The Fractal Flame Algorithm

Scott Draves
Spotworks, NYC, USA

Erik Reckase
Berthoud, CO, USA

September 2003, Last revised November 2008

Abstract

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 attractor 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.

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. We found that preserving information maximizes aesthetics.

1 Overview

Some examples appear in Figure 1. 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 3, and then further extended with post transforms and final transforms in Section 3.1. Section 4 describes how log-density display works and its importance, and Section 5 covers the coloring algorithm. These three sections cover the core innovations. Sections 6 and 8 explain other important properties of the implementation, and Section 7 shows how to create symmetric flames. The appendix is a catalog of the variations including formulas and examples.

2 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 Hutchinson's recursive set equation [3]:

$$S = \bigcup_{i=0}^{n-1} F_i(S)$$

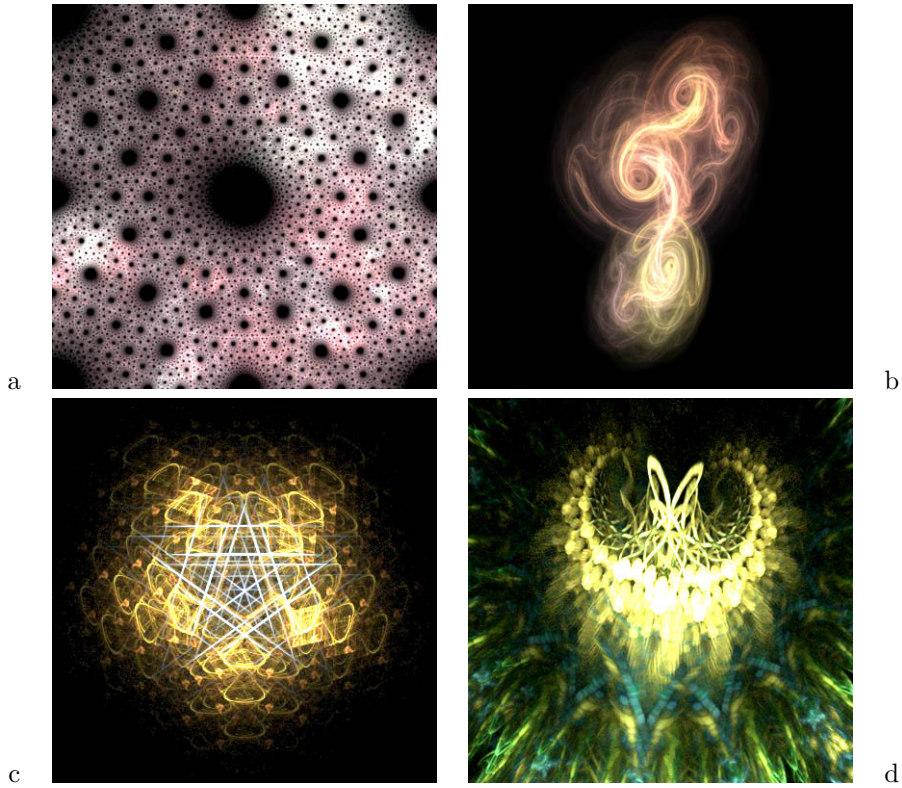


Figure 1: Example fractal flame images. The names of these flames are: a) 206, b) 191, c) 4000, and d) 29140. These images were selected for their aesthetic properties.

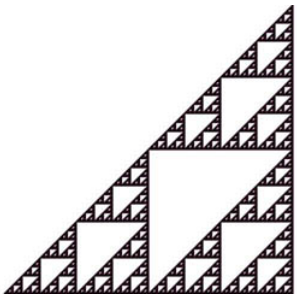


Figure 2: Sierpinski's Gasket, a simple IFS. X and y increase towards the lower right. The recursive structure is visible as the whole image is made up of three copies of itself, one for each function.

As implemented and popularized by Barnsley [1] the functions F_i are linear (technically they are affine as each is a two by three matrix capable of expressing scale, rotation, translation, and shear):

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

For example, if the functions are

$$F_0(x, y) = (\tfrac{x}{2}, \tfrac{y}{2}) \quad F_1(x, y) = (\tfrac{x+1}{2}, \tfrac{y}{2}) \quad F_2(x, y) = (\tfrac{x}{2}, \tfrac{y+1}{2})$$

then the fixed-point S is Sierpinski's Gasket, as seen in Figure 2.

In order to facilitate the proofs and guarantee convergence of the algorithms, the functions are normally constrained 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*. Useful guarantees of this become difficult to provide when the functions are non-linear. Instead we recommend using a numerically robust implementation, and simply accept that some parameter sets result in better images than others, and some result in degenerate images.

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

(x, y)= a random point in the bi-unit square
iterate {
    i = a random integer from 0 to $n - 1$ inclusive
    $(x, y) = F_i(x, y)$
    plot(x, y) except during the first 20 iterations
}

The bi-unit square are those points where x and y are both in [-1,1]. The chaos game 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 will be generated eventually because an infinite string of random symbols (the choices for i) contains every finite substring of symbols. This is explained in more formally in Section 4.8 of [1].

No sufficient 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 fractal flames, the number of samples is specified with the more abstract parameter *quality*, or samples per output pixel. That way the image quality (in the sense of lack of noise) remains constant in face of changes to the image resolution and the camera.

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 allows 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 to draw symmetric flames as shown in Section 7.

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.

3 Variations

We now generalize this algorithm. The first step is to use a larger class of functions than just affine functions. Start by composing a non-linear function V_j from R^2 to R^2 with the affine functions:

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

We call each such function V_j a *variation*, and each one changes the shape and character of the solution in a recognizable way. The initial variations were simple remappings of the plane. They were followed by *dependent variations*, where the coefficients of the affine transform define the behaviour of the variation. More recently, *parametric variations* have been introduced, which are variations controlled by additional parameters independent of the affine transform.

Appendix A documents 49 variations. Variation 0 is the identity function. It and six more examples are:

$$
\begin{array}{rcll}
V_0(x,y) & = & (x,y) & \textit{linear} \\
V_1(x,y) & = & (\sin x, \sin y) & \textit{sinusoidal} \\
V_2(x,y) & = & \frac{1}{r^2} \cdot (x,y) & \textit{spherical} \\
V_3(x,y) & = & \big(x\sin(r^2) - y\cos(r^2), x\cos(r^2) + y\sin(r^2)\big) & \textit{swirl} \\
V_4(x,y) & = & \frac{1}{r} \cdot ((x-y)(x+y), 2xy) & \textit{horseshoe} \\
V_{17}(x,y) & = & (x + c\sin(\tan 3y), y + f\sin(\tan 3x)) & \textit{popcorn} \\
V_{24}(x,y) & = & (\sin(p_1 y) - \cos(p_2 x), \sin(p_3 x) - \cos(p_4 y)) & \textit{pdj}
\end{array}
$$

where

$$r = \sqrt{x^2 + y^2}$$

An example of a dependent variation is Popcorn, V_{17}, which is dependent on the c and f coefficients of the affine transform.

The PDJ variation, V_{24}, is an example of a parametric variation. PDJ relies on four external parameters (p_1, p_2, p_3, p_4) to fully characterize its behaviour.

Variations can be further generalized by replacing the integer parameter j with a blending vector v_{ij} with one coefficient per variation. Then

$$F_i(x,y) = \sum_j v_{ij} V_j(a_i x + b_i y + c_i, d_i x + e_i y + f_i)$$

With this generalization, if are n functions, then there are $87n$ parameters that specify it. The parameters consist of $49n$ variational coefficients v_{ij}, $31n$ parametric coefficients, $6n$ matrix coefficients a_i through f_i, and n weights w_i. Significantly fewer parameters are required, however, if unused parameters are assumed to be 0.

3.1 Post Transforms

To this point, applying a transform to a set of coordinates involves first applying an affine transformation to the coordinates, and then applying a non-linear variation function to the result of the affine transformation. We further generalize this with the addition of a secondary affine transformation, a *post transform*, to be applied after the non-linear function. This provides the ability to change the coordinate systems of the variations. If the post transform is

$$P_i(x,y) = (\alpha_i x + \beta_i y + \gamma_i, \delta_i x + \epsilon_i y + \zeta_i)$$

then we redefine the F_i as follows:

$$F_i(x,y) = P_i(\sum_j v_{ij} V_j(a_i x + b_i y + c_i, d_i x + e_i y + f_i))$$

3.2 Final Transforms

We can now introduce the concept of a *final transform*, which is an additional function $F_{final}(x, y)$ that is *always* applied regardless of the value of i in the iteration loop. The final transform is like a non-linear camera. The result of the application of the final transform function is not 'in the loop', and there can be only one final transform per flame. The final transform can have a post transform associated with it. In pseudocode, we now have:

(x, y)= a random point in the bi-unit square
iterate {
    i = a random integer from 0 to $n - 1$ inclusive
    $(x, y) = F_i(x, y)$
    $(x_f, y_f) = F_{final}(x, y)$
    plot (x_f, y_f) except during the first 20 iterations
}

Adding the post transforms to the earlier parameter count, we now have $93n$ parameters necessary to fully specify n functions. If a final transform is desired, then the total number of parameters necessary is $93(n + 1)$.

4 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 proportional to an exponent of the 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.

The flame algorithm addresses 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, for example, densities of 3,000 and 5,000 in one part of the image and 30 and 50 in another part.

