Glitch Flames

Published by ricksidwell View all posts by ricksidwell

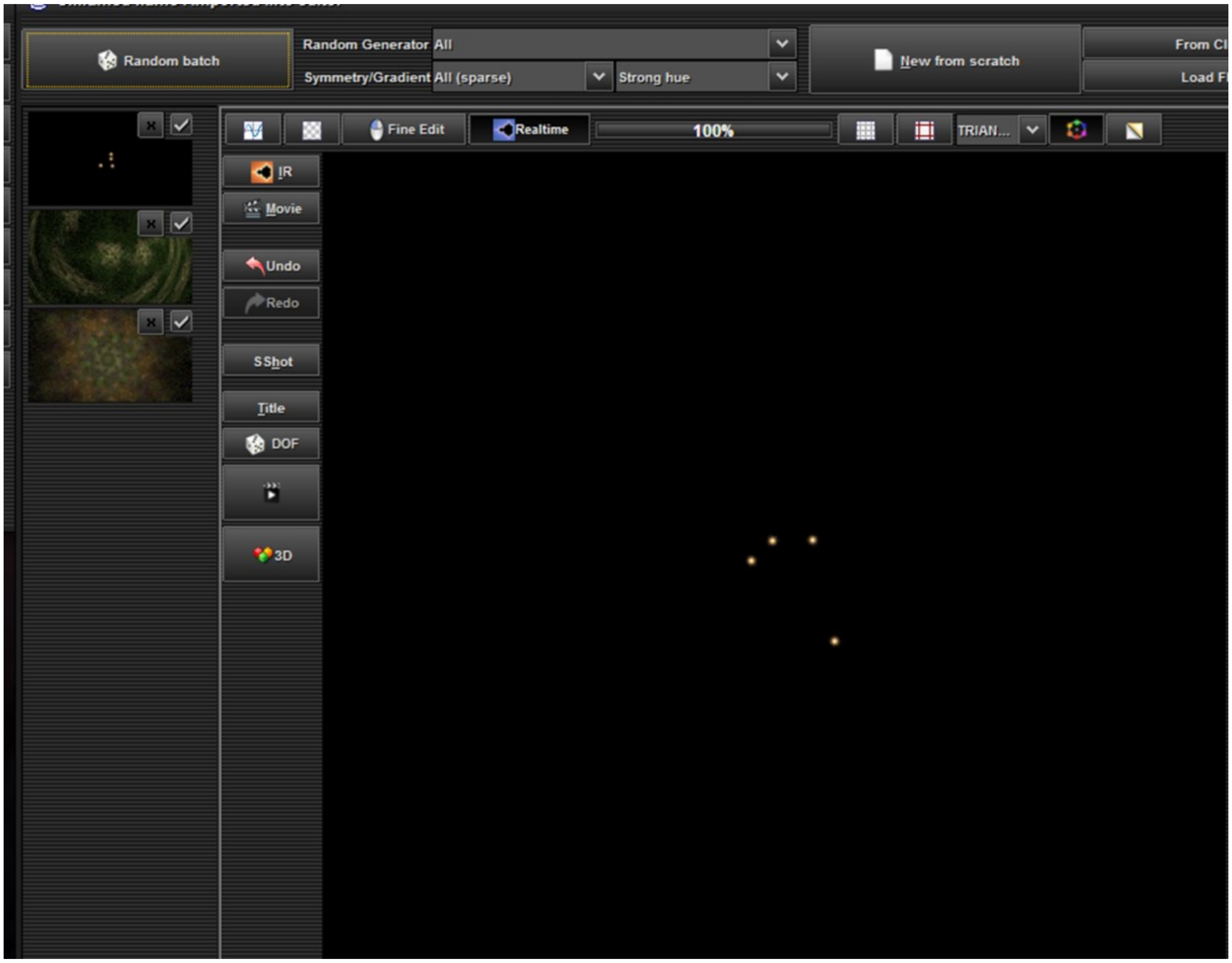
The glitch flame style was first described by Jessica Darling (aka FarDareisMai) in a DeviantArt journal post (https://fardareismai.deviantart.com/journal/What-is-Glitch-Style-392968745) in August 2013. But it really doesn't give a clear definition, stating just that a glitch style flame contains "patterns and textures that almost never occur naturally in apophysis". I'm a bit more pragmatic, so after experimenting with her parameter pack and reading between the lines of her description, I've come up with my own definition: A glitch flame is one that is very sensitive to the initial points of its orbit. When a flame is rendered, a random point is chosen to start the orbit. Most flames are attractors, so repeatedly iterating any random initial point will produce the same flame. Glitch flames are not; the orbits of two different initial points may each contain hundreds of unique points, but have no common points. This isn't a property that Jess mentions in her post, but it does tend to produce unique patterns and the items in her bullet points do tend to create flames that render differently, so I think we're mostly talking about the same thing here. This is the definition I'll be using in this article.

