Flame Fractal Variation Guide

Version 1.0, by Rick Sidwell

Several guides have been made to help people understand and compare the many variations available for use in flame fractal programs like Apophysis and JWildfire. They typically take a base flame and show the results of the different variations on it. This guide takes a different approach: Rather than using a common base flame, it uses one that best shows the effect of the particular variation. The hope is that it will make it more clear just what each variation does.

This guide is far from comprehensive! Besides only explaining about a third of the variations, it just shows the effect of a single application. Interactions with other variations, use cases, and above all, the effect of iterations are all beyond the scope of this guide. The variations selected for inclusion were basically the ones I understood and could describe in a quarter to half page. No consideration was given to how useful a variation is in creating fractal art. It's a subjective consideration anyway, but some of the variations included here will rarely be used, while other very useful ones are not listed.

For each variation, the name is shown along with a note whether it is 2D or 3D, and how it works. A normal variation will transform the (x,y) or (x,y,z) point to another (x,y) or (x,y,z) point. A "blur" will ignore the input points (so no starting point is shown) and generate a shape. A "half blur" will also generate a shape, but requires some input points, and will often preserve their colors in the output. For 3D variations, the effect on z is described: "transforms z" means the z value is modified, "sets z" means z is set, ignoring the starting z value, and "passes z" means z is just passed with no change except multiplication by the variation value (so it is technically a 2D variation).

Also for each variation, the support is shown for that variation in commonly used flame fractal programs: Apophysis 2.09, Apophysis 7X version 15B (the last version before the 3D paradigm was changed), Apophysis 7X version 16, JWildfire 2.00, and Chaotica 1.5.2. Possible values are "yes", "no", and "dll" (meaning a plug-in can be used). The plug-in name, if applicable, is listed below the table. Note that every built-in variation in Apophysis 7X version 16 will pass z; this is not noted in the guide.

The variations are listed in alphabetical order of their base names, which is the name after removing the prefixes "pre_", "post_", "dc_", and "Z_". Most variations with these prefixes have a normal version as well, so this puts the versions together. But some don't, so for example, post_rotate_x is located where "rotate_x" would be if it existed.

It is worth noting that pre_ and post_ variations will have no effect by themselves; they need to be combined with a normal variation. Similarly, some variations affect only z, not x and y. In these cases, linear is used along with the variation under study to produce a useful result. (The linear variation itself just passes (x,y) without modification, so is not shown in the guide.)

For most variations, the variation value is just multiplied by the result, affecting only the size. This is useful when combining variations on one transform to set the proportional amount of each. But some variations use the variation value as part of their logic. In this guide, a variation value of 1 is used unless noted otherwise.

Many variations have variables that can be set to adjust the effect. Not all variables are always described, but when they are, the names are shown in italics. The values of the variables used for the examples are listed for each variation, so it should be possible to reproduce any of them using the accompanying test flame file. All but blurs are made by adding a final transform to one of the test flames (adding linear as well when it is needed as discussed above). For 3D variations, the pitch of each example is listed.

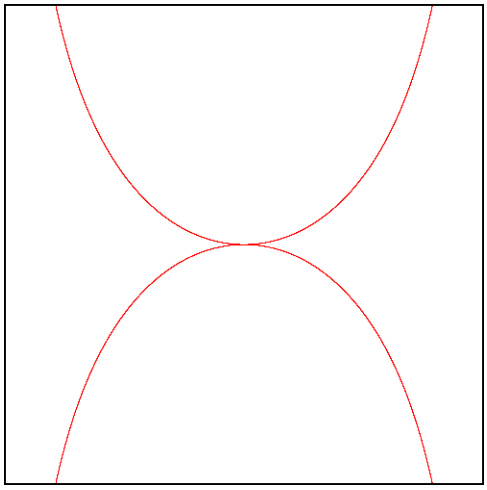
My primary purpose in assembling this guide was to help my own efforts in understanding the bewildering array of variations available for flame fractals. I can't promise I got everything correct, especially for the variations where no source code was available. I've learned a lot putting this together. I hope others will find it useful as well.

Rick Sidwell, November 2014

arch (2D blur)

2.09	7X15B	7X16	jwf	ch
no	no	no	yes	yes

Variation values less than 2 give a partial curve; 1 gives right half; value of 2 shown.



Z_arch (2D blur)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	no	dll

ZArch.dll
Uses variable instead of variation value

auger (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	yes	yes	yes

auger.dll

Creates a wave effect that gets stronger further from the origin. Variable *sym* controls how much x is affected; see the two right examples which are the same except for *sym*.

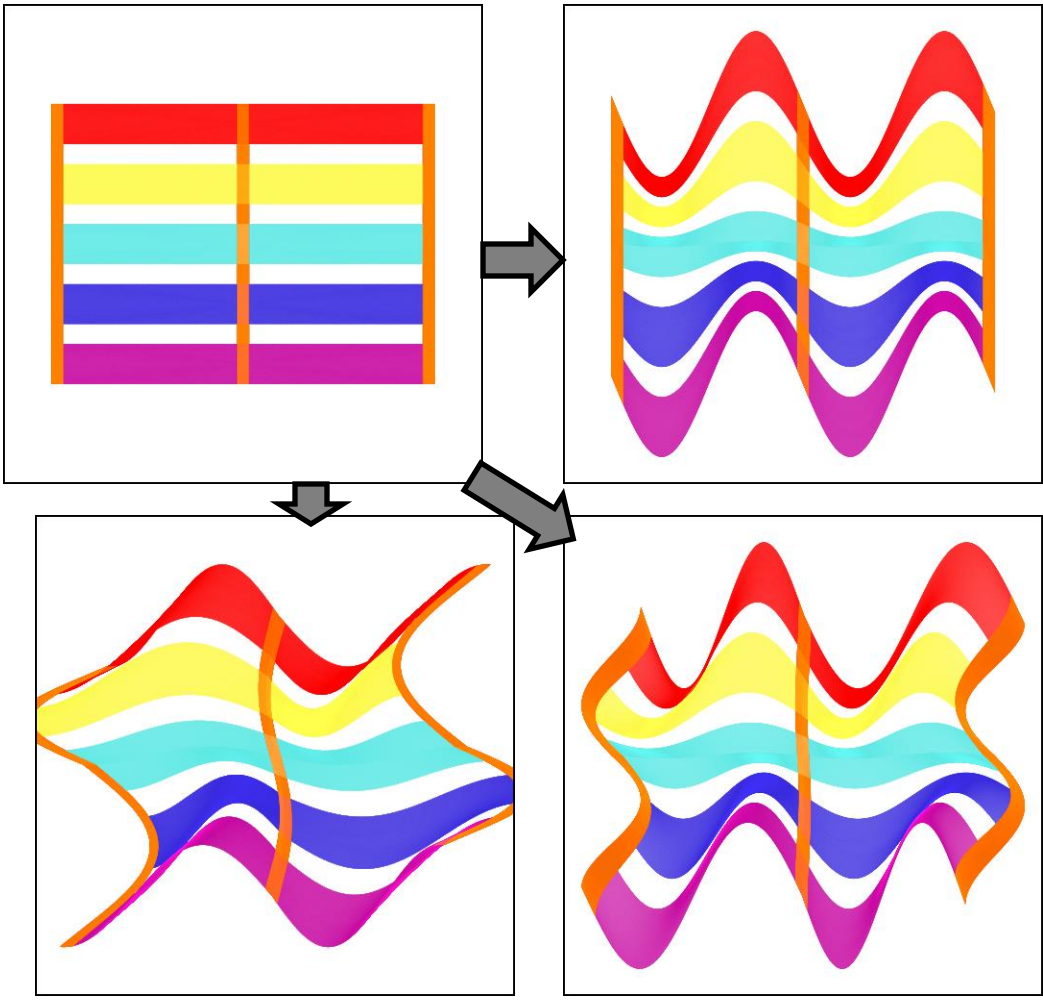
Top right: *sym* = 0 (so x is not affected), *weight* = 0.5, *freq* = 4, *scale* = 0.1

Bottom right: *sym* = 0.3 (mild x effect), *weight* = 0.5, *freq* = 4, *scale* = 0.1

Bottom left: *sym* = 1, *weight* = 0.3, *freq* = 3, *scale* = 0.5

The stripes have the appearance of rippled ribbons, but that is just an illusion; auger is 2D only.

Compare waves2, waves2b, wavesn.

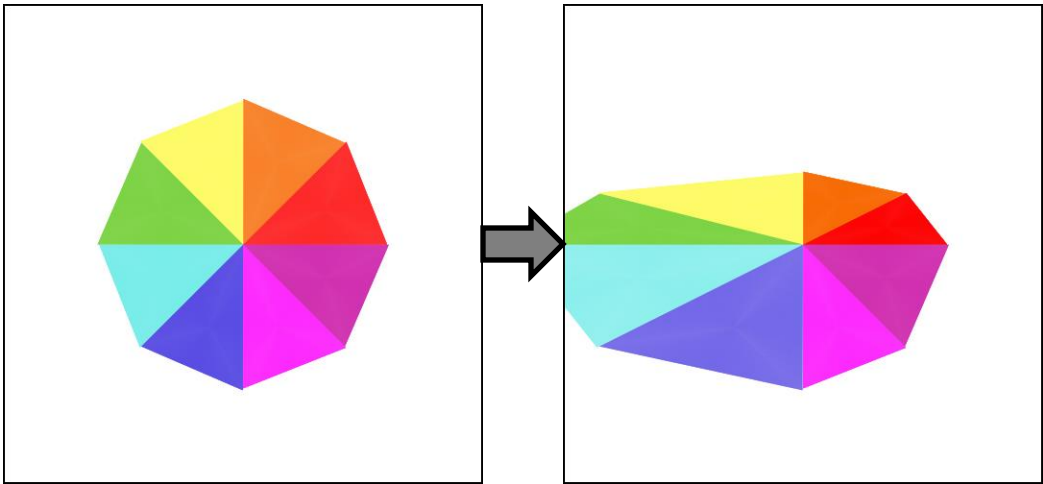


bent (2D)

2.09	7X15B	7X16	jwf	ch
yes	dll	dll	yes	yes

bent.dll
Doubles negative x (towards the left).
Halves negative y (towards the top).

Same as bent2 with *x* = 2 and *y* = 0.5.

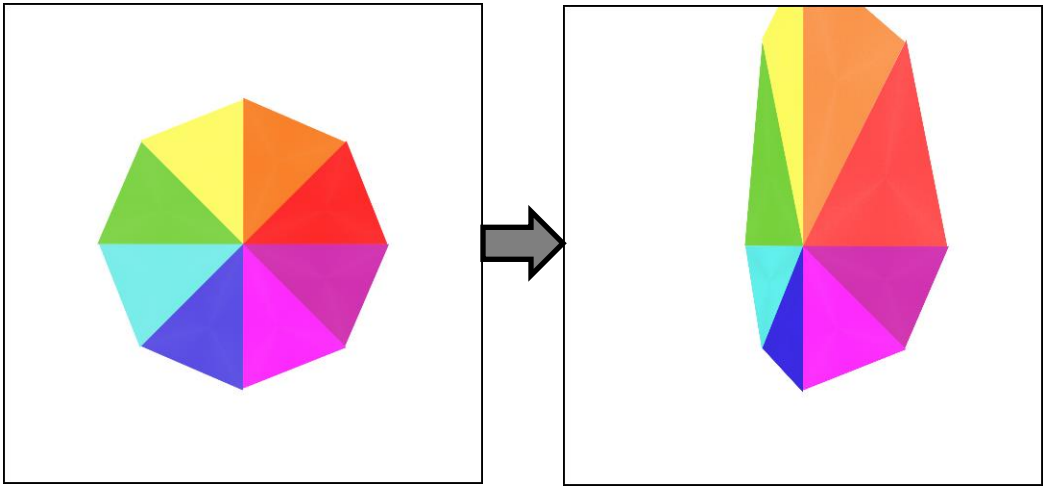


bent2 (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	dll

bent2.dll
Scales negative x (left) and y (up) as defined by the variables. Negative values are allowed.

Example uses *x* = 0.4 and *y* = 2.

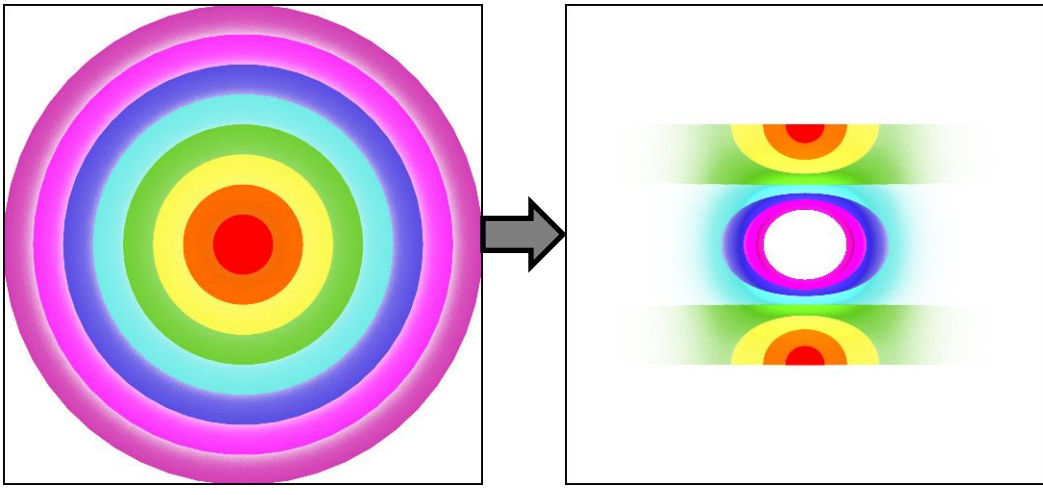


bipolar (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	yes	yes	yes

bipolar.dll
The interior of the unit circle is stretched horizontally, split, and put at the top and bottom. The rest of the plane is turned inside-out, stretched, and put in the middle.

The output wraps at the top and bottom, making this variation useful for vertical tiling.



blob (2D)

2.09	7X15B	7X16	jwf	ch
yes	no	no	yes	yes

blob_fl (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	no	no

blob_fl.dll

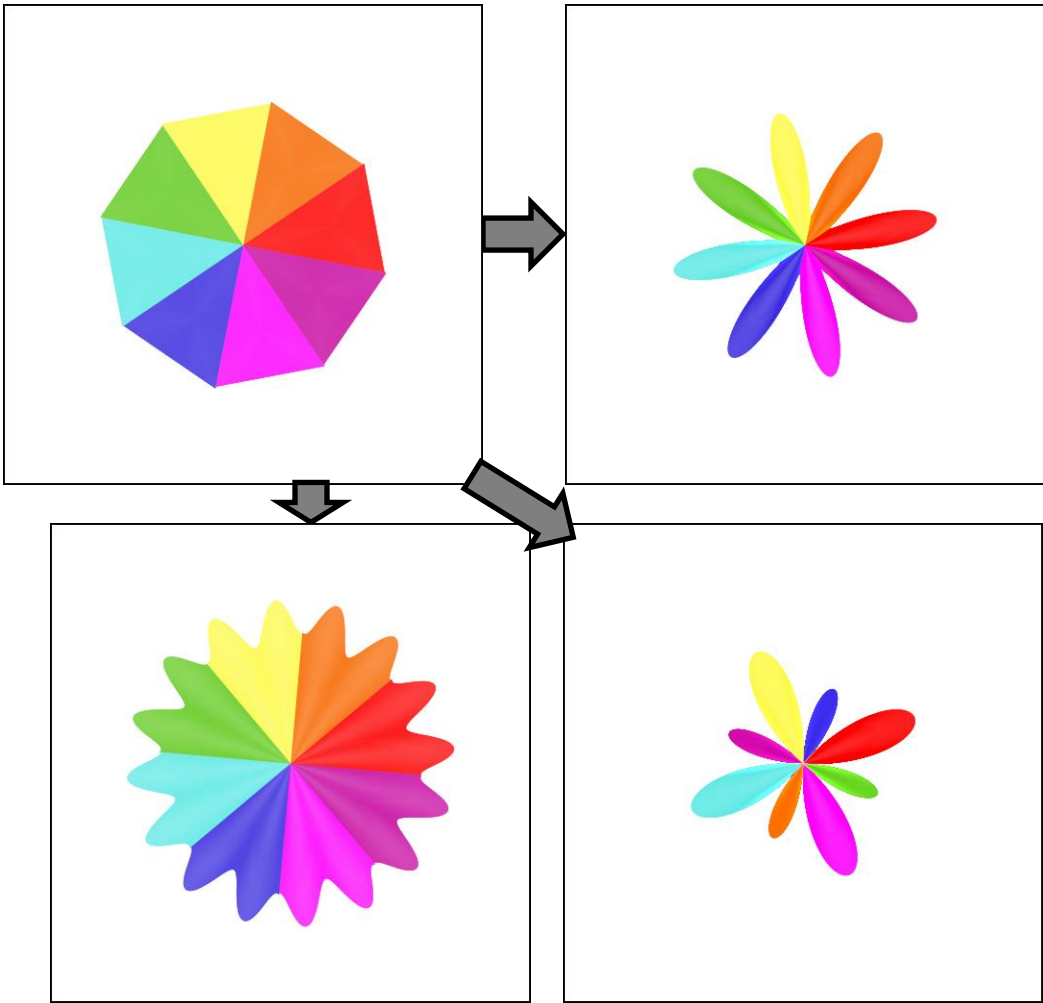
Pinches the plane according to the variables: *waves* is the number of pinches, and the height varies between *low* and *high*. The built-in version in Apo 2.09 requires an integer value for *waves*; the others do not.

Top right: *low* = 0.2, *high* = 1, *waves* = 8

Bottom left: *low* = 0.9, *high* = 1.2, *waves* = 16

Bottom right: *low* = -0.6, *high* = 0.9, *waves* = 4

Compare with cardiod.

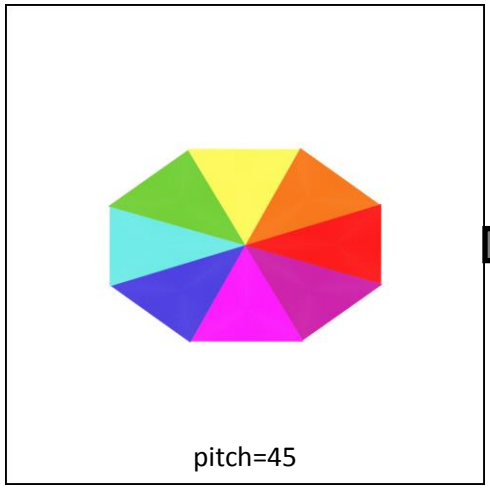


blob3D (3D, sets z)

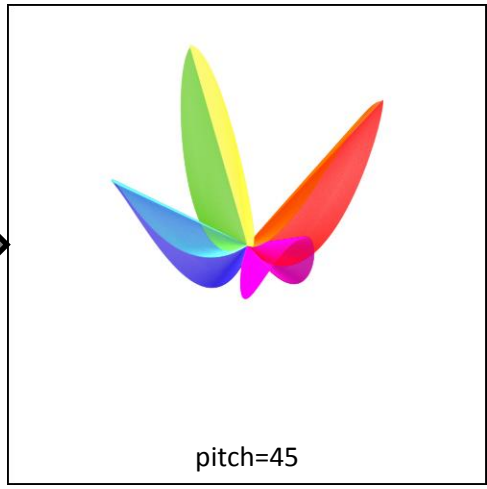
2.09	7X15B	7X16	jwf	ch
no	no	no	yes	no

A 3D version of blob. Looks the same as blob if pitch is 0.

Example uses *low* = 0.1, *high* = 1, and *waves* = 4 with a pitch of 45.



pitch=45



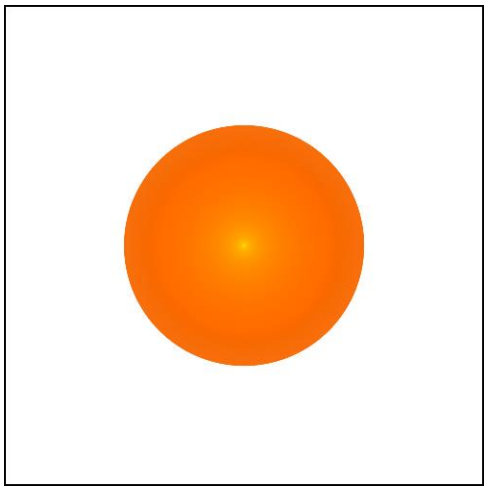
pitch=45

blur (2D blur)

2.09	7X15B	7X16	jwf	ch
yes	yes	yes	yes	yes

A circle with a bright center. (The spot doesn't show with all colors.) For a circle without the bright center, use circleblur. See also sineblur.

pre_blur is a pre_ version of gaussian_blur, not blur.
blur_circle is a circle without a bright center, but different versions are inconsistent.



blur_heart (2D blur)

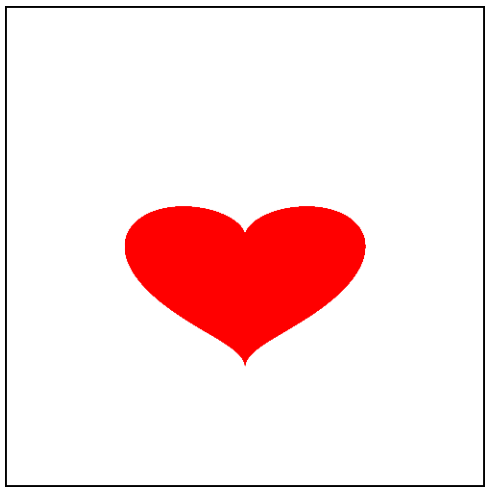
2.09	7X15B	7X16	jwf	ch
dll	dll	dll	no	dll

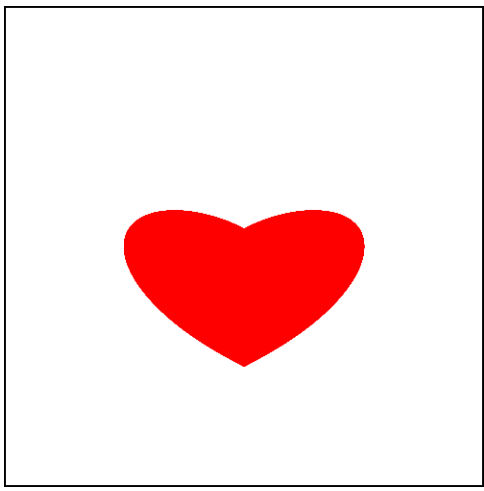
blur_heart.dll

A heart-shaped blur. Three parameters to vary the shape.

Left (default): *p* = 0.5, *a* = -0.6, *b* = 0.7

Right: *p* = 0.9, *a* = -0.3, *b* = 0.4





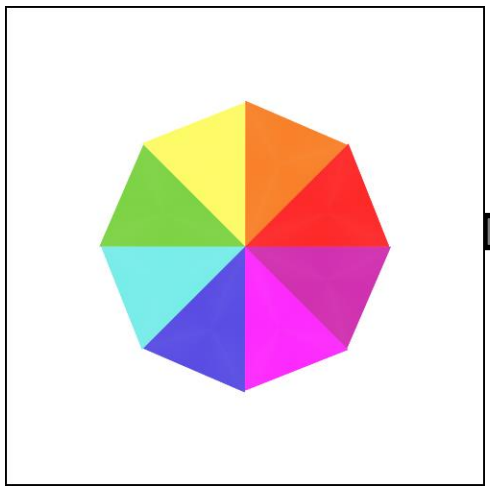
blur_linear (2D)

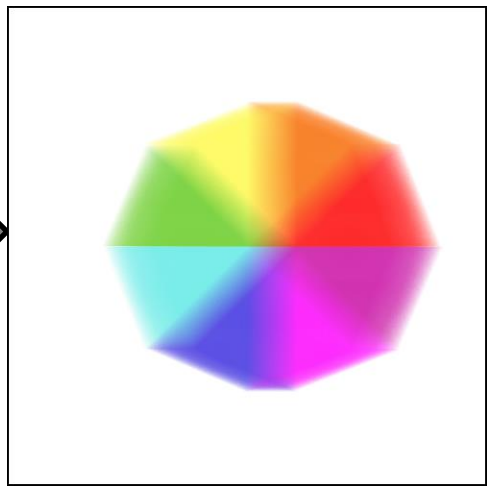
2.09	7X15B	7X16	jwf	ch
dll	dll	dll	no	yes

blur_linear_jf.dll

Creates a motion blur effect.

Two variables:
length – size of the blur effect
angle – angle of the blur (in radians)





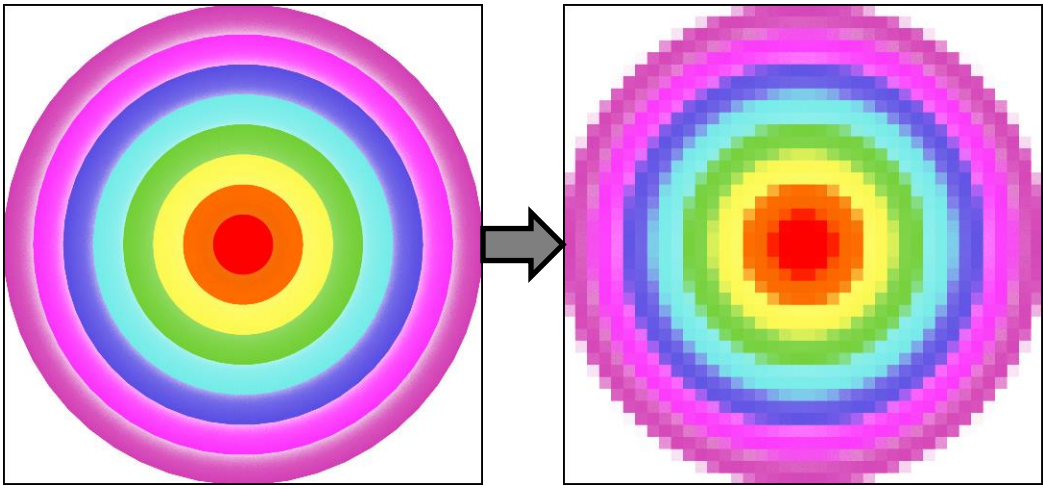
blur_pixelize (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	yes	no	yes

blur_pixelize.dll

averages colors in an area to make large square pixels.

Two parameters:
size – specifies the size of each pixel
scale – allows resizing the pixels



blur_zoom (2D)

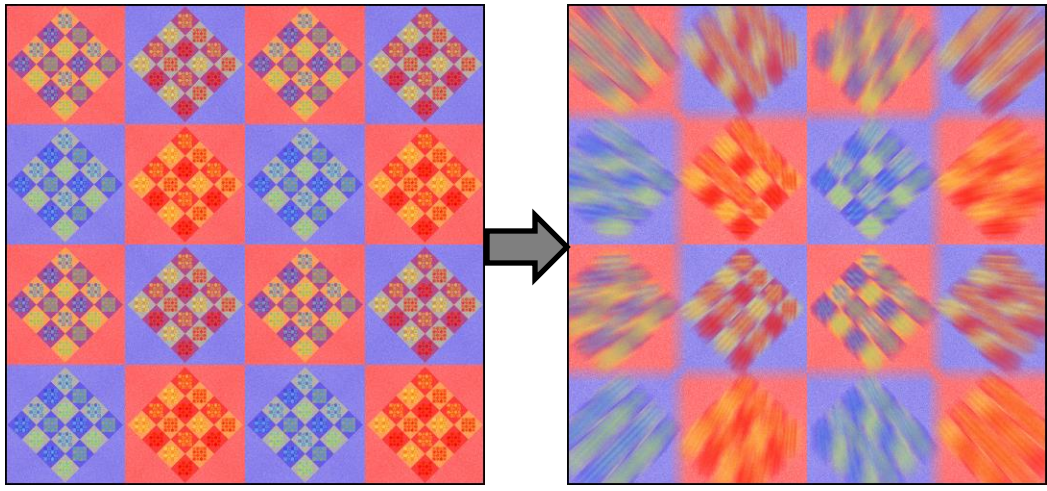
2.09	7X15B	7X16	jwf	ch
dll	dll	yes	yes	yes

blur_zoom_jf.dll

zooms from a center point outward

Three parameters:
zoom_length – length of the zoom (0.15 for this example)
x and *y* – center point of the zoom

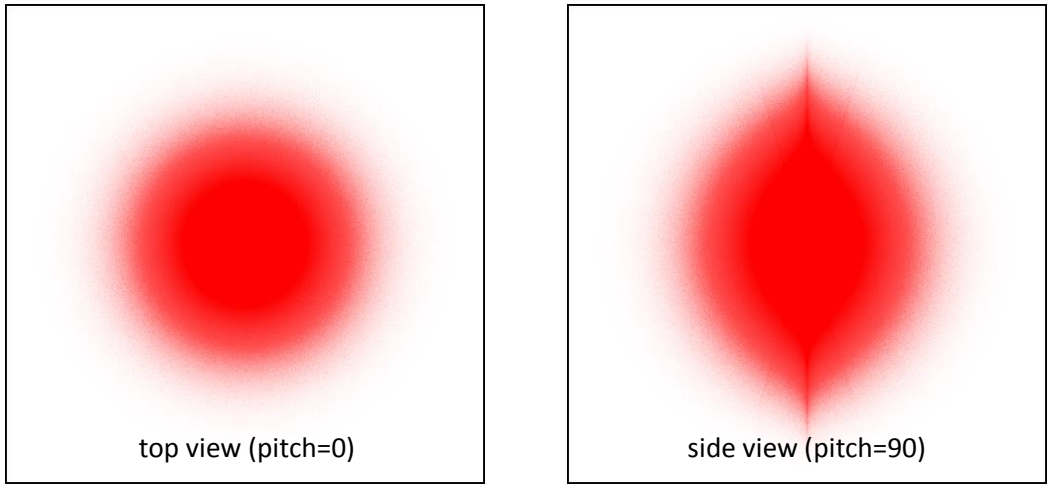
See falloff2 with *type* = 1.



blur3D (3D blur)

2.09	7X15B	7X16	jwf	ch
no	yes	yes	yes	no

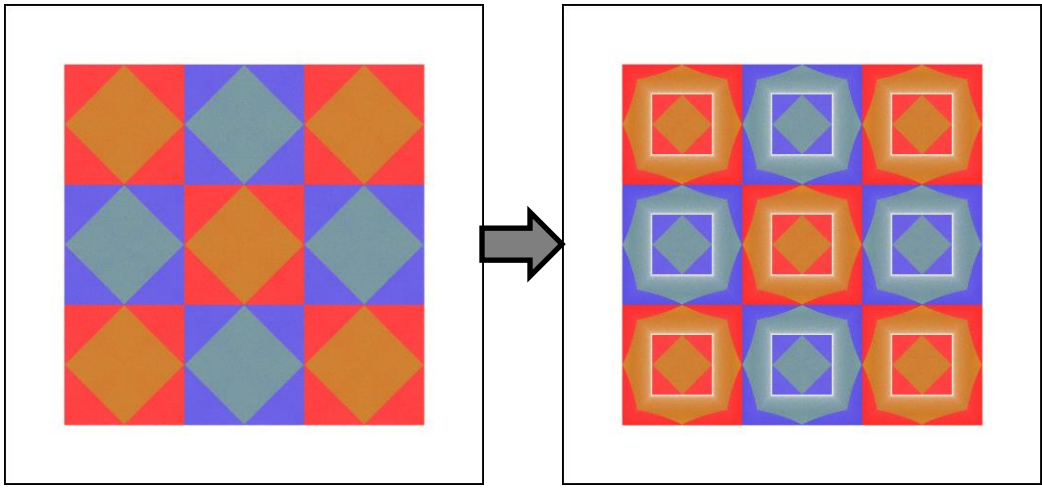
Three dimensional Gaussian blur



boarders2, pre_boarders2 (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	dll

boarders2.dll, pre_boarders2.dll
Divide the plane into squares of size 1, and make a copy of each. Shrink one copy and keep in the middle. Poke a square hole in the other and expand it to make a frame around the first.
Set all three variables to 0.5 (shown here) to make it work like boarders (its predecessor with no variables).
See tri_boarders2 and xtrb.