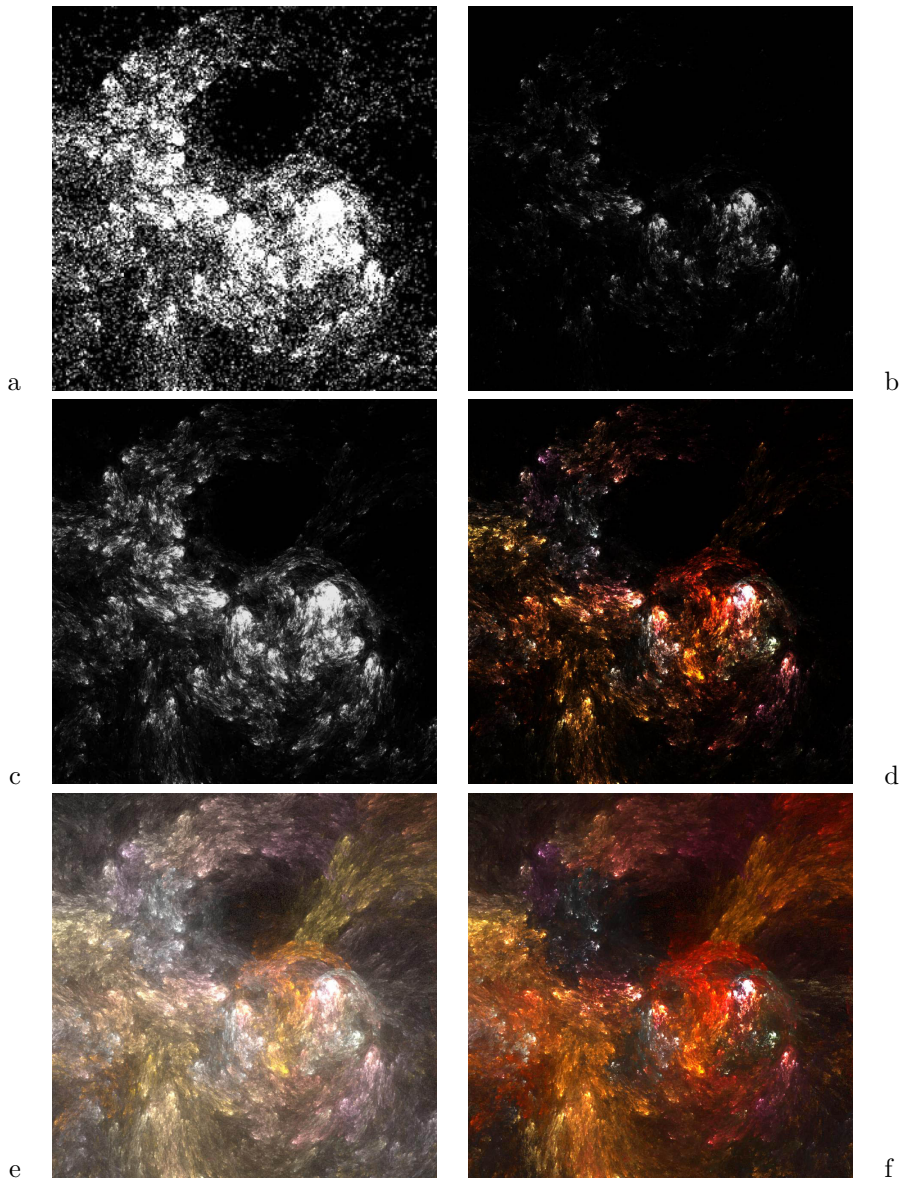


Figure 3: Successive refinements of the rendering technique starting with a) binary membership, then b) linear, c) logarithmic, d) with color, e) with gamma factor, and finally f) with vibrant colors. The parameters to create this image are given in Appendix B.

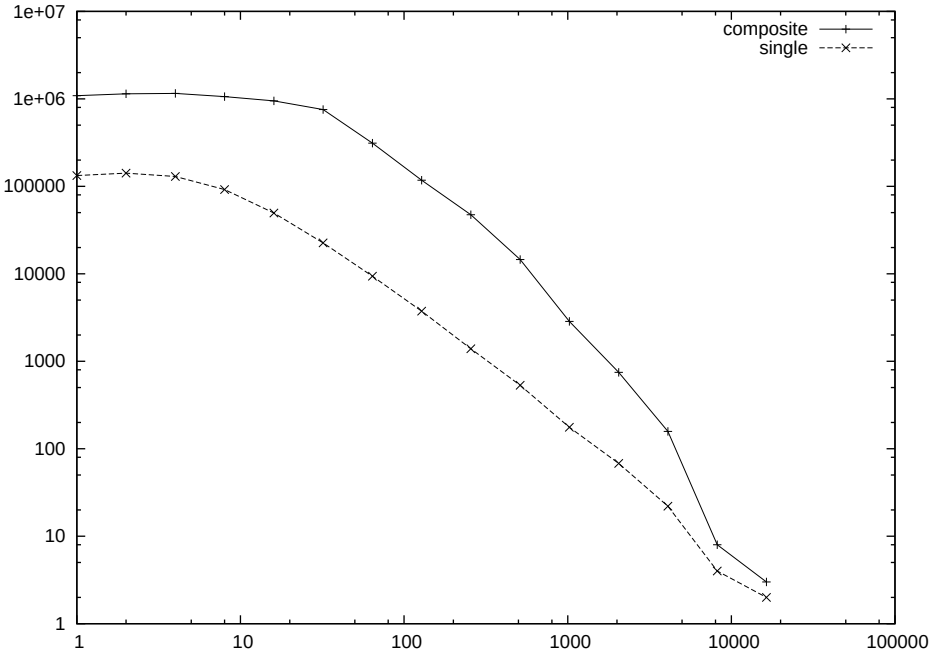


Figure 4: Plots showing that the distribution of densities in an IFS follows the power law. The density (on the horizontal axis) is the number of hits by the system in a pixel, the frequency (on the vertical axis) is the number of pixels with that density (or up to the next power of two). The line is from the image in Figure 3, the other is the composite of 19 old, 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. Each image was computed with 9.2e6 samples on a 900x900 grid.

The display of high dynamic range images like these by *tone mapping* is studied in computer graphics [4]. The logarithm used here (combined with the gamma factor, described below) is just an ad-hoc tone-map.

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.

5 Coloring

There is more information to be wrung from the attractor. In particular, which parts of the attractor 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.

To achieve this we assign a color c_i to each function F_i (and a corresponding c_{final} if a final transform is present) and add an independent coordinate to the chaos game:

(x, y)= a random point in the bi-unit square
c = a random point in [0,1]
iterate {
    i = a random integer from 0 to $n-1$ inclusive
    $(x, y) = F_i(x, y)$
    $c = (c + c_i)/2$
    $(x_f, y_f) = F_{final}(x, y)$
    $c_f = (c + c_{final})/2$
    plot (x_f, y_f, c_f) 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.

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.

6 The Gamma Factor

Accurate display of any digital image on a Cathode Ray Tube (CRT) requires gamma correction to account for the non-linear response of screen phosphors to voltage. 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: $b_{corrected} = b^{1/\gamma}$ where γ normally about 2.2.

But 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.

Because the red, green, and blue phosphors respond independently, 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.

7 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 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.

8 Filtering

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

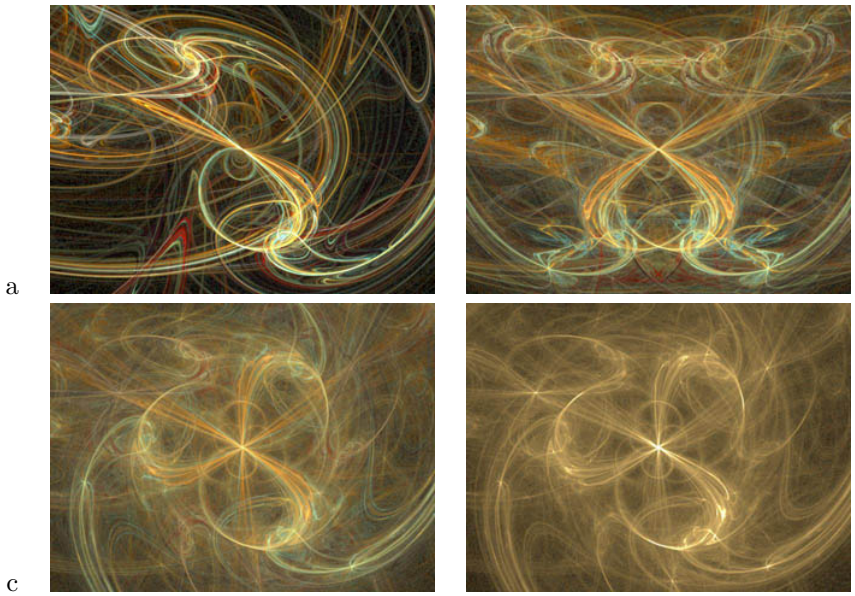


Figure 5: Examples of symmetry. Image d shows how the colors wash out without special treatment of the color coordinate for symmetry transforms.

the desired image at high resolution, and then filter it down to display resolution. This is known as supersampling, and its cost is linear in time and memory (though the sampling is normally applied in two dimensions, so 3 by 3 supersampling 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 small, 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 well-sampled parts of the image blurry. This can be addressed with a form of Density Estimation [5]. We have implemented a dynamic filter where a blur kernel of width inversely proportional to the density of points in the histogram is applied to the samples [6]. The blur kernel is scaled based on the supersampling level, and is applied after the log density scaling. This variable width filter allows higher density areas to remain in focus, while lower density areas are significantly smoothed. Figure 6 illustrates the effect of density estimation.

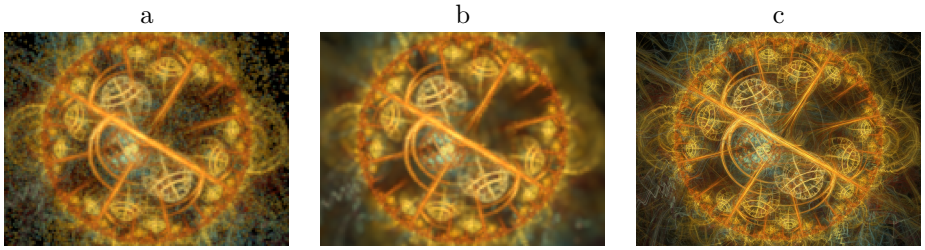


Figure 6: Demonstration of density estimation: (a) a low-resolution, low-quality, zoomed image rendered without density estimation, and (b) with density estimation, and (c) a high quality render at high resolution with density estimation.

9 Motion Blur

Motion blur, or temporal anti-aliasing, is not so easy to do correctly. Supersampling can be achieved by 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 supersampling for free! With linear density display, this would work exactly.

The non-linearity of the logarithm complicates things. Consider a pixel with density 8. Assume for simplicity the logarithm is base 2, so its assigned brightness is 3. Now put the fractal in motion so it blurs across two pixels, each should get brightness 1.5. But if the motion takes place before the logarithm then the 8 samples would be divided into two pixels of density 4 each, whose logarithm is 2, not 1.5. Objects in motion would appear unnaturally bright.

