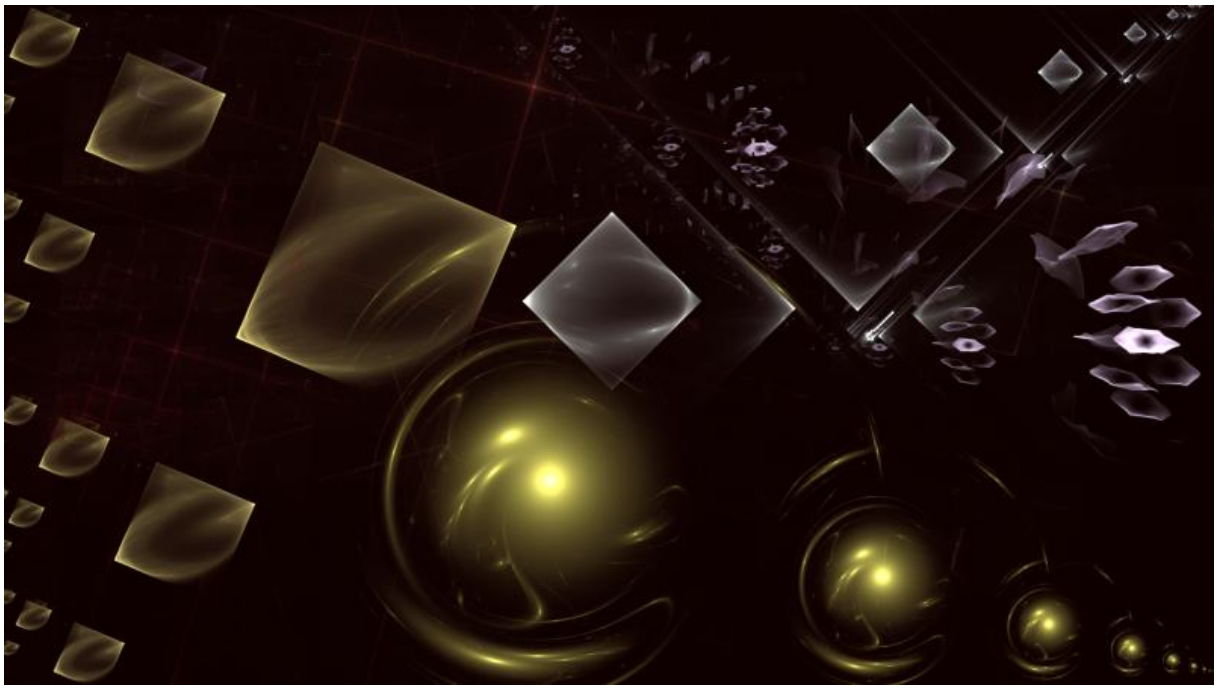


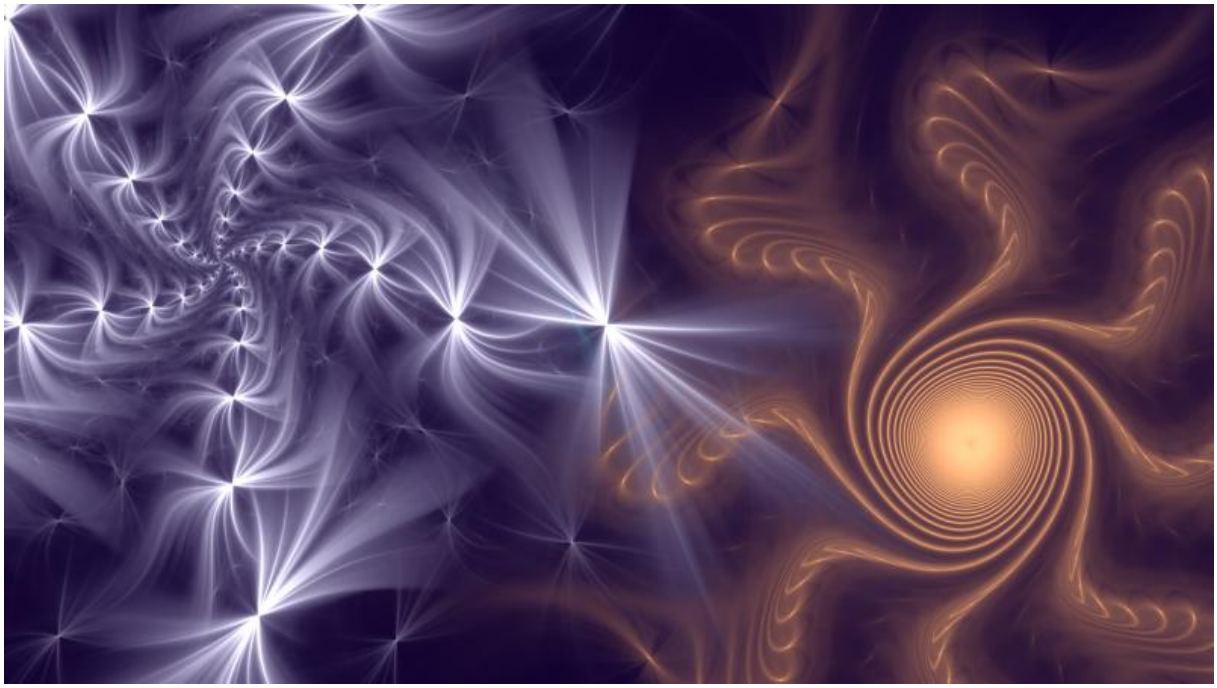
JWildfire tutorial: Mastering "xaos" (relative weights) to compose fractals

The following tutorial describes a technique to compose one fractal of several other fractals where the building fractals do NOT affect others. As you may have guessed this technique relies upon the use of relative weights also called "xaos" :-)

The tutorial is not intended to just show how to move some triangles and enter some numbers to finally come out with a fractal similar to one shown here - it is more about anything under the hood. Therefore you should really get some Scotch and/or coffee, sit back and take some time for it :-)

You will need either JWildfire or Apophysis to reproduce things, but I will describe them only for JWildfire in detail, but there should be no problems to adapt things to Apophysis.
In any case feel free to ask :-)

1 Understanding relative weights



1.1 Understanding the "chaos game"

Before we attempt to compose different fractals we should remind how a simple fractal is constructed. Lets start with a really simple *thought* example:

- create one transformation ("triangle")
- decorate it with one or more nonlinear functions
- move, rotate and scale the triangle to the "right" place
- do the same with the post transformation if needed

The chaos game

So what is going on under the hood to produce the visual result? An algorithm called the "chaos game" which works (simplified) as follows:

- a point at random position in space is selected (therefore "chaos game")
- now for a given number of iterations ("samples") there occurs the following:
  - the point is transformed according to the "triangle" of the (affine) transform (rotate, shear, move)
  - the resulting point is transformed using the nonlinear function(s)
  - the resulting point is transformed according the "triangle" of the post transform (again rotate, shear, move)
  - the resulting point is projected into some kind of canvas (the result of this projection will later contribute to the visual image (which is another difficult process))
  - continue with next step (i. e. transform the (raw, i. e. not projected) point again and again ...)

1.2 Understanding weights at the transform level

Now lets create a more complicated still *thought* example:

- create two transformations ("triangles")
- decorate them with nonlinear function(s)
- decorate them with weight factors (see below)
- move, rotate and scale the triangles to the "right" place
- do the same with the post transform if needed

Weight factors = probabilities

Beneath the fact that we now have more than transform there is one more important thing: the weight factors.
The weight factor is used in the chaos game to select a certain transform applied at a certain step (sample). Transforms with higher weights are executed more often than transforms with lower weights. In terms of our chaos games the weight factors are nothing else than probabilities.