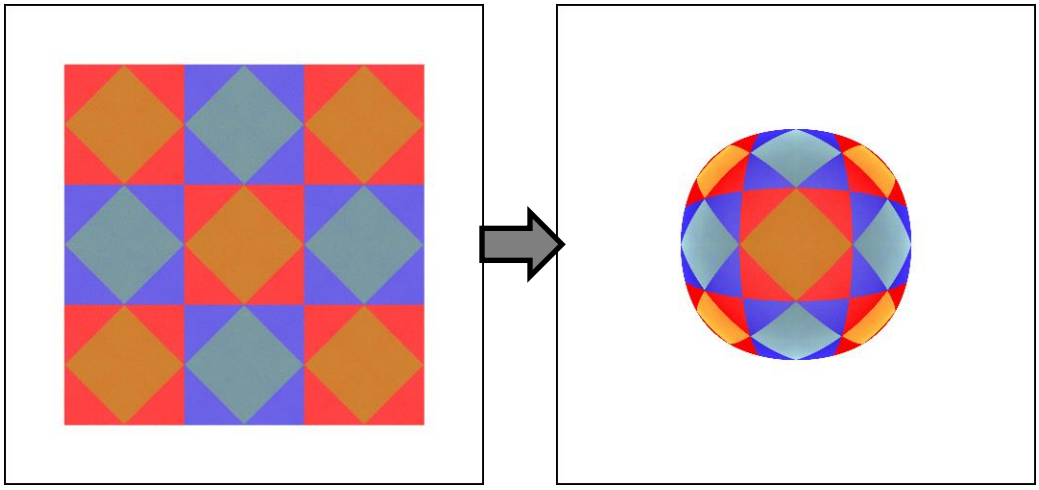
bubble (2D/3D, sets z)

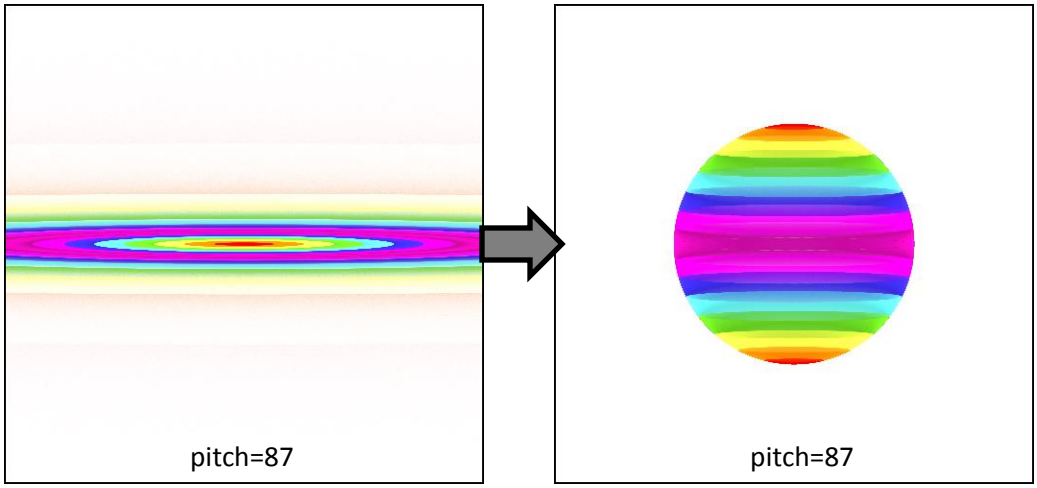
2.09	7X15B	7X16	jwf	ch
yes	yes	yes	yes	yes

Projects the flame onto a sphere. Bubble was a staple long before the first 3D version of Apophysis was developed, and has many uses even in 2D. The top example demonstrates how it works.

But in programs that support 3D, bubble sets z to make a true sphere (any previous value of z is ignored). The bottom example shows its effect on coincentric rings, with Pitch set to 87° (nearly edge on). Eight equal sized rings in the middle (the purple one goes from1.75 to 2) transform to the top half. These are then repeated in reverse with proportional sizes (the red one goes from 8 to infinity, and can't really be seen in the original) transform to the bottom half.

See hemisphere.





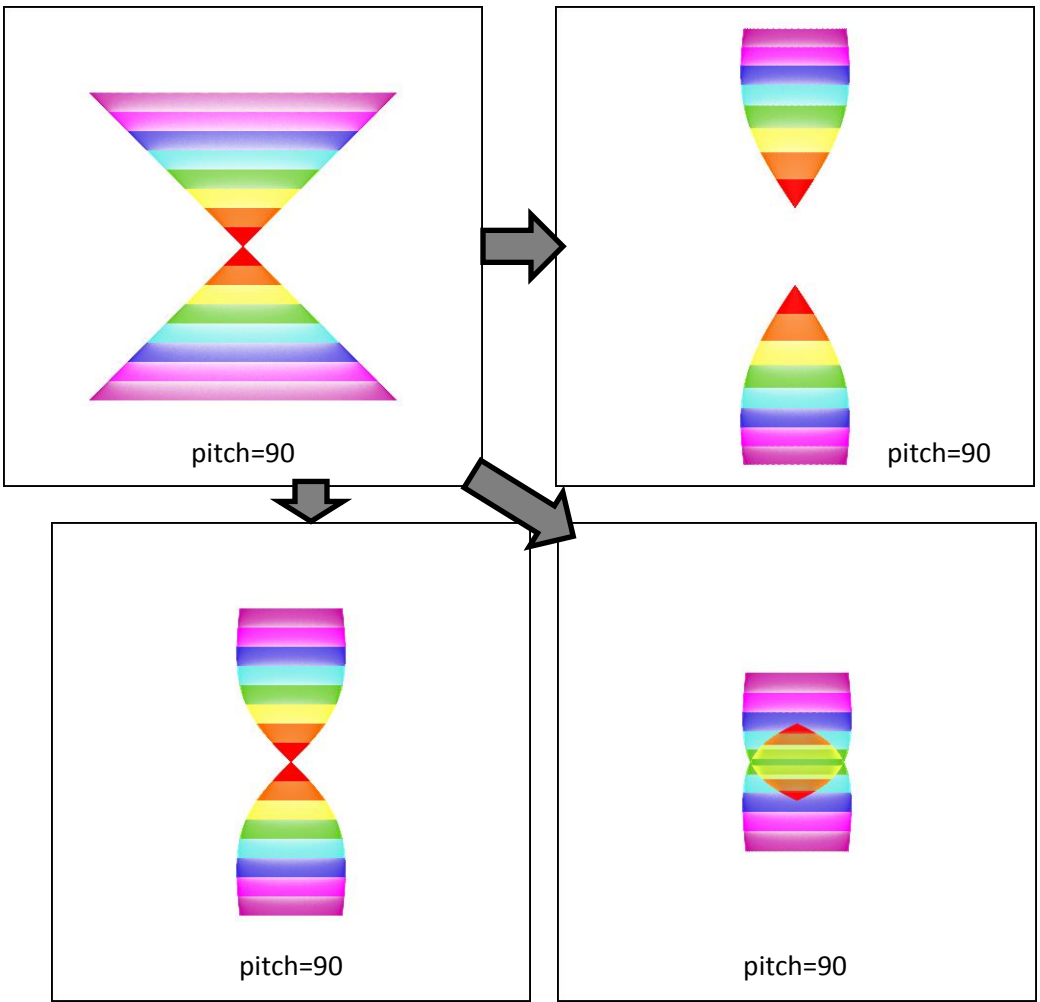
bubble2 (3D, transforms z)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	dll

For the x-y plane, bubble2 works just like bubble except that there are two variables, *x* and *y*, that scale the result.

But bubble2 transforms z based on the variable *z*, unlike bubble which sets it to form a sphere. This means other transforms must have already set a *z* value for it to transform.

The examples shown here are all at pitch 90, so the vertical axis is z instead of y. With variable *z* = 0, the z value is just passed on (bottom left; x changes but z does not). When *z* is positive (top right, *z* = 0.5), the top and bottom halves of the flame are separated by a gap. When *z* is negative (bottom right, *z* = -0.5), the top and bottom halves overlap.

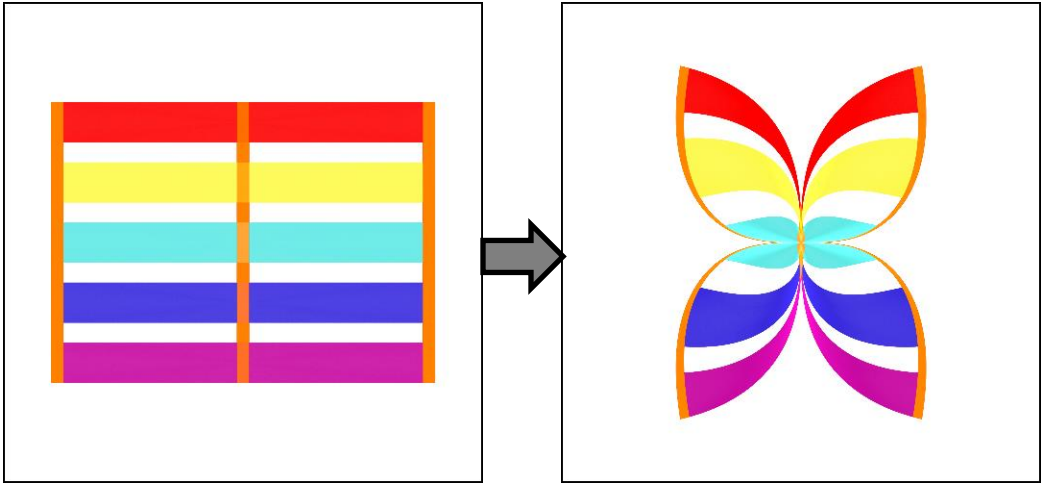


butterfly (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	yes

butterfly.dll

A four-way pinch effect that resembles a butterfly.



bwraps7 (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	dll

bubble_wrap_S7.dll

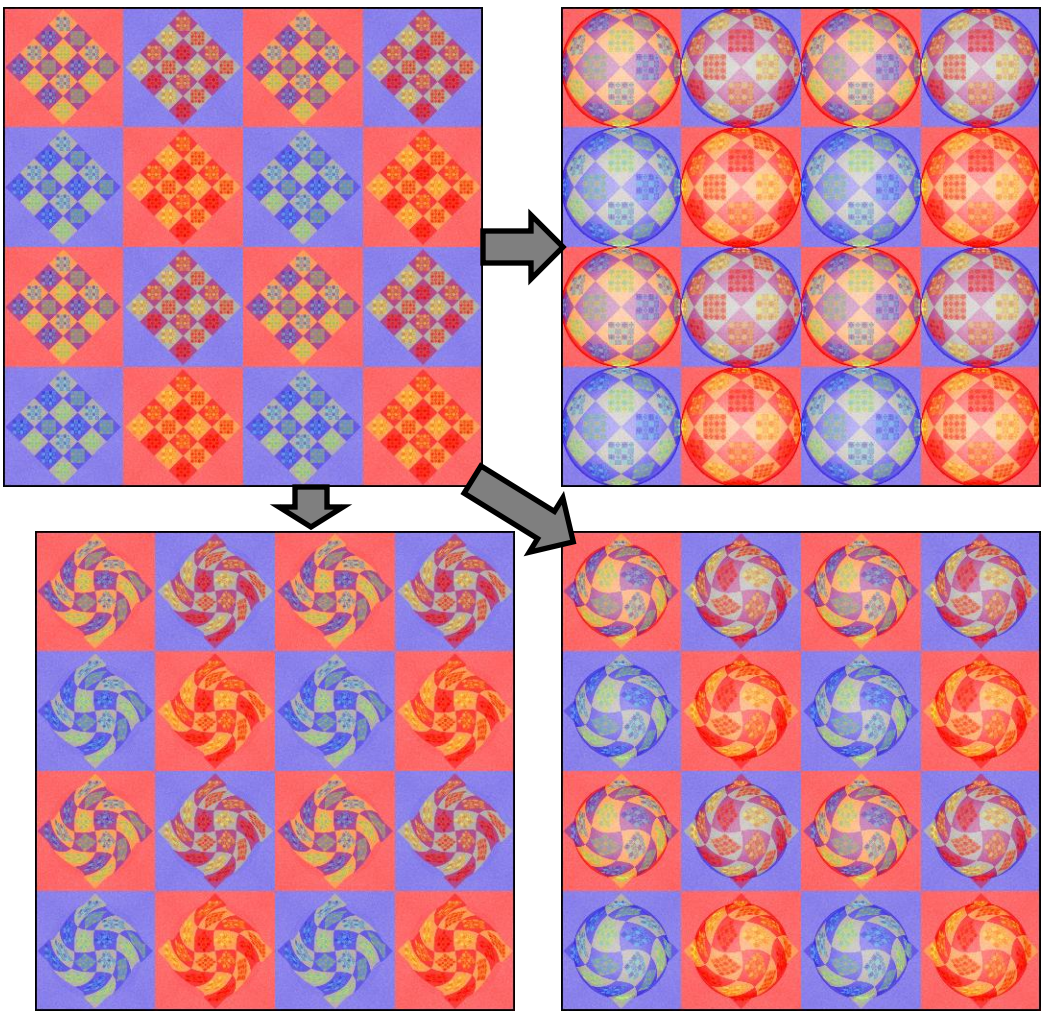
Overlays the flame on a grid of bubbles, like bubble wrap. Note this is 2D only; z is ignored.

Top right uses the default parameters:
cellsize = 1 matches the sample's size
space = 0 makes the bubbles touch
gain = 2 for normal bubbles
inner_twist = 0 so no inner twist
outer_twist = 0 so no outer twist

Bottom right shows space and twist:
space = 0.5 puts space between bubbles
inner_twist = 1 to twist the insides

Bottom left reduces the gain:
space = 0.5 (same as bottom right)
gain = 1 flatten bubbles a bit
inner_twist = 1 (same as bottom right)

Other versions: **bwraps** and **bwraps2** are nearly the same as bwraps7, but the *gain* variable works differently. These versions also have pre_ and post_ variations.



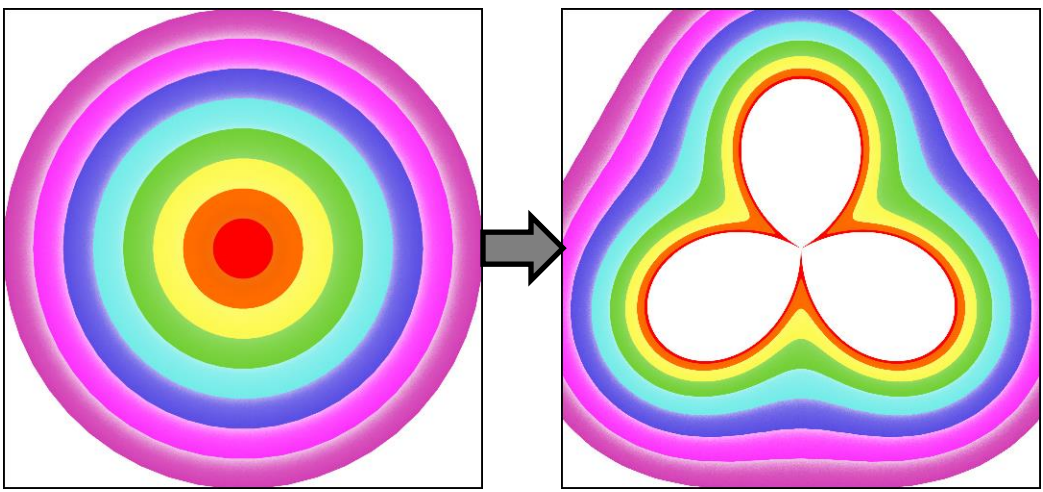
cardiod (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	no	dll

cardiod_mf.dll

Applies a cardiod curve, pushing the flame outward from the center. The single variable, *a*, determines the number of petals. The default, 1, gives a standard cardiod. Shown is *a* = 3.

Compare with blob, which pushes from the outside in.



checks (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	dll

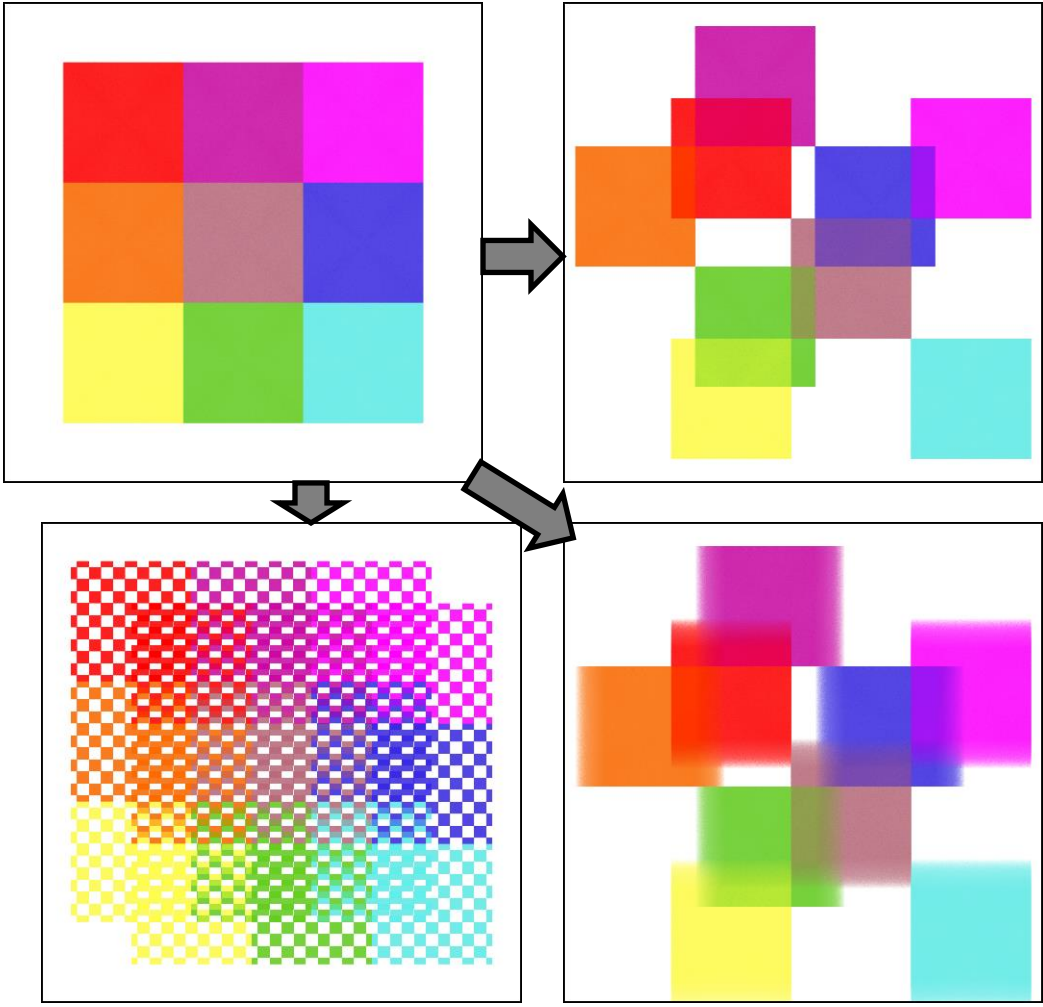
checks.dll

Divides the flame into a checkerboard (square size is determined by the *size* variable). Shifts half of the squares up and left, and the other half down and right according to the *x* and *y* variables (negative values are allowed and shift the other direction).

Top right shows basic operation with *size*=1 (the size of the squares in the original), *x* = 0.4, *y* = 0.3, and *rnd* = 0. The outside middle squares moved up and left; the corners and center moved down and right.

Bottom right shows the effect of the *rnd* variable, set to 0.25 here.

Bottom left has a much smaller square size: *size* = 0.1, *x* = 0.252, *y* = 0.176, and *rnd* = 0.



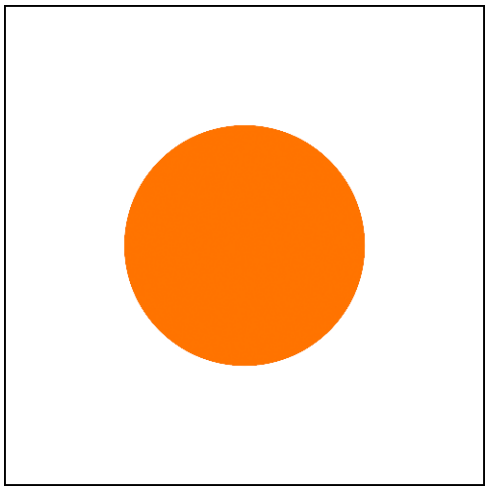
circleblur (2D blur)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	dll

circleblur.dll

A flat circle (no bright center like blur can have).

A similar variation, **blur_circle**, does the same thing but different versions are not consistent.

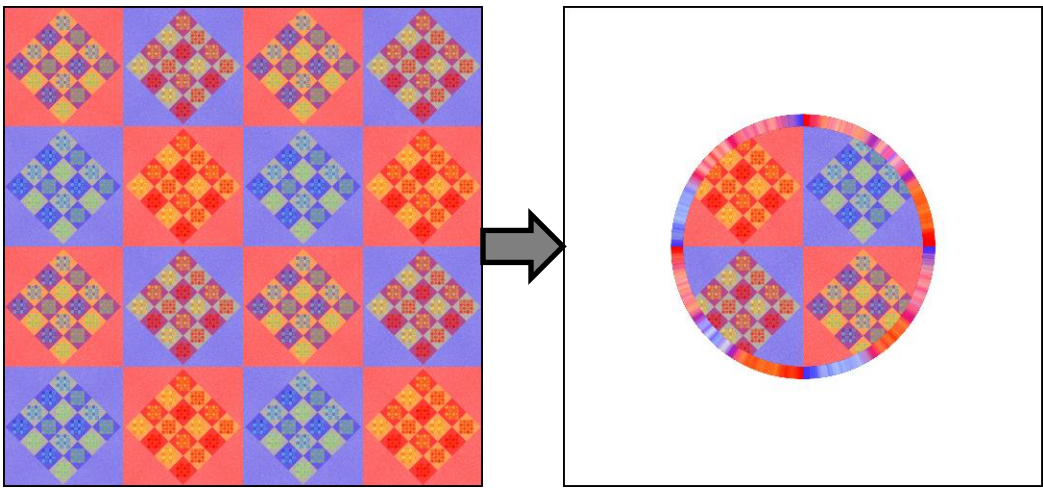


circlecrop, pre_circlecrop, post_circlecrop (2D, passes z)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	dll

circlecrop.dll
Crops a flame to a circle. Variables allow adjusting the circle size and position. Variable *scatter_area* specifies the size of the border. Set variable *zero* to 1 for no border at all. Example uses *radius* = 1, *x* = 0, *y* = 0, *scatter_area* = 0.2, *zero* = 0.

See crop, cropn.



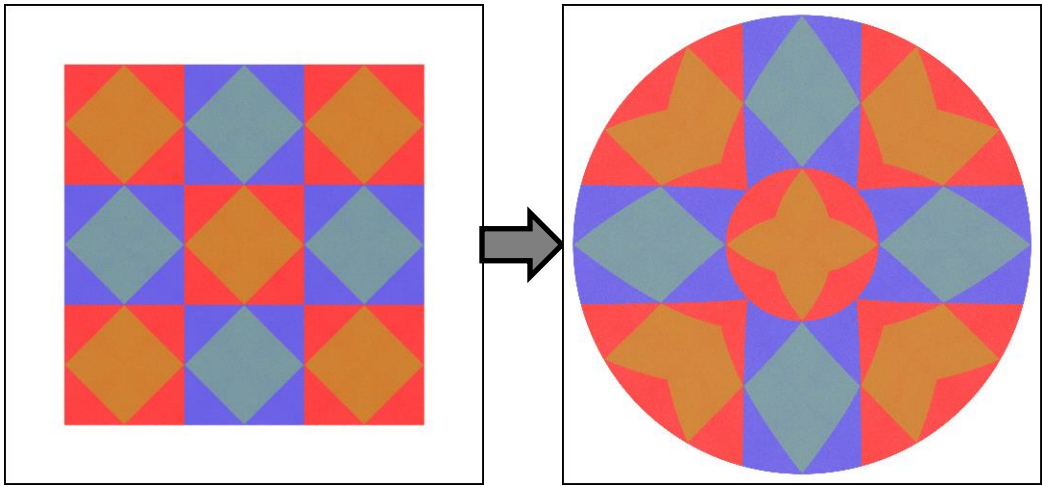
circlize (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	dll

circlize.dll

Maps squares that are centered at the origin with sides parallel to the x and y axes to circles.

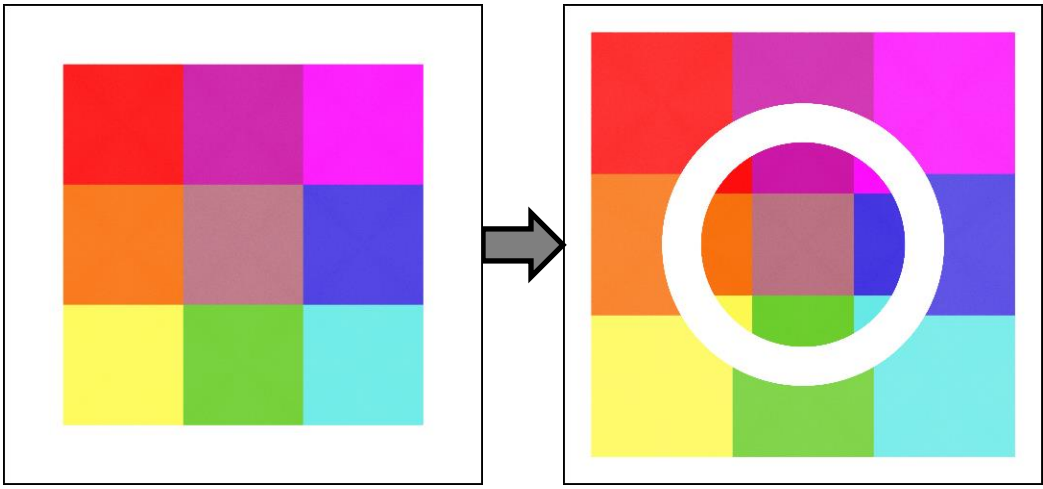
See squarize, which does the opposite.



circus (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	dll

circus.dll
Scales the unit circle and the rest of the flame separately according to the variable *scale*. When less than 1, the circle is shrunk and the rest is expanded, leaving a ring in the middle. When more than 1, the opposite occurs and the two overlap. The example uses the value *scale* = 0.85.