A proper solution requires the use of 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, applying the density estimation filter in the process. After all samples are completed, the second buffer is filtered down into the final image.

The drawback of this approach, however, is the computational effort required to apply the density estimation filter repeatedly to the linear buffer; multiple applications of the filter can easily double the rendering time.

After experiments with a single buffer yielded acceptable results, we decided to default to single buffer renders, while retaining the option of using the extra buffer.

9.1 Directional Motion Blur

Uniformly distributing the samples among time steps works well for an animated series of frames, but the illusion of motion of an individual frame animation may be improved by providing a sense of direction. Early attempts at directional

a b

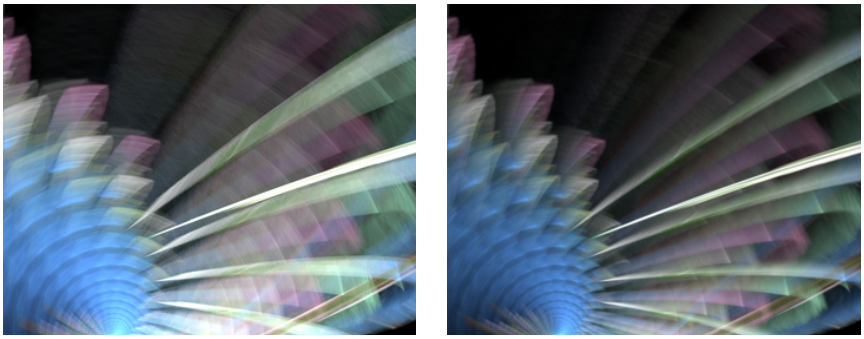


Figure 7: (a) motion blur and (b) directional motion blur.

motion varied the number of samples used at each time step, but since the density estimation filter was more aggressive during less dense time steps, the result appeared unnatural.

Instead, the color of points accumulated during earlier time steps are scaled in intensity, with the scaling constant approaching 1.0 as the time steps progress. A rendering parameter can be supplied to the renderer to change how aggressively the blending varies throughout the time steps. See the effects of directional motion blur in Figure 7.

10 History and Acknowledgements

In 1987 at Brown University Bill Poirier showed Scott Draves what he called 'Recursive Pictures' a kind of two-dimensional IFS. Poirier used a formulation that included perspective transforms, but lacked a software implementation. In response Draves created the first of many IFS algorithms. It was written in Postscript and ran on the Laserwriter, producing high resolution line drawings. Draves reimplemented this idea in a variety of ways over the three years he spent in the Computer Graphics Research Group at while still at Brown.

The first implementation to include all three definitive characteristics of fractal flames (non-linear variations, log-density display, and structural coloring) was created in the summer of 1991 while Draves 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 General Public License (GPL), making it the first open-source art. 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, Oxidizer, and Qosmic).

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 Draves in 2003.

In 2004, Mark Townsend released Apophysis, a translation of Draves' C code into Delphi Pascal, with the addition of a GUI for interactive design. Erik Reckase became more involved with the flame algorithm after the release of Apophysis, and became an official developer and maintainer in 2005. Reckase's contributions have focused on improved image quality, code optimization, and keeping up with the wealth of variations being developed in the Apophysis community.

In 2006 Peter Sdobnov added final transforms to Apophysis and they were soon after adopted by our implementation to maintain compatibility.

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 [2], a follow-on art project by Draves. 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.

References

[1] Michael Barnsley: Fractals Everywhere. Academic Press, San Diego, 1988.

[2] Scott Draves. The Electric Sheep Screen-Saver: A Case Study in Aesthetic Evolution. Applications of Evolutionary Computing, LNCS 3449, 2005.

[3] Hutchinson, J. *Fractals and Self-Similarity.* Indiana University Journal of Mathematics. 30, 713-747, 1981.

[4] Jack Tumblin, Holly Rushmeier. Tone Reproduction for Realistic Images. IEEE Computer Graphics and Applications. November/December 1993 (Vol. 13, No. 6) pp. 42-48.

[5] B. W. Silverman, Density Estimation for Statistics and Data Analysis, Chapman and Hall, London, 1986.

[6] F. Suykens and Y. D. Willems. Adaptive filtering for progressive Monte Carlo image rendering. In Eighth International Conference in Central Europe on Computer Graphics, Visualization and Interactive Digital Media (WSCG 2000), Plzen, Czech Republic, February 2000.

Appendix: Catalog of Variations

For each variation we give its formula and its name, and provide a list of parameters for parametric variations. Variables used in these formulas are:

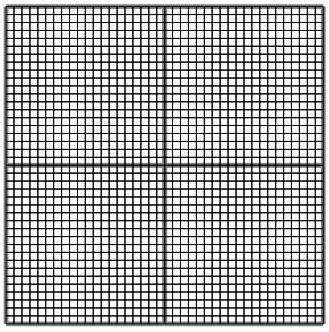
$$\begin{aligned} r &= \sqrt{x^2 + y^2} \\ \theta &= \arctan(x/y) \\ \phi &= \arctan(y/x) \end{aligned}$$

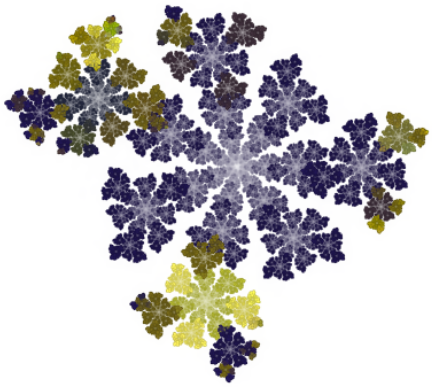
(a, b, c, d, e, f) are considered to be the affine transform coefficients for a variation, and are used in dependent variations. Ω is a random variable that is either 0 or π. Λ is a random variable that is either -1 or 1. Ψ is a random variable uniformally distributed on the interval $[0, 1]$. The 'trunc' function returns the integer part of a floating-point value.

A visualization of the distorted coordinate grid and a representative flame using this variation are also supplied. As in Figure 2, x and y increase towards the lower right, to match the coordinate system of the output image. Note that these sample flames are selected based on their characteristic shape and not for any aesthetic qualities. The representative images attempt to use a single variation, but in general, variations can be mixed when creating flames. For random-based variations, a scatterplot is substituted for the coordinate grid.

Linear (Variation 0)

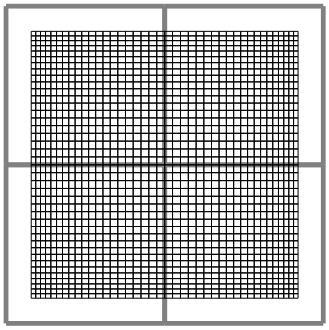
$$V_0(x, y) = (x, y)$$





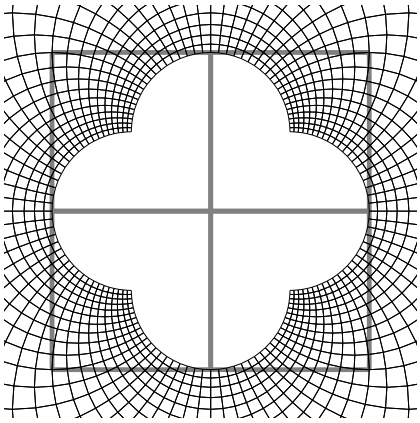
Sinusoidal (Variation 1)

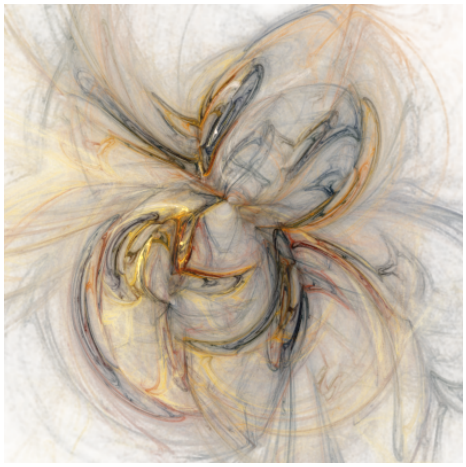
$$V_1(x, y) = (\sin x, \sin y)$$



Spherical (Variation 2)

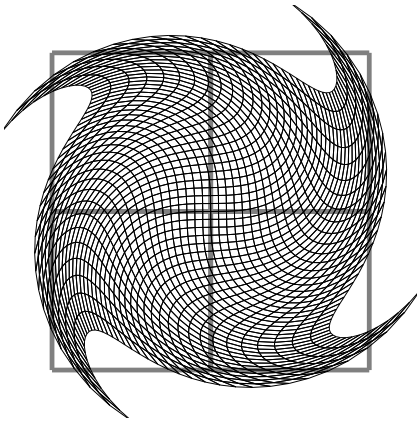
$$V_2(x, y) = \frac{1}{r^2} \cdot (x, y)$$

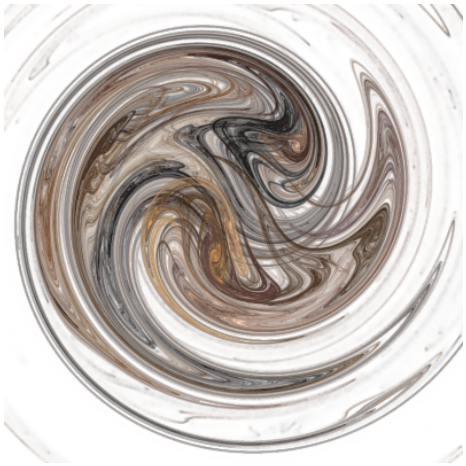




Swirl (Variation 3)

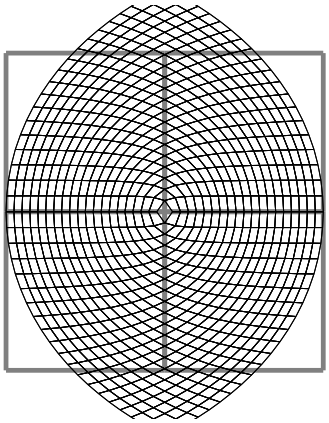
$$V_3(x, y) = \big(x \sin(r^2) - y \cos(r^2), x \cos(r^2) + y \sin(r^2)\big)$$