The most obvious consequence of this is that a glitch flame looks different every time it is rendered since different initial points are chosen each time. This is most apparent when making smooth changes to the flame, such as rotating a triangle or dragging to increase a variation parameter. As we do this, the fractal program does a series of quick renders so we can judge the effect of our action. For normal flames, the change is fairly smooth; each quick render is nearly the same as the previous one. But the change is very jumpy for glitch flames; each quick render starts with a different random point, so results in a very different flame.

Flames that change with every render sound unstable, perhaps even "broken". It does make them hard to deal with since test renders are irrelevant. Even trying to compose an image with glitch flames seems impossible since simple changes like rotating or shifting the flame cause a new render with a different appearance. There are some solutions, but let's first see how these flames work and what makes them glitchy. Their nature is more evident in JWildfire than Apophysis (we'll see why as we explore them), so let's start by making a simple glitch flame in JWildfire.

Start like making any new flame from scratch. Here's the process I use: Click "New from scratch" to make a new flame, then on the Transformations tab click "Add" to add a transform. Some dots will appear in the flame window; if they are not easily visible (or you just don't like the color), change the gradient to something that works better; I usually just click "Rnd grd" on the Gradient tab a few times until the dots are clearly visible. (This probably won't be the final gradient; it's just a starting point. Choosing a better one will come much later in the process.)

Before going on, let's pause a moment and analyze what we see. It's just some random dots, but we can learn a lot from them. When JWildfire renders a flame, it initiates a number of threads running in parallel. Each runs the flame fractal algorithm by choosing a random point and iterating it using the defined transforms. At this point, we only have one transform, and it does nothing to the point. So the orbit of each point is simply the initial random point repeated over and over. That shows up as a number of random dots, one for each thread. A partial screen shot is shown below; there are four dots because my PC has four cores. A computer with eight cores would show eight dots.



Note that the dots in the main flame viewer and the dots in the thumbnail to the left (the top one) are different. This is because they are rendered separately. Since the orbits just repeat the initial points, they are obviously sensitive to them; by the definition above, this is a glitch flame! Admittedly not a very interesting one, but it shows how they work. Try right-clicking the thumbnail to force a re-render. The dots change since new random initial points are chosen.

Let’s continue with making a more interesting glitch flame. Click the transform to select it, then go to the Nonlinear tab and change the variation from linear3D to spherical. There are still a bunch of random points, but now there are twice as many. Spherical basically reflects the flame across the unit circle, so the orbit for each initial point now consists of two points: the random initial point and its reflection across the unit circle, alternated over and over.