So, now with two transforms our *chaos game* algorithm looks as follows:

- a point at random position in space is selected
- now, again for a given number of iterations ("samples"):
    - one of the given transforms is randomly chosen where transforms with higher weight are more often selected
    - the point is transformed according to the "triangle" of the chosen transform (rotate, shear, move)
    - the resulting point is transformed using the nonlinear function(s) of the chosen transform
    - the resulting point is transformed according the "triangle" of the post transform (again rotate, shear, move)
    - the resulting point is projected into some kind of canvas
    - continue with next step (i. e. transform the point again and again ...)

As we can see: every transform affects every other transform because transforms are randomly chosen and after any transform any other transform may be chosen. Specifying weights at transform level can only affect how often a certain transform may be chosen in comparison to another transform.

1.3 Understanding weights at the transition level ("relative weights")

While the approach of specifying weights at the transform level is a very powerful construct (e. g. most of the gnarl stuff works by having very large differences in those weights) it would be also convenient in some cases to be able to "chain" transforms.

I. E. to have a transform where you could specify that another certain transform always should follow. An example could be to make one certain transform a little bit noisy or deformed. Then you would (naturally) want to define one more transformation and link it to the transform which is to be deformed.

Fortunately, there is also a solution for this approach keeping our framework of the chaos game nearly untouched. It is often referred as "xaos", I call it "relative weights" or "transition weights".

This solution works as follows: every transform is extended to carry a transition weight to any other transition and itself. I. E., every transform has now a number of individual weights which number is equal to the total number of transforms. This brings some state into the chaos game: at each iteration where the next transform is chosen now both the absolute weight at transform level and the transition weights of the current transform are taken into account (multiplied).

In JWildfire and Apophysis the relative weights are typically initialized with 1 which results in the same behavior as no relative weights were present.

Chaining transforms

Our example of a chained transform is now easily to solve. Forbidden transitions can be achieved by simply setting the transition weight to zero. And allowed transitions keep unchanged with a value of 1.0.

Let's imagine two transforms A and B where a third transform C should be linked to transform A, i. e. the transform C occurs always A and never occurs twice.

The initial distribution of relative weights without any constraints for transitions would be as follows:

transform A	Transform B	Transform C
weight A->A: **1.0**	weight B->A: **1.0**	weight C->A: **1.0**
weight A->B: **1.0**	weight B->B: **1.0**	weight C->B: **1.0**
weight A->C: **1.0**	weight B->C: **1.0**	weight C->C: **1.0**

Now let us eliminate all transitions which should not be allowed:

- A->A and A->B are invalid because C should occur after A
- A->C is obviously valid
- B->A and B->B do not concern our constraint and are therefore valid
- B->C is invalid because C should only follow A
- C->A and C->B are valid
- C->C is invalid because C should not occur twice

According to the eliminated transitions our weights look at follows

transform A	Transform B	Transform C
weight A->A: **0.0**	weight B->A: **1.0**	weight C->A: **1.0**
weight A->B: **0.0**	weight B->B: **1.0**	weight C->B: **1.0**
weight A->C: **1.0**	weight B->C: **0.0**	weight C->C: **0.0**

Relative weights = probabilities

Before we continue with the practical part of the tutorial lets resume that relative weights (similar to weights at the transform level) are probabilities. They affect how often a certain transform is involved in the chaos game. Therefore any number greater or equal zero is possible and there are no limits for you to experiment :-)