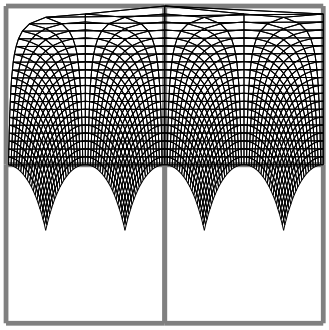
Horseshoe (Variation 4)

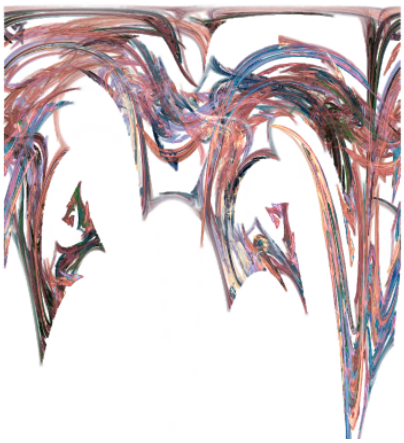
$$V_4(x, y) = \frac{1}{r} \cdot ((x - y)(x + y), 2xy)$$



Polar (Variation 5)

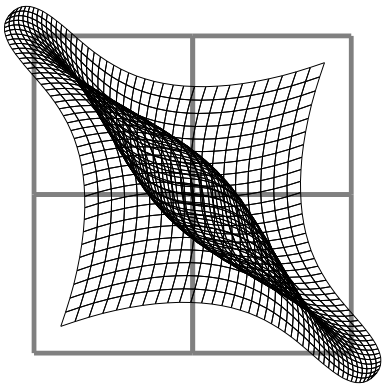
$$V_5(x, y) = \left(\frac{\theta}{\pi}, r - 1\right)$$

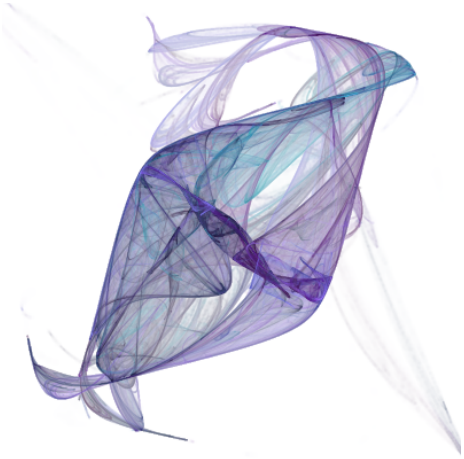




Handkerchief (Variation 6)

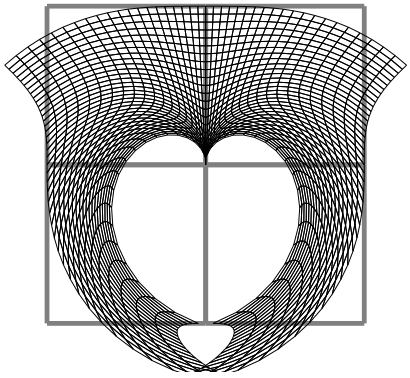
$$V_6(x, y) = r \cdot (\sin(\theta + r), \cos(\theta - r))$$





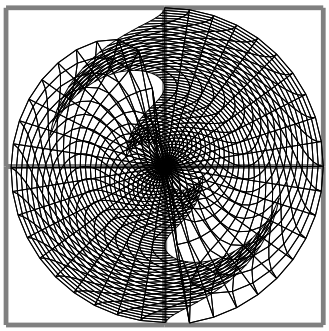
Heart (Variation 7)

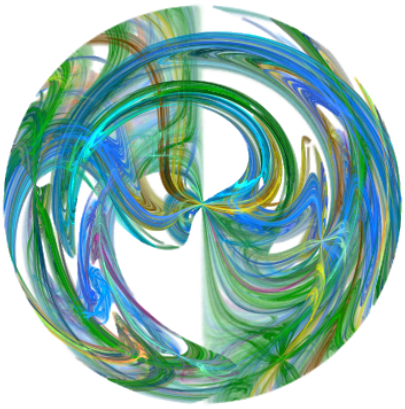
$$V_7(x, y) = r \cdot (\sin(\theta r), -\cos(\theta r))$$



Disc (Variation 8)

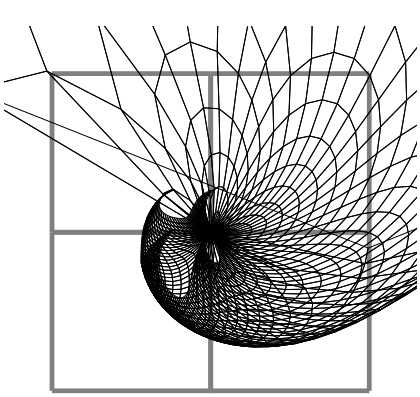
$$V_8(x, y) = \frac{\theta}{\pi} \cdot (\sin(\pi r), \cos(\pi r))$$

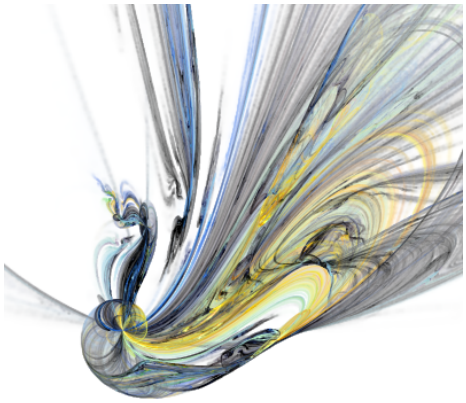




Spiral (Variation 9)

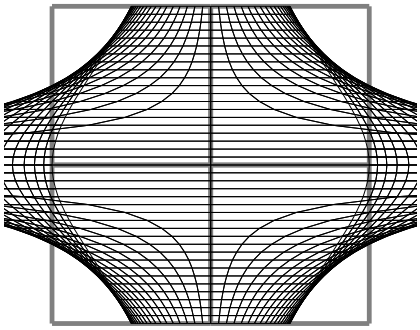
$$V_9(x, y) = \frac{1}{r}(\cos\theta + \sin r, \sin\theta - \cos r)$$





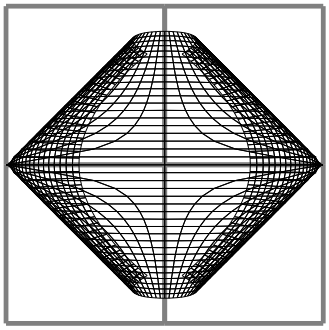
Hyperbolic (Variation 10)

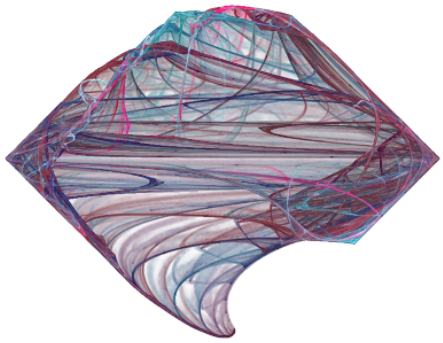
$$V_{10}(x, y) = \left(\frac{\sin\theta}{r}, r\cos\theta\right)$$



Diamond (Variation 11)

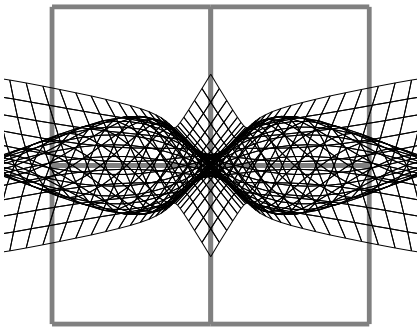
$$V_{11}(x, y) = (\sin\theta\cos r, \cos\theta\sin r)$$

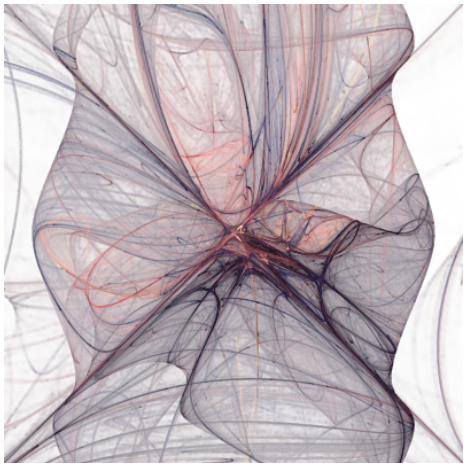




Ex (Variation 12)

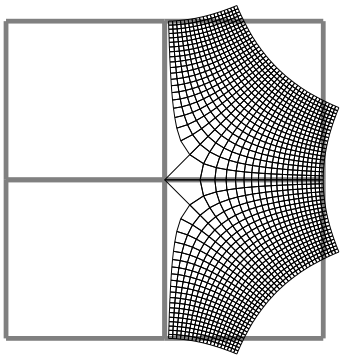
$$p_0 = \sin(\theta + r), p_1 = \cos(\theta - r)$$
$$V_{12}(x, y) = r \cdot (p_0^3 + p_1^3, p_0^3 - p_1^3)$$

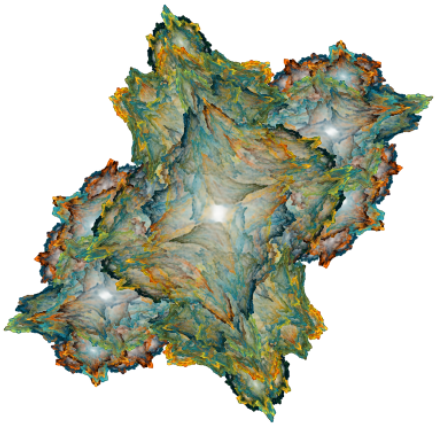




Julia (Variation 13)

$$V_{13}(x,y) = \sqrt{r} \cdot (\cos(\theta/2 + \Omega), \sin(\theta/2 + \Omega)$$

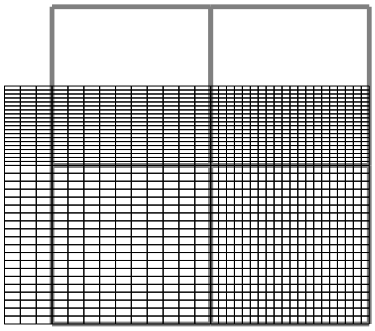


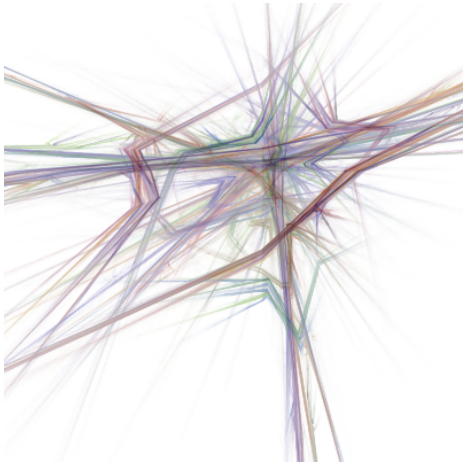


(Note: The grid visualization for the julia variation only includes data for $\Omega = 0$.)

