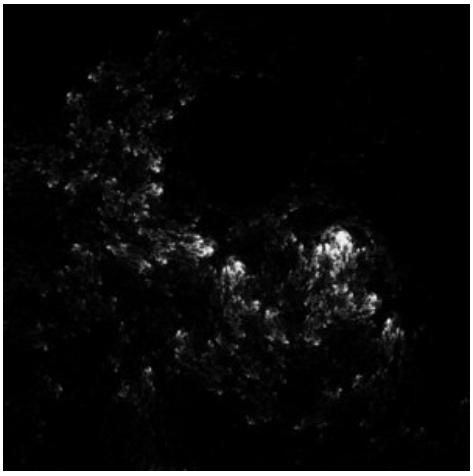
This log-density mapping is the source of a 3D illusion. On sight people often guess that fractal flames are rendered in 3D, but as just described the algorithm works strictly with a 2D buffer.

However, where one branch of the fractal crosses another, one may appear to occlude the other if their densities are different enough because the lesser density is inconsequential in sum. For example branches of densities 1000 and 100 might have brightnesses of 30 and 20. Where they cross the density is 1100, whose brightness is 30.4, which is hardly distinguishable from 30.

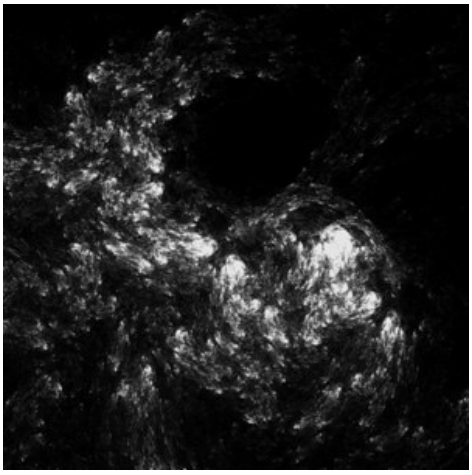
The display of high dynamic range images is a sub-field of computer graphics unto itself [XXX]. Its techniques such as tone mapping may be more effective than using a logarithm.



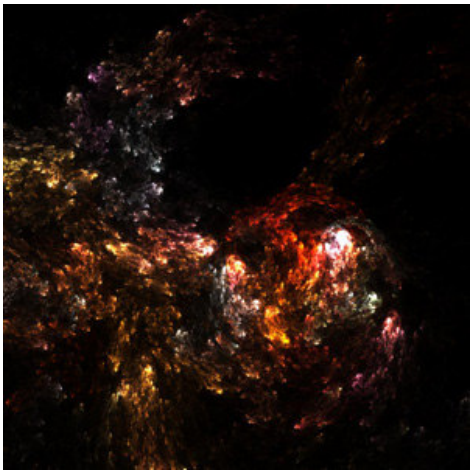
a) binary



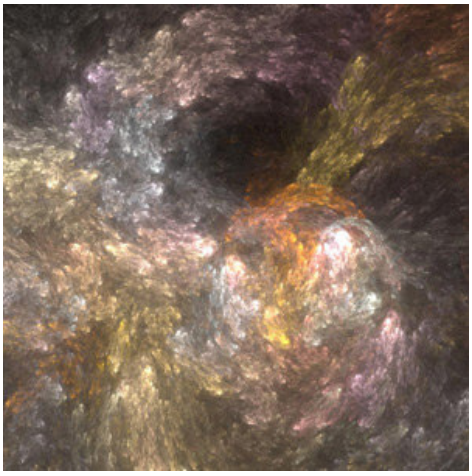
b) linear



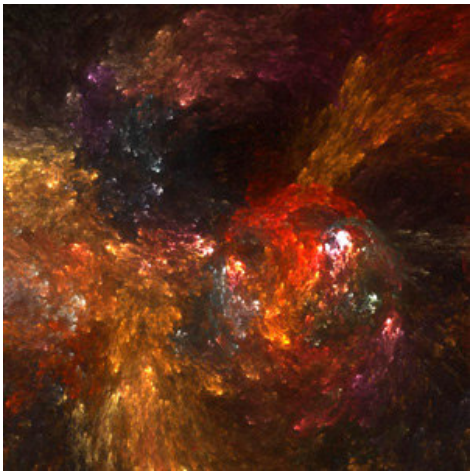
c) logarithmic



d) in color



e) with gamma



e) vibrant

Figure 3. Successive refinements of the rendering technique. The parameters to create this image are given in Appendix B.