Now click the “L” button to add a linked transform. The points will jump as the flame is re-rendered, but nothing much will change. Let’s add some more points. With the new linear3D transform selected, go to the Affine tab, enter “20” into the rotation field, and click the “Rotate left” button. That will add a lot more points: nine times as many to be exact. This flame is still simple enough we can predict the orbit for each initial point. Since the transforms are linked, they are executed alternately. So for a given initial point, transform 1 is applied and the first orbit point is its spherical reflection. Then transform 2 is applied and rotates the point 20° counterclockwise, becoming the second orbit point. Then transform 1 is applied, reflecting that point to make the third orbit point. This continues forever, but after eighteen rotations the orbit point has circled back to the initial point, so the orbit points repeat after that. (Eighteen is the number of degrees in a full circle, 360, divided by our rotation, 20; 360/20=18.) Only nine of the points actually appear because the first transform is hidden (that’s part of adding a linked transform).

So now we have lots of points, and sometimes they are even randomly arranged in interesting patterns. But let’s add a little more complexity. Select transform 2 and in the Nonlinear tab change the linear3D to bCollide. The default values for bCollide (num=1 and a=0) make it work the same as linear, so nothing will change except the arrangement of the dots. But change parameter a to 0.3 and a more interesting pattern will appear. Maybe; if not, try re-rendering to get something different. Here are three images I got with this process. They all use the same parameters; I just re-rendered to get different results.

Note that the basic structure is the same; it comes from the spherical-bCollide combination. But the details are very different. This is really difficult to work with! If you find something you like, you can’t change the resolution or the color since that would force a new render. Fortunately, there are a couple of ways to make them more stable.

As we’ve seen, glitch flames are unstable because they are very dependent on the initial random points chosen by the flame rendering algorithm. What we need is a way to replace the initial points with points of our own. There are a couple of ways to do this. We start with a method that is somewhat tedious and has some weaknesses, but is instructive. It relies on a trick: transforms with no variations, which create a point at the origin that we can move with post transforms. Let’s repeat the glitch flame from above with this trick:

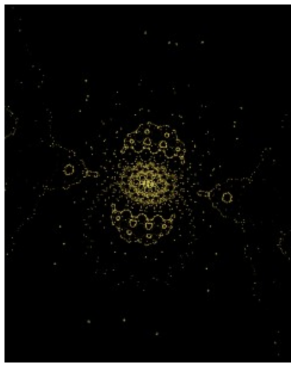
Start with the usual: “New from scratch” followed by “Add” and a quick gradient check. Now change the linear3D to nothing by scrolling to the very top of the variation list and choosing the very first item, which has no name. The random dots from the linear3D are replaced with a single dot at the center. We’ll want a few of these, so click “Dupl” three times so there are four transforms with no variation. (You can make more if you want, but we’ll stick with four here.)

Next, we move the dots around. Click the “Move triangles” icon and then look at the “Toggle post transform mode” button; if it isn’t highlighted, click it to enable post transform mode. Select transform 1, then move its triangle somewhere. You’ll see a point move from the center. Select transform 2 and do the same, then repeat for transforms 3 and 4. The flame will now consist of four dots positioned however you moved the triangles. It looks much the same as a new flame with just a linear3D variation (if your computer has four cores), except that the dots don’t change every time you render; they stay where you placed them.

Click “Add” to add the fifth transform, and change the variation from linear3D to spherical. As above, this doubles the number of points, so now there are eight points. One extra step is needed here: change the weight from 0.5 to 500. We’ll see why this is needed in a moment.

Finally, as before, click “L” to add a linked transform, and on the Affine tab uncheck Post TF and rotate the pre transform left 20°. Change the variation to “bCollide”, and change the parameter a to 0.3. The flame now looks like one of the ones we had before, but this time it is stable; re-rendering it doesn’t change it since the initial points are now chosen by the first four transforms, not randomly.

