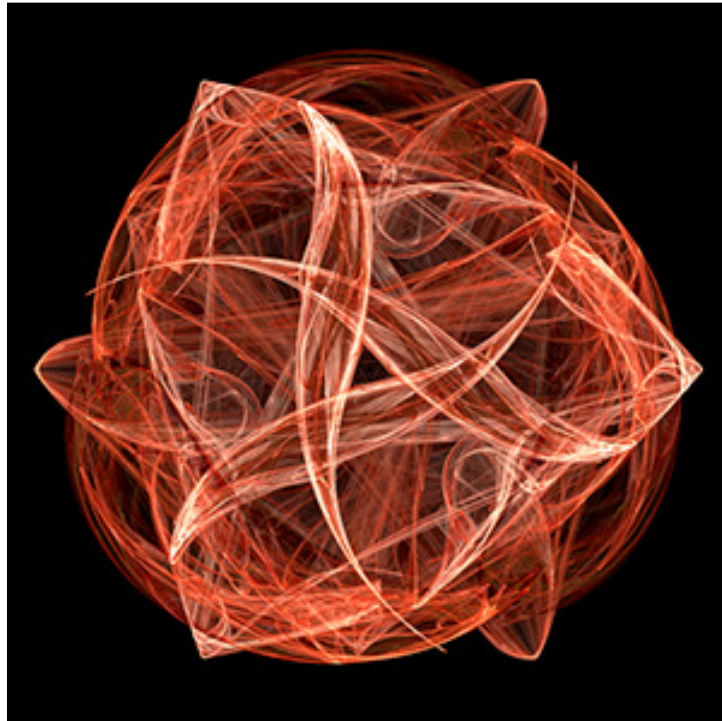


QS Flame



About the Program

“Cosmic recursive fractal flames” are a form of chaotic attractor which first were created by Scott Draves in 1992. Their popularity has made them ubiquitous throughout the digital world. This program was written in 1999 for Windows 95, and as far as Draves knew at the time, it was the first port of his UNIX code to Windows. Since then, numerous versions of flame software have appeared for most platforms, even including PhotoShop plug-ins. In its current state of revision and update, *QS Flame* is a 32-bit program which should continue to work in current 64-bit Windows versions.

More information about Draves's numerous artistic exploits can be found at his Web site:

<http://scottdraves.com/>

Specific information about flames can be found at:

<http://www.flam3.com/>

Apophysis is a highly sophisticated and recommended flame program for Windows which can be found at:

<http://apophysis.org/>

Draves intended for his code to be open source, and he released it under the GNU General Public License. The source code for this program can be obtained by contacting the author at the e-mail address at the end of the file, and a copy of the GPL is provided as the file: “GNU General Public License - Version 3 - 2007.pdf.”

Contents of this file

- 1: Using the Program**
 - a: Getting Started**
 - b: Menu Items**
 - c: Zooming**
- 2: Flame Background**
- 3: Acknowledgments**
- 4: Disclaimer**



1a: Using the Program: Getting Started

This program uses pseudo-randomization heavily. The easiest way to generate new images is to select parameters with the **New Random Set** option. Then you can check the **Image Parameters** and **Coefficients** dialog boxes to see what types of parameters correspond with various types of images. Finally you can experiment with changes in these parameters to adjust the images to your preference.

1b: Using the Program: Menu Items

Load Palette: The program utilizes 256-color palettes, which then are blended by the filtering process, or are adjusted with the brightness and gamma options. There is a default palette, but other palettes can be loaded in the form of standard Fractint MAP files. These are text files which list (up to) 256 colors in red, green and blue triplets. The following fragment shows the first 5 lines of a hypothetical MAP file, which would represent pure black, pure red, pure green, pure blue and pure white:

```
0 0 0 SAMPLE.MAP
255 0 0 anything after the triplets is treated as a comment
0 255 0
0 0 255
255 255 255
```

You can create a MAP file manually in a text file editor, use the palette editor in Fractint, use any number of freeware and shareware palette editors, create your own programs to write palettes in MAP file format, or utilize the huge number of MAP files available on the Internet. There is a small set that comes with Fractint, and there is an extensive collection called FracXtra which is available on many fractal-oriented Web sites. I've also placed a ZIP file containing a small set of MAP files at my Web site.