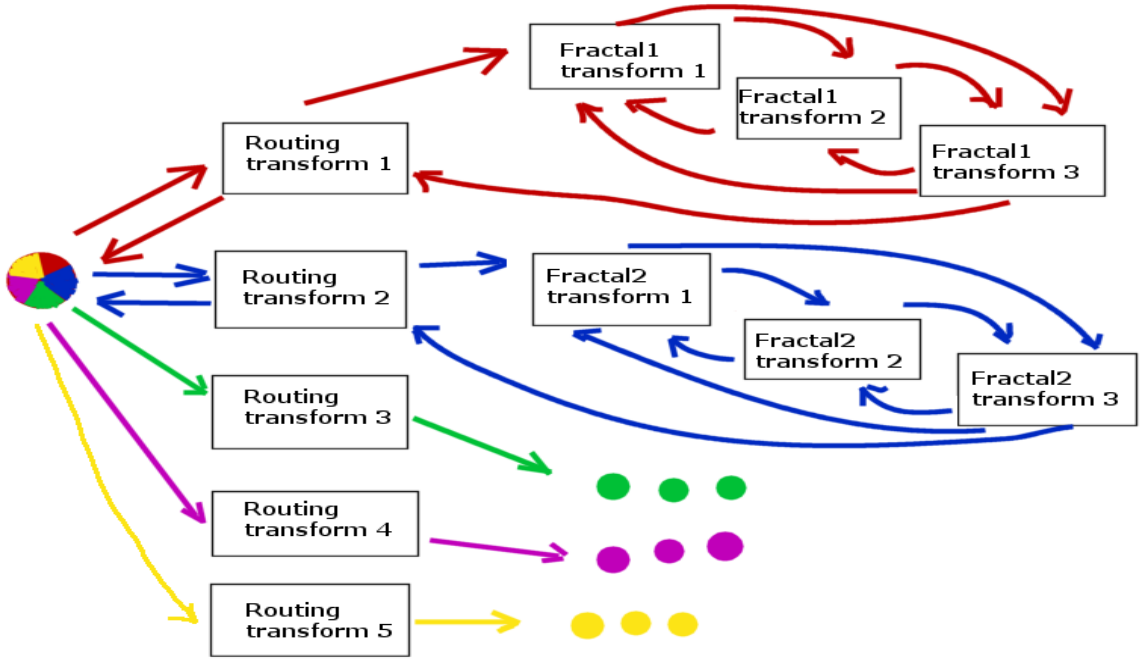
But in this tutorial we will use only relative weights of 0.0 or 1.0 because it's all about enabling/disabling transform transitions.

2 Composing fractals

2.1 Idea

Assume we want to compose N fractals. Then we create at first N "routing" transforms marked as invisible. Each "routing transform" corresponds to one of those fractals and is directly linked with the first transforms of "its" fractal.

All relative weights are distributed in such a way, that the "chaos" travels only between the routing fractal. And as a route is selected only the transforms of the selected fractal are processed. After processing this "leaf" the next route is selected and so on.



2.2 Example

Having put such effort in theory we want to create a non-trivial example now :-) And, even better we create a reusable skeleton.

2.2.1 The skeleton

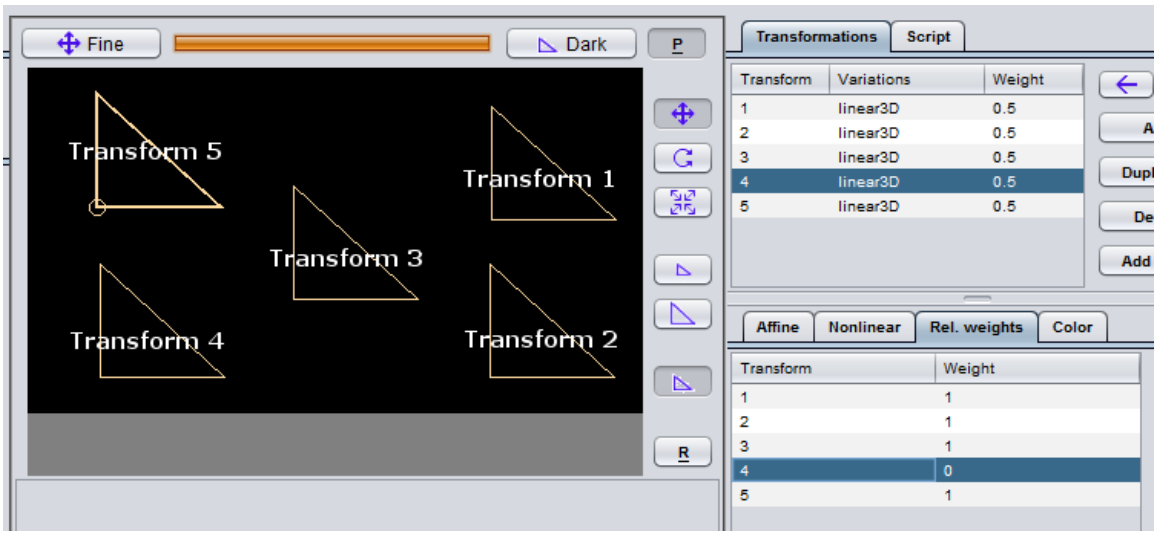
We choose 5 fractals, each composed of 3 transforms. For each fractal we choose a simple Sierpinsky triangle (composed of 3 transforms) as template at first. This has the advantage that the triangles which are linked are also clustered in the triangle view. And it simple to recognize which triangle forms which edge etc.