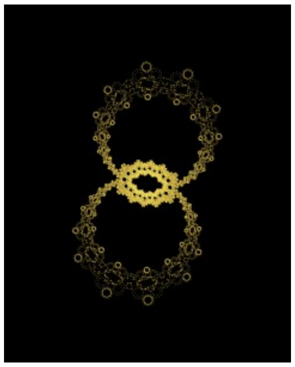
To change our new flame, select one of the first four transforms, go back into post transform mode, and move it around. Move the others around to change it further. Now when we find something we like, we can make refinements and it will still keep its basic appearance. We can also save it and get the same flame back when we reload it.

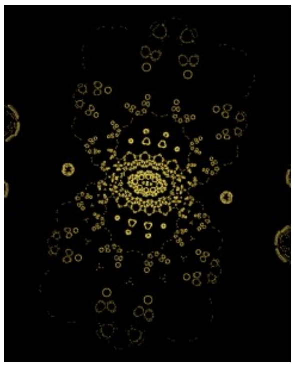


But you may have noticed that the new stable flame doesn't have as much detail as the unstable one we are trying to emulate. To see why, let's consider a possible orbit for this flame. We start with a random point and apply transform 1, which replaces it with a different point specified by its post transform. Then the next transform is chosen; the bCollide transform is linked to the spherical one, so it won't be chosen, but any of the others might be. Since the spherical transform has a huge weight compared to the others, it is most likely to get chosen; let's say it is. The point is reflected, then the linked bCollide transform is applied, which rotates it and does the bCollide operation. It is again time to select a new transform to apply; again, it will be something besides bCollide, with the spherical transform having the highest probability. The computation bounces back and forth between the spherical and bCollide transforms until randomly one of the other point transforms is chosen, which resets the point to whatever is specified by its post transform, and the process repeats.

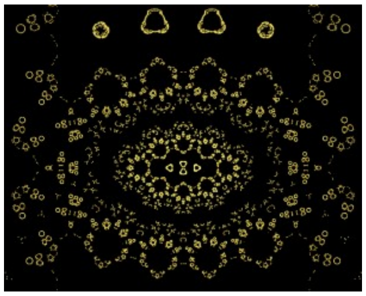
Here's the problem: the computation starts over again each time the spherical transform is not chosen. Even though it is a thousand times more likely to be chosen than the point transforms, getting the details we are missing requires more than a thousand iterations. But setting the spherical weight more than a thousand times higher than the point weights won't work due to the way the rendering algorithm is implemented.

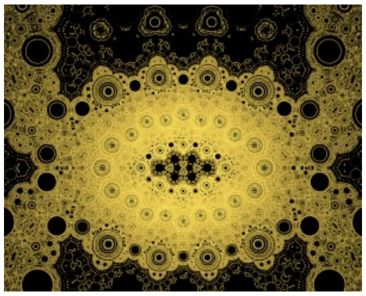
Fortunately, there is another way that is both easier and overcomes this limitation. Starting with JWildfire version 3.30, there is a variation named pre_stabilize that helps to stabilize glitches (I call it "glitch repellant"). Just add it to the first variation. So for the original glitch flame we made above with just spherical and bCollide, select the first transform (the spherical) and in the Nonlinear tab for variation 2 select pre_stabilize. The glitch flame is no longer glitchy; re-rendering produces the same result. To get a different result, change the pre_stabilize parameter seed. This works best when realtime (progressive) preview is on, set with the blue button above the flame preview. Here are renders of the glitch flame we have been using with two different seeds:



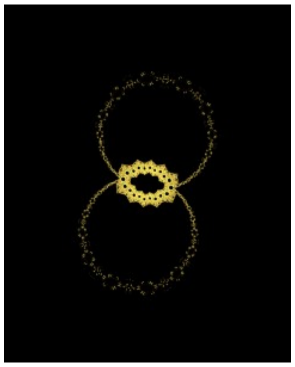


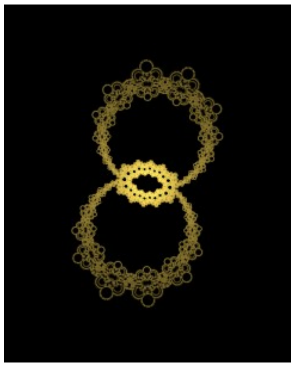
The pre_stabilize variation uses the same principle as the point variations; it chooses a set of points that replace the random ones generated by the rendering algorithm. They are random, but the sequence is repeatable, determined by the seed parameter. Each seed value generates a different set of points, but a particular seed value will always generate the same set of points, thus stabilizing the flame. The number of points is set with the n parameter; increasing it adds more initial points, which often adds more details to the resulting flame. This is useful because glitch flames often lack detail. For example, the first image below is a close-up and rotation of the last flame above, which has n set to 4. Doubling it to 8 adds significant detail, shown in the second image below.