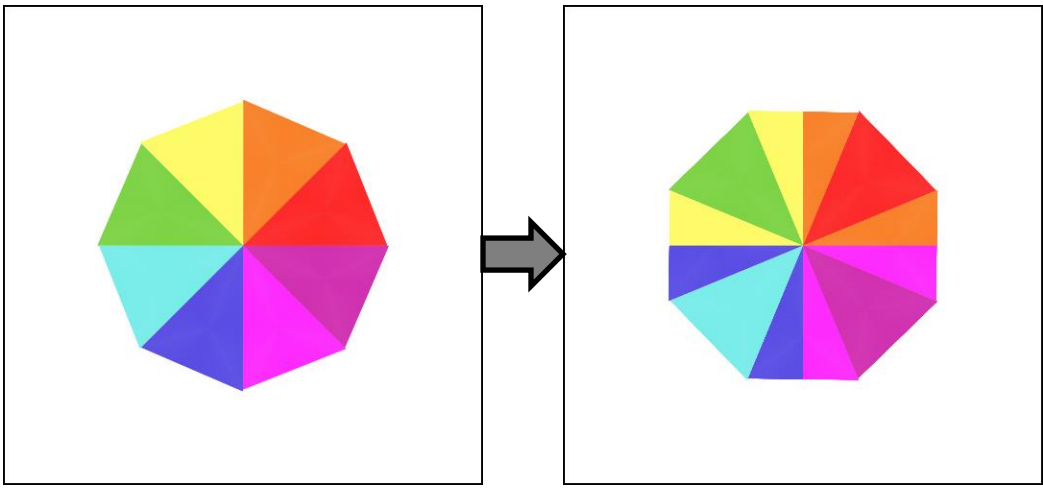
collideoscope (2D)

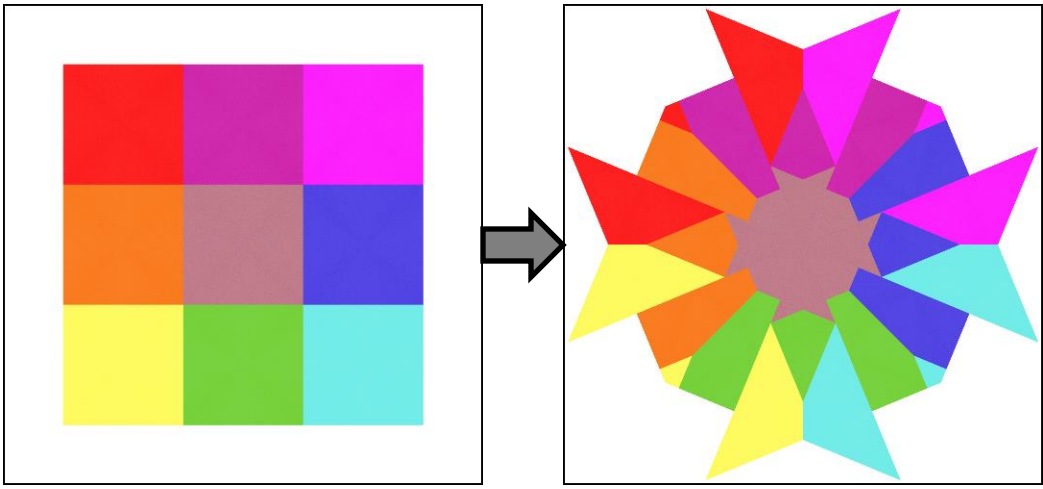
2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	yes

collideoscope.dll
Divides the top and bottom halves of the flame into *num* wedges each, then rotates each wedge according to *a*, the fraction of a complete rotation. The part rotated off the edge is cycled back to the other side. Adjacent wedges rotate opposite directions.

The top example uses *num* = 2, so the original is divided into four wedges, and *a* = 0.25 so each is rotated a quarter turn. So the green and yellow sections form one wedge that is rotated clockwise.

The bottom example uses *num* = 4 and *a* = 0.5, so divides the original into 8 wedges, each rotated halfway. For example, one wedge is made of an orange rectangle and a red triangle (half of the respective squares). The rotation makes the bottom of the rectangle and the long side of the triangle adjacent and rotated halfway through the wedge.





conic (2D half-blur)

2.09	7X15B	7X16	jwf	ch
no	no	no	yes	yes

conic2 (2D half-blur)

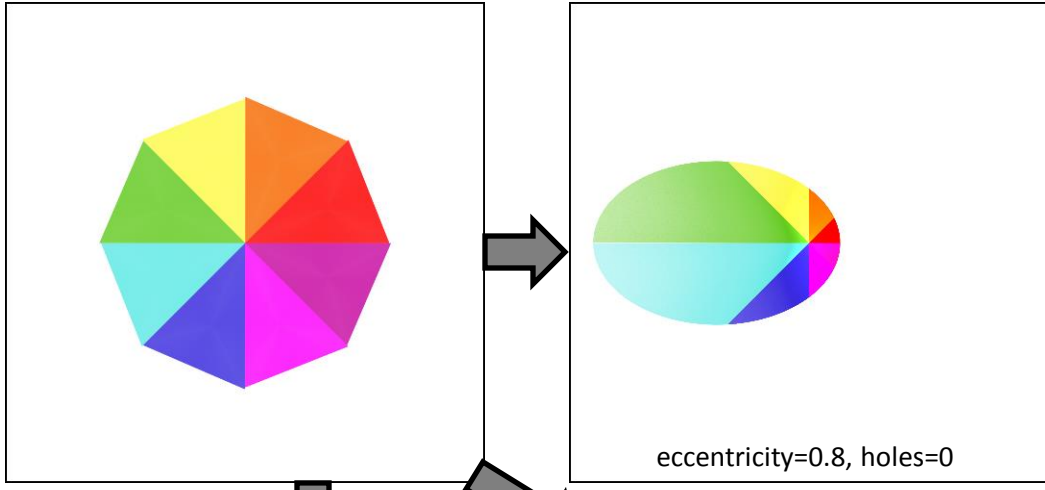
2.09	7X15B	7X16	jwf	ch
dll	dll	dll	no	dll

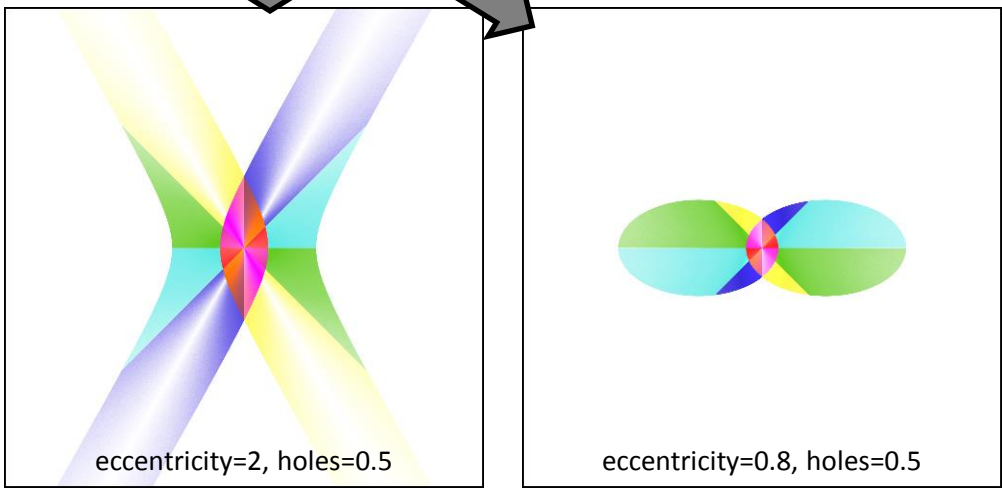
conic2.dll
Makes a conic section shape (ellipse, parabola, or hyperbola), with the focal point at the origin.

The *eccentricity* variable determines the type (1 for parabola, less for ellipse, more for hyperbola).

Setting the *holes* variable to 0.5 results in two shapes, the main one on the left and one turned 180° on the right, as in the bottom two examples. Setting to 0 or 1 shows only the left or right shapes, and going past that creates a hole at the focal point (a hyperbola includes both shapes, and already has a hole at the focal point).

(The plugin Conic.dll has the variation name conic, but is different from these.)





cpow (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	dll

cpow.dll
The name means Complex Power; it treats each point of the flame as a complex number and rasies it to the complex power specified by the variables *r* and *i*.

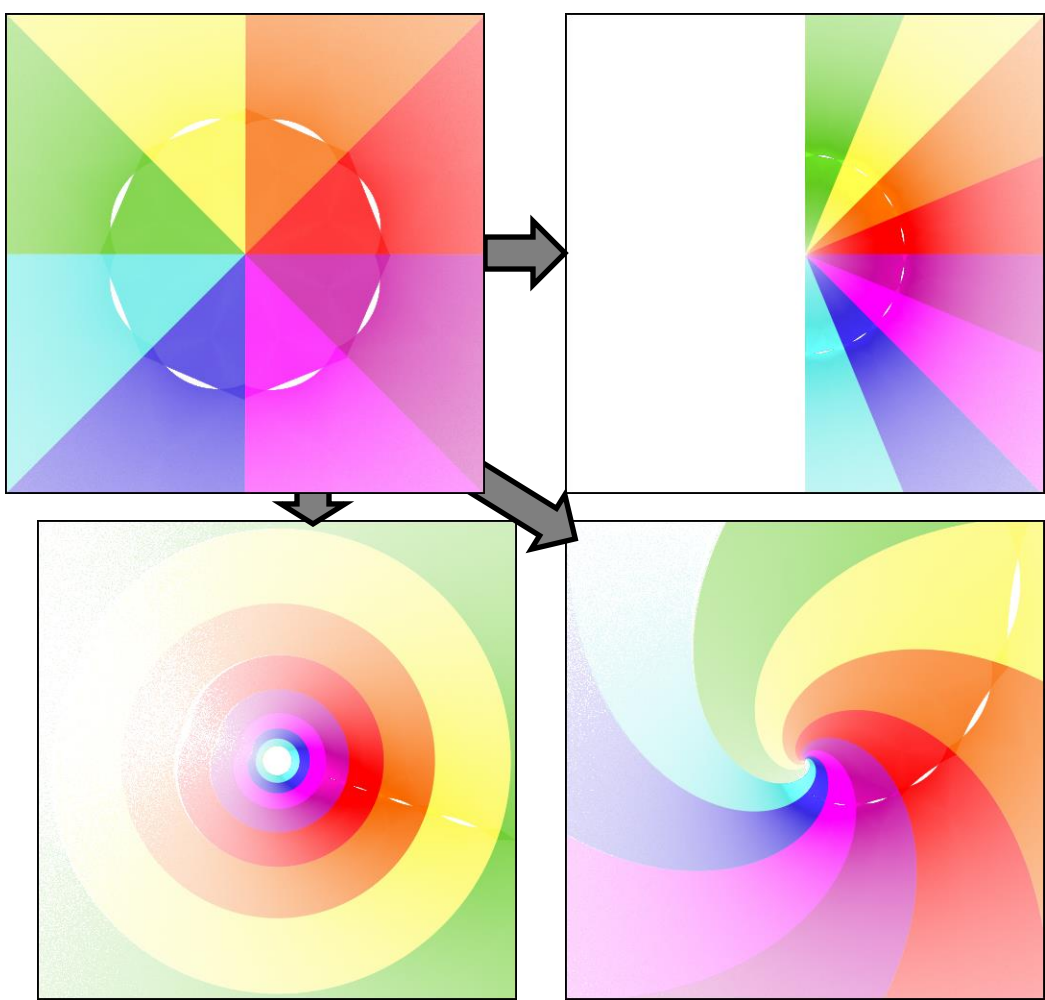
With *i* = 0, decreasing *r* from its default of 1 splits the plane along the negative x axis and rotates the top and bottom away from each other, much like folding a Japanese fan (top right, where *r* = 0.5 and *i* = 0). Increasing *r* does the opposite, causing an overlap.

With *r* = 0, setting *i* will convert wedges to rings (bottom left, where *r* = 0 and *i* = 0.5).

Setting both *r* and *i* converts wedges to spirals (bottom right, where *r* = 0.5 and *i* = 0.5).

A julian effect (see julian) is also available with the variable *power* (not shown).

Compare Juliac.



crackle (2D blur)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	yes

avCrackle.dll

A flexible (but slow) blur that can generate a variety of interesting textures.

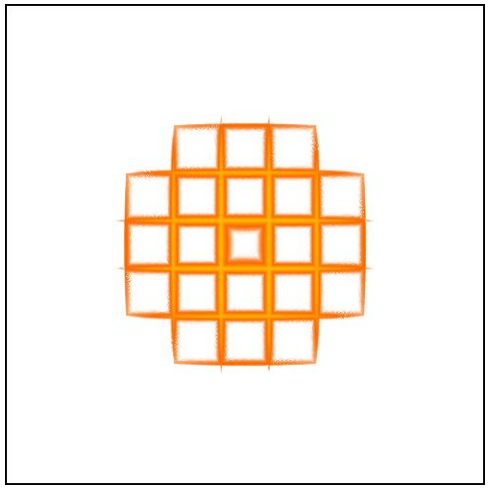
Top left: *cellsize* = 0.8, *power* = 0.1, *distort* = 0, *scale* = 1, *z* = 0

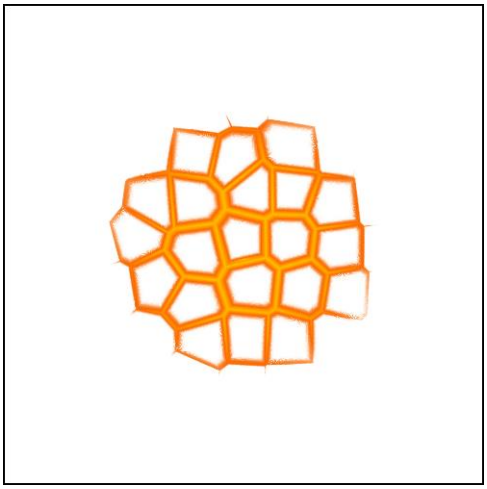
Top right: *cellsize* = 0.8, *power* = 0.1, *distort*= 0.4, *scale* = 1, *z* = 0

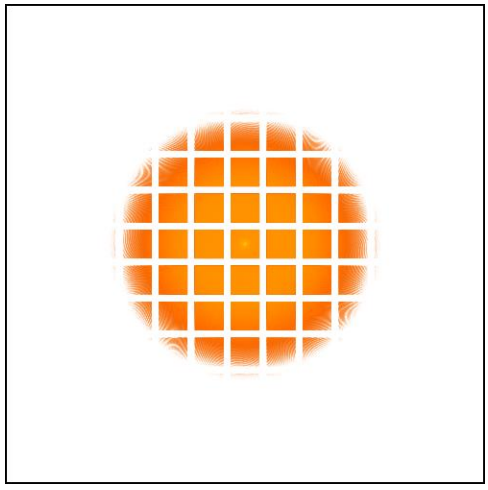
Bottom left: *cellsize* = 0.6, *power* = 1, *distort* = 0, *scale* = 0.8, *z* = 0

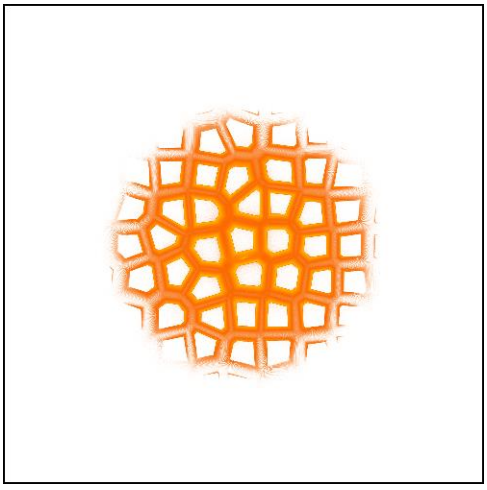
Bottom right: *cellsize* = 0.6, *power* = -0.3, *distort* = 0.4, *scale* = 0.6, *z* = 0

The *z* variable changes the distortion, and has nothing to do with 3D or the z axis.







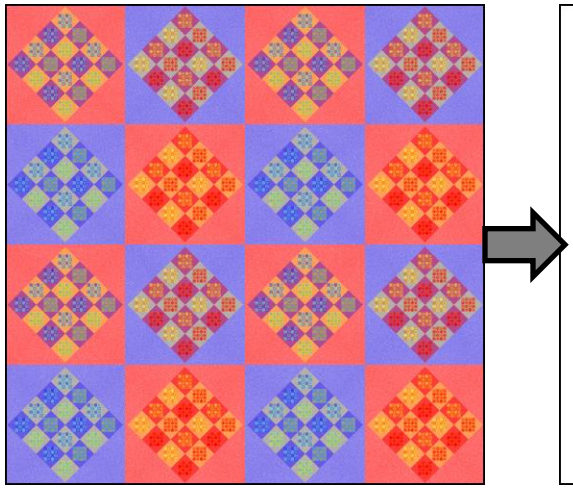


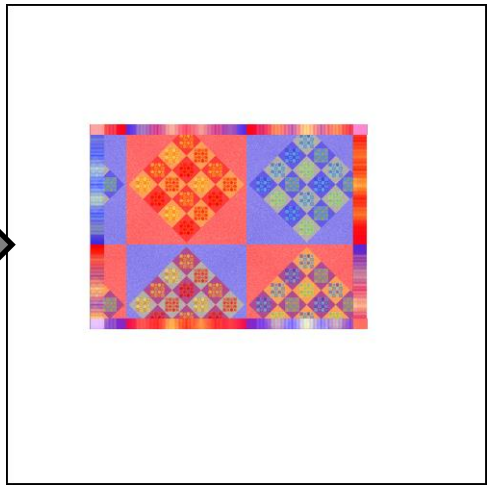
crop, pre_crop, post_crop (2D, passes z)

2.09	7X15B	7X16	jwf	ch
dll	dll	yes	yes	yes

crop.dll
Crops a flame to a rectangle. Variables allow adjusting the rectangle sides. Variable *scatter_area* specifies the size of the border. Example uses *left* = -1.3, *top* = -1, *right* = 1, *bottom* = 0.7, *scatter_area* = 0.1.

See circlecrop, cropn.





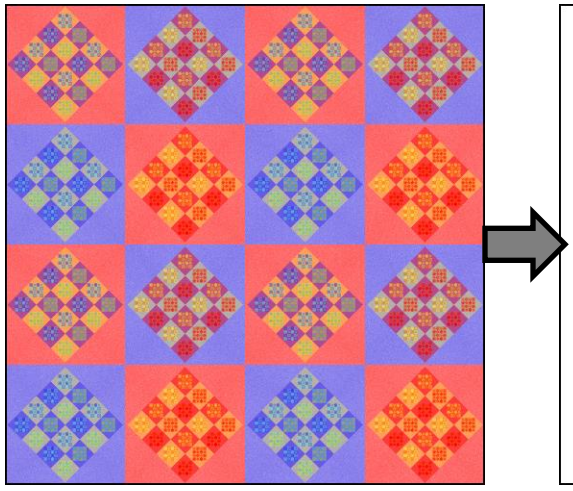
cropn (2D, passes z)

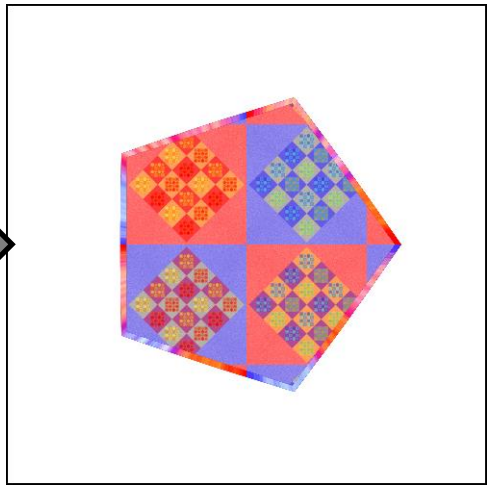
2.09	7X15B	7X16	jwf	ch
dll	dll	dll	no	yes

cropn.dll

Crops a flame to a polygon. Variables allow adjusting the *power* (number of sides, negative for a hollow crop) and *radius*. Variable *scatterdist* specifies the size of the border. Example uses *power* = 5, *radius* = 1, *scatterdist* = 0.1.

See circlecrop, crop.





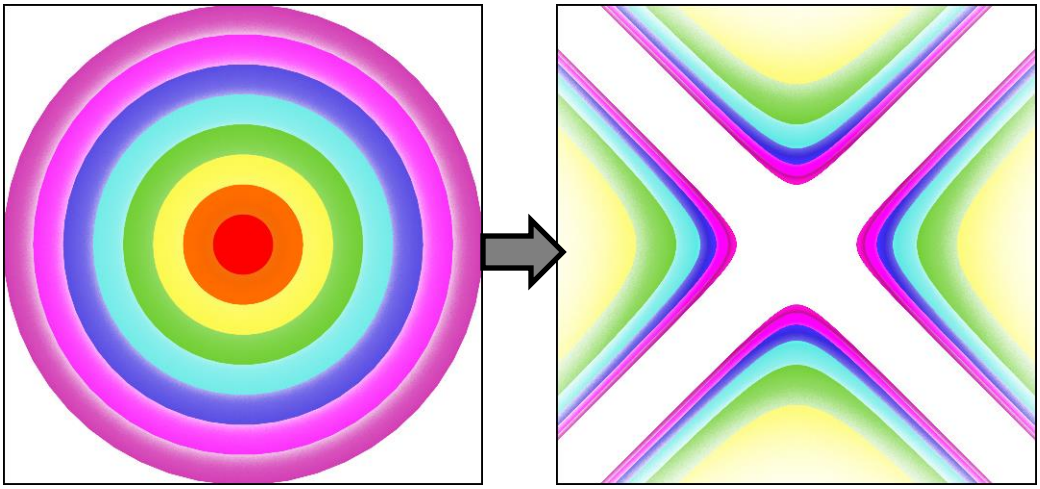
cross (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	yes	yes	yes

CrossVariationPlugin.dll

Divides the flame diagonally into four wedges, then turns each wedge inside-out.

The variations **cross** , **cross2**, and **Z_cross** all work exactly the same.



curl (2D)

2.09	7X15B	7X16	jwf	ch
yes	yes	yes	yes	yes

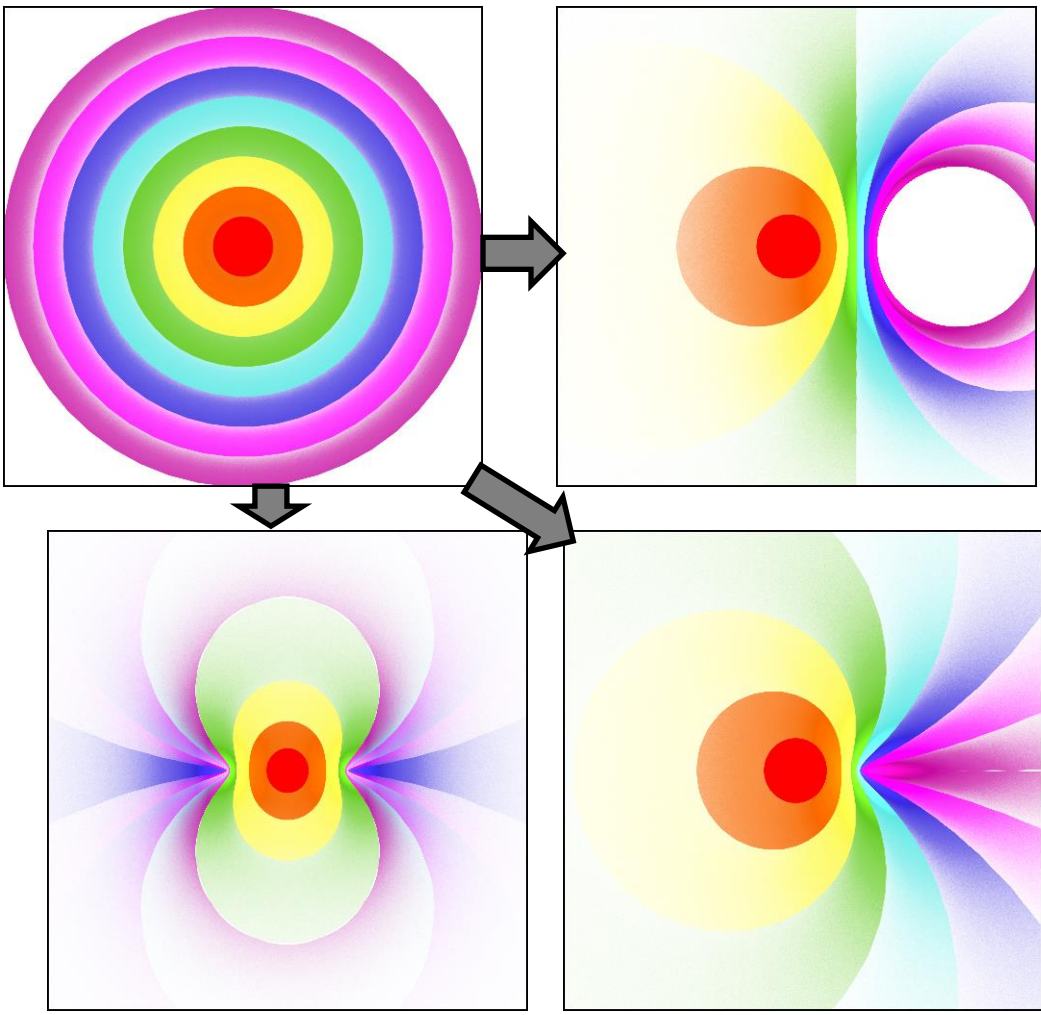
Think of the plane as a series of coincentric rings, as shown here but extending out to infinity.

As *c1* is increased, the rings shift left and uncurl, deforming first to a vertical line, then curling back into rings on the right, turning that side inside-out.
Top right: *c1* = 1, *c2* = 0, showing the state when the ring at distance 1 from the origin is a line. When *c1* is negative, the same thing happens but the opposite direction.

As *c2* is inceased, the rings stretch vertically then the middle starts to pinch in and the top and bottom spread out and rotate around until they cross then rejoin at the ends with top and bottom flipped.
Bottom left: *c1* = 0, *c2* = 0.5, showing the state when blue ring ends are crossing and the shrinking magenta ring touches the expanding green one. When *c2* is negative, the action happens side to side.

Bottom right: *c1* = 1, *c2* = 0.25

Pre_ and post_ versions also available.

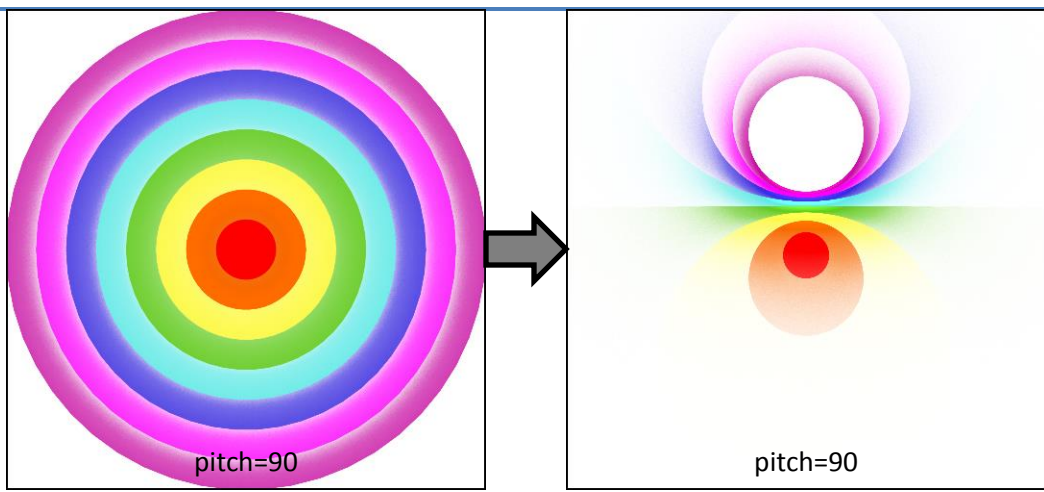


curl3D (3D, transforms z)

2.09	7X15B	7X16	jwf	ch
no	yes	yes	yes	no

Three variables, *cx*, *cy*, and *cz*, work similarly to the curl *c1* variable, but in the x, y, and z dimensions. There is no analog to *c2*. The example shows a side view of the coincentric rings from the curl example rotated upward to show the effect of *cz*. *cx* = 0, *cy* = 0, and *cz* = 1.

Post_ version also available

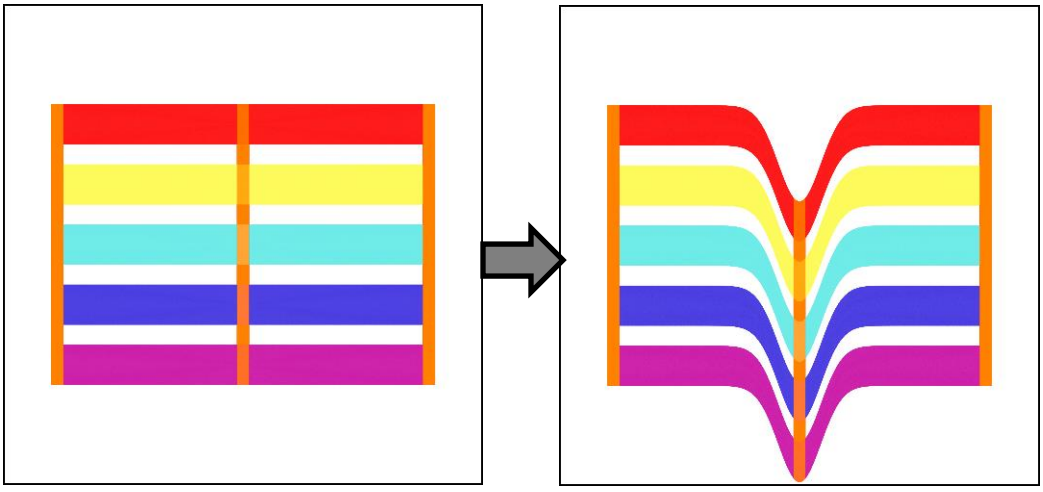


curve (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	dll

curve.dll
Puts a dimple in the flame; it can be horizontal, vertical, or both. Variables *xamp* and *yamp* specify the amplitude (positive for right or down). *Xlength* and *ylength* specify the width.