2.2.1.1 Create the routing triangles

Create 5 transforms (transform1-5) with

- Variation "linear3D" = 0.5 ("Transformations/Nonlinear" tab)
- Weight = 0.5
- Draw mode = HIDDEN ("Transformations/Color" tab)
- the relative weight for itself = 0.0 ("Transformations/Rel. weights" tab), e. g. set the first row to zero for the first transform, the second row for the seconds transforms etc.

Change to "Edit Post Transform" mode (yellow triangles) on "Transformations/Affine" tab and distribute the triangles over the canvas similar to my example in the following screenshot:



2.2.1.1 Create the first template fractal

Create 3 transforms (transform6-8) with

- Variation "linear3D" = 0.5 ("Transformations/Nonlinear" tab)
- Weight = 10.0

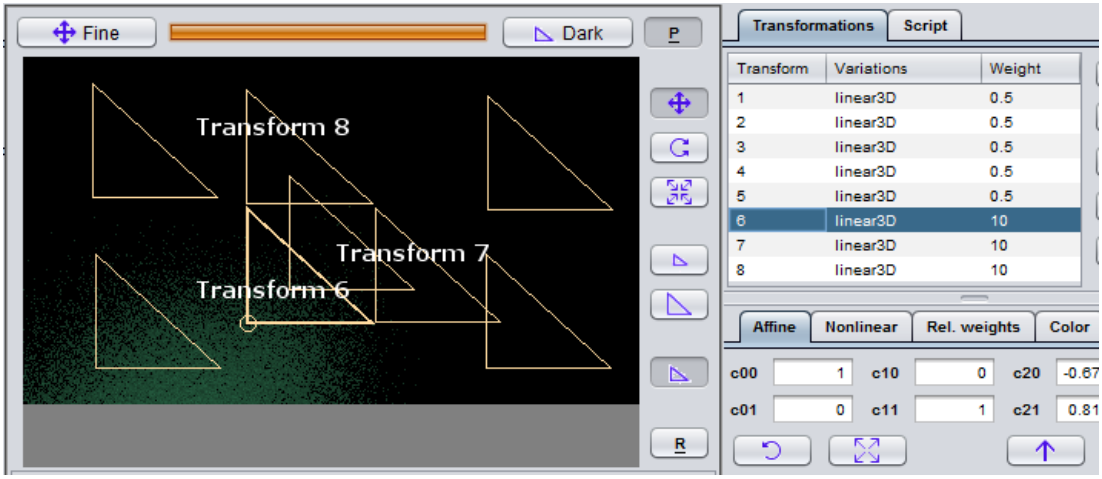
Change to "Edit Post Transform" mode (yellow triangles) on "Transformations/Affine" tab.