Bent (Variation 14)

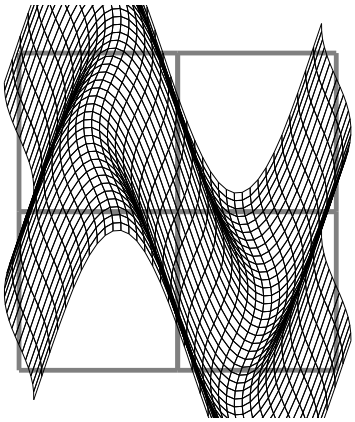
$$V_{14}(x,y) = \begin{cases} (x,y) & x \geq 0, y \geq 0 \\ (2x,y) & x < 0, y \geq 0 \\ (x,y/2) & x \geq 0, y < 0 \\ (2x,y/2) & x < 0, y < 0 \end{cases}$$

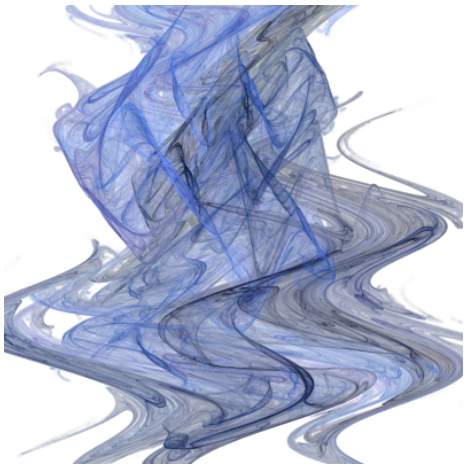




Waves (Variation 15) - dependent

$$V_{15}(x,y) = \left(x + b\sin\left(\frac{y}{c^2}\right), y + e\sin\left(\frac{x}{f^2}\right)\right)$$

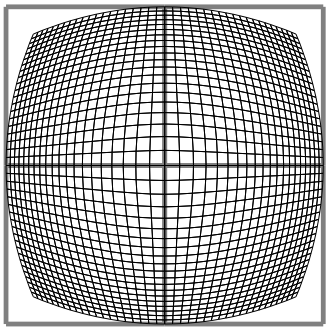


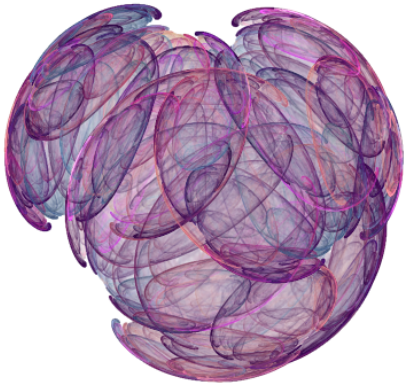


Fisheye (Variation 16)

Note the reversed order of x and y in the formula.

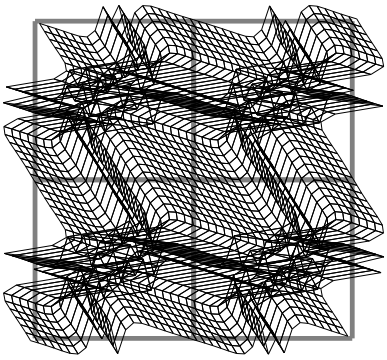
$$V_{16}(x,y) = \frac{2}{r+1} \cdot (y,x)$$





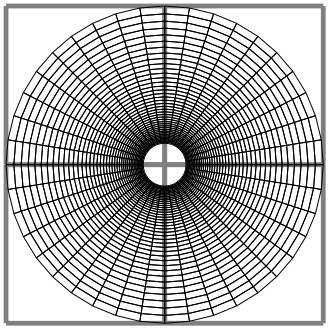
Popcorn (Variation 17) - dependent

$$V_{17}(x, y) = (x + c\sin(\tan 3y), y + f\sin(\tan 3x))$$



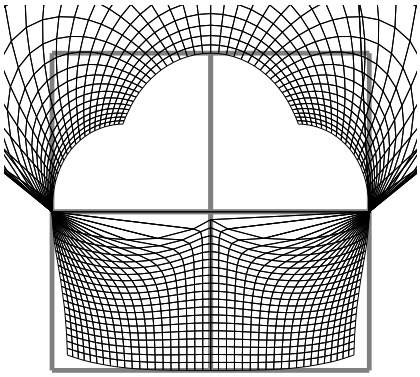
Exponential (Variation 18)

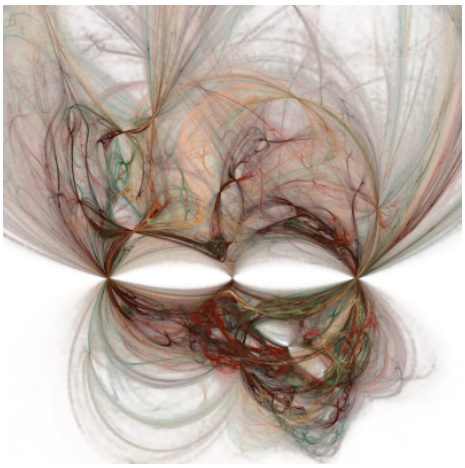
$$V_{18}(x, y) = \exp(x - 1) \cdot (\cos(\pi y), \sin(\pi y))$$



Power (Variation 19)

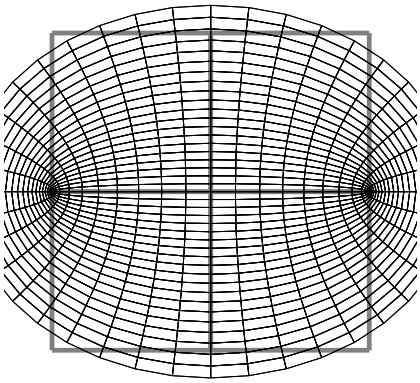
$$V_{19}(x, y) = r^{\sin\theta} \cdot (\cos\theta, \sin\theta)$$

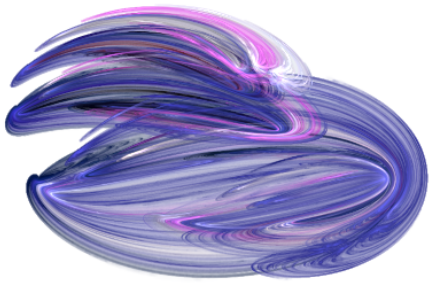




Cosine (Variation 20)

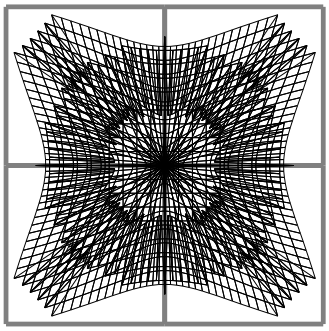
$$V_{20}(x, y) = (\cos(\pi x)\cosh(y), -\sin(\pi x)\sinh(y))$$





Rings (Variation 21) - dependent

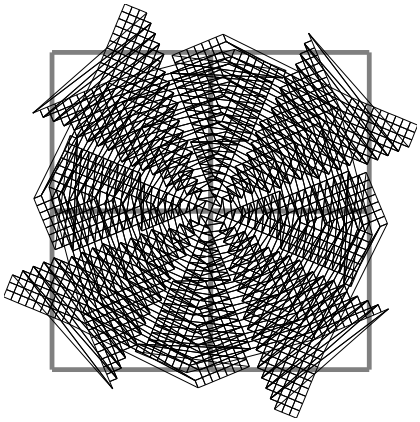
$$V_{21}(x,y) = \left((r+c^2) \bmod (2c^2) - c^2 + r(1-c^2)\right) \cdot (\cos\theta, \sin\theta)$$

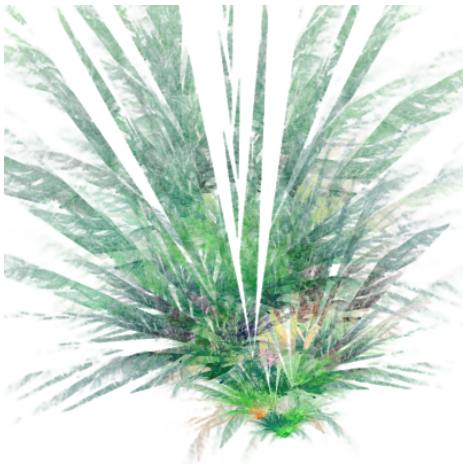


Fan (Variation 22) - dependent

$$t = \pi c^2$$

$$V_{22}(x,y) = \begin{cases} r \cdot (\cos(\theta - t/2), \sin(\theta - t/2)) & (\theta + f) \bmod t > t/2 \\ r \cdot (\cos(\theta + t/2), \sin(\theta + t/2)) & (\theta + f) \bmod t \leq t/2 \end{cases}$$

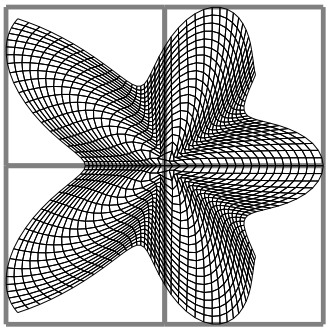




Blob (Variation 23) - parametric

$$p_1 = \text{blob.high}, p_2 = \text{blob.low}, p_3 = \text{blob.waves}$$

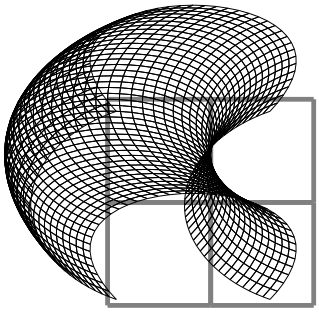
$$V_{23}(x, y) = r \cdot \left(p_2 + \frac{p_1 - p_2}{2}(\sin(p_3\theta) + 1) \right) \cdot (\cos\theta, \sin\theta)$$