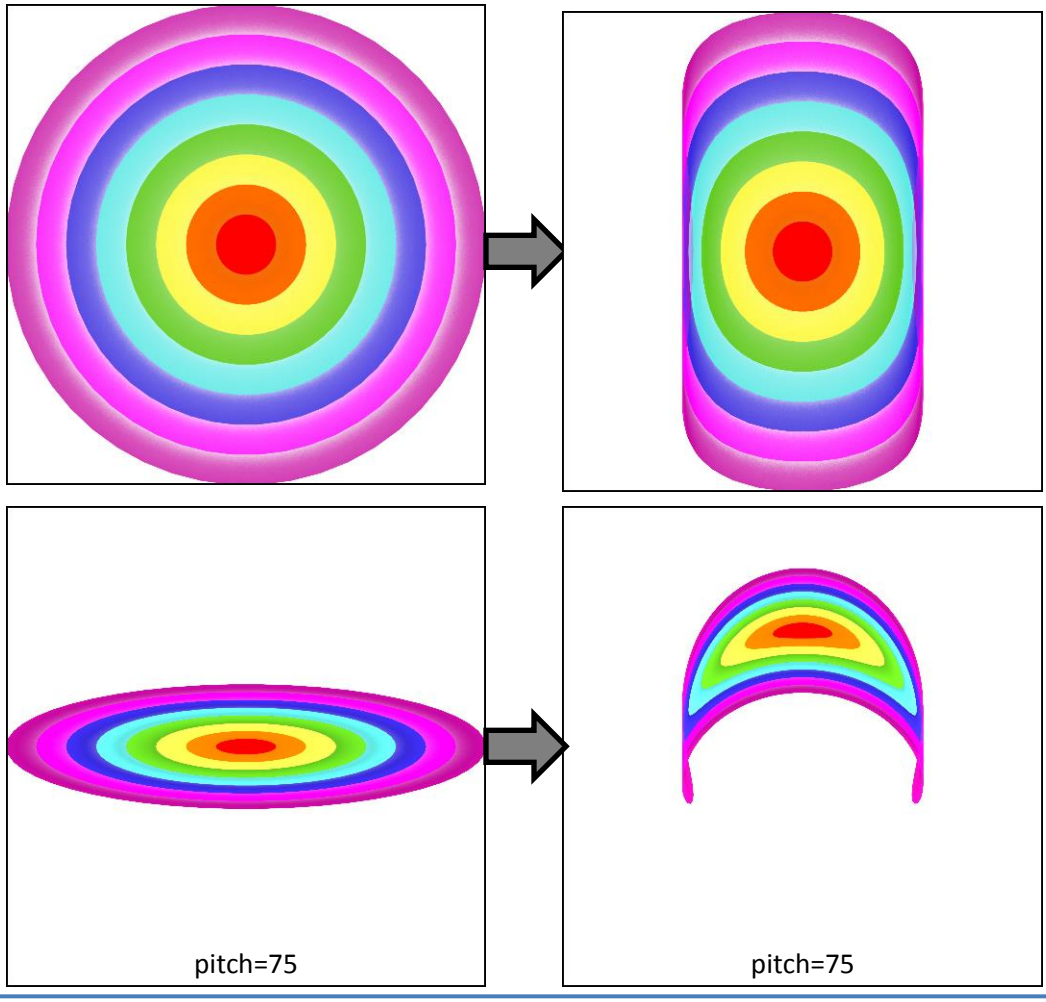
Example has *xamp* = 0, *yamp* = 0.8, *xlength* = 1, and *ylength*= 0.25



cylinder (2D/3D, sets z)

2.09	7X15B	7X16	jwf	ch
yes	yes	yes	yes	yes

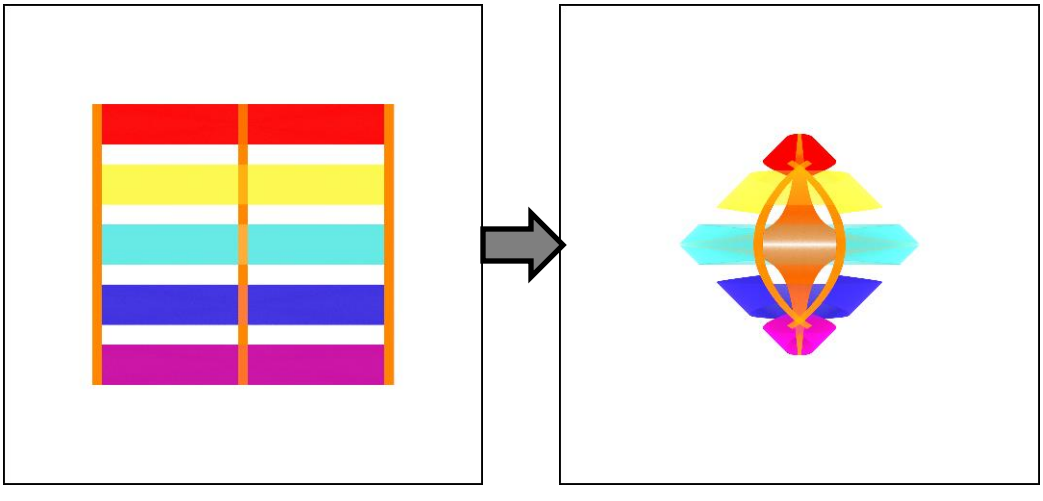
Curves the flame into a vertical cylinder. In 3D versions, it sets z (ignoring any previous value).



diamond (2D)

2.09	7X15B	7X16	jwf	ch
yes	yes	yes	yes	yes

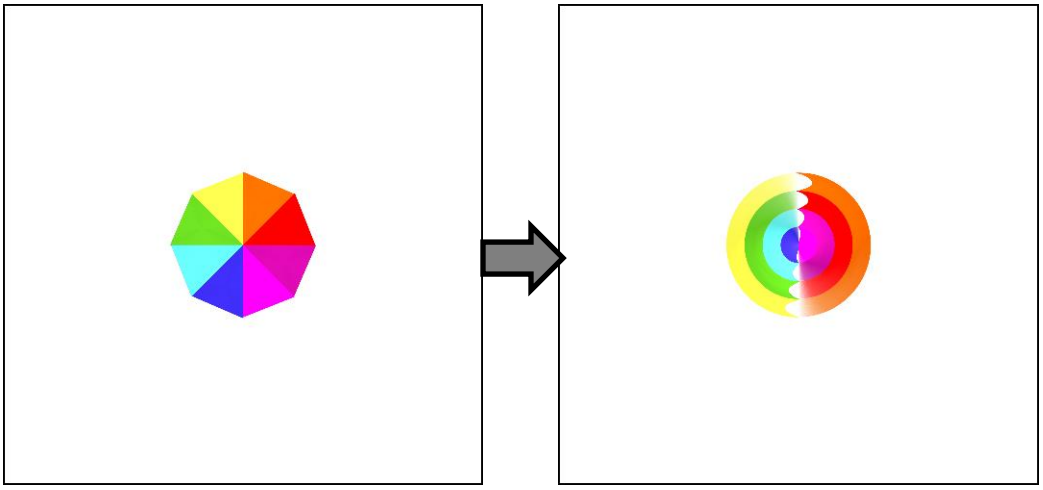
Warps flame into a diamond shape that fits in a unit circle. The example is carefully chosen to have minimal overlap, but as the original flame gets larger, diamond will warp it to fit the diamond shape.



disc (2D)

2.09	7X15B	7X16	jwf	ch
yes	yes	yes	yes	yes

Morphs the plane into a unit circle by turning wedges into arcs. The upper right quadrant starts at the bottom and arcs clockwise around the outside; the upper left quadrant starts at the top. The bottom half does the same on the inner half. The example has radius 1 so it stops before the arcs overlap to illustrate this; larger flames continue rotating around the circle. Compare idisc and wdisc.

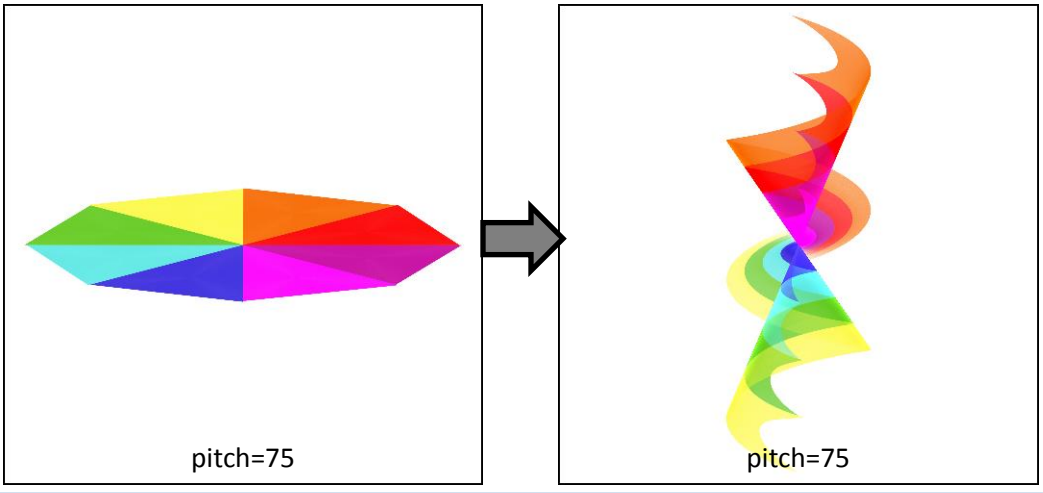


disc3d (3D, transforms z)

2.09	7X15B	7X16	jwf	ch
no	dll	dll	yes	no

disc3d.dll

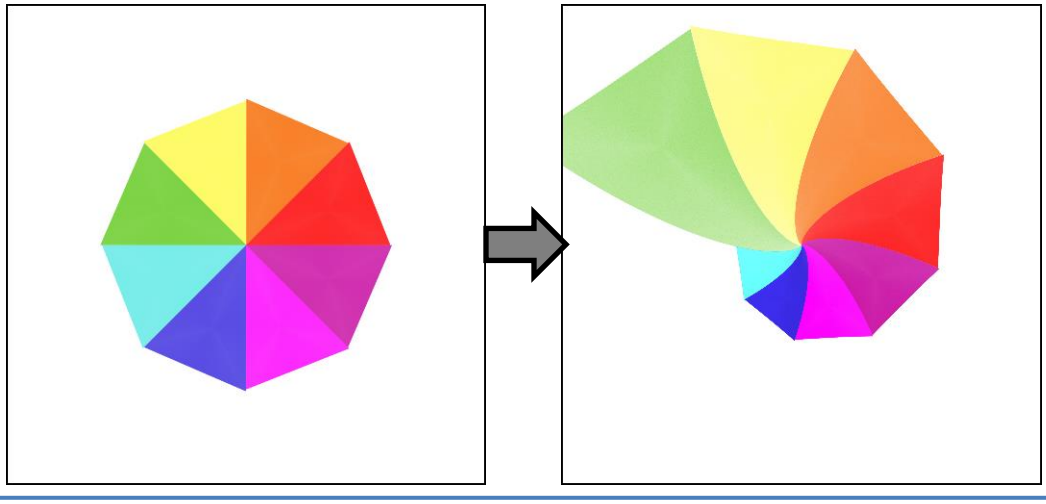
Changes x and y just like disc, but also transforms z according to the distance from the origin. The example shows the effect on a flat flame.



droste (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	no	dll

droste.dll
Takes a ring with radii *r1* and *r2* and unrolls it into a rectangle, rotates and shrinks the rectangle, then rolls it back up, now into a spiral. It does this for the entire flame, so only the ratio r2/r1 matters. The example uses *r1* = 1 and *r2* = 4.975. See escher which uses different math to produce the exact same effect.



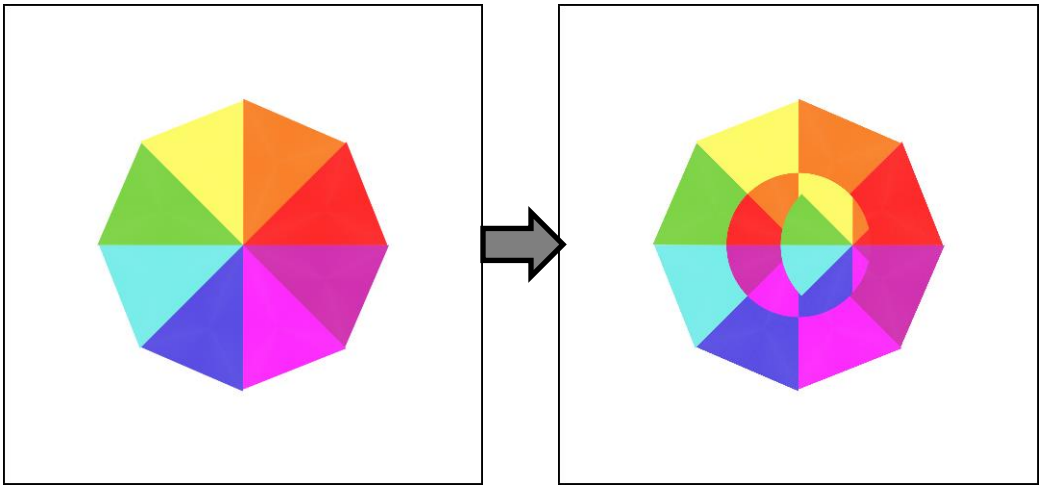
eclipse (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	dll

eclipse.dll

Shifts a circle in the middle of the flame right, filling in the gap with a mirror image of the circle. Variable *shift* controls how far the circle is shifted: 0 for no shift, 1 for halfway, 2 for all the way, negative to shift left.

Example has *shift* = 0.7.

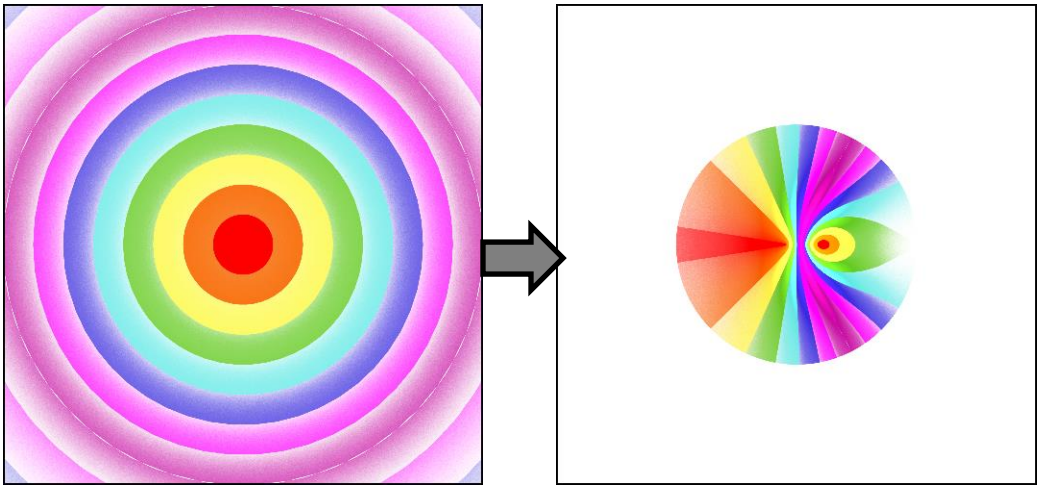


edisc (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	yes

edisc.dll

Morphs the flame into a circle based on the distance of each point from the center. The original flame here (truncated by necessity) starts with eight rings of varying color, which are then repeated in reverse order with increasing width (so red goes from 8 to infinity, orange goes from 4 to 8, etc.).

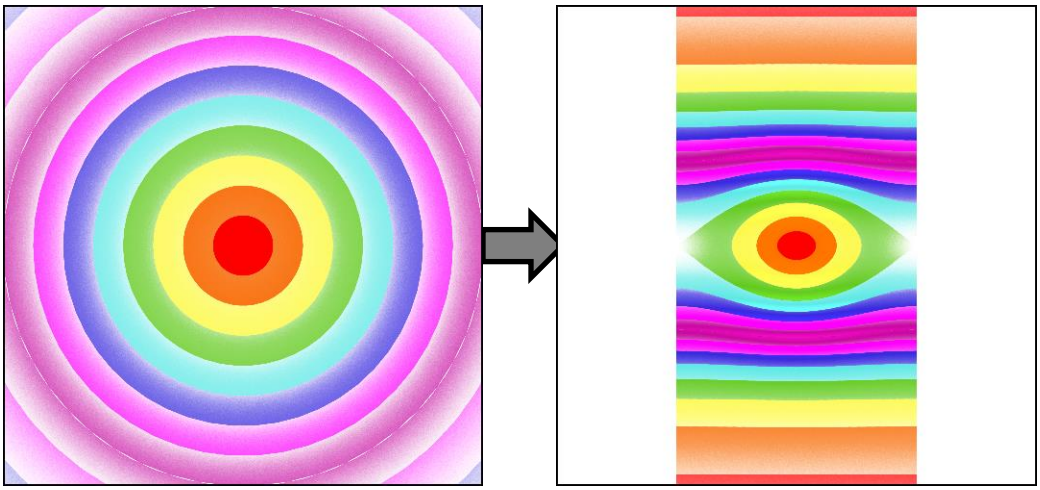


elliptic (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	yes	yes	yes

elliptic.dll

Stretches rings within distance 1 from the origin into ellipses. Splits rings further out and stretches them into progressively straighter lines. The width is the variation value.



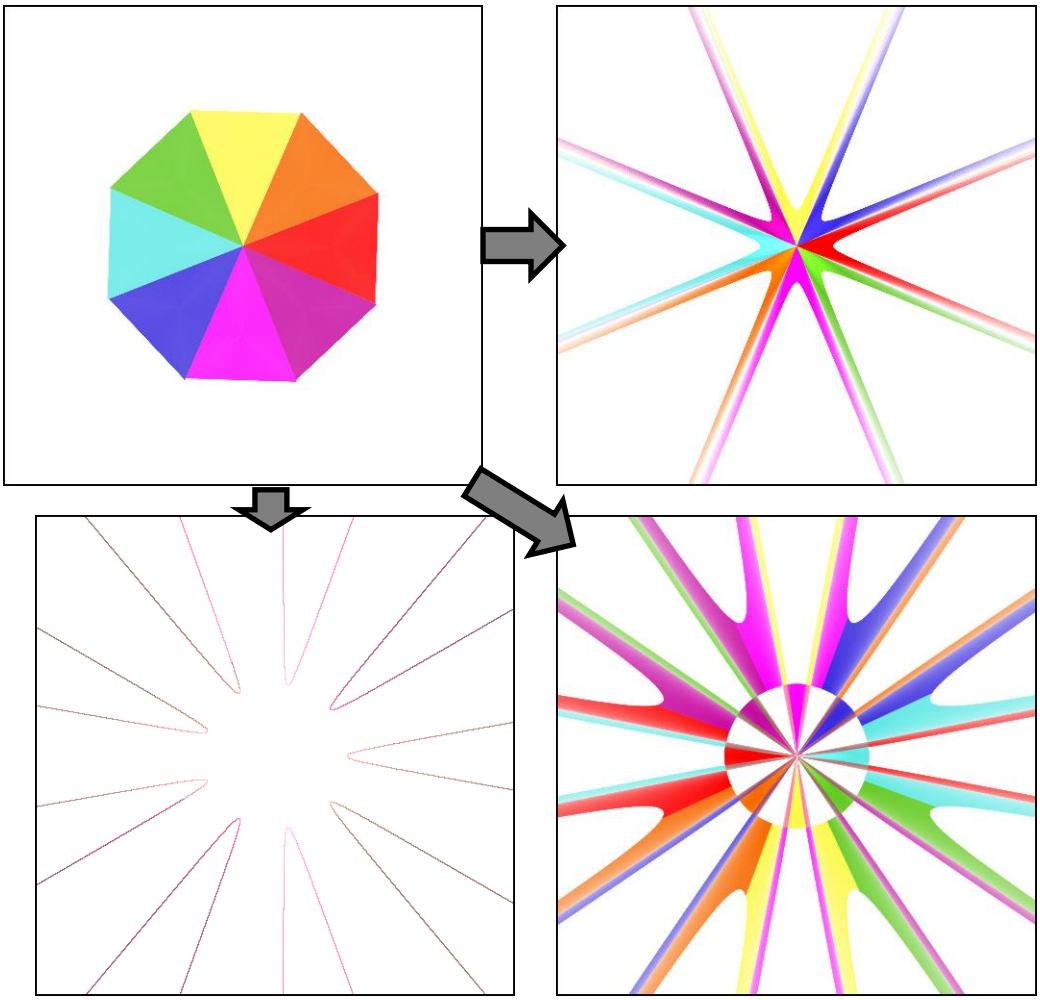
Epispiral (2D half blur)

2.09	7X15B	7X16	jwf	ch
dll	dll	yes	yes	yes

Epispiral.dll

A mathematical curve that isn't really a spiral. If *n* is odd, there are *n* sections, otherwise twice *n*. (Actually, there are always twice *n* sections, but when *n* is odd half the sections overlap.) Setting *thickness* to 0 results in a line blur (except in 7x16); otherwise a shape. Setting *holes* puts a circle in the center filled with holes and points according to the other variables.

Top right: *n* = 4, *thickness* = 0.5, *holes* = 0
Bottom right: *n* = 8, *thickness* = 1, *holes* = 1
Bottom left: *n* = 9, *thickness* = 0, *holes* = 0

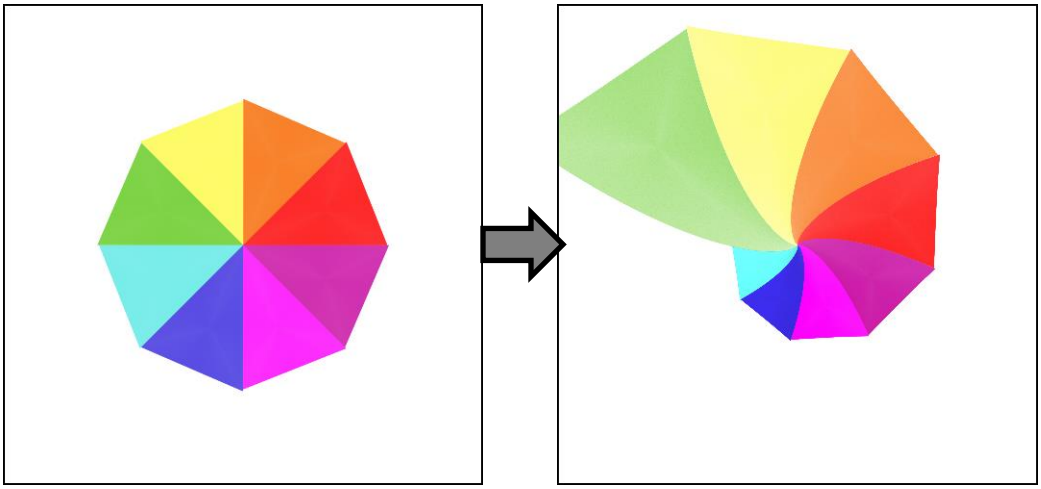


escher (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	yes	yes	yes

escher.dll
Implements Escher's Map by treating each point of the flame as a complex number and raising it to a power determined by the variable *beta*. The example shows *beta* = 0.5.

See cpow (allows using an arbitrary complex number) and droste (same effect using different math).

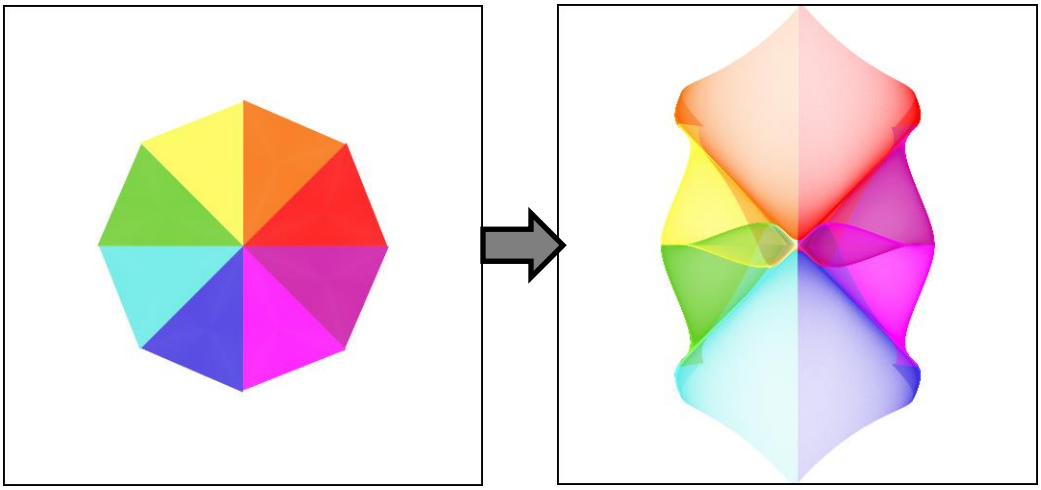


ex (2D)

2.09	7X15B	7X16	jwf	ch
yes	dll	dll	yes	yes

ex.dll

I've heard this variation is called ex because it often transforms the plane into an X shape.

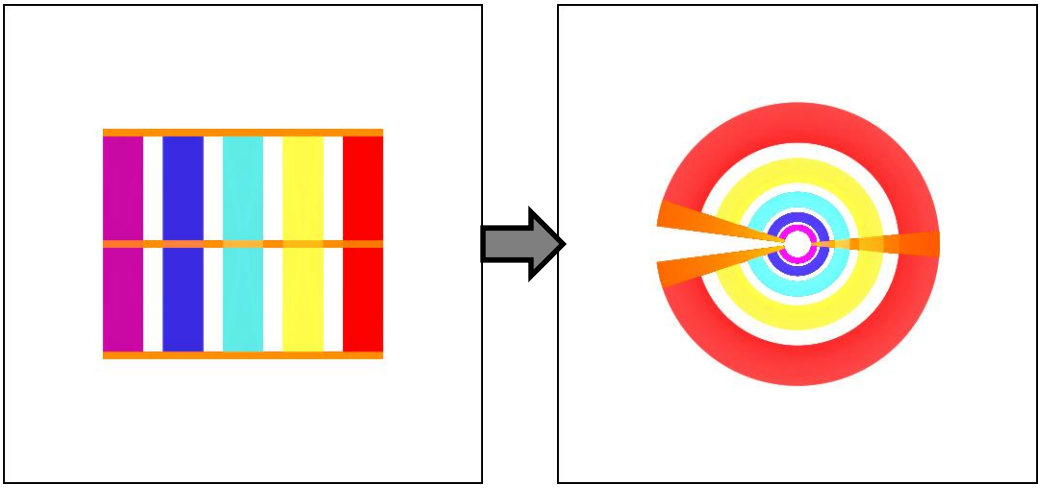


exponential (2D)

2.09	7X15B	7X16	jwf	ch
yes	dll	dll	yes	yes

exponential.dll

Polar conversion (kind of opposite of polar). Distance is e^(x-1) and angle is pi*y. This converts vertical lines to rings and horizontal lines to wedges.

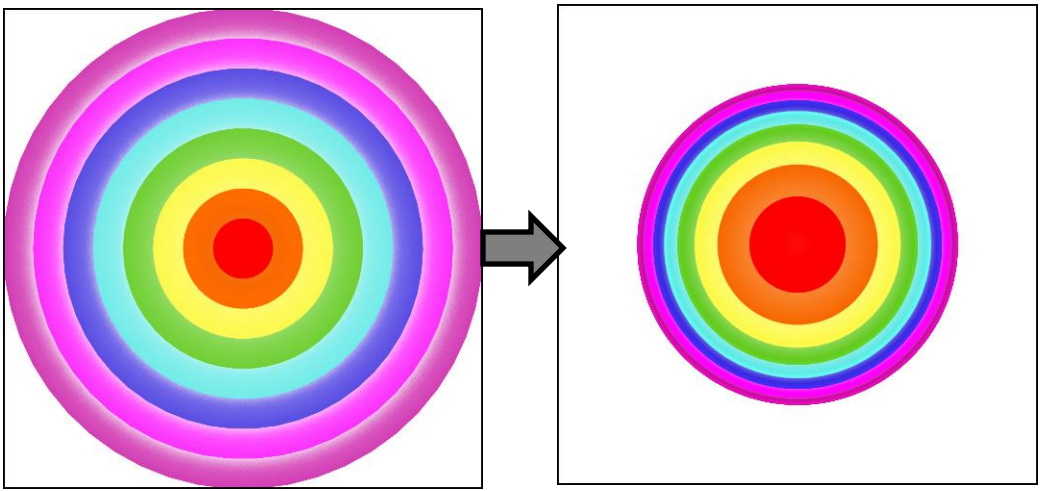


eyefish (2D)

2.09	7X15B	7X16	jwf	ch
yes	yes	yes	yes	yes

Generates a fisheye effect, expanding points close to the center and contracting points further away.

fisheye does the same, but it also swaps x and y in the result.



falloff2 (3D, transforms z)

2.09	7X15B	7X16	jwf	ch
dll	dll	yes	yes	dll

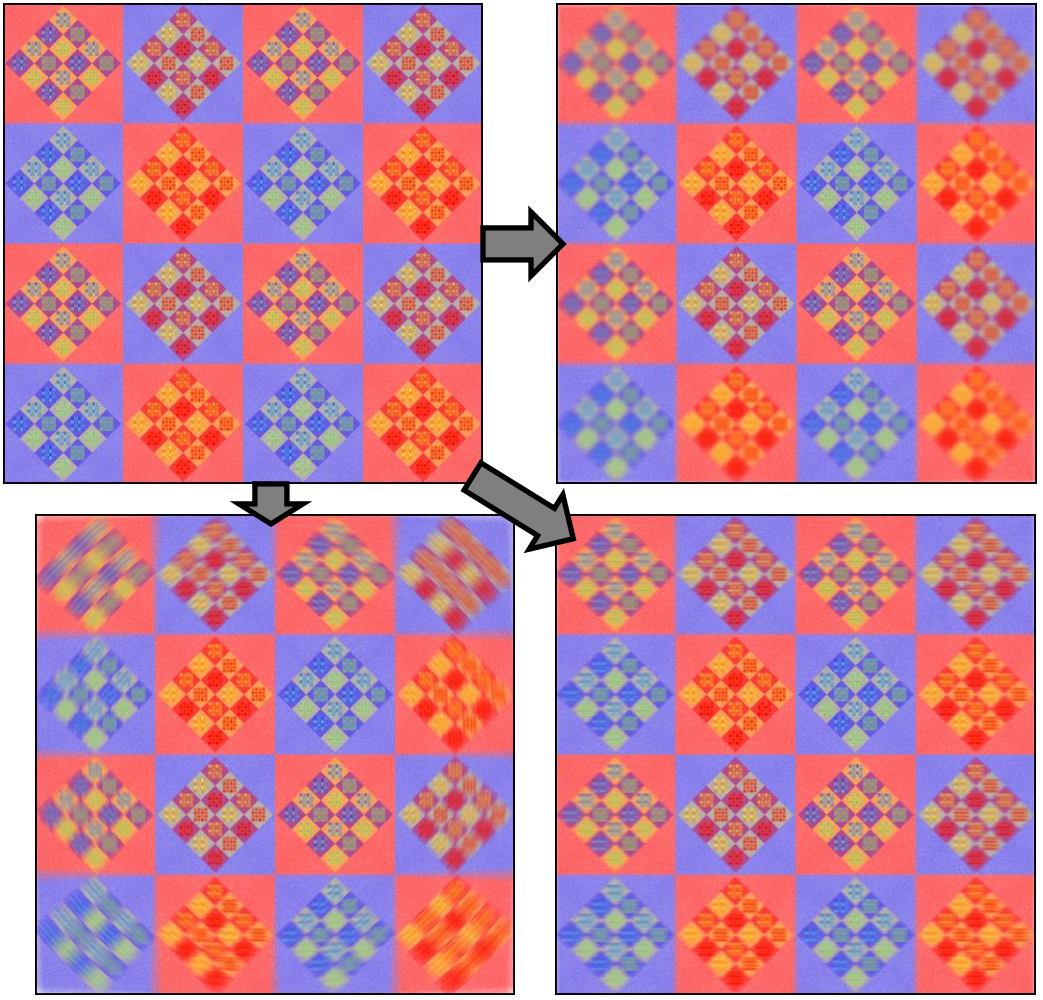
falloff2.dll, post_falloff2.dll, pre_falloff2.dll

Blurs the plane outside a sphere defined by *x0*, *y0*, *z0*, and *mindist*. Three types are available: set *type* to 0 for linear blur (top right), 1 for radial blur (bottom left), or 2 for Gaussian blur (bottom right). Blur strength is controlled by *scatter* (the default, 1, is used in all examples).