Move

- transform6: left-down
- transform7: right-down
- transform8: up

-

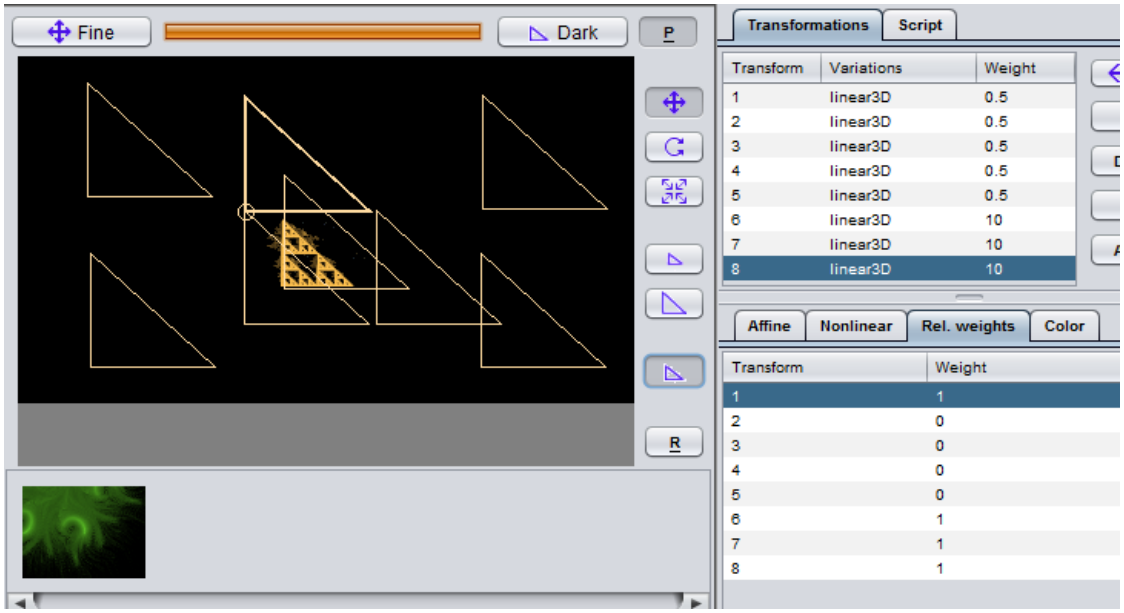
so that they form a larger triangle:



2.2.1.2 Create the routes for the first template fractal

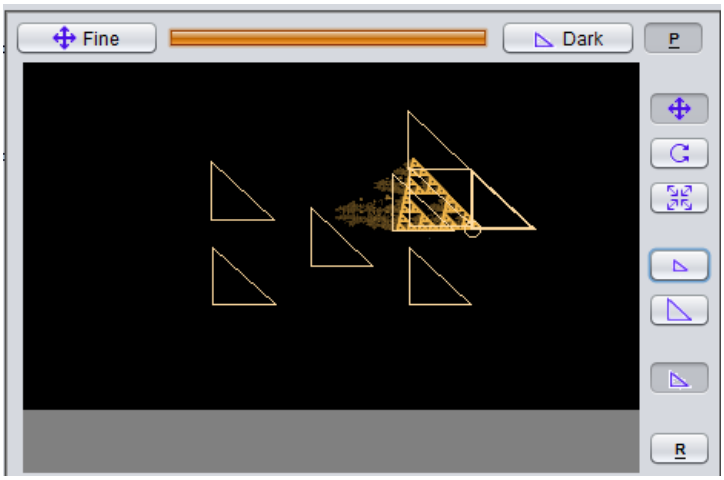
- clear the relative weights for the newly generated transforms in all routing transforms1-5, i. e. set the rows 6-8 on the "Transformations/Rel. weights" tab to zero for transform1-5
- create the route for the first fractal (re)setting the relative weight 6 of transform1 to 1, so only transform1 can call only the first transform of our sub-fractal
- "cage" the sub-fractal: clear all relative weights for routing fractals in the newly generated transforms. I. e. set rows 1-5 on the "Transformations/Rel. weights" tab to zero for transformation6-8
- allow the "jump back" to the routing transform by enabling the transition from the last transform of our sub-fractal to the corresponding routing transform, i. e. (re)set rows 1 on the "Transformations/Rel. weights" tab to 1 for transform8

We should now see a Sierpinsky triangle and are finished with the first fractal template:



Zoom out the triangle view (to shrink triangles) and move the three triangles of the template fractal on top of the corresponding routing triangle to "cluster" things on the screen.