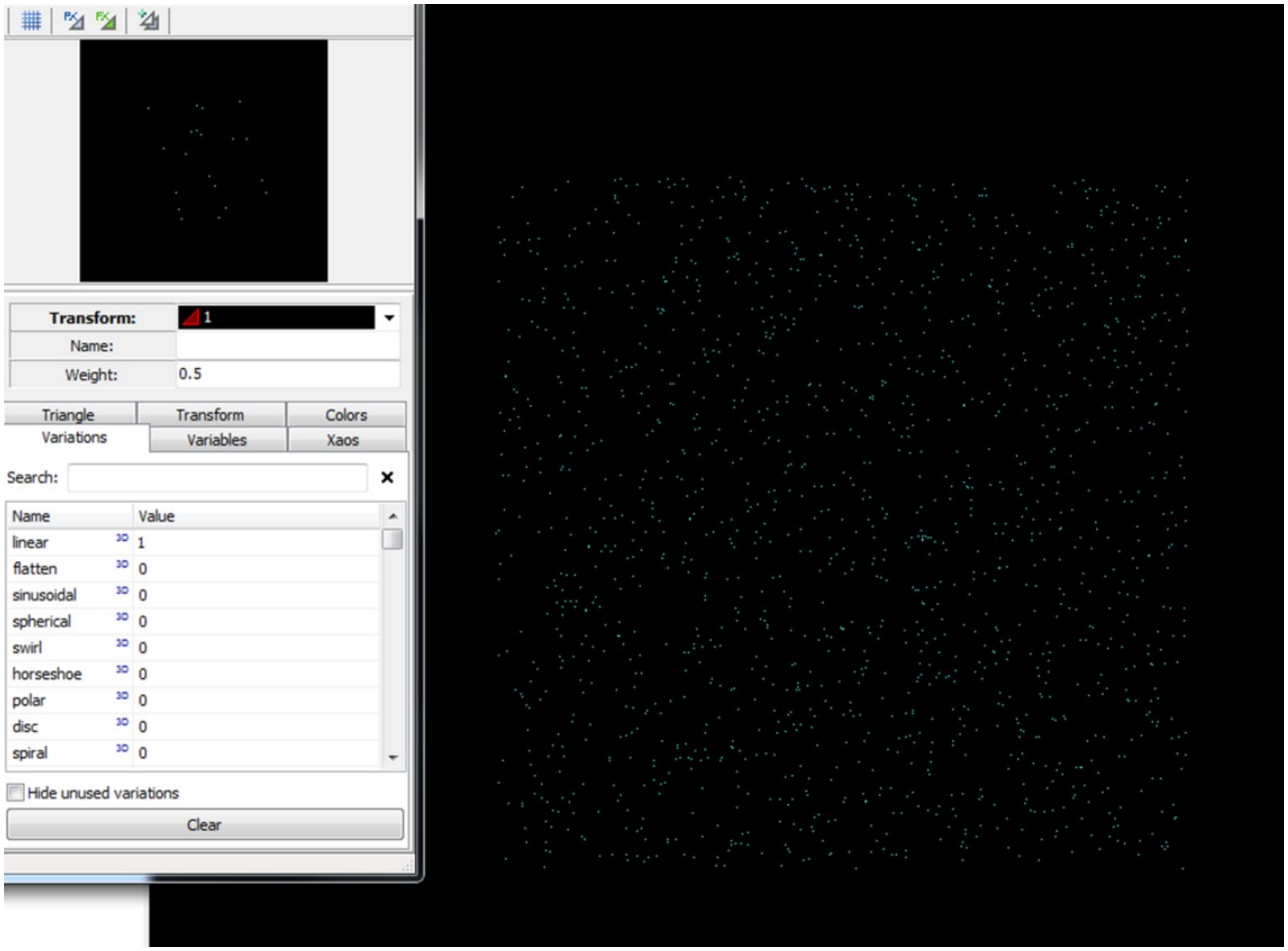


The other parameter, p, is the probability that a new initial point will be chosen; it is multiplied by 1000 for convenience. So the default of 0.1 means that for each iteration there is a probablility of 1/10,000 that the current sequence will stop and a new one start, and a 9999/10,000 chance that the current sequence will continue. Decreasing p will make the sequence continue longer before resetting. This will sometimes bring out more details, and sometimes just reduce its brightness. This is illustrated in the following images; the first has p=0.1 and the second has p=0.001. Notice the details that appear when the sequence is allowed to run longer.





It is important to realize that we’ve only used JWildfire here. Apophysis (the program Jess was using when she wrote her post on glitch flames) behaves a bit differently, as do other flame fractal programs. Let’s take a short detour and see what the differences are, starting with the first simple glitch flame containing only a transform with linear. In JWildfire, it produced one dot for each render thread. In Apophysis, it produces a lot of dots, more in the main window than the preview. Furthermore, the dots are a lot smaller.

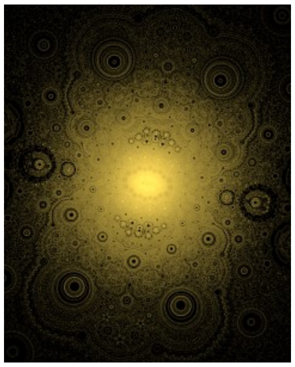


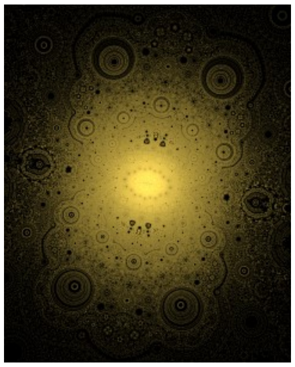
Name	Value
linear	1
flatten	0
sinusoidal	0
spherical	0
swirl	0
horseshoe	0
polar	0
disc	0
spiral	0

The JWildfire render engine starts with a random point (one per thread) and iterates that point until the render is terminated (unless it encounters an error, in which case it starts again with a new random point). But the Apophysis render engine stops when it reaches 10,000 iterations and starts again with a new random point. For most flames, both techniques produce the same image, but the difference is quite apparent in glitch flames. This is probably why Jess states that glitch flames are generally rendered at low density; a high density render would have lots more random start points, which smooth out the glitchiness of the flame.

The dots are larger in JWildfire because of the way it does anti-aliasing. This is controlled in the Anti-Aliasing/Filter tab with the Antialiasing amount and Antialiasing radius settings. You can change the default values for new flames in Preferences; if you haven't changed them they will be 0.25 and 0.5 respectively. Changing either to 0 will disable anti-aliasing and make the dots small like in Apophysis.

These differences make glitch flames rendered in the two programs appear very different, aside from their glitchiness.
The same will be true for other flame programs, such as Chaotica. Glitch flames are not typical, so different render algorithms will produce different results. The following figures show the same glitch flame described above rendered twice by Apophysis 7X16. As above, the flame itself is the same; only the random starting points are different. Although differences are apparent, they are largely the same.