For types 0 and 2, *mul_x*, *mul_y*, and *mul_z* control the strength for each axis (set to 1, 1, and 0 for the right examples). For type 1, *mul_x* controls the ray blur (center outward), *mul_y* controls circular blur around the z-axis, and *mul_z* controls circular blur around the x-axis (set to 0, 1, and 0 for the bottom left example).

Although a full 3D variation, it works fine in 2D versions; but keep *mul_z* = 0 or your flames may look different if opened in a 3D version.

Supercedes **falloff** and **post_rblur**.



fan2 (2D)

2.09	7X15B	7X16	jwf	ch
yes	yes	yes	yes	yes

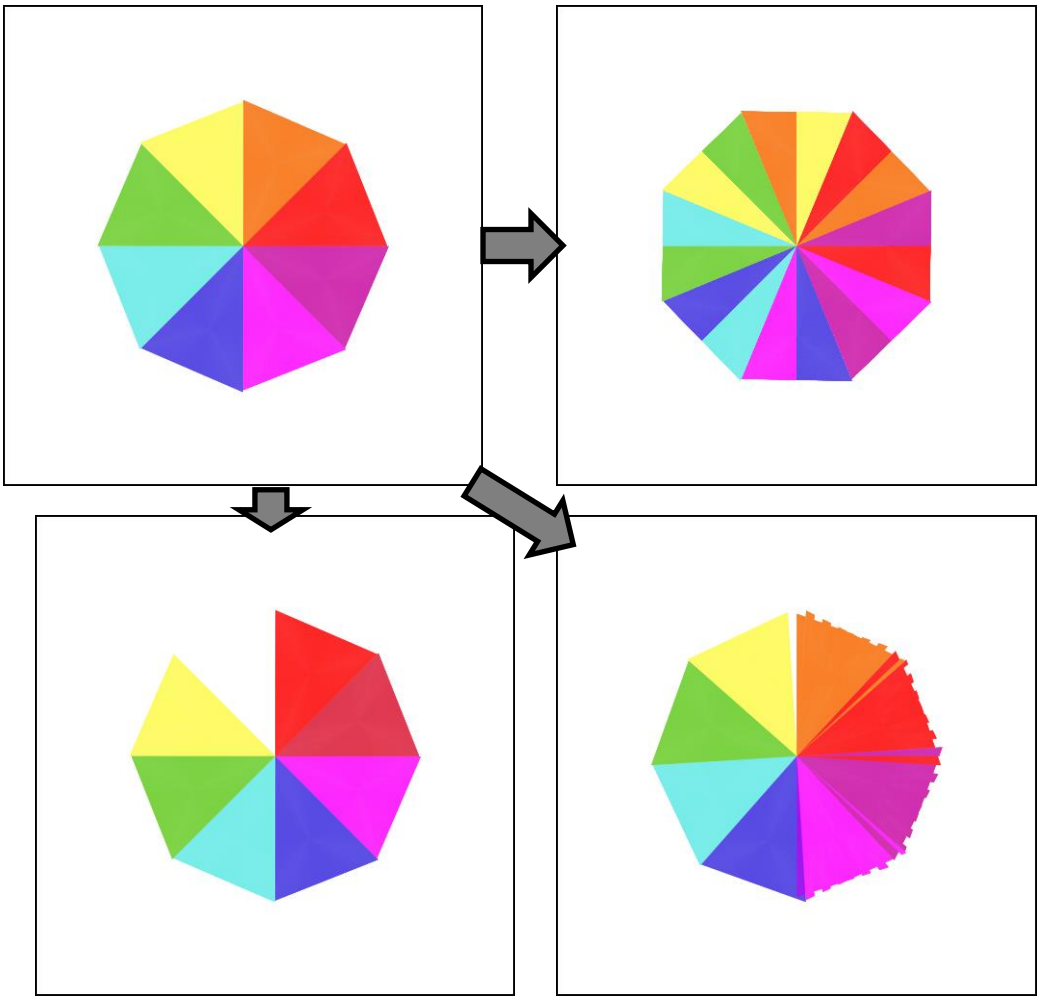
Divides the plane into wedges, then rotates them right or left. Variable *x* sets the size of the wedges. When *y* is 0, wedges on the left side are rotated left and wedges on the right are rotated alternately. Increasing *y* makes more wedges rotate alternately and decreasing *y* makes more rotate the same direction.

Top right: *x* = 0.5 (makes 16 wedges) and *y* = 3.534, which is large enough to make all the wedges alternate direction.

Bottom right: x = 0.2 (makes 100 wedges) and *y* = 0. The alternations on the right show mostly in the jagged edges. The left side wedges are evenly rotated left.

Bottom left: *x* = 0.707 (makes 8 wedges) and *y* = -1.57. The red and green wedges swapped, and the rest rotated left, leaving a space at the top and making the green and magenta wedges overlap.

Supercedes **fan**.

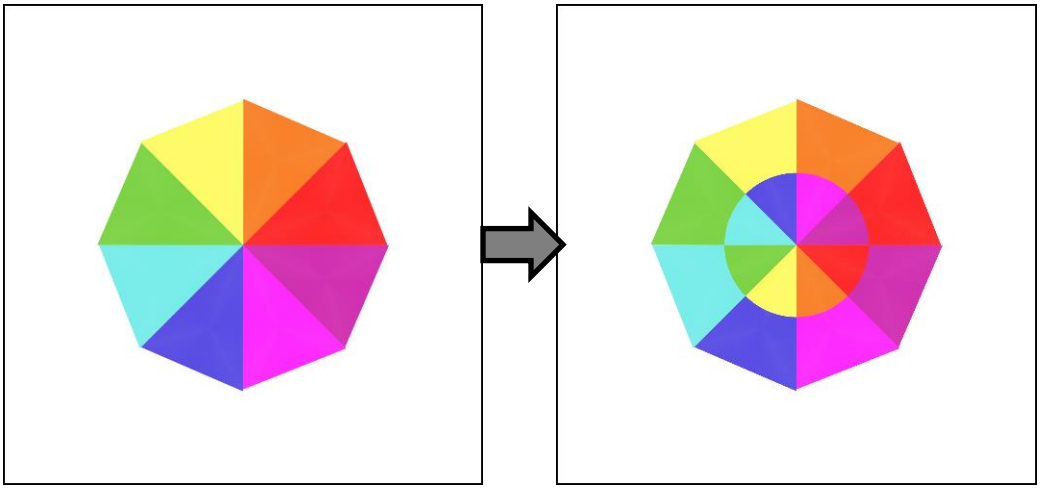


flipcircle (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	dll

flipcircle.dll

Flips points within a circle top to bottom. Radius of circle is the value of the variation.

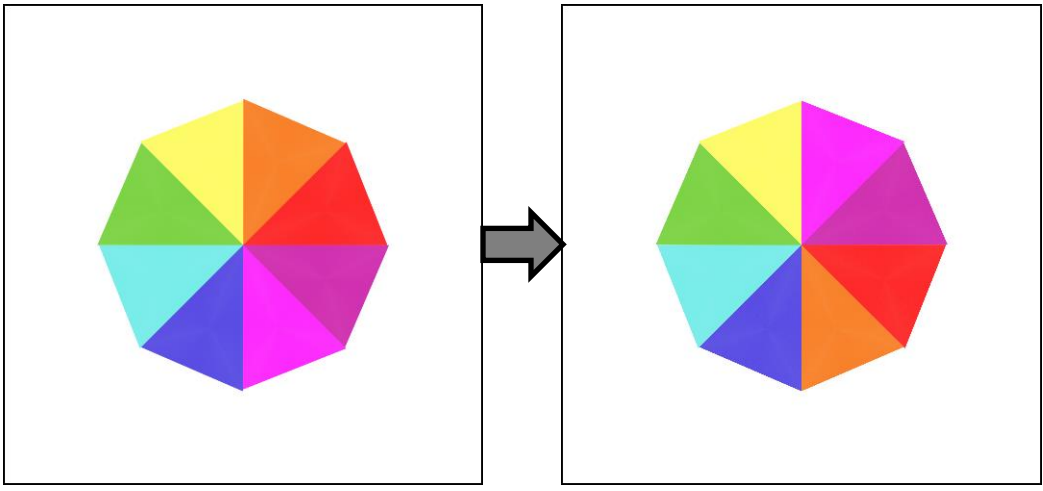


flipy (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	dll

flipcircle.dll

Flips points on the right side of the flame top to bottom.



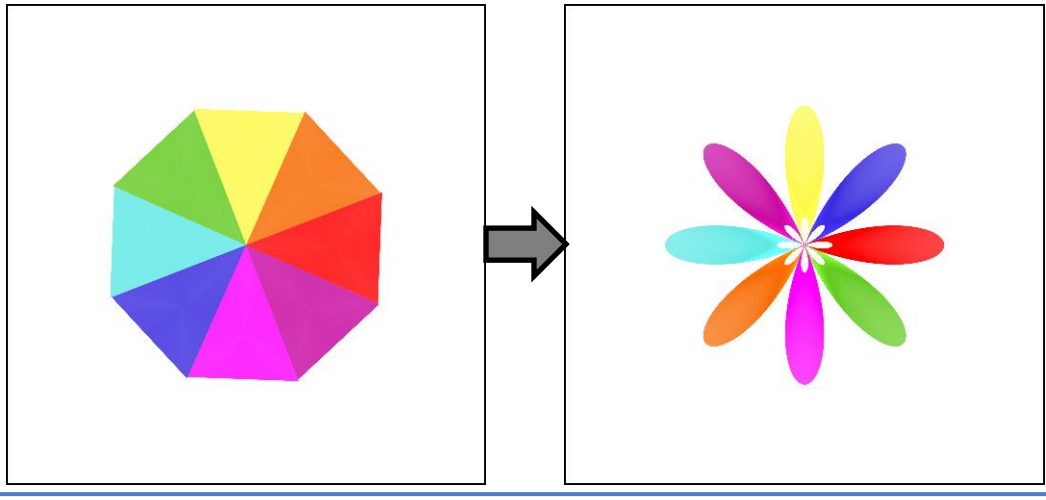
flower (2D half blur)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	yes

FlowerVariationPlugin.dll
Makes a flower shape; there are twice the variable *petals* petals, except that if it is odd half of the petals will overlap, giving only *petals* petals. Variable *holes* puts holes in the petals and affects the size.

Example has *petals* = 4 and *holes* = -0.25.

Compare epispiral; it uses similar math.



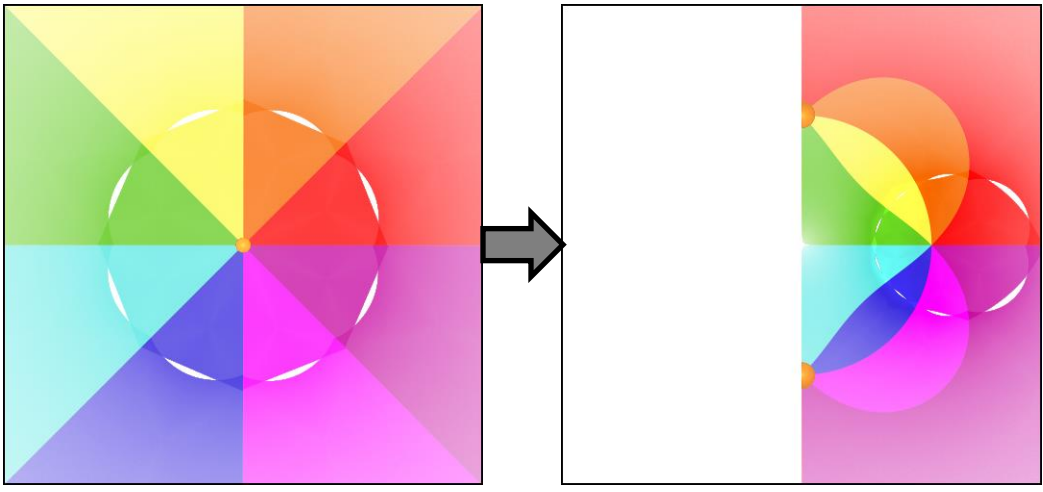
flux (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	yes

flux.dll

Converts radial lines to curved flux lines.

Variable *spread* is a scaling factor; in the example *spread* = -0.2.



foci (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	yes	yes	yes

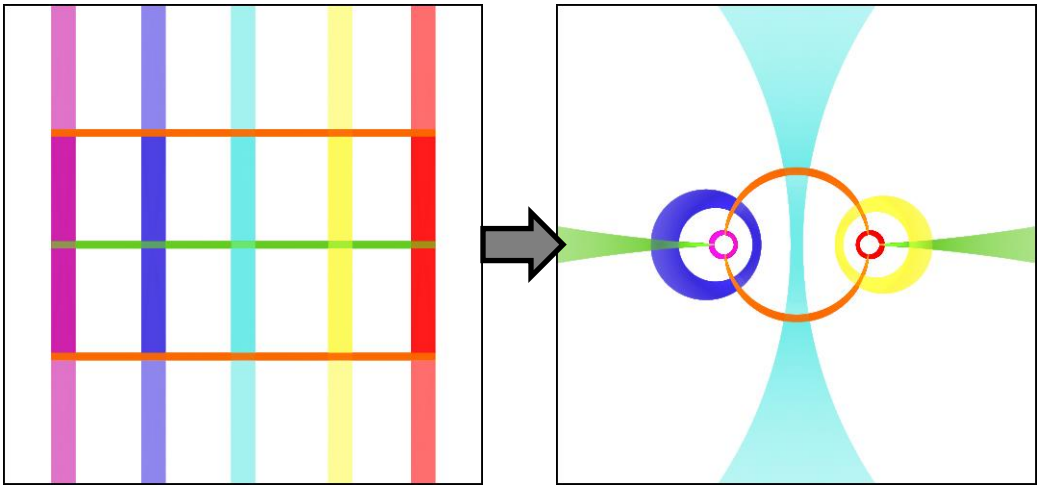
foci.dll

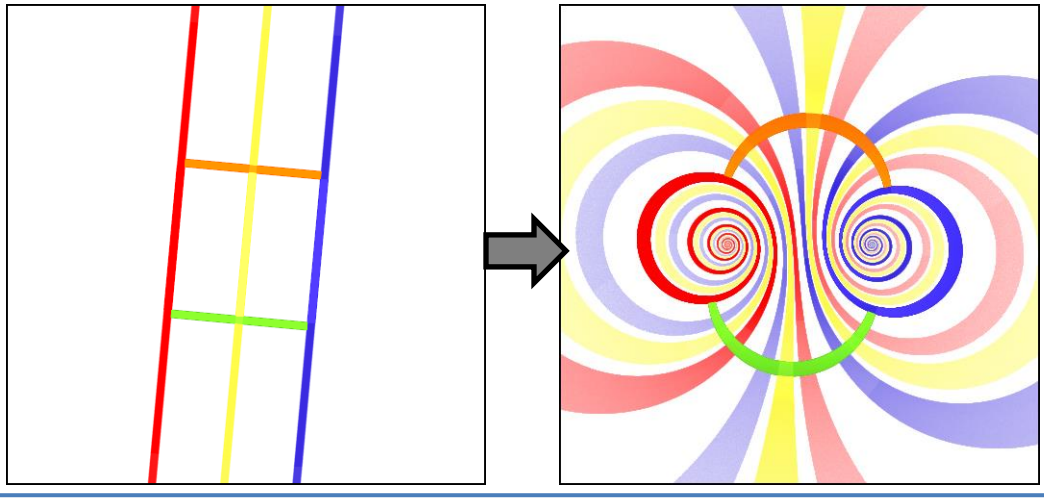
Wraps the plane around a horizontal cylinder, then tapers the ends and bends them away to form a U, looking up from the bottom.

In the first example, the vertical lines are infinite and map to smaller circles as they get away from the center. The intersection of the middle horizontal and vertical lines is at infinity (or think of it as "behind" the view point so not visible).

The second example is more complex, but better shows how foci works. The infinite vertical lines are slanted slightly, so when wrapped around the cylinder they spiral around. These spirals can be seen when the cylinder is bent and tapered.

See unpolar.



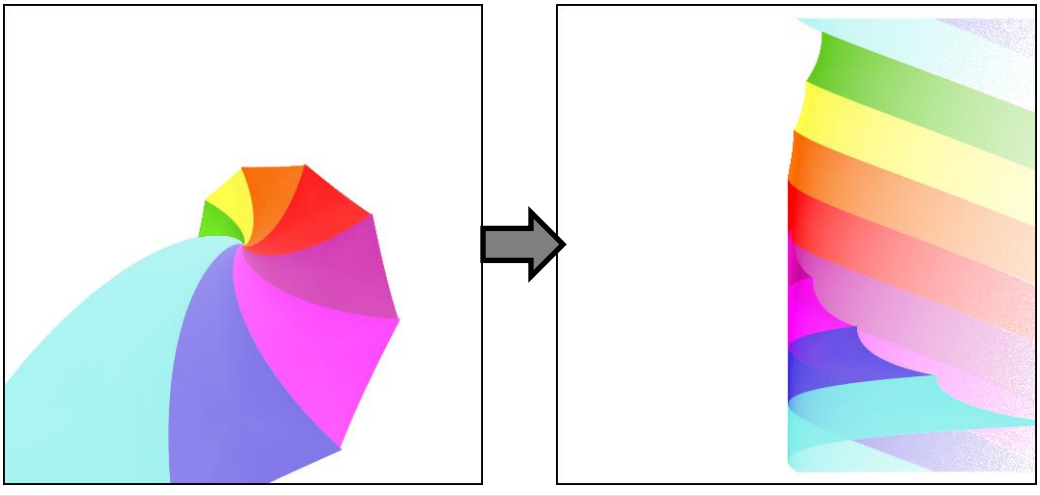


gamma (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	no	dll

Converts each point to polar coordinates and uses the angle for the y coordinate, converting wedges to horizontal ribbons. The gamma function is used to compute x from the distance. The example uses escher for the original, so the wedges aren't straight, but it illustrates the math.

Compare polar.



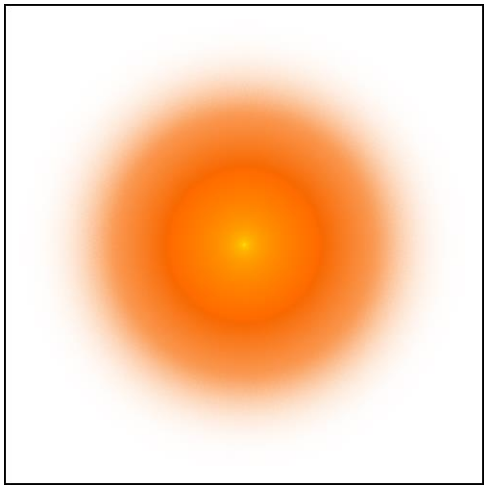
gaussian_blur, pre_blur (2D blur)

2.09	7X15B	7X16	jwf	ch
yes	yes	yes	yes	yes

A fuzzy circle with a bright center and a ring in the middle. (The spot and ring don't show with all colors.)

Compare blur and blur3D.

pre_blur is a pre_ version of gaussian_blur (not blur as the name would imply).



gdoffs (2D, passes z)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	dll

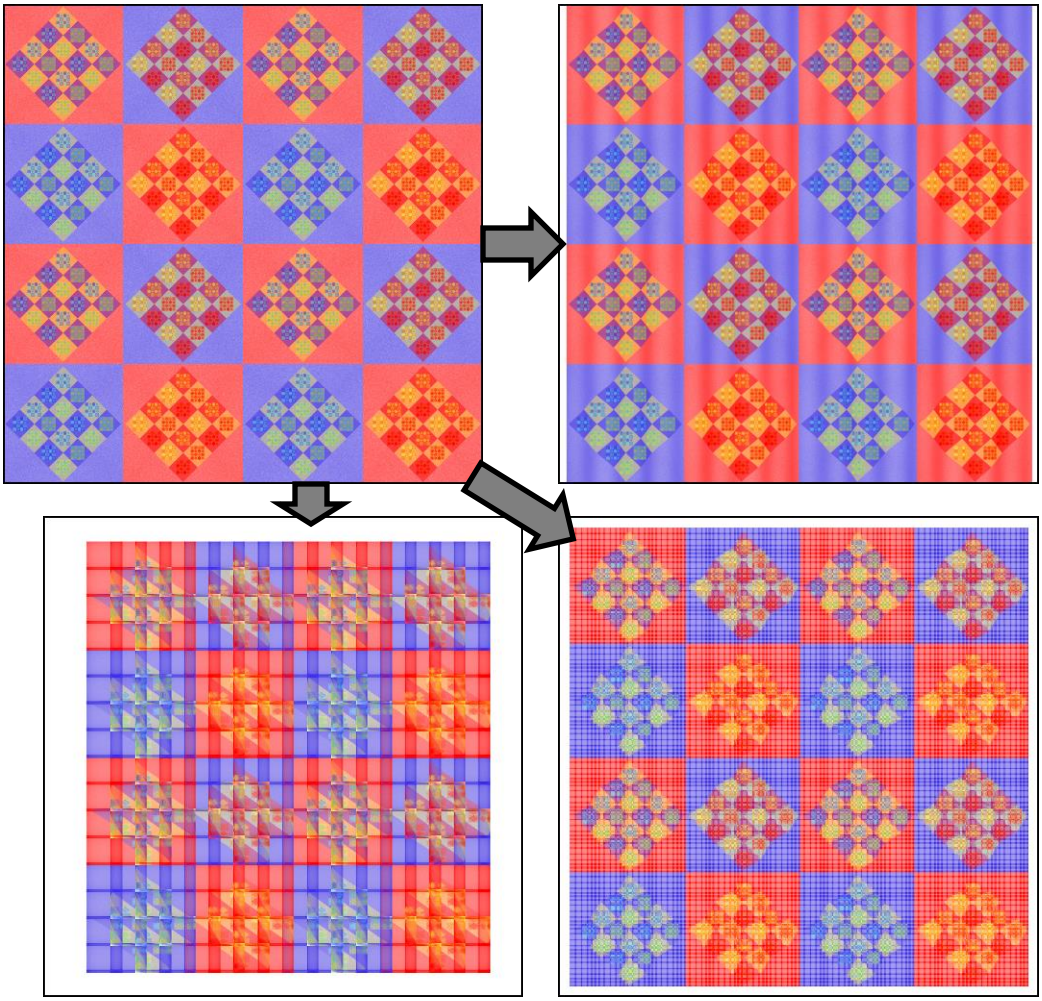
gdoffs.dll

Gdoffs is short for “grid offset”, and it can produce several effects based on a grid, including the corrugation, crosshatch, and plaid type patterns shown here. Variables *delta_x* and *delta_y* control the amount of distortion; effective range is from 0 (no distortion) to just under 5. Only the larger of *area_x* and *area_y* is used, so it’s easiest to just set *area_y* to 0, as is done for the examples here. If *square* is 1, then *delta_y* will be ignored and *delta_x* used for both dimensions.

Top right: *delta_x* = 0.4, *delta_y* = 0, *area_x* = 2.5, *gamma* = 1.

Bottom left: *delta_x* = 1, *delta_y* = 0.75, *area_x* = 1, *gamma* = 1.

Bottom right: *delta_x* = 0.5, *delta_y* = 0.5, *area_x* = 1, *gamma* = 6.

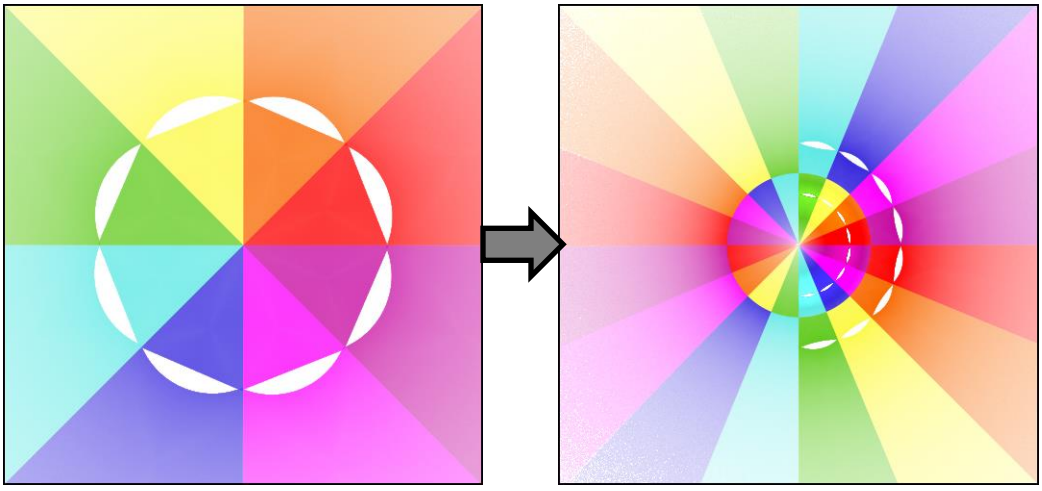


glynnia (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	yes

glynnia.dll

The unit circle maps to the left side of the result, in two places: scrunched and rotated to form an inner half-circle, and turned inside-out and rotated to form the background. The rest maps to the right side: scrunched and rotated to form the background, and turned inside-out and rotated to form the inner half-circle.

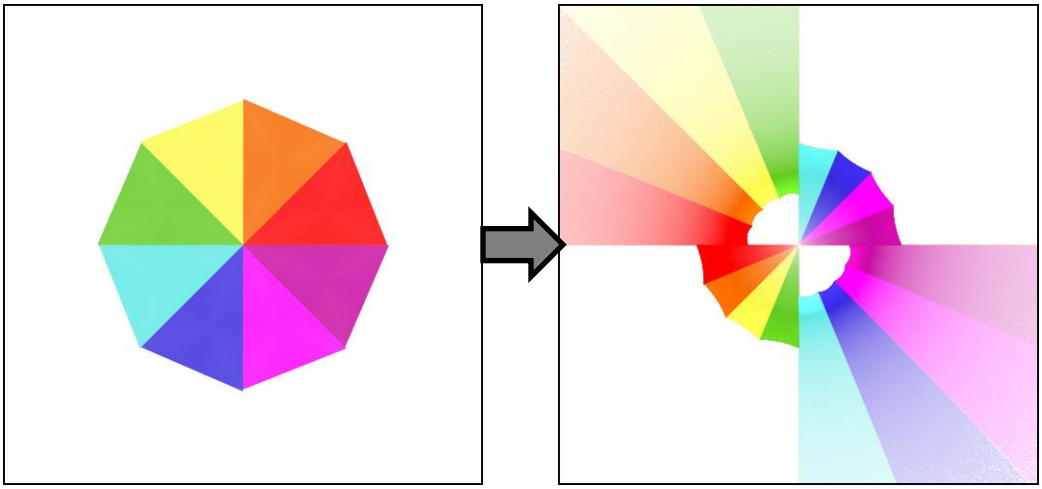


glynnia2 (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	no	dll

glynnia2.dll

The top half is scrunched and rotated to form the bottom left of the result, and flipped and turned inside-out to form the top left. The bottom half is flipped and scrunched to form the top right half, and flipped back and turned inside-out to form the bottom right.