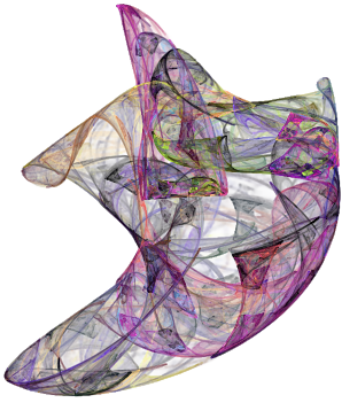


PDJ (Variation 24) - parametric

$$p_1 = \text{pdj.a}, p_2 = \text{pdj.b}, p_3 = \text{pdj.c}, p_4 = \text{pdj.d}$$

$$V_{24}(x, y) = (\sin(p_1 y) - \cos(p_2 x), \sin(p_3 x) - \cos(p_4 y))$$





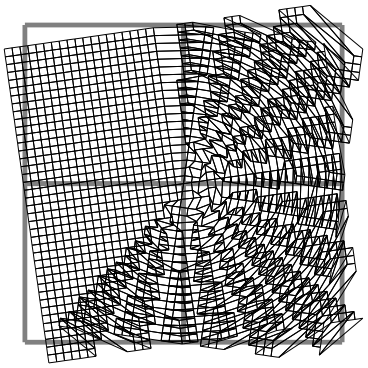
Fan2 (Variation 25) - parametric

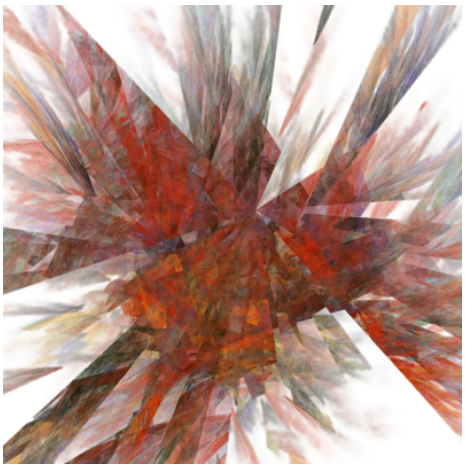
Fan2 was created as a parametric alternative to Fan.

$$p_1 = \pi(\text{fan2.x})^2, p_2 = \text{fan2.y}$$

$$t = \theta + p_2 - p_1 \text{trunc}(\frac{2\theta p_2}{p_1})$$

$$V_{25}(x,y) = \begin{cases} r \cdot (\sin(\theta - p_1/2), \cos(\theta - p_1/2)) & t > p_1/2 \\ r \cdot (\sin(\theta + p_1/2), \cos(\theta + p_1/2)) & t \leq p_1/2 \end{cases}$$





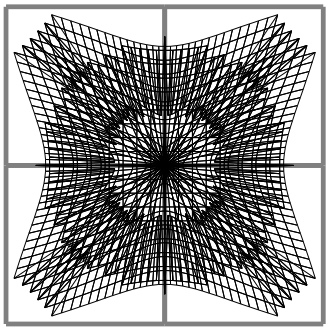
Rings2 (Variation 26) - parametric

Rings2 was created as a parametric alternative to Rings.

$$p = (\text{rings2.val})^2$$

$$t = r - 2p\text{trunc}\left(\frac{r+p}{2p}\right) + r(1-p)$$

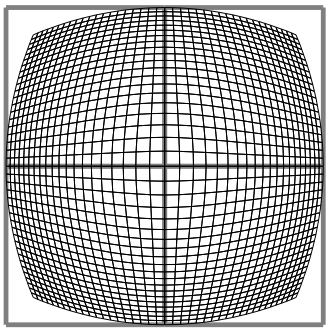
$$V_{26}(x,y) = t \cdot (\sin\theta, \cos\theta)$$



Eyefish (Variation 27)

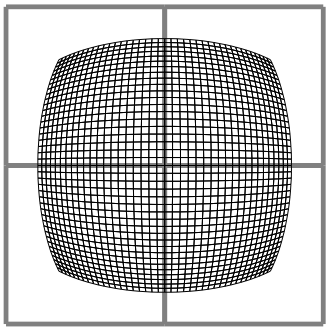
Eyefish was created to correct the order of x and y in Fisheye.

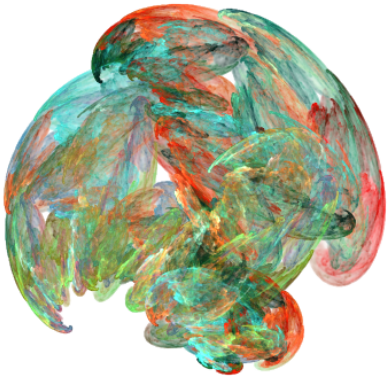
$$V_{27}(x,y) = \frac{2}{r+1} \cdot (x,y)$$



Bubble (Variation 28)

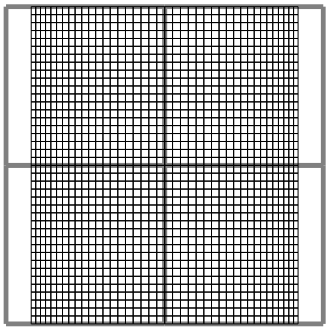
$$V_{28}(x, y) = \frac{4}{r^2 + 4} \cdot (x, y)$$





Cylinder (Variation 29)

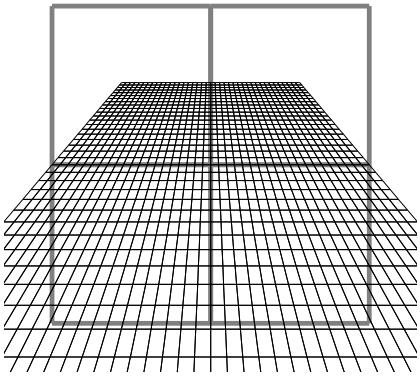
$$V_{29}(x, y) = (\sin x, y)$$

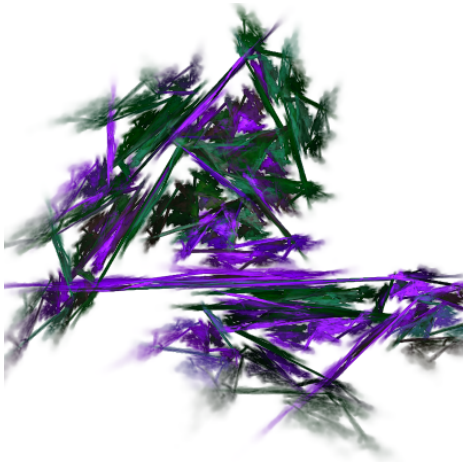


Perspective (Variation 30) - parametric

$$p_1 = \text{perspective.angle}, p_2 = \text{perspective.dist}$$

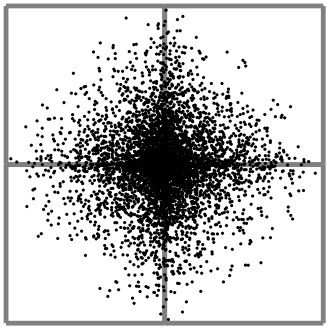
$$V_{30}(x, y) = \frac{p_2}{p_2 - y \sin p_1} \cdot (x, y \cos p_1)$$

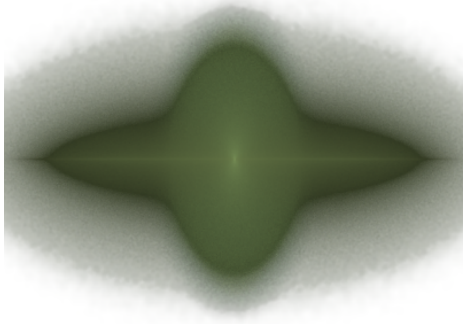




Noise (Variation 31)

$$V_{31}(x, y) = \Psi_1 \cdot (x \cos(2\pi\Psi_2), y \sin(2\pi\Psi_2))$$





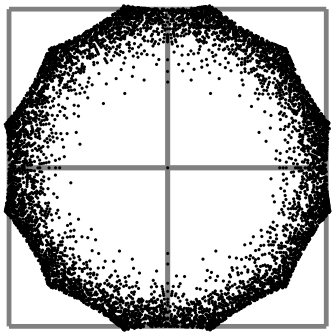
JuliaN (Variation 32) - parametric

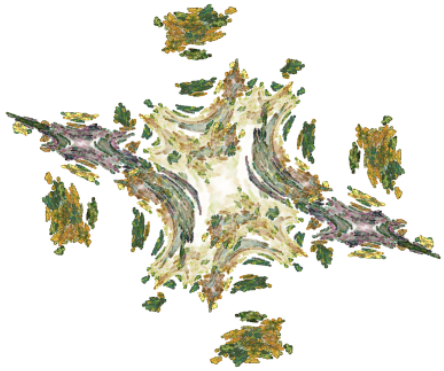
$$p_1 = \text{juliaN.power}, p_2 = \text{juliaN.dist}$$

$$p_3 = \text{trunc}(|p_1|\Psi)$$

$$t = (\phi + 2\pi p_3)/p_1$$

$$V_{32}(x, y) = r^{\frac{p_2}{p_1}} \cdot (\cos t, \sin t)$$





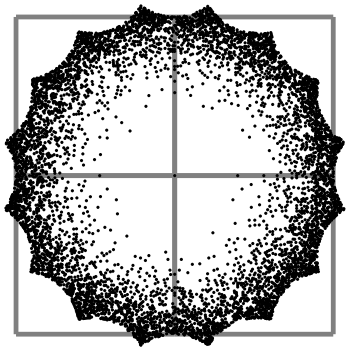
JuliaScope (Variation 33) - parametric

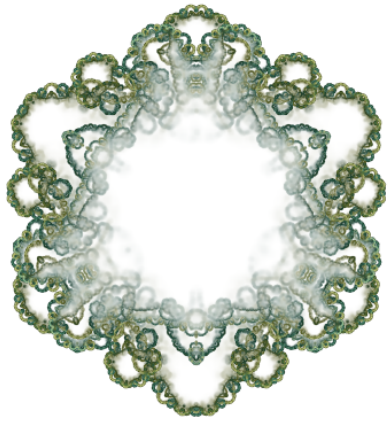
$$p_1 = \text{juliaScope.power}, p_2 = \text{juliaScope.dist}$$

$$p_3 = \text{trunc}(|p_1|\Psi)$$

$$t = (\Lambda\phi + 2\pi p_3)/p_1$$

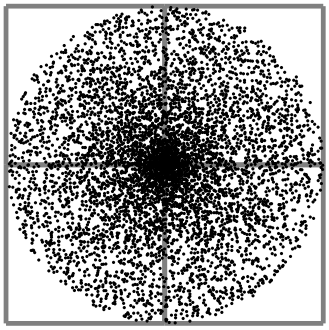
$$V_{33}(x, y) = r^{\frac{p_2}{p_1}} \cdot (\cos t, \sin t)$$