Glitch flames are surprisingly hard to find, or to tweak once you find one. Here is a list of things that deglitch a flame; things to do if you want a stable flame, or avoid if you want a glitch flame:

- Adding blur variations, which are variations that ignore the input points, like “blur”. Instead, they generate a random point within the shape the generate. Obviously, if the input point is ignored, flames that use them are not at all sensitive to the initial points of the orbits.
- Allowing a choice between transforms. Glitch flames usually use Xaos settings to organize the tranforms into a chain, with transforms executed in a particular order. To see why allowing a choice between transforms kills glitches, consider a simple flame with two transforms, either of which may be selected for any iteration. The first iteration will transform the initial point to one of two points, depending on which transform is used. The second iteration will transform the resulting point into one of two points; the result after two iterations is that the initial point is transformed to one of four points. The number of possible points is exponential; three iterations will result in one of eight points, four iterations in one of sixteen, ten in one of 1024, etc. This removes the sensitivity of the flame on the initial points, thus deglitching it.
- Using a variation that maps to multiple points. For example, julian with n=2 will map the input point to one of two output points. The number of points increases exponentially with each iteration as described above.
- Resizing or moving a transform’s triangle without compensating for it, or changing the variation amounts which is usually the same as resizing the post transform triangle. If you shrink the pre or post transform triangle, successive orbit points will be attracted to (get closer and closer to) the origin or some other point. If you make it larger or move it, successive orbit points will get larger and larger (put another way, they will be attracted to infinity). But we can sometimes do these things if we are careful to compensate for it. For example, shrinking the pre affine transform by some amount and increasing the post affine transform by the same amount is a useful technique, though it depends on the variation(s) used.
- Pulling points to an attractor. There are multiple ways to do this, but all will remove glitching. Some variations, like elliptic, will nearly always do this. Some variations, like mobius, will do it with particular parameter combinations, but glitch fine with others.

This isn’t to say to never do these things; doing one of these to a glitch flame might result in something very nice, just not a glitch flame.

With these points in mind, let’s tweak our glitch flame to make it look different. The main shape comes from the bCollide variation, so lets replace that with something else. Specifically, let’s change it to waves2. The default waves2 parameters are way too powerful for a glitch; we need to set scalex and scaley to very small values, like 0.002. On the other hand, it works well to set freqx and freqy to fairly high values, like 100 and 75. Depending the pre_stabilize seed, we get something like the first image below. The rotation of the waves2 transform is 20°, so there are 18 of the shapes, 9 on the outside and 9 in the inside of the unit circle (the reflection caused by the spherical transform). If we change the rotation to 45° (click Reset TF, then enter 45 and click the Rotate left button), there will be 8 shapes, 4 on the outside and 4 on the inside, as shown in the second image. But if we change it to something not so easily divisible into 360, there will be lots of overlapping shapes; the third image below uses a rotation of 16°. Although we can get different combinations by changing the pre_stabilize seed parameter, they are all basically fuzzy coincentric circles. We can make them more interesting using a final transform. Let’s start with a julian (it maps to multiple points, but in a final transform it won’t affect the orbits, so the glitch flame remains a glitch), shown in the fourth image. That doesn’t do much itself since it just maps fuzzy circles to fuzzy circles. But adding just a touch of another variation creates interesting shapes. The last two images show julian with linear and ngon, but lots of other variations work here; try atan (mode 2, increase stretch), bCollide (small amount, like 0.2, and set a to 0.5), butterfly (try different amounts), the list goes on.

Many other variations work with spherical besides bCollide and waves2 that we have used here. Try eMod, eMotion, hexes, and ortho; others work as well. Modify the parameters (and the pre_stabilize seed) to find something interesting.