4. Coloring

There is more information to be wrung from the attractor. In particular, which parts of the fractal come from which functions? The flame algorithm uses color to convey this. The result is a substantial aesthetic improvement. Color could be assigned according to the density map

and although the result is increased visibility of the densities relative to grayscale, the internal structure of the fractal remains opaque. Furthermore, for animation the eye prefers that the color of each part of the attractor remain unchanged over time, otherwise the illusion of an object in motion is compromised. The fractal flame algorithm uses an original means to accomplish this: adding a third coordinate to the iteration.

Naturally we want to use a palette or color-map which we define as a function from [0,1] to (r,g,b) where r, g, and b are in [0,1]. A palette is classically specified with an array of 256 triples of bytes. I have found that continuous color-maps work best, that is, colors that are near each other in index should also be near each other in color space.

We assign a color c_i to each function F_i and add an independent coordinate to the chaos game:

```
(x, y) = a random point in the biunit square
c = a random number in [0,1]
iterate {
  i = a random integer from 0 to n − 1 inclusive
  (x, y) = F_i(x, y)
  c = (c + c_i) / 2
  plot (x, y) with color c except during the first 20 iterations
}
```

This has the important property that the most recently applied function makes the largest difference in the color index and also in spatial location. Indices make less difference as they recede in time. Hence colors are continuous in the final image.

If there were three or fewer functions in the system it would be possible to relate each function to a color channel (red, green, or blue), or more generally to assign a different color to each function. But using a palette provides more variation, rather than a mostly gray blur. It also makes importing existing palettes convenient.

Color plotting is naturally implemented by keeping three counters per pixel instead of one, and adding the current color to the three of them instead of incrementing a single density counter. That is not enough information for proper log-density display, however. Taking the logarithm of each channel independently grossly alters the colors. Instead one must add a fourth channel of so-called alpha (α), or transparency values.

So then to plot a point the color is added to the three color channels, and 1 is added to the alpha channel. After all the samples have been accumulated each channel is scaled by $(\log \alpha) / \alpha$. See Figure 3d for the result. The resulting alpha values can be output with the image if the file format supports them, or they can be used for compositing the fractal with a background immediately, or they can be discarded.

5. Gamma Correction

Accurate display of any digital image on a Cathode Ray Tube (CRT) requires gamma correction, that is, accounting for the non-linear response of screen phosphors to voltage [XXX]. Without correction the darker parts of an image appear too dark. If the brightness b of a pixel is a value between 0 and 1, the corrected brightness is simply:

$$b_{\text{corrected}} = \sqrt[\gamma]{b}$$

where γ is normally about 2.2. I have found that depending on the specific image, gamma values as large as 4 improve visibility of fine structure in the attractor. Large gamma values also substantially increase the visible noise, and so require longer rendering times to compensate. The parameter is therefor left to the user to set according to taste and circumstance. See Figure 3e for the result of applying gamma 4 to our running example.

Some operating systems and image display programs automatically gamma-correct images, but more likely than not, it must be handled by the application. In fact this is advantageous to us because images are normally quantized to 8 bits when they are stored or displayed. Gamma correction after this quantization results in the loss of 4 bits of resolution at the low end! My implementation does the correction on internal 16-bit buffers, hence no resolution is lost (at gamma 2 on an 8-bit display).

Gamma correction is normally applied to each color channel independently. However if the gamma value is unnaturally large this has the effect of washing out the colors of the image (it becomes pastel or ghostly off-white). This is because saturated colors occur due to large difference between channel values. But gamma correction boosts any small values towards one, leaving less difference, and hence less saturation.

While one may desire this effect, to preserve bright colors the gamma correction may be applied the same way the logarithm is: by calculating a scale factor on the alpha channel, and then applying it to the three color channels.