A dialog box displays your selected palette and offers you the chance to confirm it, try again or cancel. After loading a palette, you can start a new image with this palette by selecting the **New Image** or **Infinite Iteration** options. Or you can **Re-Iterate** the image, adding new points to the current image using your new palette, combining it with the previous palette.

Load Parameter Set: Parameter sets are saved in binary FLM files. The information stored in one of these files allows you to duplicate the image as it was when you saved it, except for the palette. By using the **Re-Iterate** option, you can render one image, load a new parameter set, and then superimpose the new image over the old one. Of course, you can start from scratch by using the **New Image** or **Infinite Iteration** options.

Save Image: Your image will be saved as a 24 bit TARGA file.

Save Parameter Set: This option saves the parameters of your current image (except for the palette) to a binary FLM file. You can use this file via the **Load Parameter Set** option to re-create the image.

Image Size: This option allows you to select the dimensions of your image by entering the desired width and height, between 200 and 2,000 pixels, in a dialog box.

Image Parameters: Each image can have one to six sets of parameters. If you select the **New Random Set** option, the number of sets is determined at random. Each set uses one of the seven types of formulae (*linear, sinusoidal, spherical, swirl, horseshoe, polar or bent*) and has its own 3 x 2 matrix of coefficients. See **Flame Background** for more details. The *probability* of any set

being chosen during each pass through the iteration loop is set by the randomization option to be equal, and the total of the probabilities is 1.0, but you can try changing the probability settings.

The default *zoom parameters* include a *scale* of 1.0 (no zoom) and *x center* and *y center* values of 0.0. These values can be changed manually in this dialog box, or by using the mouse-driven zoom box, explained below in the **Zooming** section. Scale values greater than 1.0 reflect zooming in, and values less than 1.0 reflect zooming out.

The *oversampling factor* can be 1, 2 or 3. If the factor is 1, the calculated points are mapped directly to pixels within the dimensions of the image. If the factor is 2 or 3, a virtual image is created with dimensions that are twice or three times the dimensions of the final image. In effect, what is happening is that the image is being divided into sub-pixels. The significance of this process is that when you apply the filter, it operates on the sub-pixel level to give a more finely-detailed image. The expense of this process is, of course, CPU time and memory. It seems that no matter how much RAM you have, when the buffers for this memory are first allocated, Windows may decide to relegate them to virtual memory on your hard disk. The rendering process may slow down significantly and you will hear the hard disk being accessed repeatedly. If this happens, then as you continue to render images or re-iterate, the memory will be given higher priority until you are using true RAM, which will restore the speed and efficiency of the process. As you might suppose, a typical trick is to experiment with small images with no oversampling, and then, when you get something you like, increase the image size and add oversampling.

Coefficients: As explained in the **Image Parameters** section, each parameter set has a 3 x 2 matrix of coefficients. You can alter these to your heart's content. I can't give you any methodical way to do this, so you're on your own.

Color Parameters: The *filter radius* varies from 1 to 100 and, depending on the oversampling factor, operates either on a pixel or sub-pixel level. The larger the radius, the blurrier the image and the slower the process of filtering.

The *brightness* factor increases or decreases the red, green and blue attributes of each pixel by the selected value from -100 to +100. Keep in mind that if you have two pixels with attributes of (245, 246, 247) and (254, 254, 254), and you apply a brightness factor of +10, both pixels will end up as (255, 255, 255). If you then try to decrease the brightness, both pixels will behave identically from then on, so you've eliminated their differences.

The *gamma* factor applies standard gamma correction curves to the image, affecting color attributes near the extremes of 0 and 255 minimally, and affecting attributes in the middle most significantly. Imagine the curves of the family of equations: $y = x$ to the $(1.0 / \text{gamma})$ power between $x = 0.0$ and $x = 1.0$. The range of gamma values is from 0.1 to 10.0.