The lazysusan variation also makes interesting glitch flames, both by itself and in combination with other variations. Start with a new flame, add a transform, change the variation to lazysusan, and set all of the lazysusan parameters to 0. This makes it work the same as linear. Set variation 2 to pre_stabilize and change n to 10. Now go to the affine tab and rotate the transform 90° left and move it 0.75 units down. So far, we just have a bunch of dots as the points generated by pre_stabilize are rotated and moved. To center them, go to the Camera tab and set both CentreX and CentreY to 0.38.

Although the dots technically form a glitch flame, to make it interesting change the spin and twist parameters of lazysusan. Set the spin parameter to any value between 0 and π; it is an angle in radians. (Values outside this range work, but the result will have a large hole.) Change the twist parameter to distort the flame; negative values work well, but the further from 0 will make a more solid, less glitchy flame. As usual with pre_stabilize, change the seed parameter for variations of the pattern, increase n to make it denser or decrease n to make it more sparse.

Lazysusan glitch flames work well in combination with some other variations; just add a linked transform and change the variation to something else. Here are a few to try:

- popcorn2: start with x=0.25, y=0.25, and c=0.05, but change them to find something you like.
- stripes: keep space close to 0, like -0.002; negative values work well for both space and warp.
- voron: keep k near but not equal to 1, for example 1.01.
- waves2: keep scalex and scaley very close to 0, for example 0.003. Try values for freqx and freqy between 20 and 80 (though larger and smaller values also work).

We have one last topic: coloring. So far, all we have seen are monochrome flames. Quoting Jessica Darling’s post referenced above: “Glitch style fractals are VERY difficult to color”. That is certainly a true statement! Here’s the problem: Flame fractals are normally colored based on the sequence of transforms taken to generate a particular orbit point. Besides mapping the previous orbit point to the next one, a transform also maps the previous color to the next one based on the transform’s color and speed settings. When the sequence of transforms is random, the colors of the generated points will vary. But the sequence of transforms for glitch flames is fixed; the same transforms are executed in the same order, over and over. So the colors of the points doesn’t change.

There are a couple of ways to try to get some color variety in glitch flames:

- Add a direct color variation such as dc_crackle, dc_linear, or dc_perlin to the first transform in the chain, with an amount of 0, and click the “post” button to make it a post-transform. Direct color overrides the calculated color for a transform, so it can make a flame colorful. However, the coloring doesn’t follow the structure of the flame, so tends to look artificial. But a highly textured coloring variation with a relatively sparse glitch flame can look OK.
- Use a color map instead of a normal gradient. This essentially replaces the gradient with an image, and uses the colors of the 2D image, which are more varied than a linear gradient. But again, the coloring doesn’t follow the structure of the flame.

- If the flame has more than one transform, the process of linking them makes only the last one visible; the rest are hidden. Sometimes, we can make more than one visible by changing the draw mode from HIDDEN to NORMAL. Frequently, the result will be messy; that's why only one of a linked transform chain is normally visible. If the result is acceptable, set the color values for the transforms to different colors and set the speed for both to -1. This will make the points generated by the two transforms have different colors. Two colors is better than one, and there will actually be other colors where different colored points overlap. But it is still not a lot of variety. An example is shown in the first image below.
- Use two or more layers with the same glitch flame that have different pre_stabilize seed values and different colors. This can be effective, but still has limited color variety and makes it difficult to modify flames since the same changes need to be made to each layer. This method is used in the second image below.
- Set the color speeds of all of the transforms to 1, which will make them use the previous color. With no way to initialize the color, that will be the random initial color selected by the render, so it will change every render. To stabilize this, pre_stabilize has a direct color parameter, dc, in versions of JWildfire above V3.30. Set it to 1 to enable it, and each of the initial points it generates will have a stable color. The last image below was colored using this method.

The last option is the easiest to use, but I didn't think of it earlier so it isn't available in JWildfire V3.30 (the current one as I am writing this). If you don't want to wait for the next JWildfire release, you can download the code from https://pastebin.com/yfvvc0R2 and use it in a custom_wf_full variation.

Behind the Scenes