The name of the parameter that selects this is called *vibrancy*, and it can take any value from 0, meaning to apply gamma to channels independently, to 1, meaning to apply gamma from the alpha channel to each channel.

6. Symmetry

The human mind responds to symmetric designs at a fundamental level. When the matrix coefficients are chosen at random, the chances of a symmetric design appearing are vanishingly small. We can easily inject such functions intentionally, however. The result appears in Figure 5.

There are two kinds of symmetries: rotational and dihedral. First we cover rotations. Adding a function to the system that rotates by 180 degrees makes a 2-way rotational symmetry appear. The weight of this function should be equal to the sum of the weights of all other functions in the system. That way half of the jumps in the chaos game are between the two halves, and hence the two halves will have equal density. If the the rotation function is given the same weight as the other functions, then one of the two halves will only be a shadow of the other.

Adding a rotation by 120 degrees does make a 3-way symmetry appear. The three branches do not have equal density however, and no weighting would balance them. That's because in order to get into the 240-degree branch in the chaos game, one has to pick the rotational function twice in a row, which is only 25% probable, but the 120-degree branch is 50% probable.

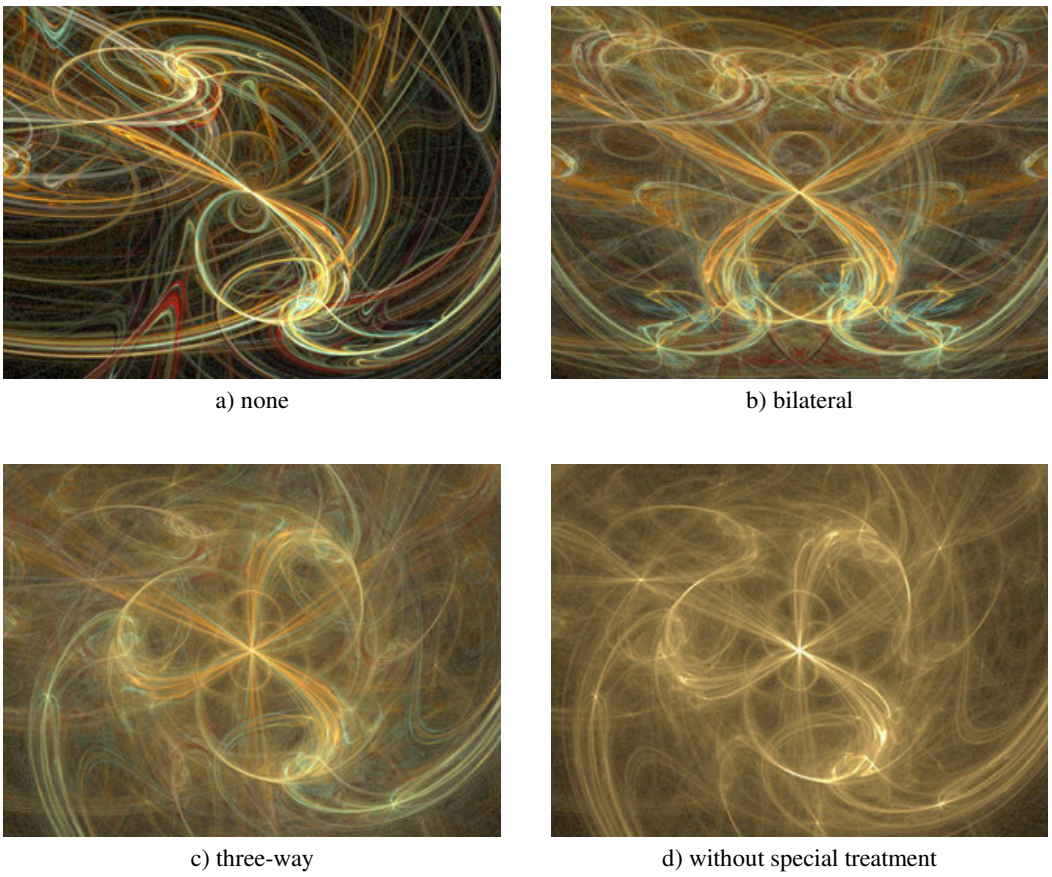
Instead one must introduce two transforms, one by 120 degrees and one by 240, and give them both weight equal to the sum of the others. Then all three branches will have the same probability. In general, to produce n-way symmetry, n-1 additional transforms are necessary to balance the densities.