If brightness is set to 0 or gamma is set to 1.0, the **Apply Brightness** or **Apply Gamma** options will be disabled, since there would be no effect in applying the options at those settings.

New Random Set: This option causes a new image to be rendered using randomly-selected numbers of sets, image formulae and coefficients. See the **Image Parameters** section for more information.

New Image: This option causes a new image to be rendered, using the currently-selected parameters and palette. The previous image will be erased.

Re-Iterate - Superimposing Images and Palettes: When an image is rendered, the program cycles through an iteration loop 100,000 times. On most computers, this will be quite fast, yet it will develop the image sufficiently to demonstrate most of its qualities. However, for a final image, you will probably want to continue to iterate several more times, so that more points will be hit. The Re-Iterate option allows you to do this as many times as you wish.

However, my friend Jeff Field has observed that "the qualities of some images will only become apparent after many iterations. For some random parameters, at first only a kind of blurry mass of dots will appear. But after re-iterating about 15 - 20 times, intricate sets of intersecting curves emerge."

If you zoom deeply into an image, you may encounter a blank screen. This can happen because the first time around, the random selection of the various parameter sets may result in no hits in the region you are zooming into. So, select the Re-Iterate option several more times if necessary, and eventually you should see the desired portion of the image.

The **Re-Iterate** option also allows you to render and manipulate an image, then load another parameter set which you can *superimpose* over the first image. You also can combine two or more palettes with the same or different parameter sets. If you want to try this technique, remember *not* to select the Render New Image option, which will erase everything and start a new image from scratch. There is no simple formulaic way to superimpose images and palettes, so the best approach is to experiment freely and save often.

Infinite Iteration: A number of users have requested this option, which allows the iteration loop to continue indefinitely. The loop is terminated when a key is pressed or a mouse button is clicked.

Un-Zoom: This option restores the default zoom values to a scale of 1.0 and center values of 0.0. You can then select either the **New Image**, **Re-Iterate** or **Infinite Iteration** options.

Apply Filter: This option will apply a Gaussian-type filter to the current image. The pixel (or sub-pixel) radius of the filter is selected in the **Color Parameters** dialog box. If the *oversampling factor* is set to 2 or 3 in the **Image Parameters** dialog box, this option will be disabled after one application. This is because the result of the filtering is displayed on the screen, but the sub-pixels in the filter buffer are not changed, so repeated applications of the filter would just regenerate the same image. As soon as you **Re-Iterate** the image or render a new image (**New Image**), the filter buffer will be updated and this option will be enabled again.

Note: If you change the oversampling factor and answer "No" when the dialog box asks you whether you want to proceed with a brand new image, you then can apply various options to the old image, save it, or superimpose a new image (with the new oversampling factor) over the old image. But as soon as you apply the filter, the old portion of the image will be lost, as only the newly-rendered component of the image is stored in the filter buffer, and that is what the filtered image is derived from.

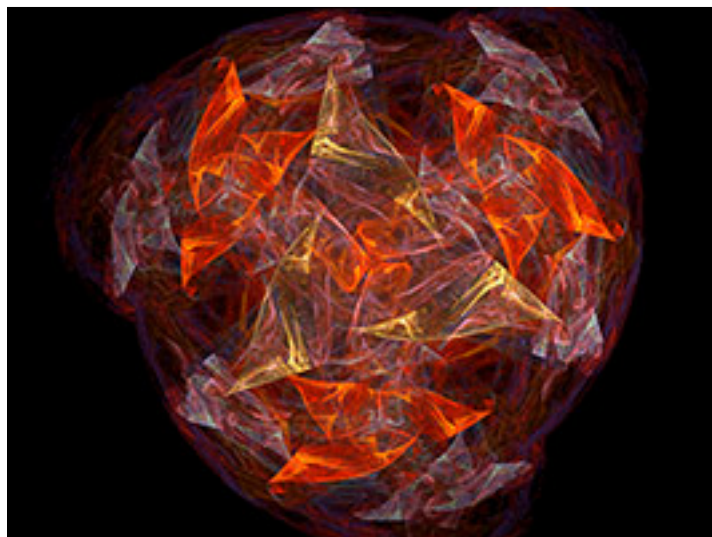
Apply Brightness and Apply Gamma: These options are adjusted in the **Color Parameters** dialog box. If brightness is set to 0 or gamma is set to 1.0, the respective options will be disabled, since there would be no effect in applying the options at those settings.