GlynnSim1 (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	dll

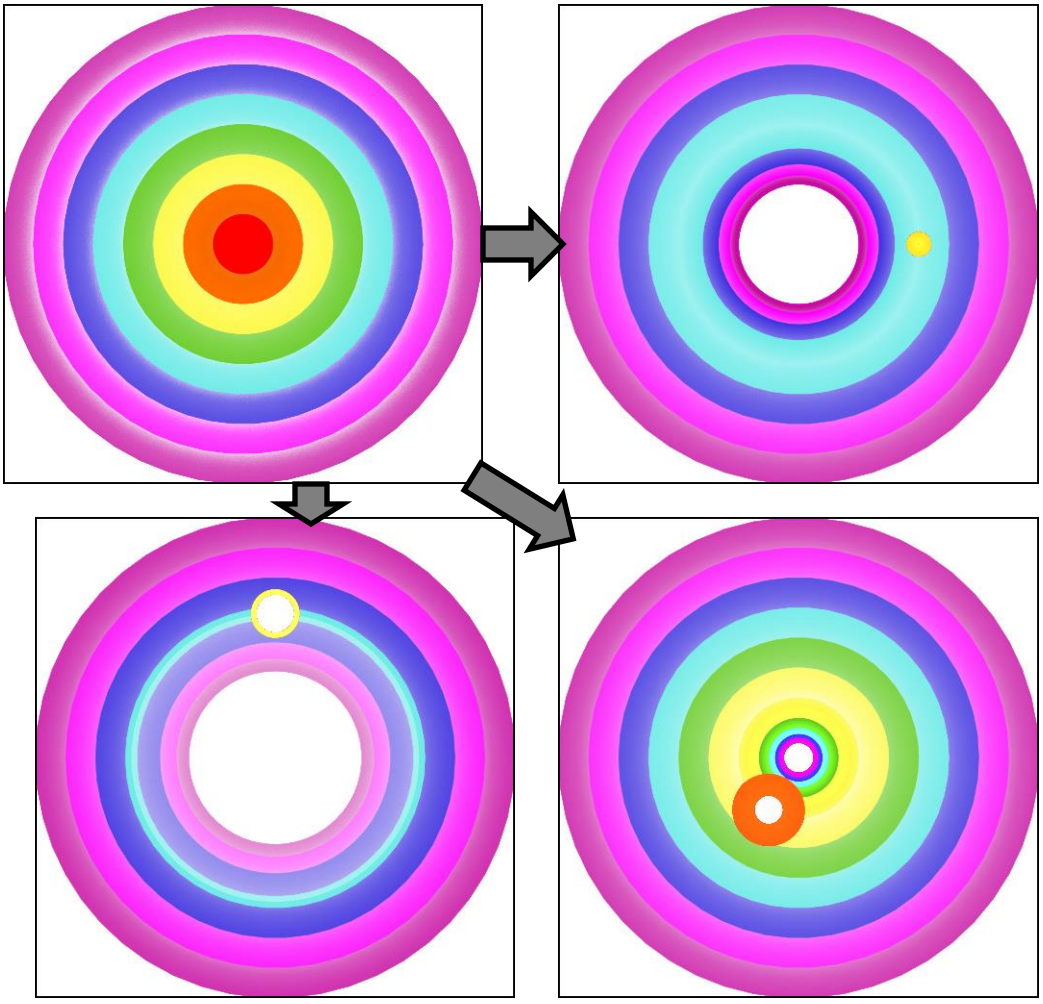
GlynnSim1.dll
The GlynnSim variations pass the part of the plane outside *radius* intact, and replace the inside with an everted copy of the outside. Variables *contrast* and *pow* control the density balance between inside and outside. Compare with spher.

GlynnSim1 converts the area inside *radius* to a circle of radius *radius1* with its center at distance *radius* and angle *Phi1* (in degrees). It has a hole in the center with a size defined by *thickness* (from 0, no hole, to 1, just the circle outline).

Top right: Default values, *radius* = 1, *radius1* = 0.1, *Phi1* = 0, *thickness* = 0.1, *pow* = 1.5, and *contrast* = 0.5.

Bottom left: *radius* = 1.2, *radius1* = 0.2, *Phi1* = -90, *thickness* = 0.8, *pow* = 5, and *contrast* = 0.1.

Bottom right: *radius* = 0.5, *radius1* = 0.3, *Phi1* = 120, *thickness* = 0.4, *pow* = 1, and *contrast* = 1.



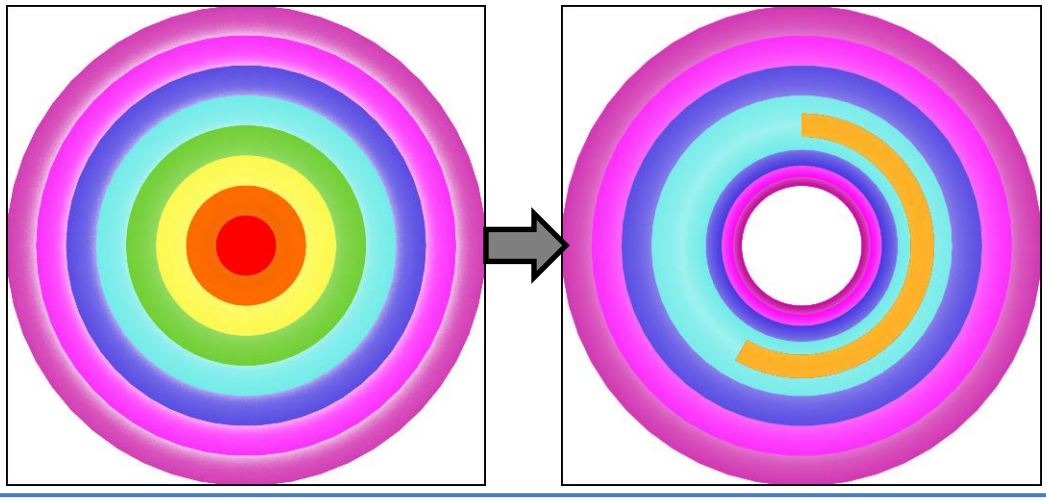
GlynnSim2 (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	dll

GlynnSim2.dll

Similar to GlynnSim1, but instead of a circle it makes an arc between *Phi1* and *Phi2* (in degrees) with thickness *thickness*.

Example uses *radius* = 1, *thickness* = 0.1, *contrast* = 0.5, *pow* = 1.5, *Phi1* = 120, and *Phi2* = -90.



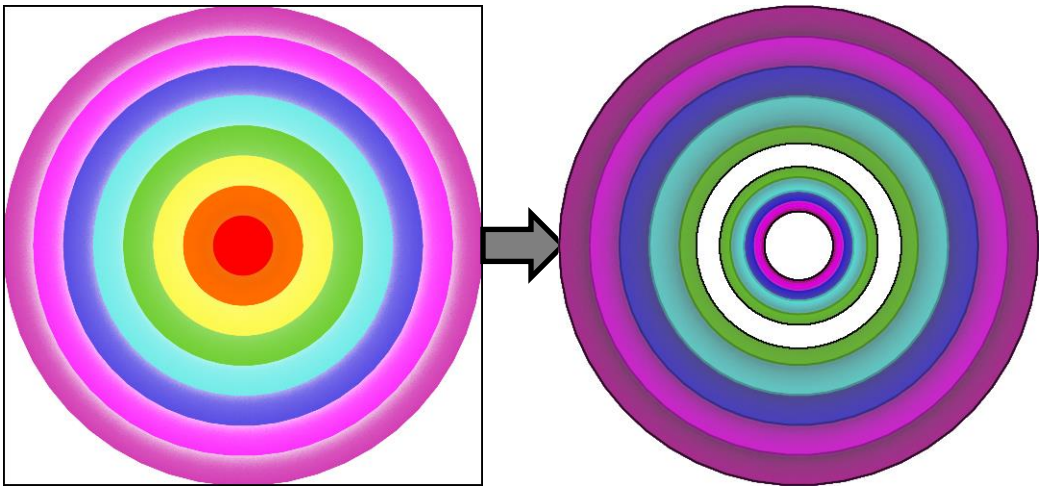
GlynnSim3 (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	dll

GlynnSim3.dll

Similar to GlynnSim1, but instead of a circle it leaves a space at *radius* with thickness *thickness*. (Variable *thickness2* is not used.)

Example uses *radius* = 0.75, *thickness* = 0.1, *thickness2* = 0.1, *contrast* = 0.5, and *pow* = 1.5.



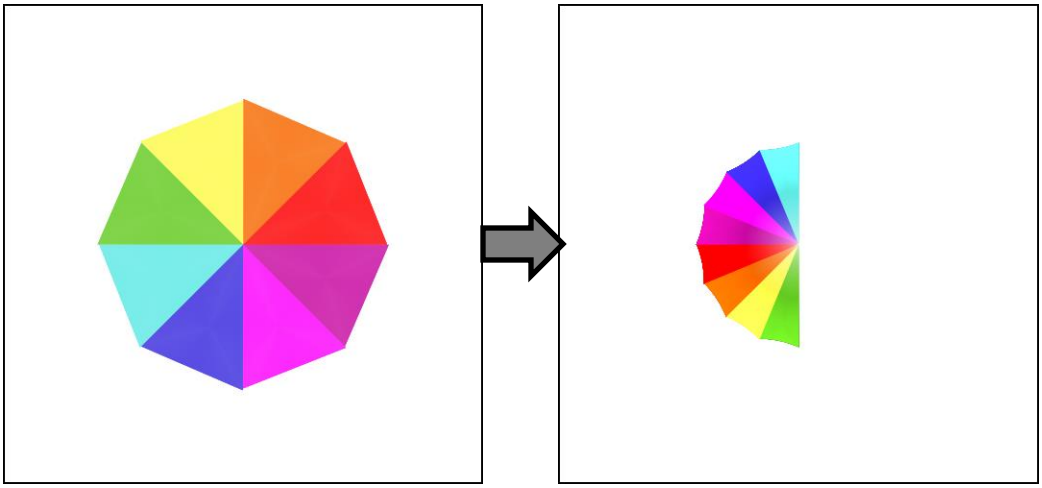
Half_Julian (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	no	dll

Half_Julian.dll

Like julian, but doesn't replicate the result to fill the hole. Unlike julian, *power* does not need to be an integer.

Compare cpow with *i* = 0 and *r* = 1/*power*.

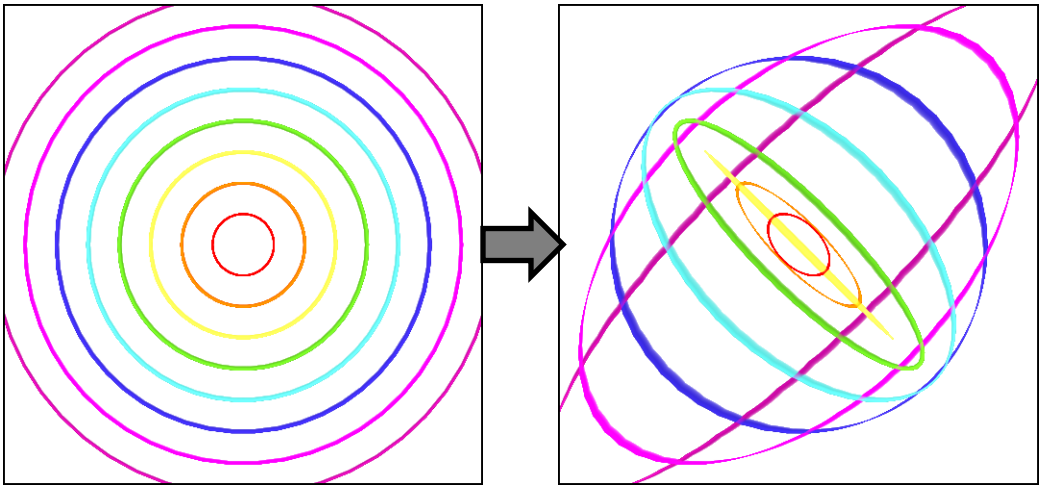


handkerchief (2D)

2.09	7X15B	7X16	jwf	ch
yes	dll	dll	yes	yes

handkerchief.dll

Transforms circles centered at the origin to ellipses. Circles (like the blue one in the example) whose radius is a multiple of π/2 will remain circles; circles (like the yellow one) whose radius is an odd multiple of π/4 become diagonal lines. The overall effect is reminiscent of folding a handkerchief.



post_heat (3D, transforms z)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	no	dll

post_heat.dll

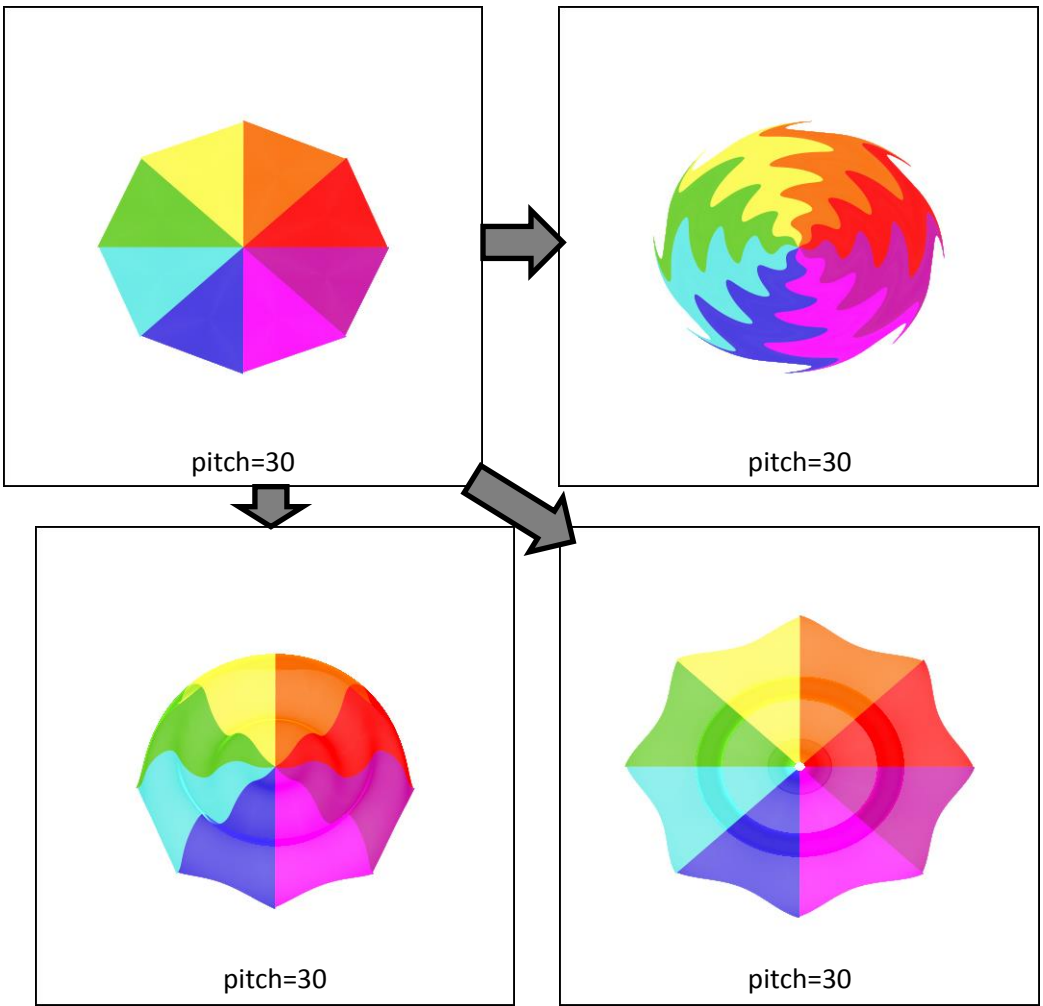
Applies a combination of three wave functions: theta is circular around the origin, phi is up and down (z-axis), and r is in and out from the origin (which usually shows up as a density variation). The examples show them individually for clarity.

There are three variables for each wave that control the period (smaller values for higher frequencies), phase (0 to π), and amplitude (set to 0 for none of that kind of wave).

top right: *phi_amp* = *r_amp* = 0, *theta_period* = 0.5, *theta_phase* = 0, and *theta_amp* = 0.3

bottom left: *theta_amp* = *r_amp* = 0, *phi_period* = 1, *phi_phase* = 1.5, and *phi_amp* = 0.3

bottom right: *theta_amp* = *phi_amp* = 0, *r_period* = 1, *r_phase* = 1.5, *r_amp* = 0.4

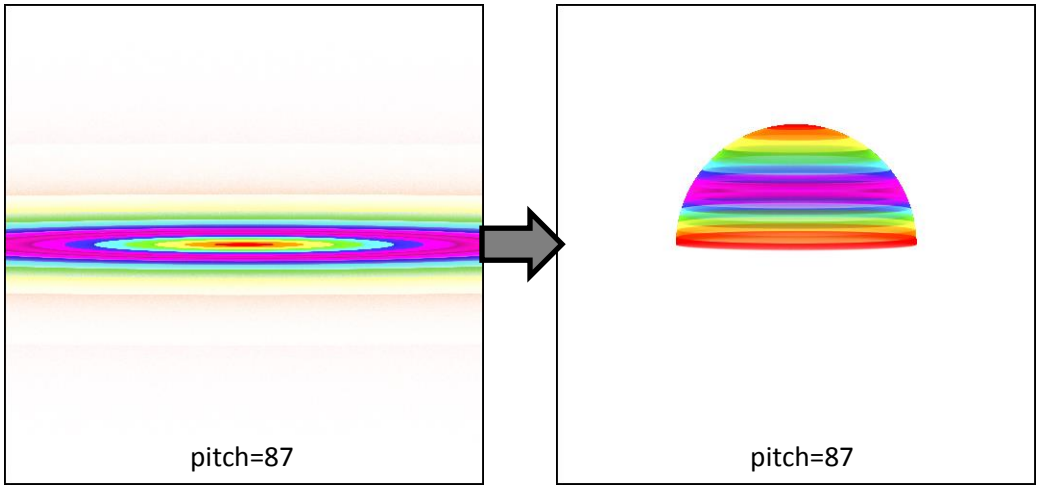


hemisphere (3D, sets z)

2.09	7X15B	7X16	jwf	ch
dll	yes	yes	yes	yes

hemisphere.dll

Projects the flame onto a hemisphere. It is like bubble, but with a half sphere instead of whole one.



hexes (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	dll

hexes.dll

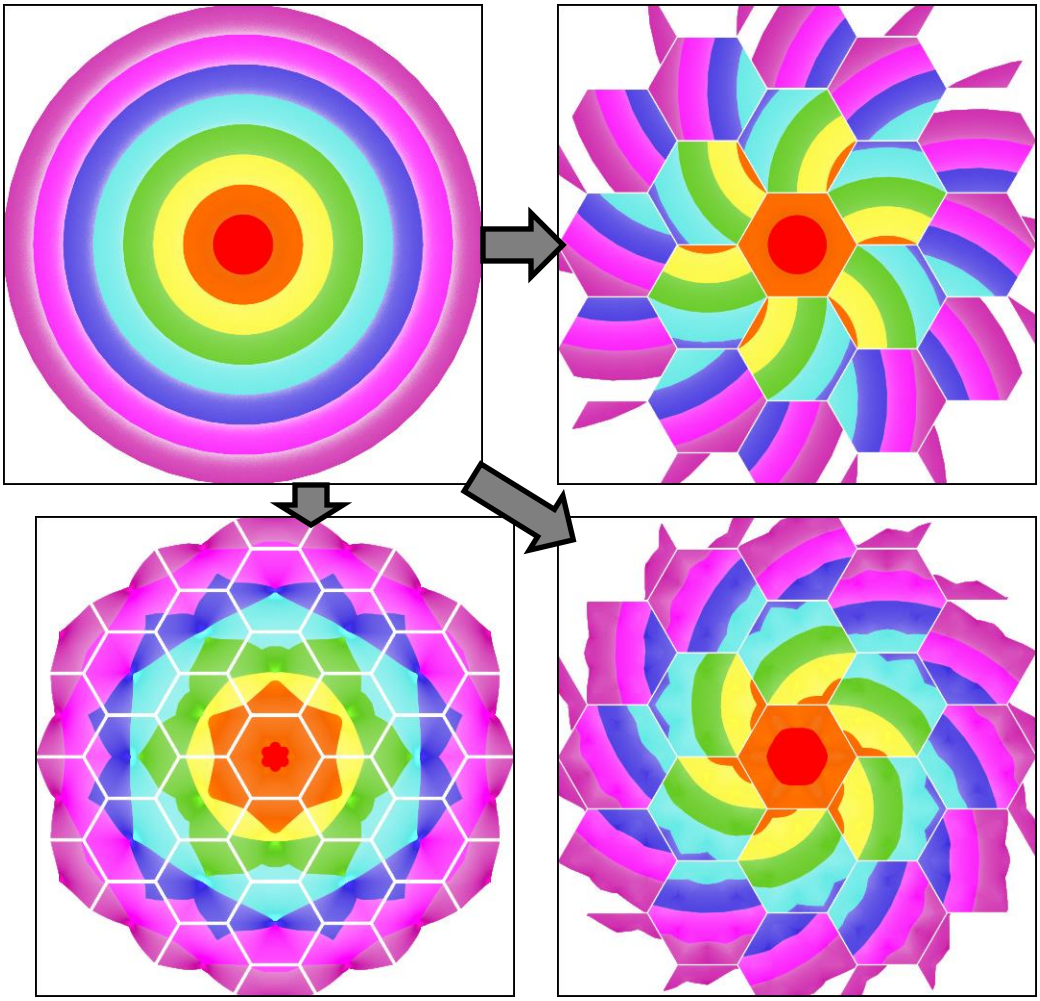
Divides the plane into hexagon tiles and rotates and scales each in place.

cellsize – the size of each hexagon
power – distorts each cell using a power function (1 for normal, 0 for outline only)
rotate – factor (0 to 1) to rotate each tile. If not a multiple of 1/6 (0.1667), the contents will be distorted to fit (see bottom right example).
scale – scale factor for each hexagon; less than 1 will leave spaces; greater than 1 will make them overlap.

Top right: *cellsize* = 0.5, *power* = 1, *rotate* = 0.166, *scale* = 0.98

Bottom right: *cellsize* = 0.5, *power* = 1, *rotate* = 0.0833 (1/12), *scale* = 0.98

Bottom left: *cellsize* = 0.4, *power* = 3, *rotate* = 0, *scale* = 0.95 (no rotation to demonstrate distortion effect of *power*)

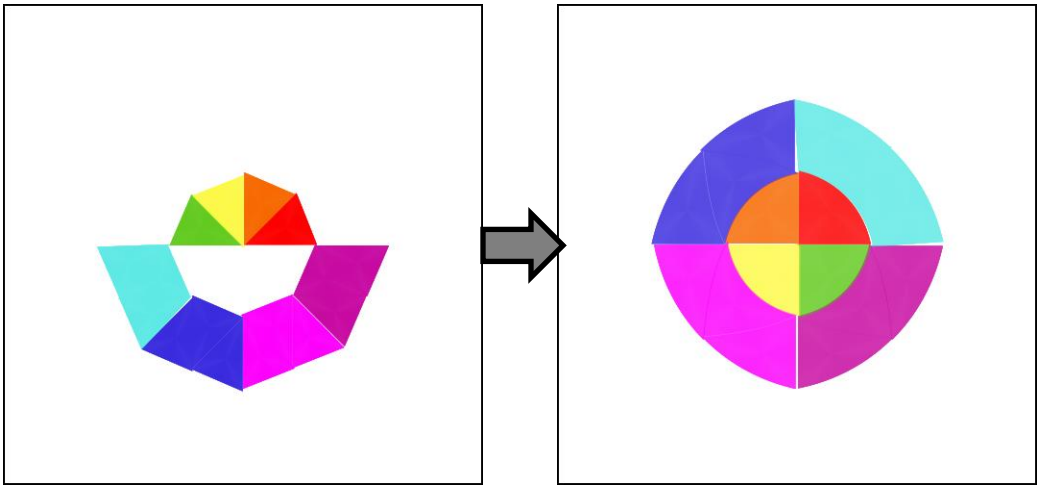


horseshoe (2D)

2.09	7X15B	7X16	jwf	ch
yes	yes	yes	yes	yes

Takes the top half of the plane and stretches it around from the left side to the right, and takes the bottom half and stretches it around the opposite way. The two halves normally overlap, although the contrived example leaves blank space so the effect on both halves can be seen.

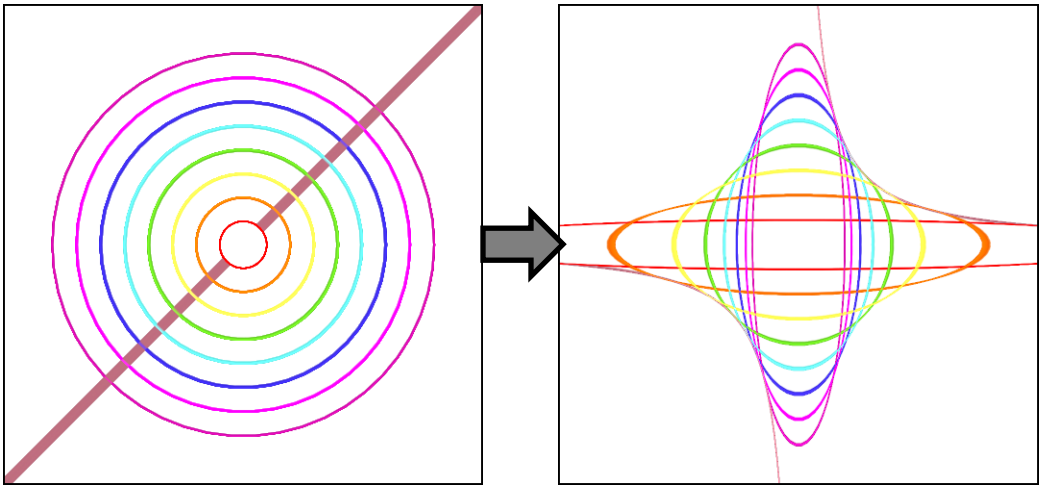
The name horseshoe may come because it transforms straight lines into U shapes.



hyperbolic (2D)

2.09	7X15B	7X16	jwf	ch
yes	yes	yes	yes	yes

Transforms circles centered at the origin to ellipses so the plane is mapped between hyperbolas. The y-coordinate and quadrant of the original point are preserved. The main diagonals (gray in the example) map to the boundary hyperbolas.



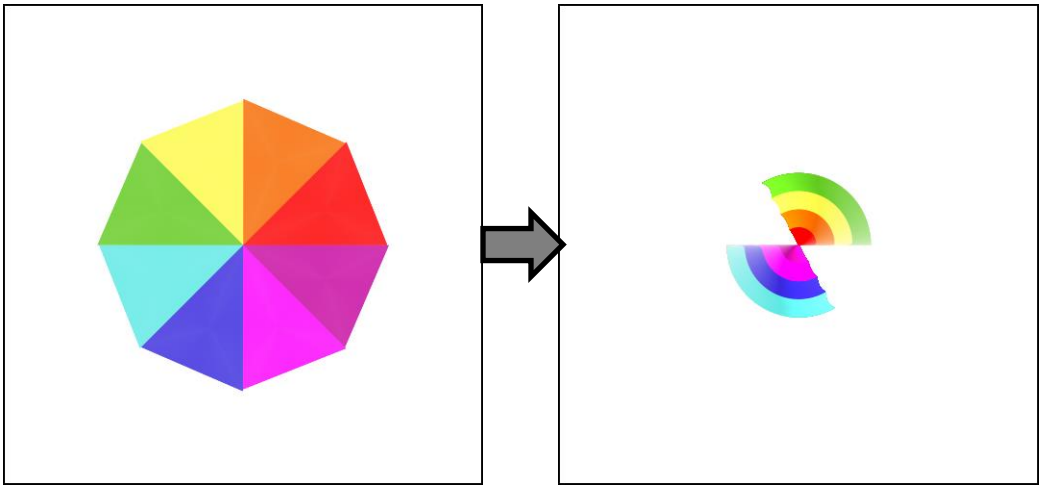
idisc (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	no	yes

idisc.dll

Morphs the plane into a unit circle by turning wedges into arcs. Unlike disc, this one doesn't overlap. Wedges in the top and bottom halves map to counter-clockwise arcs in the corresponding half of the result.

Compare disc, wdisc.

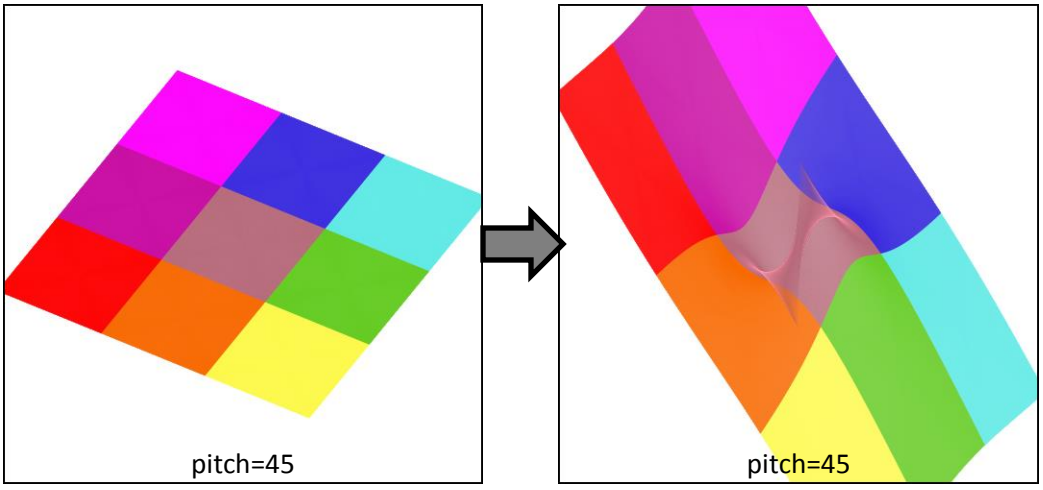


inflateZ_1 (3D, transforms z only)

2.09	7X15B	7X16	jwf	ch
no	dll	dll	yes	no

inflateZ_1.dll

Sets z based mostly on y, but with a sort of saddle shape in the center. The example is rotated 60° right (yaw=60) to show the profile.

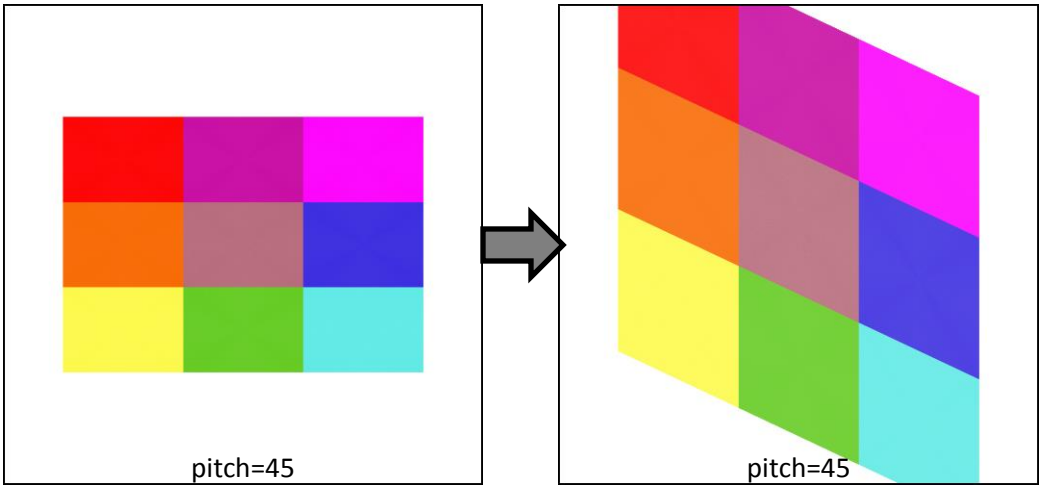


inflateZ_2 (3D, transforms z only)

2.09	7X15B	7X16	jwf	ch
no	dll	dll	yes	no

inflateZ_2.dll

Tilts the plane, bringing the top left up and the bottom right down. But as with all of the inflateZ variations, only z is set; linear is used in these examples to pass x and y.