A dihedral symmetry is created by adding a function that inverts the x coordinate only (producing bilateral symmetry). Again it is given weight equal to the sum of all the other weights combined. Combining this function with rotation functions gives all the dihedral symmetries. Dihedral symmetries are named with negative integers, so the simple bilateral symmetry is -1, and snowflake symmetry is -6. This just follows the isomorphism between multiplication on integers and composition of symmetries.

It is also possible to introduce symmetries by modifying the chaos game to support them directly instead of adding symmetric functions. For example, one can just add an integer symmetry parameter, and then after picking a function at random, pick a rotation at random. But then how to interpolate between symmetries without discontinuities is problematic.

Getting good colors with the symmetry functions requires special treatment. The problem is the symmetry functions can bring a point back onto itself after two or more applications. If the color is modified by these functions and then plotted at its original location, different colors get

averaged, and the image loses color diversity. The solution is to not change the color coordinate when applying symmetry functions. This also makes the colors symmetric as well as the shape.



a) none b) bilateral

c) three-way d) without special treatment

Figure 5. Examples of symmetry. Image d shows how the colors wash out without special treatment of the color coordinate for symmetry transforms.

7. Filtering and Motion Blur

Aliasing in spatial and temporal directions is visually disturbing and also indicates information loss because one cannot tell if an artifact in the image is an original or an alias. With anti-aliasing, there is no ambiguity.

The chaos game lends itself to anti-aliasing. Consider spatial aliasing first, that is the elimination of the jaggie edges. The normal technique is to draw the desired image at high resolution, and then filter it down to display resolution. This is known as over-sampling, and its cost is linear in time and memory (though the sampling is normally applied in two dimensions, so 3 by 3 over-sampling means 9x). With the chaos game however we can achieve this effect by just increasing the number of buckets used in the histogram without increasing the number of iterations. There is a substantial though sublinear cost in time though because the time spent filtering is significant, and the increased memory usage also means increased cache misses during iteration. The effect on visual quality, however, is dramatic. Gamma correction should be done at this filtering step, when the maximum number of bits of precision is still available.

Despite this filtering, the logarithm and gamma factor may cause low-density parts of an image to appear dotted or noisy. A wider filter would solve this, but at the expense of making the bright parts of the image blurry. Some form of density estimation [XXX] may solve this problem. One such technique is to make the filter width variable and inversely proportional to the sample density, or more precisely, making the filter big

enough to enclose a fixed number of samples.

Motion blur, or temporal anti-aliasing, is not quite so easy. One may hope that over-sampling could be achieved by just varying the parameters over time while running the chaos game. That is, if 5M samples are used to draw the frame at time t, to instead use 1M samples at time t-0.5, 1M samples at t-0.25, 1M at t, 1M at t+0.25 and 1M at t+0.5, all the while accumulating into the same buffer. That would be 5x oversampling for free! With linear density display, this would work.

The non-linearity of the logarithm complicates things. One must use an extra buffer: the first buffer is linear and accumulates the histogram. After each temporal sample, take the logarithm of this buffer and accumulate it into the second one. After all samples are completed, the second buffer is filtered down into the final image.

8. History and Acknowledgements

In 1986 at Brown University Bill Poirier showed the author what he called "Recursive Pictures", his own reinvention of two-dimensional IFS. Poirier used a formulation that included perspective transforms, but lacked a software implementation. In response the author created the first of many IFS implementations.

The first implementation to include all three definitive characteristics (non-linear variations, log-density display, and structural coloring) was created in the summer of 1991 while the author was an intern at the NTT-Data Corporation in Tokyo, Japan and was generously allowed to pursue his own projects.

That version was released on the then-nascent world wide web in 1992 under the GPL, an open source licence [GPL], and it remains available as such [XXX]. It has since been incorporated into and ported to many environments, including: the Gimp, Photoshop (as Kai's Power Tools FraxFlame), After Effects, Digital Fusion, Ultra Fractal 3, screensavers for Macintosh, Windows, and Linux, as well as stand-alone programs (Apophysis).