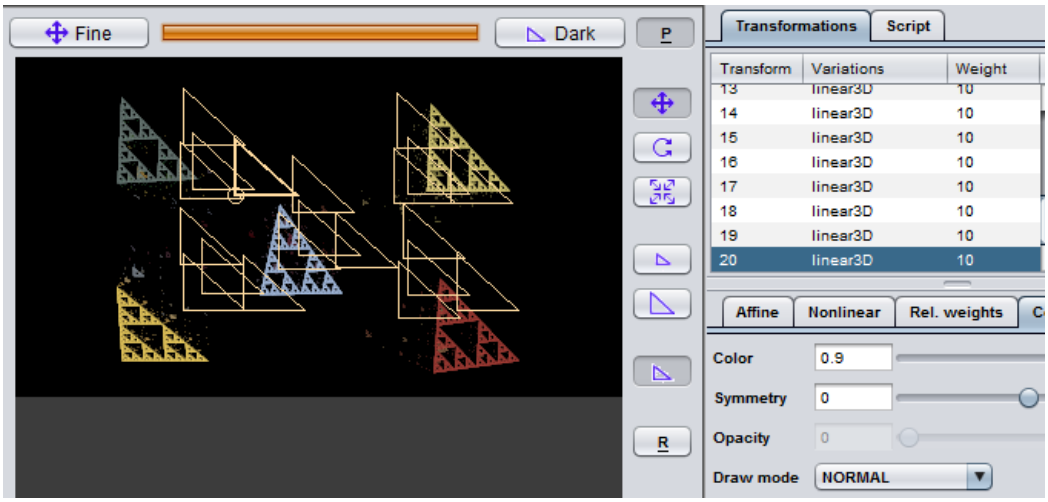
Your triangles should look like:



2.2.1.3 Create the remaining template fractals

Repeat the steps described in the last two chapters to create the remaining template fractals. Because this is now somewhat boring you could also load the provided template (`compose_example5.flame`).

You should end with a view like this:



2.2.2 Playtime! :-)

Now it's time to fill the template with your ideas! Almost anything except post transforms is possible. In some cases it may be tricky to design a certain sub-fractal if there are lots of other triangles are present. In such cases you may find the "trick" described in the next chapter useful.

2.2.2.1 Using prepared fractals

You can also use fractals which you have created before by just performing some Copy&Paste in flame files. The idea is to replace the 3 transforms of one certain sub-fractal in the template by 3 transforms of the external fractal. This only requires special treatment of relative weights.

The external fractal must not have a final transform and should consist of three transforms. If they contain of less just create dummy linear transforms to end with 3 transforms.

Open a copy of the template in a text editor and locate 3 the transforms you want to replace. You achieve this by searching for the text "<xform". The first occurrence is the first transform. The second occurrence is the second transform and so on. Each transform ends with "</xform>" or "/>".

Mark and cut the 3 transforms and paste them in a new file/buffer.

Open the external fractal and locate the 3 transforms to paste in the copy of the template.

Now for each of the pasted transforms:

- search for the text "chaos=" and delete all content between the both quotes after the equal sign
- search for the text "chaos=" in the corresponding transform of the template inside the new generated file/buffer
- mark all content between the both quotes after the equal sign
- paste this content into the copy of the template at this place where you previously deleted all

At any stage you may copy the edited flame file into the clipboard and paste it into JWildfire to check if the file is still valid.

So far for now - I hope you enjoyed this tutorial and will have fun to try things out! :-)