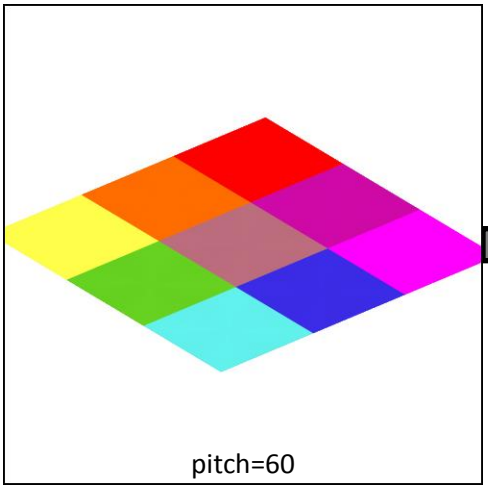


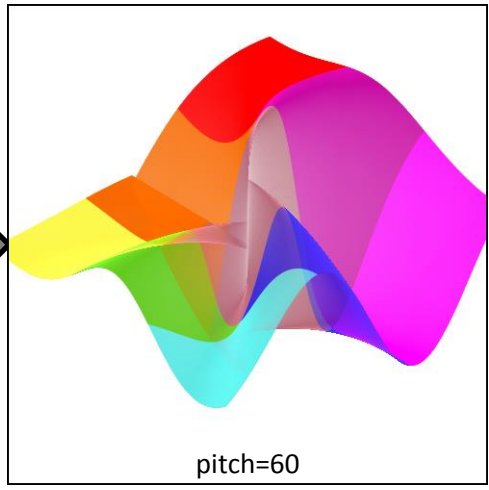
inflateZ_3 (3D, transforms z only)

2.09	7X15B	7X16	jwf	ch
no	dll	dll	yes	no

inflateZ_3.dll

Warps z to give a strong 3-dimensional shape. The example is rotated 50° left (yaw = -50).



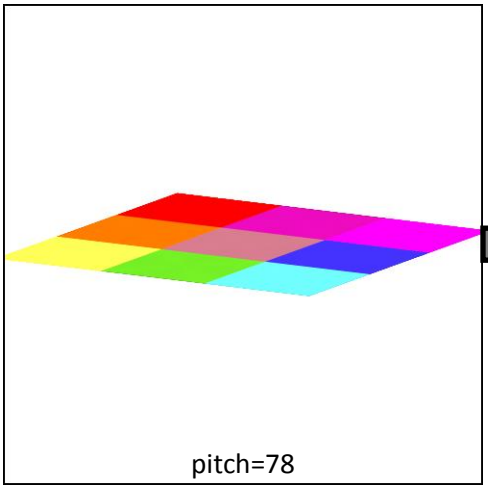


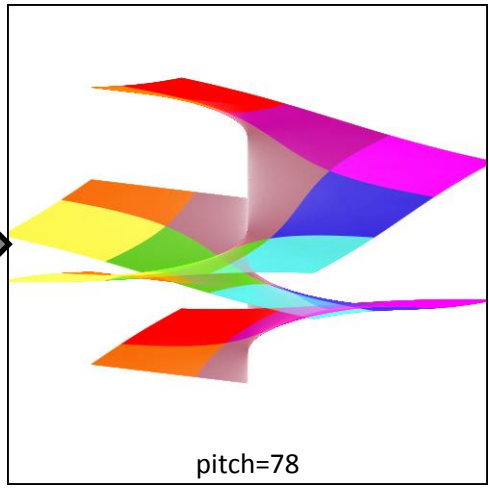
inflateZ_4 (3D, transforms z only)

2.09	7X15B	7X16	jwf	ch
no	dll	dll	yes	no

inflateZ_4.dll

Duplicates the plane and makes two interleaved helix shapes. On one, the top left is raised a lot and the bottom left is lowered a bit; on the other, the bottom left is raised a bit and the top left is lowered a lot. The example is rotated 30° left (yaw = -30).



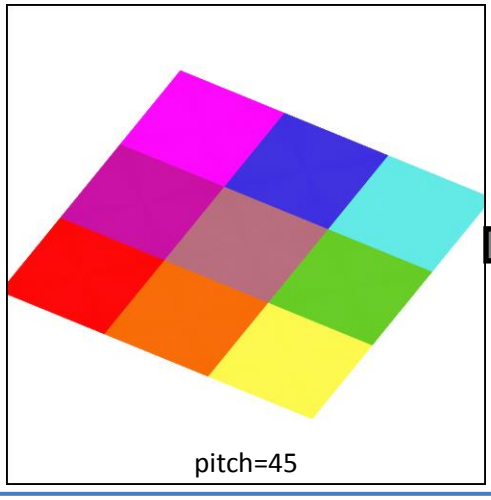


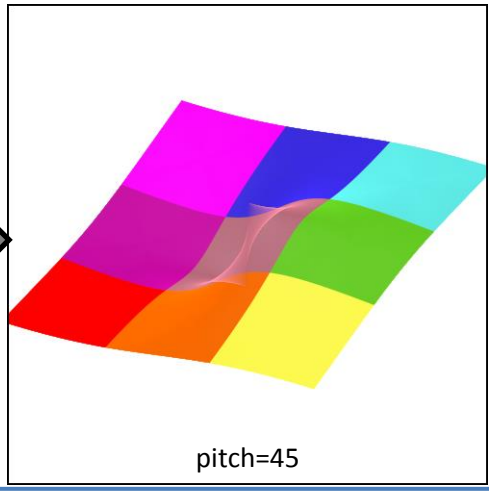
inflateZ_5 (3D, transforms z only)

2.09	7X15B	7X16	jwf	ch
no	dll	dll	yes	no

inflateZ_5.dll

Sets z to give a gentle 3-dimensional wave shape. The example is rotated 60° right (yaw=60) to show the profile.



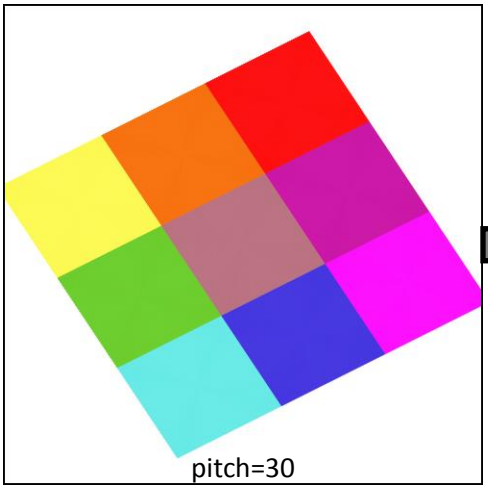


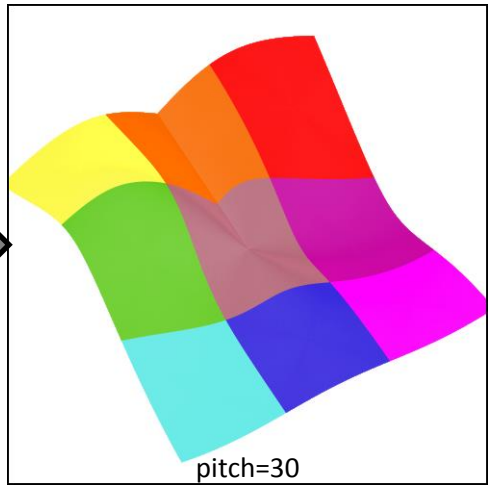
inflateZ_6 (3D, transforms z only)

2.09	7X15B	7X16	jwf	ch
no	dll	dll	yes	no

inflateZ_6.dll

Sets z to give a rolling shape with a fold along the negative x-axis. The example is rotated 60° left (yaw = -60) to show the profile.





julia3D (3D, transforms z)

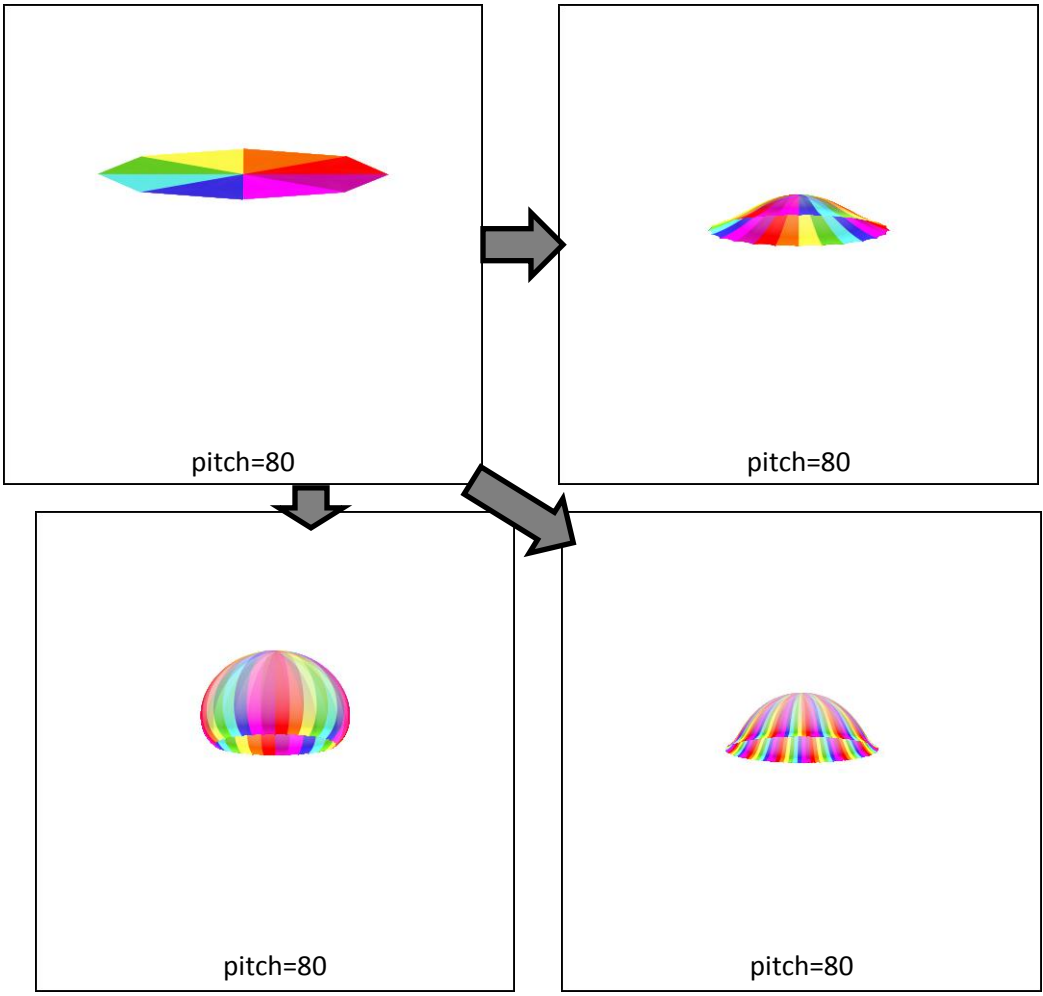
2.09	7X15B	7X16	jwf	ch
no	yes	yes	yes	no

Works like julian (with *julian_dist* = 1), but maps the result to a 3D shape, scaled by z. When *power* is positive, the middle part is stretched in and up, forming a bowl shape. When viewed face on (pitch =0), it is similar to julian but the middle is filled in. When *power* is negative, the outside edge is stretched up and in, forming a dome shape with a turned-in lip. When viewed face on, it is very different from julian since the middle is filled in.

The 3D shape is scaled by z, which is 1 in the examples here. When the z value is negative, the bowl is flipped from what is shown here. If z is 0 throughout the original, then julia3D is exactly the same as julian.

Top right: *power* = 3
Bottom right: *power* = 13
Bottom left: *power* = -4

A related variation, **julia3D_fl**, allows non-integer values for *power*.



julia3Dz (3D, transforms z)

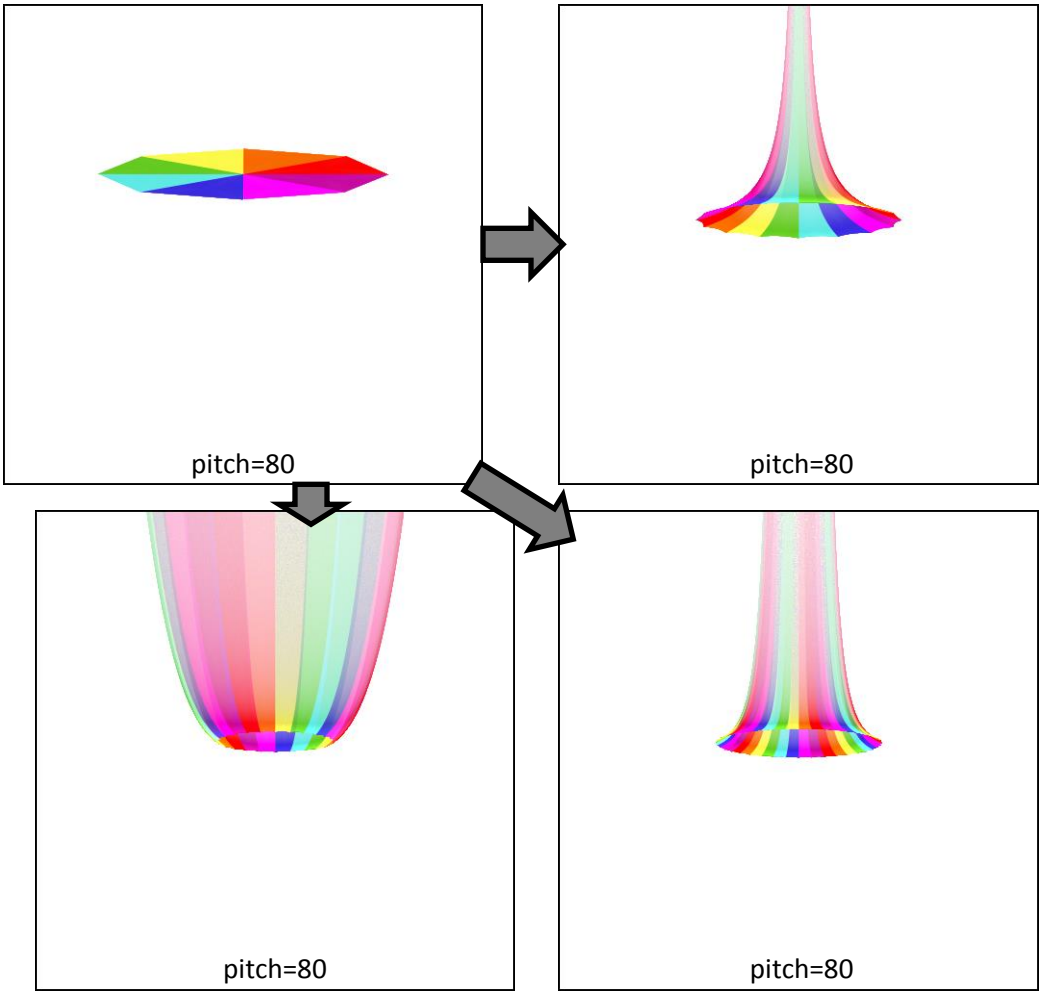
2.09	7X15B	7X16	jwf	ch
no	yes	yes	yes	no

Works like julian (with *julian_dist* = 1). But it also scales z to map to a horn shape. When *power* is larger, the "barrel" of the horn gets wider. When *power* is negative, the fractal is turned inside-out. When pitch is 0, it appears the same as julian.

The original shown here has z set to 1. Larger values make it larger. Negative values make the shape point down. When z is 0, it stays 0.

Top right: *power* = 2
Bottom right: *power* = 5
Bottom left: *power* = -3

A related variation, **julia3Dz_fl**, allows non-integer values for *power*.



Juliac (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	dll

Juliac.dll

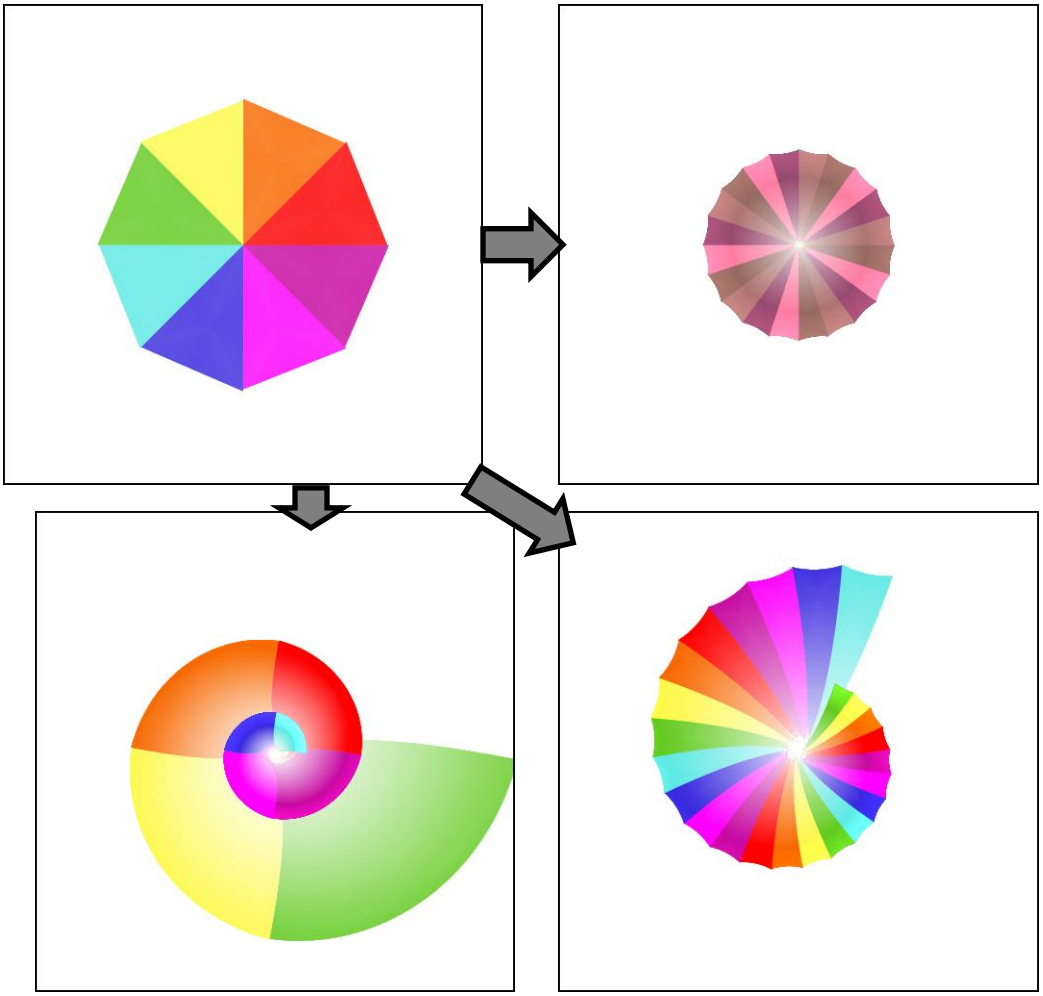
Like julian, but the *julian_power* variable is a complex number with real and imaginary parts (variables *re* and *im*) *Re* can't be zero; it doesn't need to be an integer, but the repetitions will overlap if it isn't, as shown in the top right example.

When *im* is not zero, the result will have a spiral shape, as shown in the bottom examples.

Top right: *re* = 2.5, *im* = 0, *dist* = 1
Bottom right: *re* = 3, *im* = -5, *dist* = 1
Bottom left: *re* = 0.5, *im* = 33, *dist* = 0.125

The math here is similar to cpow with *cpow_r* = 1/*re*, *cpow_i* = *im*/100, and *cpow_power* = 1, except that cpow doesn't add repetitions or have a *dist* variable.

juliac works the same as horseshoe when *re* = 0.5, *im* = 0, and *dist* = 0.5.



juliacomplex (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	no	dll

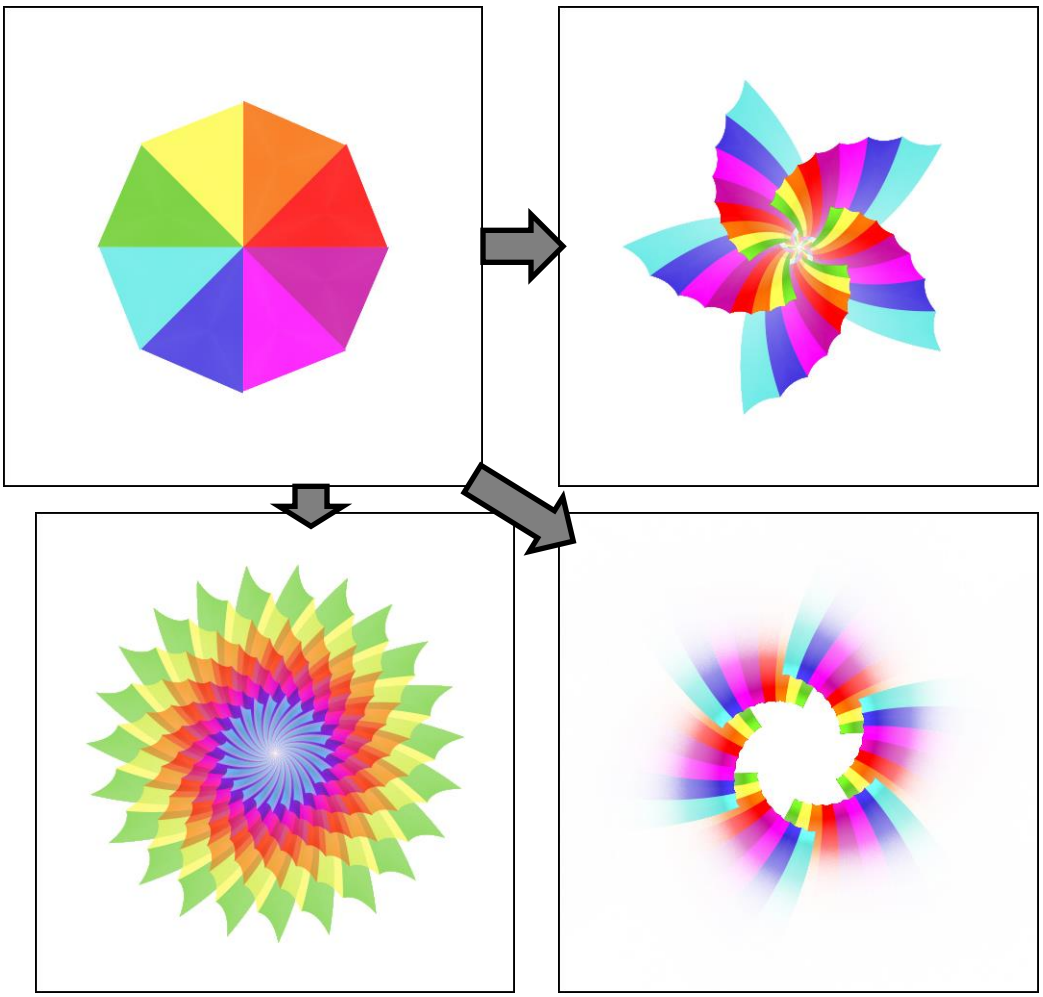
juliacomplex.dll

Similar to Juliac, but the spiral effect from *Im_p* is applied to each repetition, giving a pinwheel effect.

Top right: *Re_p* = 2.5 = 5/2, so there are five repetitions. *Im_p* = 2, so each one spirals from green in the center to cyan. *dist* = 1 so no distortion.

Bottom left: *Re_p* = 2.3 = 23/10, so there are 23 repetitions. *Im_p* = -3, so each one spirals from cyan in the center to green on the outside. This is difficult to see since most of it is overlapped by the next repetition. *dist* = 1, so no distortion.

Bottom right: *Re_p* = -6, so there are six reversed and inside-out repetitions. *Im_p* = 2.5, so each one spirals from green to cyan, but the first few colors are overlapped by the next repetition. *dist* = 1.5, making the result slightly larger, and actually causing the overlap in this case since *Re_p* is an integer.



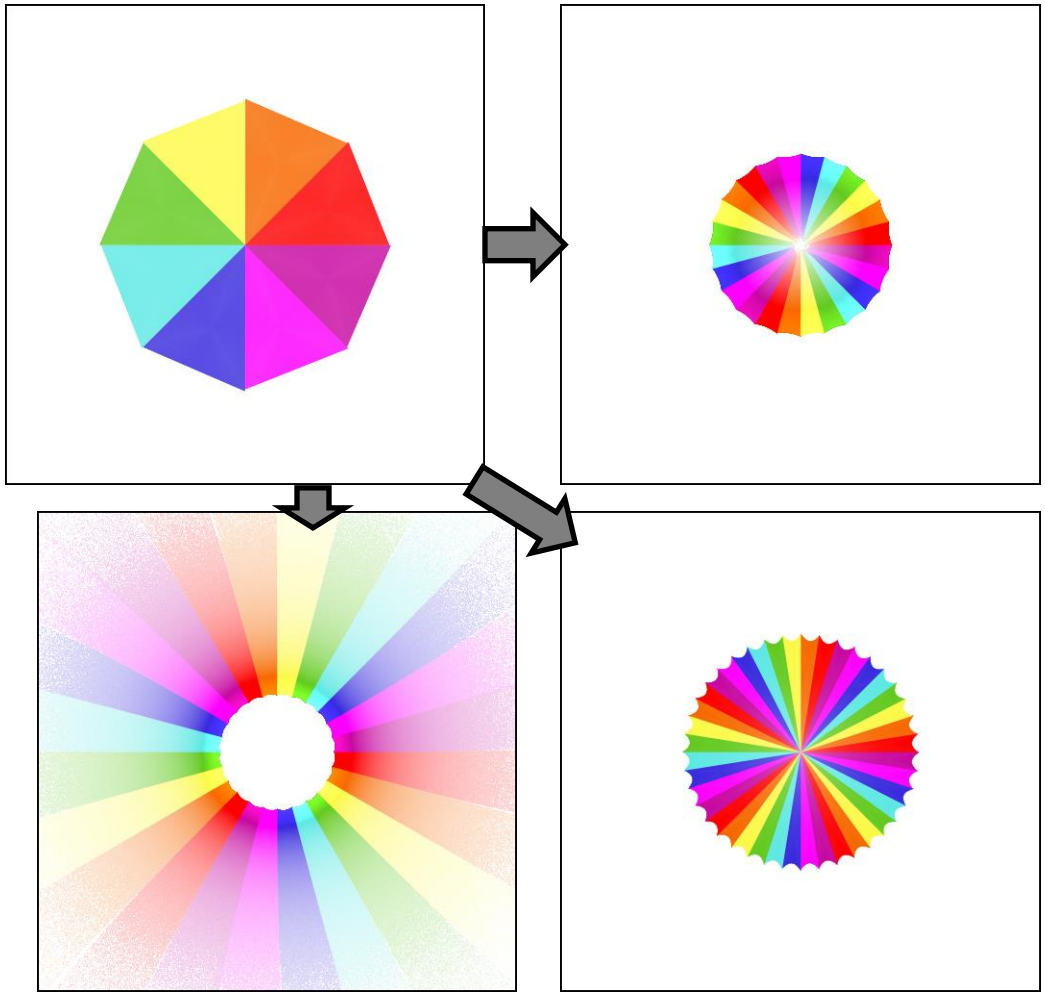
julian (2D)

2.09	7X15B	7X16	jwf	ch
yes	yes	yes	yes	yes

Cuts the plane along the negative x axis and squishes it around so it only takes up 1/*power* of the original space, then repeats that *power* times to fill up the gap. The top right example shows this with *power* = 3 and *dist* = 1. The result is also shrunk radially, making it smaller and opening a hole in the middle. The distortion variable *dist* can stretch it back out, filling in the hole but distorting the shape, as in the bottom right example with *power* = 5 and *dist* = 3.5. Values of *dist* less than one will shrink it further.

When *dist* is negative, the result is turned inside-out. When *power* is negative, the result is both turned inside-out and reflected across the x axis (see the bottom left example, with *power* = -3 and *dist* = 1). When both are negative, the result is just reflected.

Supercedes **julia**, which is julian with *power* = 2 and *dist* = 1 except that most versions swap x and y. Variation **julian_fl** allows non-integer values for *power*.

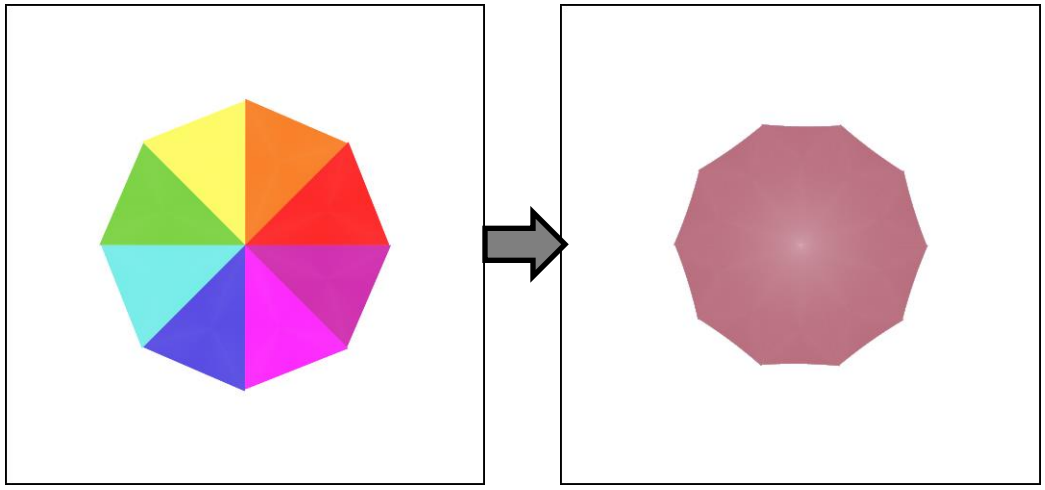


juliaq (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	dll

juliaq.dll
Divides the plane into *divisor* wedges and repeats each of them *power* times around the origin, thus converting a shape with *divisor* edges into one with *power* edges. It is the same as julian when *divisor* = 1. The example has *power* = 10 and *divisor* = 8.