The combined-channel gamma feature and the vibrancy parameter that controls it were introduced in 2001. The symmetries were introduced in 2003. Variations 7 to 12 were developed by Ronald Hordijk for his screen-saver version of the flame algorithm, then ported to the Ultra Fractal version by Erik Reckase, and adopted into the original version (with some modifications) by the author in 2003. Thanks to Hector Yee for suggesting tone mapping as the general solution to the dynamic range problem, and David Hart for suggesting density estimation as an improved filter technique.

The fractal flame algorithm is also the seed that spawned the Electric Sheep distributed screen-saver [YLEM], a follow-on art project by the author. In this system, thousands of idle computers from all over the world are harnessed into rendering (and evolving) fractal flames. The work of all participating clients is shared alike.

Appendix A: Catalog of Variations

For each variation, we give the formula, its descriptive name, a grid visualization, and an example final image.

The formulas have been designed to keep the action in the biunit square.

$$V_0(x, y) = (x, y) \qquad \textit{linear}$$

$$V_1(x, y) = (\sin x, \sin y) \qquad \textit{sinusoidal}$$

$$V_2(x, y) = \left(x / r^2, y / r^2\right) \qquad \textit{spherical}$$

$$V_3(x, y) = (r\cos(\theta + r), r\sin(\theta + r)) \qquad \textit{swirl}$$

$$V_4(x, y) = (r\cos(2\theta), r\sin(2\theta)) \qquad \textit{horseshoe}$$

$$V_5(x, y) = (\theta / \pi, r - 1) \qquad \textit{polar}$$

$$V_6(x, y) = (r\sin(\theta + r), r\cos(\theta - r)) \qquad \textit{handkerchief}$$

$$V_7(x, y) = (r\sin(\theta r), -r\cos(\theta r)) \qquad \textit{heart}$$

$$V_8(x, y) = (\theta\sin(\pi r) / \pi, \theta\cos(\pi r) / \pi) \qquad \textit{disc}$$

$$V_9(x, y) = ((\cos\theta + \sin r) / r, (\sin\theta - \cos r) / r) \qquad \textit{spiral}$$

$$V_{10}(x, y) = ((\sin\theta) / r, (\cos\theta) r) \qquad \textit{hyperbolic}$$

$$V_{11}(x, y) = ((\sin\theta)(\cos r), (\cos\theta)(\sin r)) \qquad \textit{diamond}$$

$$V_{12}(x, y) = \left(r\sin^3(\theta + r), r\cos^3(\theta - r)\right) \qquad \textit{ex}$$

$$V_{13}(x, y) = \left(\sqrt{r}\cos(\theta / 2 + \Omega), \sqrt{r}\sin(\theta / 2 + \Omega)\right) \qquad \textit{julia}$$

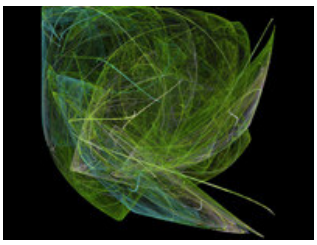
where

$$r = \sqrt{x^2 + y^2}$$
$$\theta = \text{atan}(y / x)$$

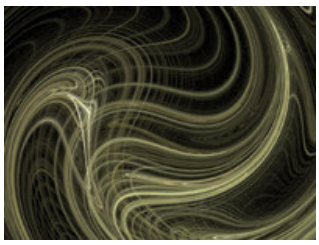
and Ω is a random variable that is either 0 or π. This random value just selects one of the two possible branches of the complex square root function that this variation really is.

Note that many of these formulas have singularities in them. Those resulting from division are avoided by adding a small constant to the denominator. The atan function has a singularity at the origin. In this case it just returns 0.