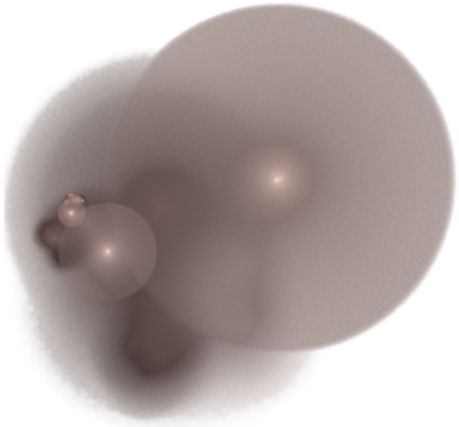




Blur (Variation 34)

$$V_{34}(x, y) = \Psi_1 \cdot (\cos(2\pi\Psi_2), \sin(2\pi\Psi_2))$$

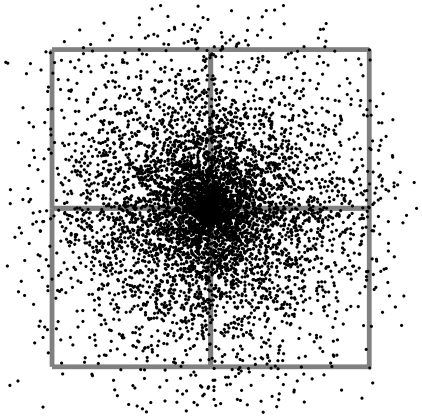


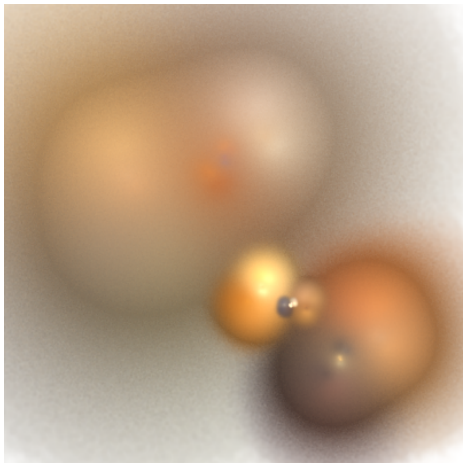


Gaussian (Variation 35)

Summing 4 random numbers and subtracting 2 is an attempt at approximating a Gaussian distribution.

$$V_{35}(x, y) = \left(\sum_{k=1}^{4} \Psi_k - 2\right) \cdot (\cos(2\pi\Psi_5), \sin(2\pi\Psi_5))$$



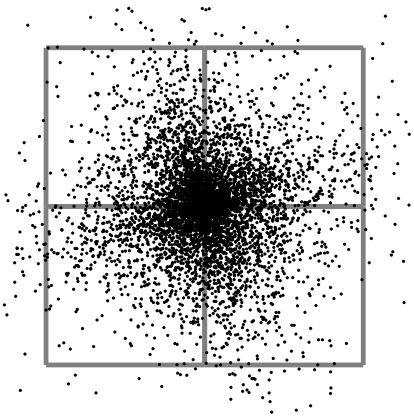


RadialBlur (Variation 36) - parametric

$$p_1 = (\text{radialBlur.angle}) \cdot (\pi/2)$$

$$t_1 = v_{36}(\sum_{k=1}^{4} \Psi_k - 2), t_2 = \phi + t_1 \sin p_1, t_3 = t_1 \cos p_1 - 1$$

$$V_{36}(x, y) = \frac{1}{v_{36}} \cdot (r \cos t_2 + t_3 x, r \sin t_2 + t_3 y)$$



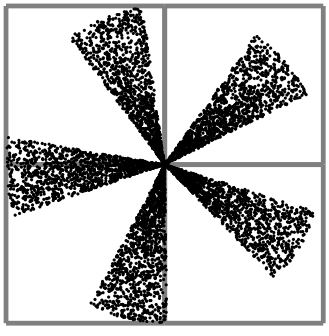
Pie (Variation 37) - parametric

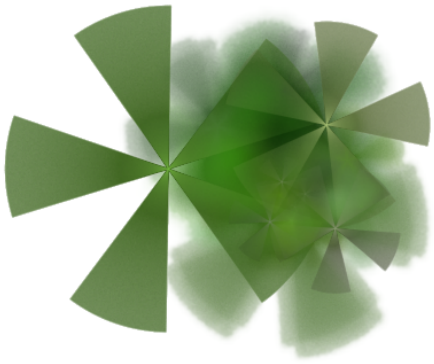
$$p_1 = \text{pie.slices}, p_2 = \text{pie.rotation}, p_3 = \text{pie.thickness}$$

$$t_1 = \text{trunc}(\Psi_1 p_1 + 0.5)$$

$$t_2 = p_2 + \frac{2\pi}{p_1}(t_1 + \Psi_2 p_3)$$

$$V_{37}(x, y) = \Psi_3(\cos t_2, \sin t_2)$$





Ngon (Variation 38) - parametric

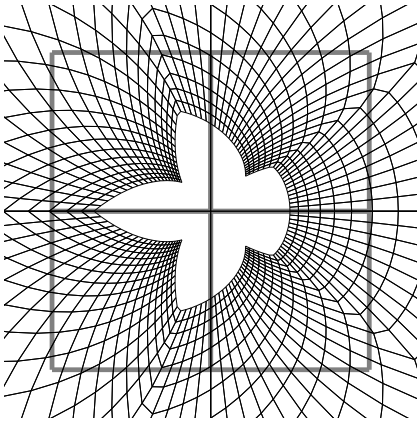
$p_1 = \text{ngon.power}, p_2 = 2\pi/\text{ngon.sides}, p_3 = \text{ngon.corners}, p_4 = \text{ngon.circle}$

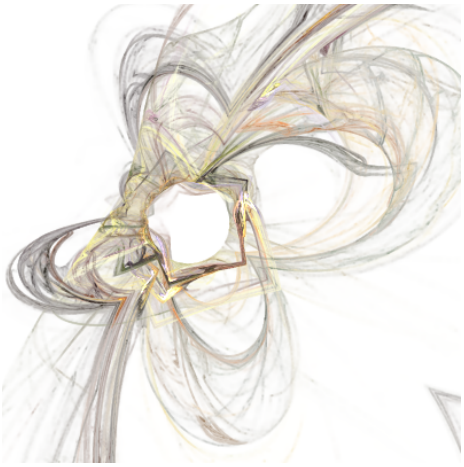
$$t_3 = \phi - p_2 \lfloor \phi / p_2 \rfloor$$

$$t_4 = \begin{cases} t_3 & t_3 > p_2/2 \\ t_3 - p_2 & t_3 \leq p_2/2 \end{cases}$$

$$k = \frac{p_3 \left(\frac{1}{\cos t_4} - 1 \right) + p_4}{r^{p_1}}$$

$$V_{38}(x, y) = k \cdot (x, y)$$



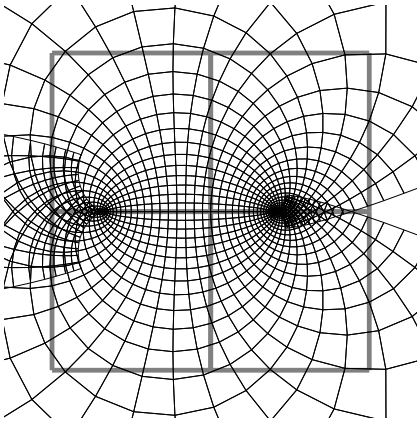


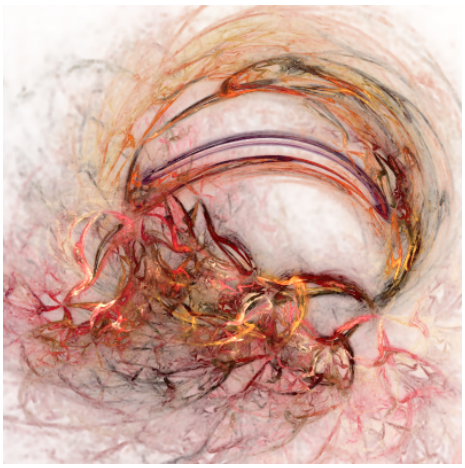
Curl (Variation 39) - parametric

$$p_1 = \text{curl.c1}, p_2 = \text{curl.c2}$$

$$t_1 = 1 + p_1 x + p_2(x^2 - y^2), t_2 = p_1 y + 2p_2 xy$$

$$V_{39}(x, y) = \frac{1}{t_1^2 + t_2^2} \cdot (xt_1 + yt_2, yt_1 - xt_2)$$

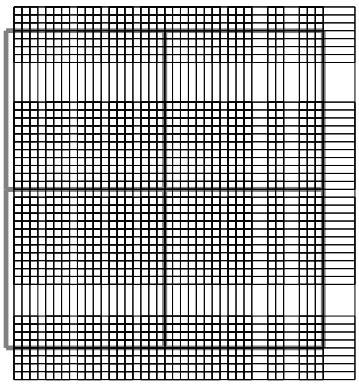


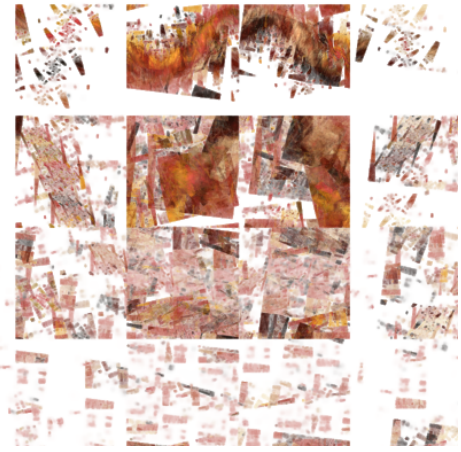


Rectangles (Variation 40) - parametric

$$p_1 = \text{rectangles.x}, p_2 = \text{rectangles.y}$$

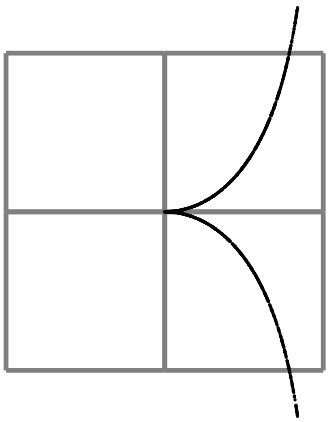
$$V_{40}(x, y) = (2\lfloor x/p_1 \rfloor + 1)p_1 - x, (2\lfloor y/p_2 \rfloor + 1)p_2 - y)$$