1c: Using the Program: Zooming

You can zoom manually from the **Image Parameters** dialog box. This is the only way to zoom *out*, which you can do either to see more of the image or to re-center it. When the *scale* value is set to a value less than 1.0, you will zoom out, and when it is set to a value greater than 1.0 you will zoom in.

You also can zoom in using the mouse-driven zoom box. Locate the mouse cursor over one corner of the area you want to zoom into. Then keep the left mouse button depressed as you drag the zoom box to the desired size and shape, then release the button. At this point, if you click the left button, a new image will be rendered. If you change your mind, click the right button and the zoom box will be erased. The aspect ratio of the zoom box will be displayed in the caption bar of the window. This will allow you to maintain the aspect ratio of the image (width / height).

If you zoom manually with the dialog box, you can use the **Re-Iterate** option to superimpose images of different scales.



2: Flame Background

Flame images are combinations of chaotic attractors which are generated by an Iterated Function System (IFS) technique. The original flame program was created by Scott Draves for the UNIX-based GNU platform in 1992. His Web site

<http://scottdraves.com/>

contains many beautiful examples of flame images, as well as considerable information about the process of rendering them. Since Scott was kind enough to make the source code of his program publicly available, I was able to use it to make a port to the Windows 95 platform. In the next paragraphs I will attempt to summarize the basic concepts, and how I've applied them to my program. Any text in **red** is a direct quotation from the files at Scott's Web site. I refer you there for more detailed explanations.

Unlike most methods for rendering images of attractors, which use one formula and one set of related parameters, the flame technique uses up to six parameter sets. Each set uses one of seven types of formulae (linear, sinusoidal, spherical, swirl, horseshoe, polar or bent) and has its own 3 x 2 matrix of coefficients. At each iteration of the rendering loop, one of the sets is chosen at random, the resulting point is plotted, and the point is plugged into another randomly-chosen set for the next iteration. It is possible to use the same formula for each set, or any other possible combination. The **New Random Set** option assigns an equal probability that any of the sets will be chosen for any given iteration, but you are free to alter the probabilities in the **Image Parameters** dialog box.

I've added the ability to combine multiple images, or different scales of the same image, by superimposing the currently-selected image over the previous one. You also can combine multiple palettes. You can make a soft blur of one image to serve as a background for a sharper, more detailed foreground image. For more information on superimposing images and palettes, see the description of the **Re-Iterate** option.

The filtering option can serve several purposes. No matter how many times you “re-iterate” 100,000 cycles, many images will still have some discontinuities. Since the filter is a variation of a Gaussian blur, it can smooth out those discontinuities. As mentioned above, you can make a rather severe blur as a background for a superimposed image. The most subtle effects of filtering involve “oversampling”. This consists of plotting the image to a virtual screen of double or triple the dimensions of the final image. In effect, this divides the pixels of the eventual image into four or nine “subpixels”. The filter is then applied on this subpixel level, resulting in a much more subtle blending of colors, and finer details suitable to the concept of fractals. (Increasing the gamma value can help to bring out these details.) This doesn't mean that these oversampled images are always “better” from an aesthetic viewpoint, but it's worth the extra stress on your computer system to experiment with this technique and explore the possibilities.

Here is some further discussion by Scott:

Flame is the name I use for my iterated function system (IFS) software. Its primary purpose is to create shape. The basic idea of IFS is that from a handful of simple functions on the plane (e.g. a 3 x 2 matrix, or the complex log function), one can generate a picture. Every set of functions defines a different picture. By continuously varying the parameters, the picture can be animated. The shapes appearing in the pictures are fractals, strange attractors in particular.