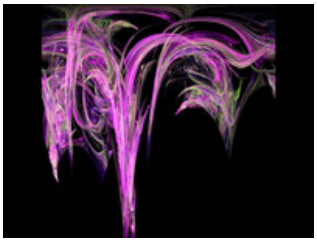

0. linear

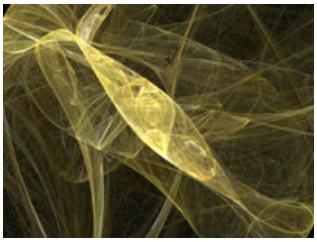

1. sinusoidal


2. spherical

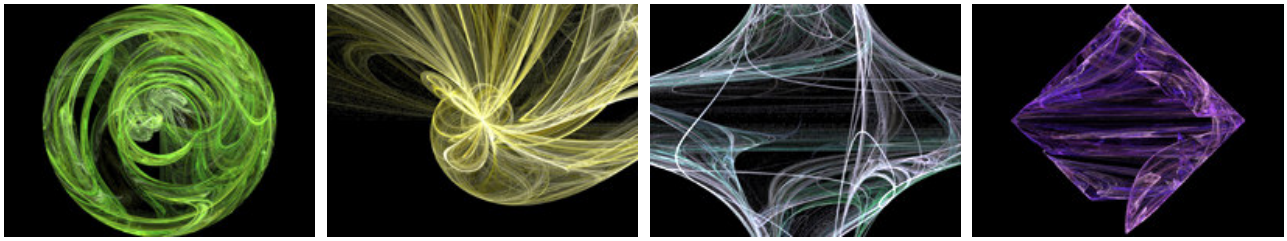

3. swirl


4. horseshoe

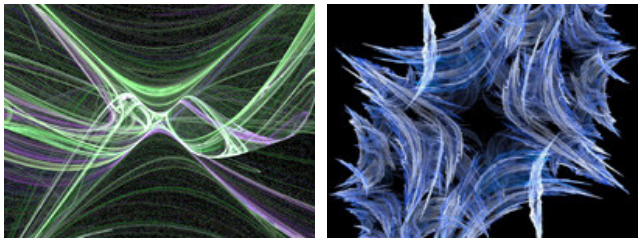

5. polar


6. handkerchief


7. heart

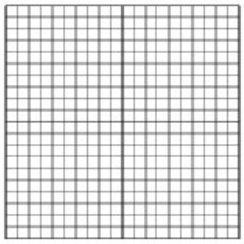


8. disc 9. spiral 10. hyperbolic 11. diamond

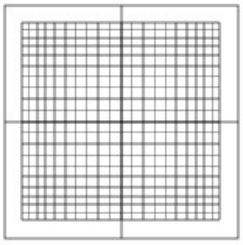


12. ex 13. julia

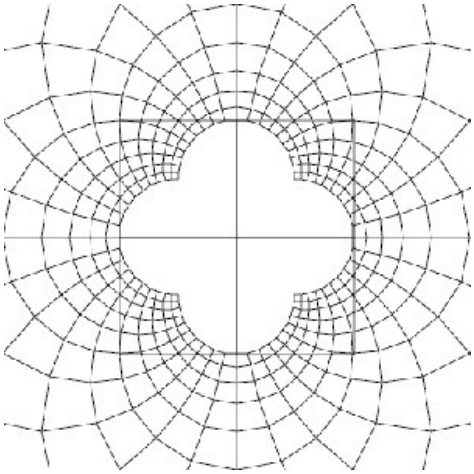
Figure 6. Examples of each of of the variations. The images were selected for being characteristic rather than for their aesthetic qualities.