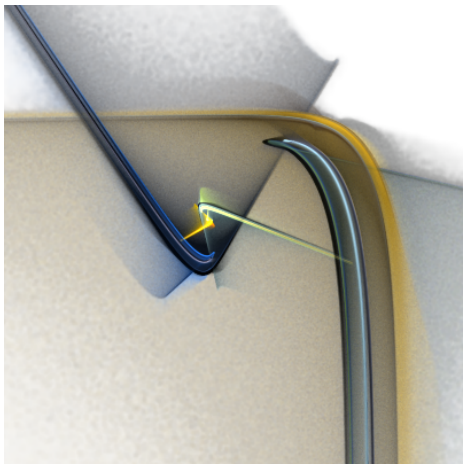




Arch (Variation 41)

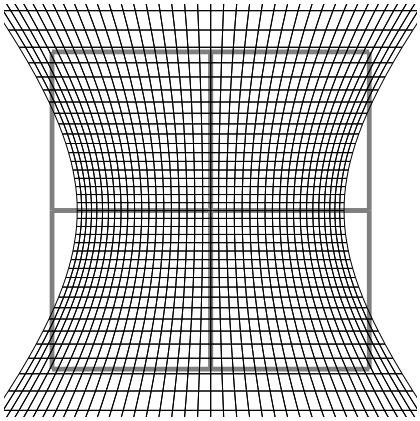
$$V_{41}(x, y) = (\sin(\Psi\pi v_{41}), \sin^2(\Psi\pi v_{41})/\cos(\Psi\pi v_{41}))$$

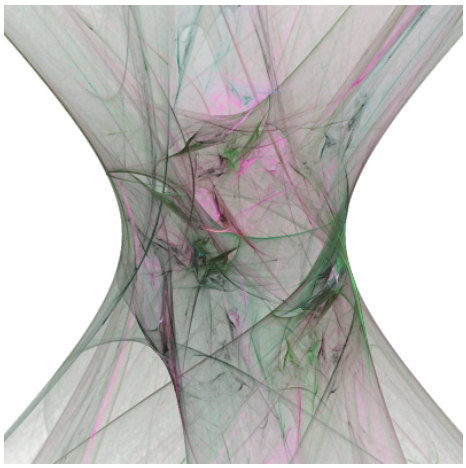




Tangent (Variation 42)

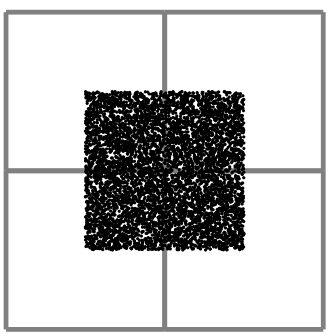
$$V_{42}(x, y) = \left(\frac{\sin x}{\cos y}, \tan y\right)$$

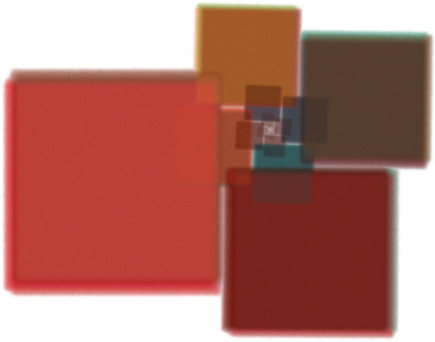




Square (Variation 43)

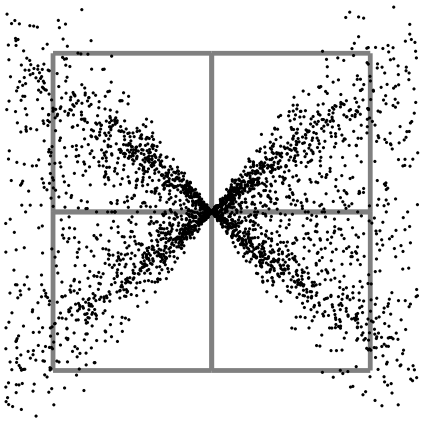
$$V_{43}(x, y) = (\Psi_1 - 0.5, \Psi_2 - 0.5)$$

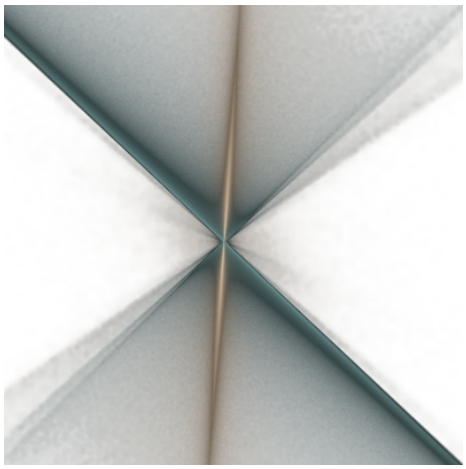




Rays (Variation 44)

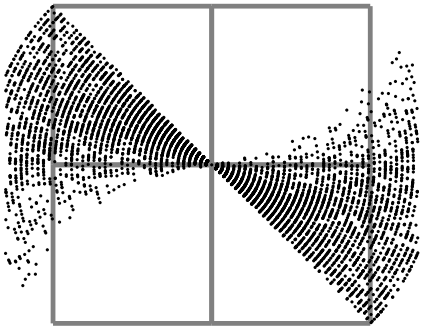
$$V_{44}(x, y) = \frac{v_{44} \tan(\Psi \pi v_{44})}{r^2} \cdot (\cos x, \sin y)$$

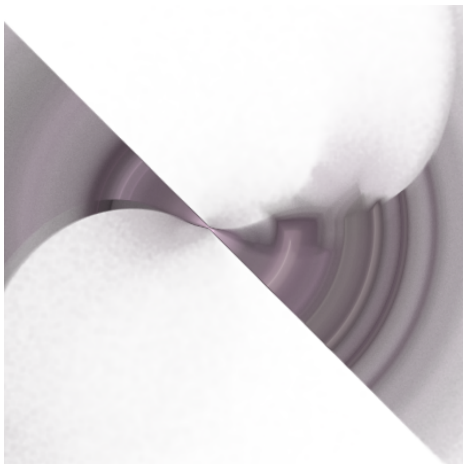




Blade (Variation 45)

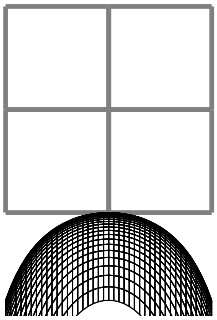
$$V_{45}(x, y) = x \cdot (\cos(\Psi r v_{45}) + \sin(\Psi r v_{45}), \cos(\Psi r v_{45}) - \sin(\Psi r v_{45}))$$

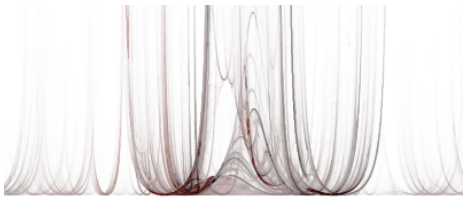


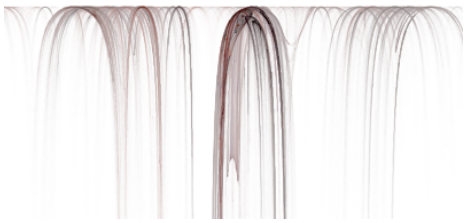


Secant (Variation 46)

$$V_{46}(x, y) = \left(x, \frac{1}{v_{46} \cos(v_{46} r)}\right)$$



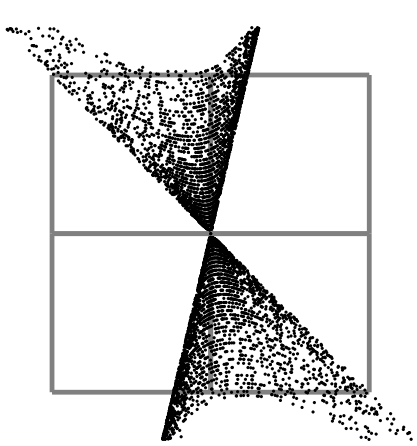


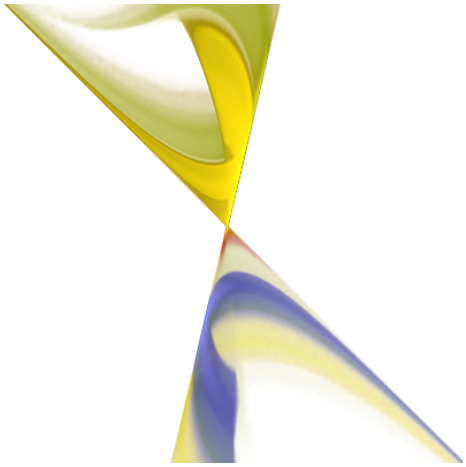


Twintrian (Variation 47)

$$t = \log_{10}\left(\sin^2(\Psi r v_{47})\right) + \cos(\Psi r v_{47})$$

$$V_{47}(x, y) = x \cdot (t, t - \pi \sin(\Psi r v_{47}))$$





Cross (Variation 48)

$$V_{47}(x, y) = \sqrt{1/(x^2 - y^2)^2} \cdot (x, y)$$