I produced my first quality monochrome images by collecting a high-resolution (typically 3000 x 4500) 2D-histogram of these points, taking its point-wise log, and then filtering down to 1000 x 1500 (the aspect ratio of 35mm film). Because fractals have such detail and texture, correct filtering is essential. The addition of color resulted in some nice pictures.

That much is easy, the hard part of making an IFS picture is picking the functions! Because I was programming with C, this problem naturally broke into two stages: parameterizing a portion of function-space, then picking the coordinates of the desired functions. Like most people, I started out with affine functions (3 x 2 matrices). To produce pictures with greater variety of character, the matrix may be composed with a “variation”. Over many months I developed six variations:

- linear: none**
- sinusoidal: component-wise sine**
- spherical: project around unit circle**
- swirl: rotate by radius**
- horseshoe: rotate by angle**
- polar: log transformation**
- bent: piecewise linear**

Note: The current implementation of “*Flam3*” has a total of 97 variations.

Rather than parameterizing with an integer specifying one of these, a vector in R7 specifies a linear combination. This has the advantage of being continuous. Thus each function has $6 + 7 = 13$ parameters.

The above parameterization contains only the tiniest sliver of function space, but already the user is confronted with as many as 75 dials to set in order to specify a picture. Furthermore, the relationship between the parameters (dials) and the picture is entirely non-invertible (as is the nature of complexity). So how can one produce interesting pictures?

I chose brute force: generate many random parameter sets, render them very quickly at low-quality, and save any good ones that appear. It was pretty easy to collect 200 that I liked. To guarantee that the matrices are contractive, their entries are chosen in the biunit cube.

This departs from the traditional creative process by factoring it into spew and filter phases. Filtering is easy because it's multiple-choice: you only have to recognize what you want. What makes a good spew? Diversity and density.

The IFS algorithm:

```
draw point x
pick random i
 $x = f[i](x)$ 
repeat
```

works because a string of random digits will, in the limit, include every possible sequence of digits as a subsequence. But how do we get the first point in P? As long as the functions are contractive, then the above algorithm will work given any x because P is an attractor. You only have to throw out the initial dozen or so iterations while x settles into the fractal. Furthermore, this gives us more than just membership in the set, it gives density: how likely are we to see the point (or one very near it) as the algorithm runs?}

This is what Scott has to say about his coloring method:

A monochrome histogram viewed with a colormap works ok ..., but it's possible to do better. Add another dimension to the IFS, and map this value. This coordinate is independent of the others, and its functions are very simple: the first function of the set averages the color value with zero, the other functions average it with one. The colormap value of a point is then simply the base-two interpretation of the string of function choices made by the IFS algorithm, a number between zero and one (the most recent digit has most significance).

The advantage of this scheme is that the coloring is structural. The disadvantage is that the histogram must now be three channels (RGB) instead of one. Note that the log only applies to intensity, so the RGB values are transformed into YUV coordinates, and the log is applied to Y only.

If you want to see what the above really means in terms of attractor formulae and coloring algorithms, I recommend that you get the most recent source code for “*Flam3*” from Scott's Web site and wade through it. Then you will have a real vested interest in comprehending what's going on.

<https://github.com/scottdraves/flam3>

Also check Scott's paper, “*The Fractal Flame Algorithm*”, at

<http://flam3.com/flame.pdf>



3: Acknowledgments

Thanks to Fractalier Listserv members Fausto Barbuto, Dave Dobbs, and Jeff Field for ongoing encouragement, testing and constructive criticism. Jeff's suggestions, including using multiple palettes in a single image, have resulted in numerous improvements in the program.

Thanks, of course, to Scott Draves for creating the concept of flame fractals to begin with, and for making his source code available. Check the **Flame Background** section for more information, and for the URL of his Web site.



Programming and help file authoring by:
Michael Sargent
South Burlington, Vermont
April 2018

msargent@uvm.edu
<http://www.uvm.edu/~msargent>

4: Disclaimer

This is unsupported, non-standard software. It is provided “as is” and any express or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed. In no event shall the author be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this software, even if advised of the possibility of such damage.