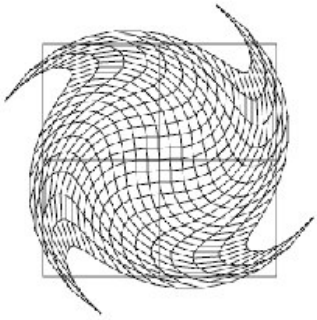
0. linear



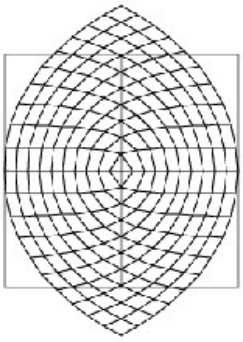
1. sinusoidal



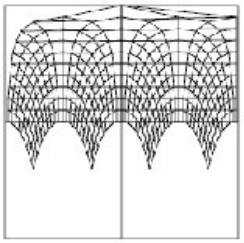
2. spherical



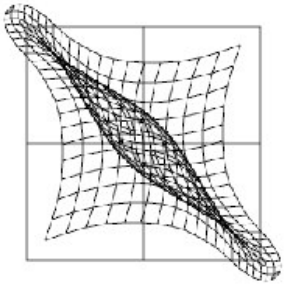
3. swirl



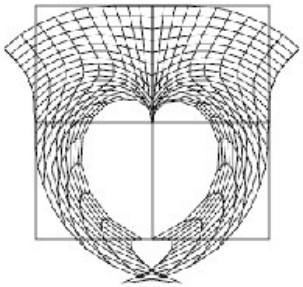
4. horseshoe



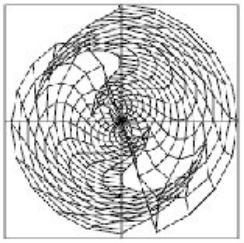
5. polar



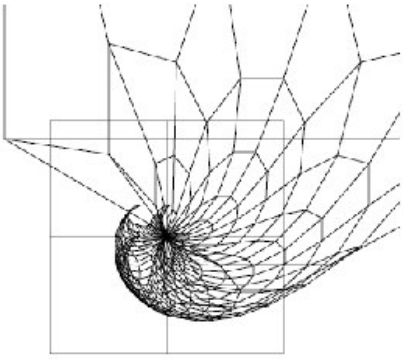
6. handkerchief



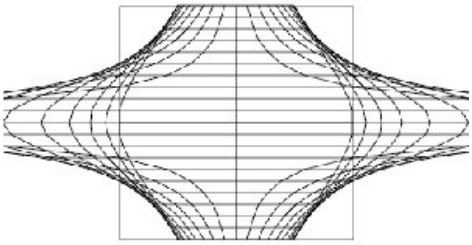
7. heart



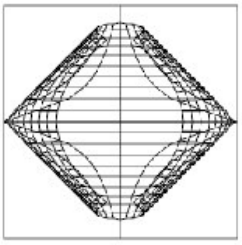
8. disc



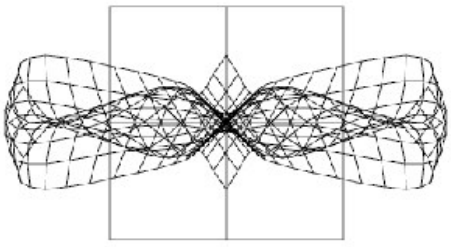
9. spiral



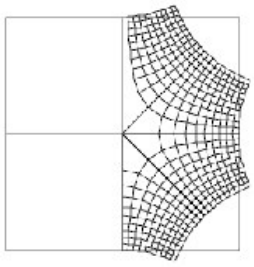
10. hyperbolic



11. diamond



12. ex



13. julia

Figure 7. Grid visualizations of each variation. The image of a grid covering the biunit square is shown. Notes: for variation 4 the grid wraps around the origin twice. For variation 13 only the half of the image corresponding to $\Omega = 0$ is shown. For variations 7, 8, and 13 the lines that cross the discontinuity of the atan function are broken at that point.

Appendix B: XML file format

This section documents the file format used by the standard implementation to specify individual images and animations. The XML meta-format is used to facilitate predictable parsing and compatibility. No formal DTD is available, but the important part is the semantics of the recognized elements and attributes, and those are given here. For example Figure 2e is produced by:

```
<flame time="0" palette="30" hue="0" size="300 300" center="0 0" scale="100"
       zoom="0" oversample="3" filter="1" quality="10" batches="1"
       background="0 0 0" brightness="5" gamma="4" vibrancy="0">
   <xform weight="0.5" color="1" var1="2"
          coefs="0.562482 0.397861 -0.539599 0.501088 -0.42992 -0.112404"/>
   <xform weight="0.5" color="0" var1="2"
          coefs="0.830039 -0.496174 0.16248 0.750468 0.91022 0.288389"/>
</flame>
```

and Figure 4c by:

```
<flame time="0" palette="77" size="320 240" center="0 0" scale="120"
       zoom="0" oversample="3" filter="1" quality="10" batches="10"
       background="0 0 0" brightness="4" gamma="4" vibrancy="1" hue="0">
   <symmetry kind="3"/>
   <xform weight="0.333" color="1" var1="0"
          coefs="0.98396 0.298328 0.359416 -0.583541 -0.85059 -0.378754"/>
   <xform weight="0.333" color="0" var1="9"
          coefs="-0.900388 0.293063 0.397598 0.0225629 0.465126 -0.277212"/>
   <xform weight="0.333" color="0.5" var1="3"
          coefs="-0.329863 -0.0855261 -0.369381 -0.858379 0.977861 0.547595"/>
</flame>
```

The flame element corresponds to a parameter set for a single IFS image as described in this paper. The *hqi* (an abbreviation of hiqh quality image) program reads flame elements from its input and renders each one as an output image. The *anim* program reads flame elements and interprets them as key-frames for an animation. It renders one output image per integer frame number, starting at time 0 and ending at the time of the last key-frame minus one. It interpolates values linearly in-between key-frames. The attributes of the flame element are:

name	type	description
time	float	The frame number when these parameters are realized. This is ignored by hqi.
palette	integer	One of 90 standard color-maps. The colors in the palette are modified by the hue attribute.
size	two positive integers	The width and height in pixels of the output image.
center	two floats	The coordinates of the center of the output image.
scale	float	The number of pixels per unit in the image plane. Together the size, center, scale, and zoom attributes specify the camera that maps the image plane to output image.
oversample	positive integer	The degree of spatial oversampling on both axes. Memory use is quadratic in this parameter. Typically less than 4.
filter	non-negative float	The radius in pixels of the the gaussian filter used to filter the oversampled image down to the output image where edge of the gaussian is arbitrarily drawn at 2.5 standard deviations (1.25%) from the center.
quality	positive float	The number of samples per pixel on average.
batches	positive integer	The number of temporal sub-samples, though also useful for preventing clamping even when not doing animation. Larger values reduce the resolution of the logarithm.
hue	float	Added to the hue component of the colors from the palette. 0.5 means to rotate 180 degrees in color space.
zoom	float	Scale the image (the camera, not the number of pixels) by this power of two. Also scales the quality to compensate.
background	3 integers	The color of the background as an RGB triple with each component in [0, 255].
brightness	positive float	Linear scale on luminance.
gamma	positive float	The inverse exponent. Values larger than one brighten the dark parts of the image. Usually between 2 and 4.
vibrancy	float	Blend between applying gamma to each channel independently (zero) for pastel and ghostly white images, and applying gamma from the alpha channel to each channel (one) for saturated colors.

Xform, symmetry, and color elements may appear inside the flame element. An xform element specifies a function in the function system, and a symmetry element specifies a collection of functions as described above. Color elements provide finer control over the colors than the palette attribute. The attributes of the xform element are:

weight	non-negative float	The relative probability that this function will be chosen by the chaos game.

color	float	The color index associated with this function, normally in [0,1].
var	up to 14 floats	The variational coefficients $v_{i\,j}$ for this function i. Any coefficients not given are assumed to be zero.
var1	integer	Shorthand for a var attribute with all coefficients zero except the one specified is one. So var1="2" is equivalent to var="0 0 1 0 0 ..." and var1="0" to var="1 0 0 ...".
coefs	6 floats	The coefficients for the affine part of the function, in row order (a d b e c f).

The symmetry element has one attribute **kind** with a non-zero integer value that is the degree of rotational symmetry. A negative value means a dihedral symmetry. Kind one means no symmetry.

The color element has two attributes **index** and **rgb**. The index value is an integer in [0,255] and the rgb value is three integers in [0,255]. It sets a single entry of the color map, so you need 256 of them. You cannot specify both a palette attribute and color elements. For example:

```
<flame ...>
   <color index="0" rgb="20 12 123"/>
   <color index="1" rgb="21 13 130"/>
   ...
   <color index="255" rgb="200 42 0"/>
   <xform .../>
   ...
<flame/>
```