3D and post_ versions ara also available (**julia3Dq** has an effect similar to julia3D).

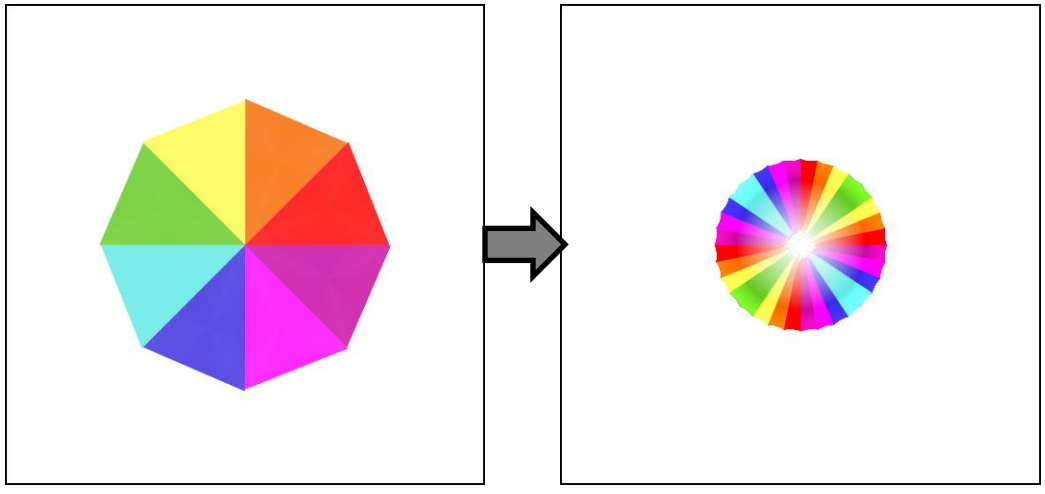


juliascope (2D)

2.09	7X15B	7X16	jwf	ch
yes	yes	yes	yes	yes

Similar to julian, except that the repetitions alternate direction. When *power* is even, the alternate repetitions match (as shown here; note how the green and cyan wedges are doubled); otherwise one repetition will not match exactly.

The example has *power* = 4 and *dist* = 1

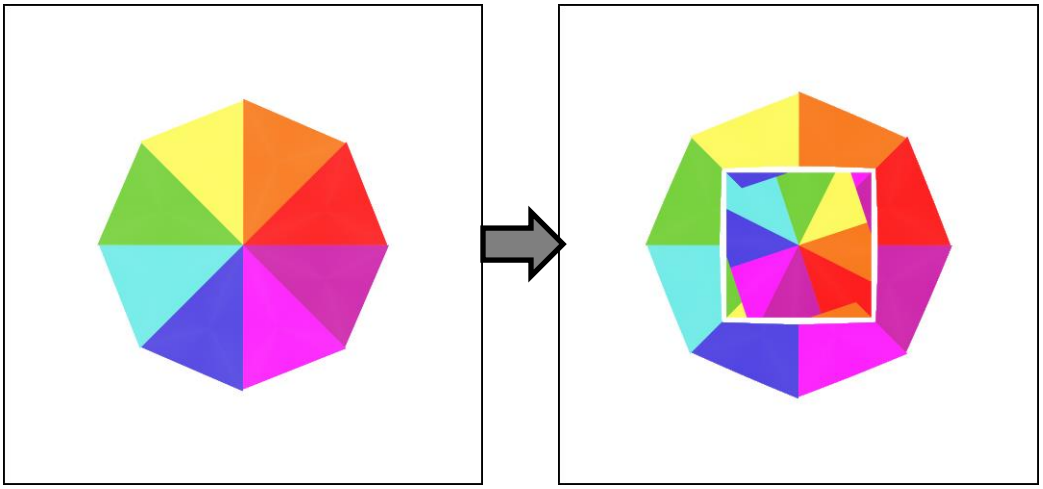


lazyjess (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	dll

lazyjess.dll
Similar to lazysusan, except a regular polygon with *n* sides is used instead of a circle. If *spin* is not set to make the corners line up exactly, they will be cut off; *corner* controls the appearance of the uncovered space (it takes values 1 to *n*).

The example uses *n* = 4, *spin* = 1.25, *space* = 0.1, and *corner* = 1



lazysusan (2D)

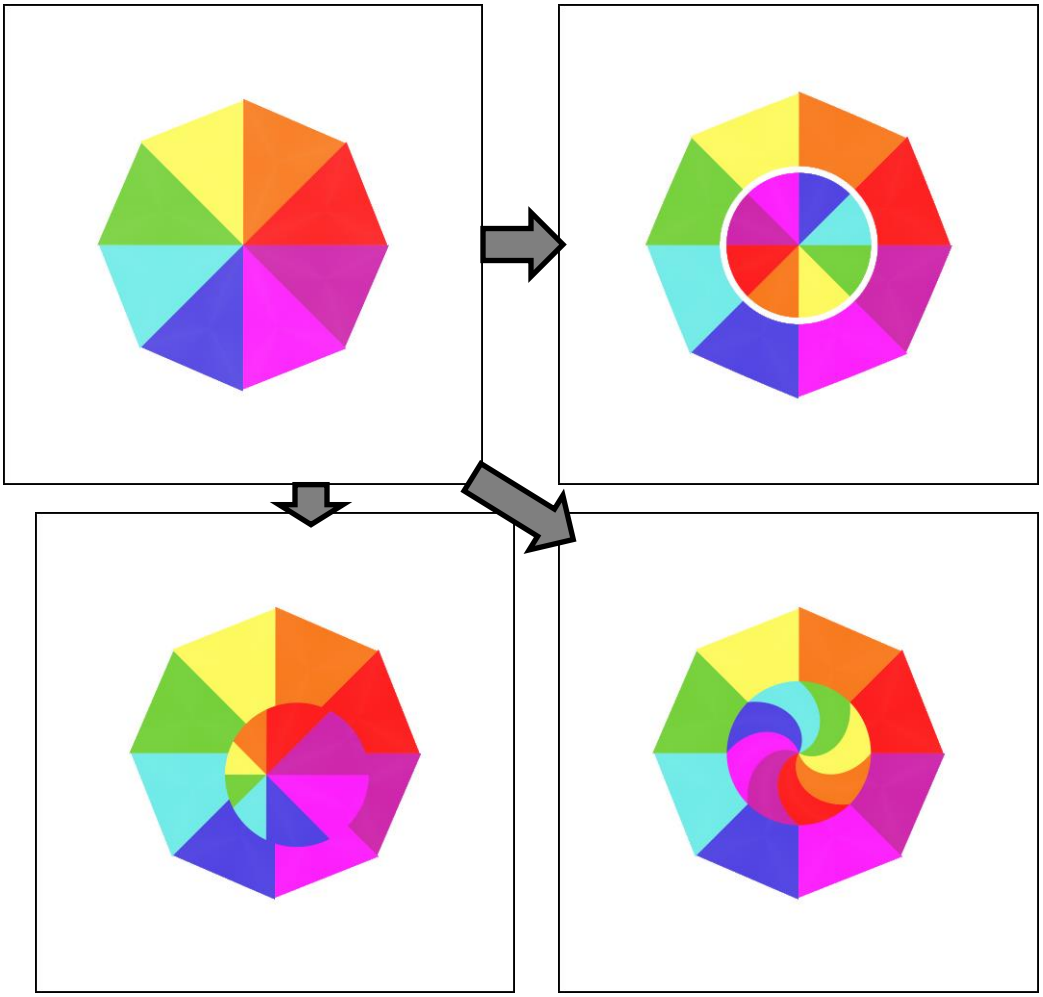
2.09	7X15B	7X16	jwf	ch
dll	dll	yes	yes	yes

lazysusan.dll

Twists and turns a circle with a radius of the variation value. The center is set by the variables *x* and *y*. It is turned clockwise by *spin* radians (0 is no turn, π is halfway around, 180°), and twisted according to *twist*.

When *space* is positive, the rest of the plane is expanded to leave a space around the circle. When negative, the rest of the plane is shrunk, causing overlap at the boundary.

Top right: *spin* = 3.14159, *space* = 0.1, *twist* = 0
Bottom right: *spin* = 1.5708, *space* = 0, *twist* = 1.25
Bottom left: *spin* = 5.5, *space* = 0, *twist* = 0, *x* = 0.3, *y* = -0.3

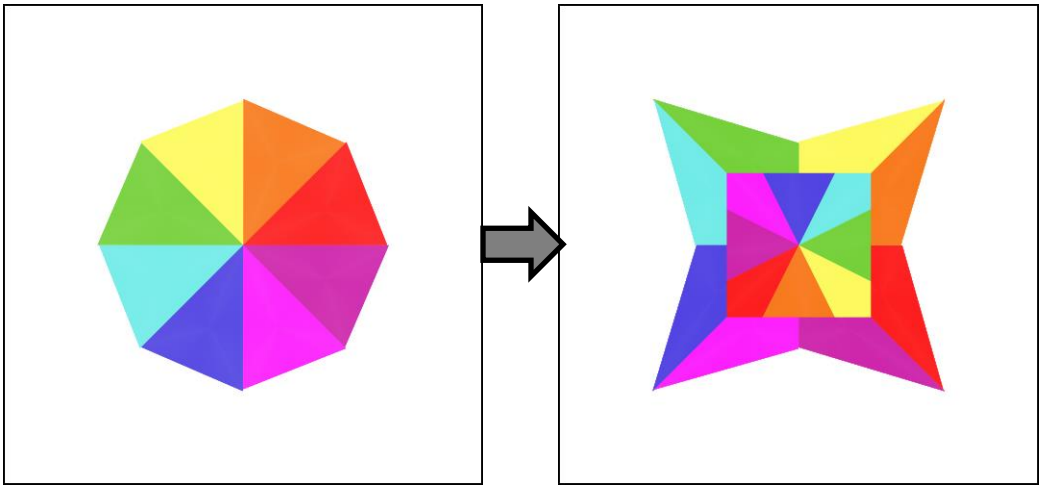


lazyTravis (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	dll

LazyTravis.dll
Spins the square centered at the origin with size 1 according to *spin_in*, which ranges from 0 (no spin) to 2 (spins all the way around). Also spins the rest of the plane according to *spin_out*, deforming it to make it fit a square..

The example uses *spin_in* = 0.875, *spin_out* = 0.25, *space* = 0



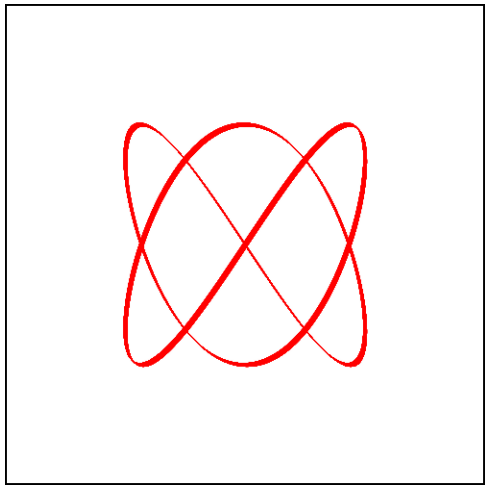
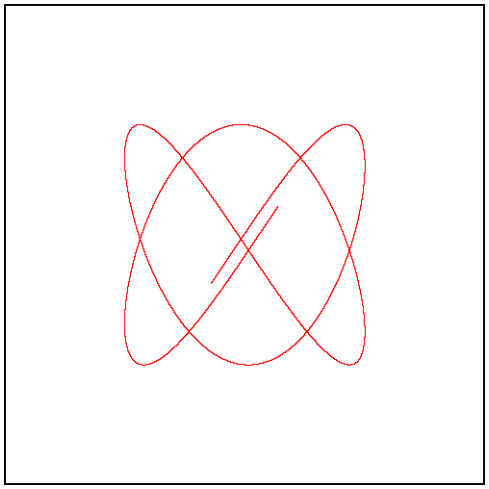
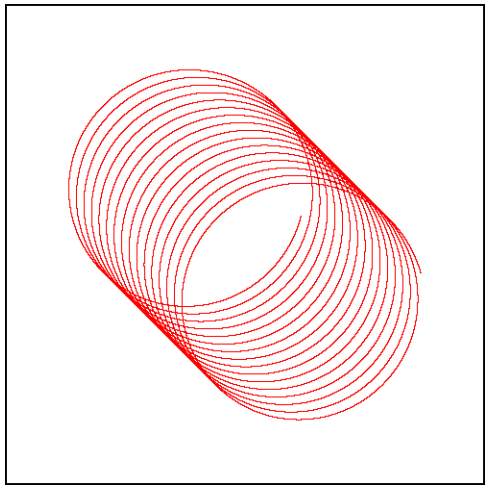
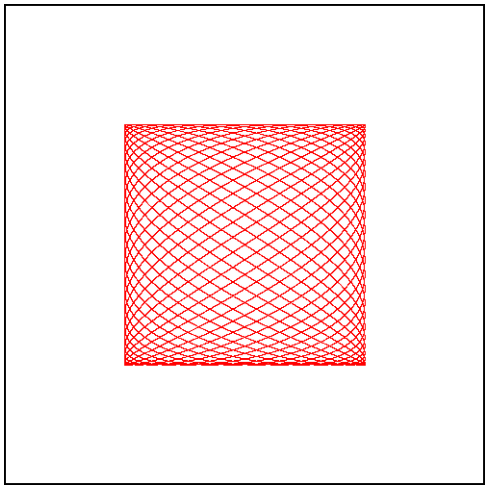
Lissajous (2D blur)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	no

LissajousVariationPlugin.dll
Generates a Lissajous curve (named after a 19th century French mathematician who studied them). The classic Lissajous shape is controlled by variables *a* and *b*, or more precisely the ratio between them. Variable *d* controls the phase difference between them (-π to π), and *e* is the thickness of the line (keep small for best results). Variable *c* adds diagonal movement to the result, as shown in the bottom left example, which would be a circle if *c* was 0 (*a* and *b* equal, with phase π/2).

Variables *tmin* and *tmax* control the extent of the generated line.

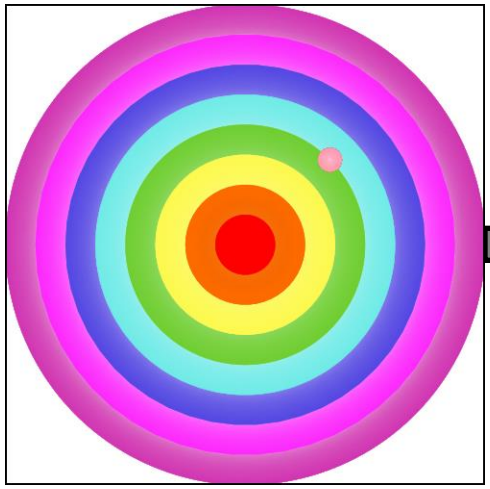
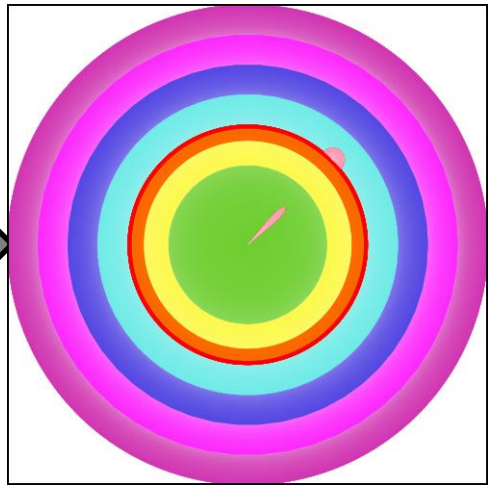
Top left: *tmin* = -3.14159, *tmax* = 3.14159, *a* = 2, *b* = 3, *c* = 0, *d* = 0, *e* = 0.03
Top right: *tmin* = -3.25, *tmax* = 3.25, *a*= 2.02, *b* = 3, *c* = 0, *d* = 0, *e* = 0
Bottom right: *tmin* = -30, *tmax* = 30, *a* = 3.125, *b* = 2, *c* = 0, *d* = 0, *e* = 0
Bottom left: *tmin* = -25, *tmax* = 25, *a* = 2, *b* = 2, *c* = 0.02, *d* = 1.57, *e* = 0

loonie (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	yes	yes	yes

loonie.dll
Turns the center of the plane inside-out, distorting it. Areas are preserved; in the example, the area of the green ring before is the same as the area of the green circle after. A circle has been added straddling the boundary to show the distortion effect. The radius of the loonie effect is the value of the transform (1 here).

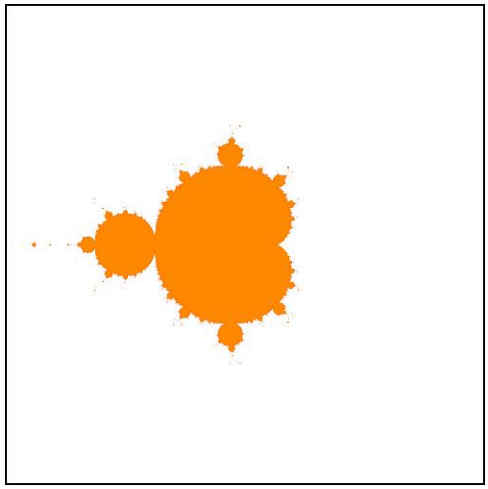
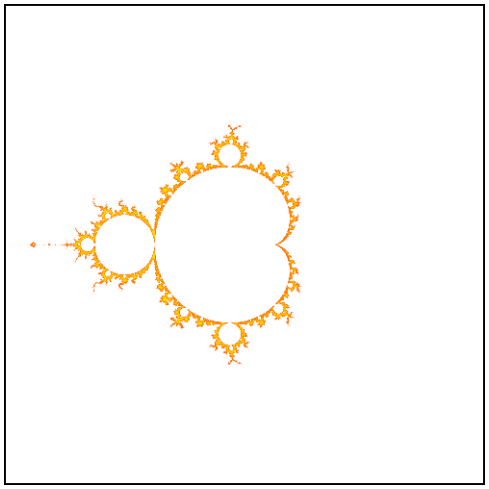



Mandelbrot (2D blur)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	no

MandelbrotVariationPlugin.dll

Generates the shape of the iconic Mandelbrot set. Note that Apophysis is not well suited for exploring that set; use an escape-time fractal program with extended precision and better coloring algorithms for that.

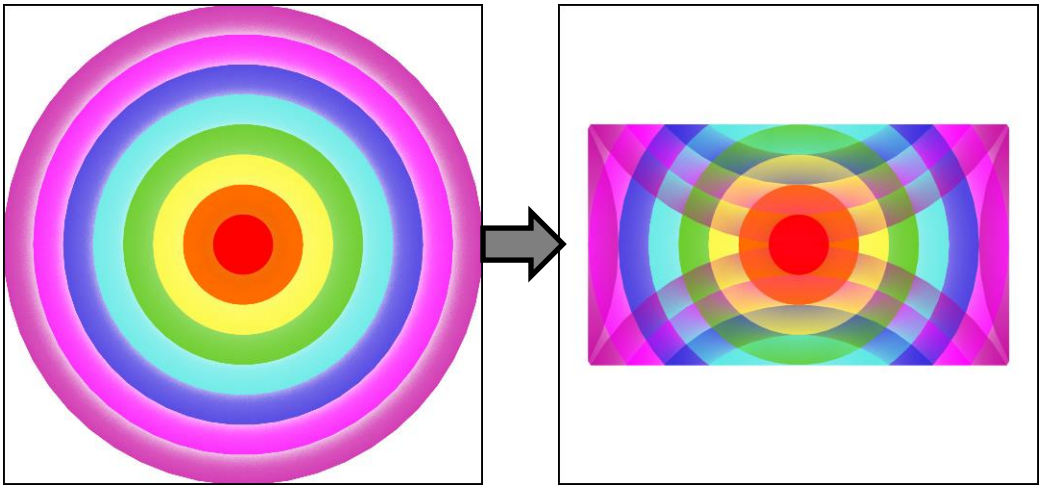
modulus (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	yes

modulus.dll

Divides the plane into rectangles, then stacks all of the rectangles on the center.

For the example, *x* = 1.75 and *y* = 1.



murl (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	dll

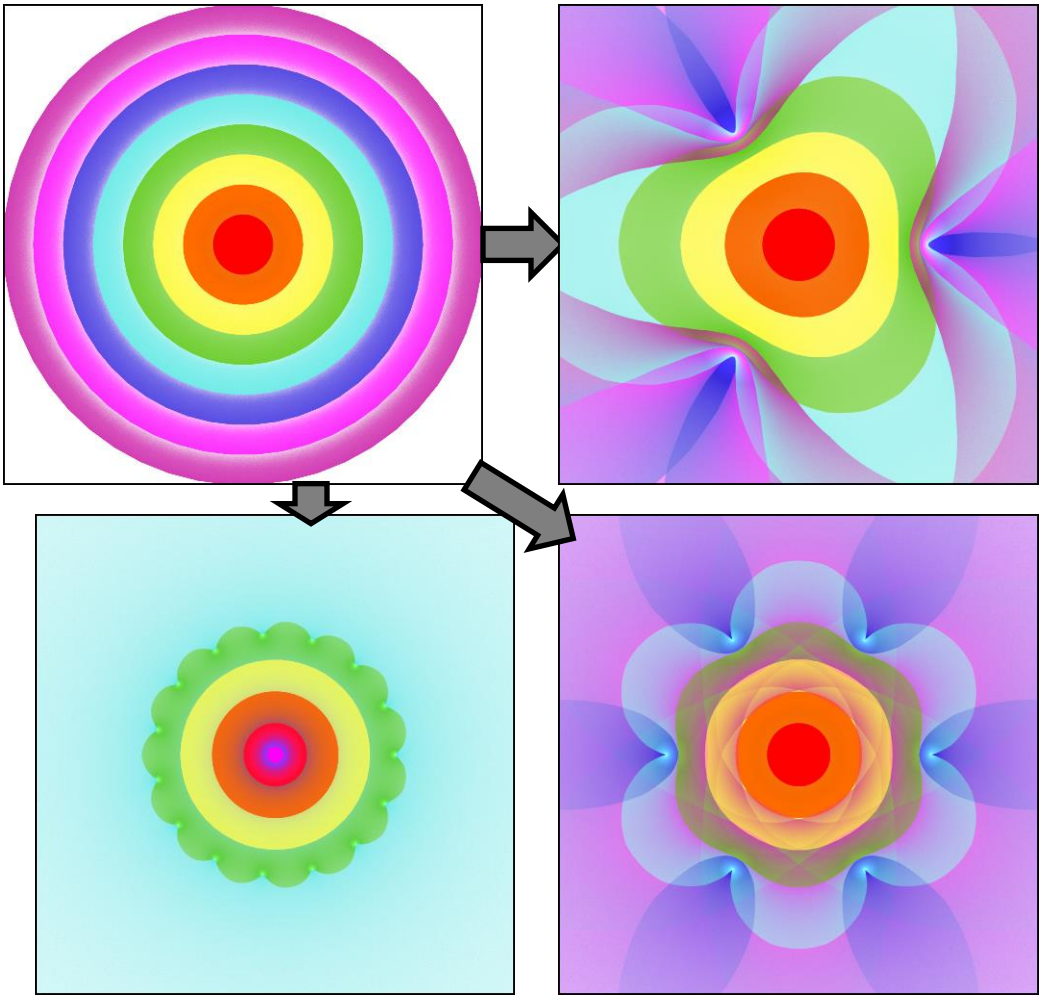
murl.dll
Based on curl. Think of the plane as a series of coincentric rings, divided into *power* lobes. As *c* increases, the spaces between the lobes pinch in and the lobes stretch out and rotate around until they cross and converge in the center.

Top right: *c* = 0.4 and *power* = 3

Bottom right: *c* = 0.25 and *power* = 6

Bottom left: *c* = 0.75 and *power* = 15

A post_murl version is also available.

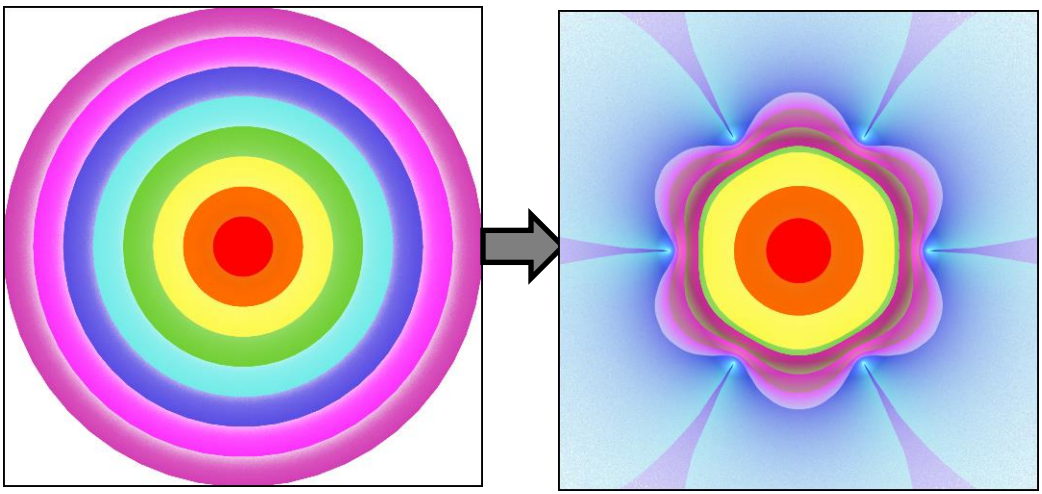


murl2 (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	dll

murl2.dll

A refinement of murl that creates less overlap with higher values of *power*. For the example, *c* = 0.25 and *power* = 6; compare with the bottom right example for murl right above it, which uses the same values.



nBlur (2D blur)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	dll

nBlur.dll

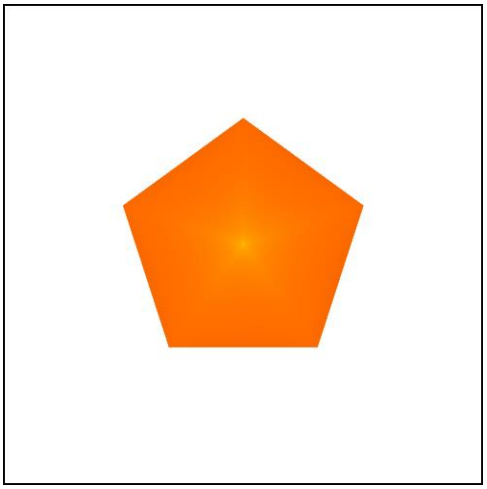
Generates regular polygons with quite a few variables to control various options such as stripes and a hole in the center. The examples show some of the possibilities; the particular variables each exemplifies are in bold.

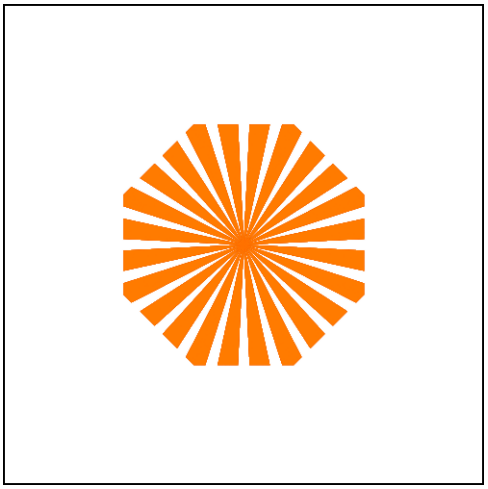
Top left: *numEdges* = 5, *numStripes* = 0, *ratioHole* = 0, *circumCircle* = 0, and ***equalBlur* = 0**

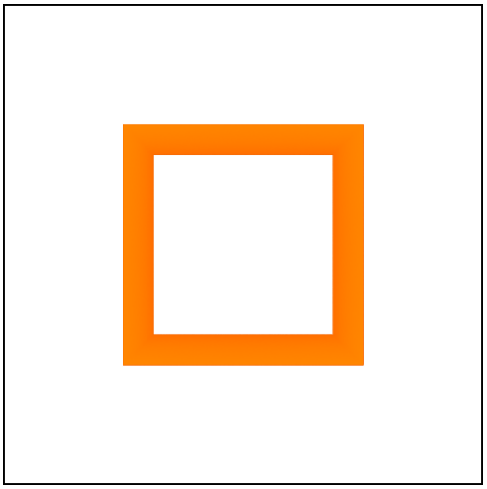
Top right: *numEdges* = 8, ***numStripes* = 3, *ratioStripes* = 1.25**, *ratioHole* = 0, *circumCircle* = 0, and *equalBlur* = 1

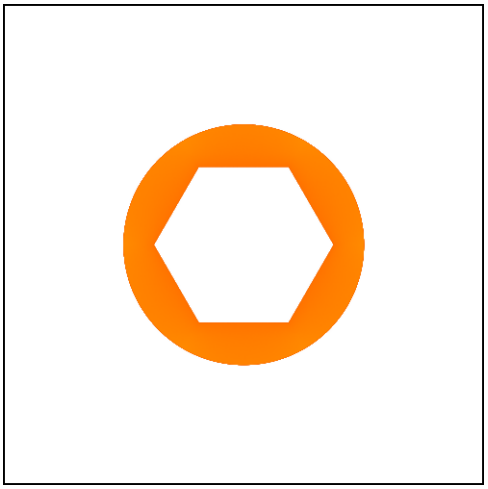
Bottom left: *numEdges* = 4, *numStripes* = 0, ***ratioHole* = 0.75**, *circumCircle* = 0, and *equalBlur* = 1

Bottom right: *numEdges* = 6, *numStripes* = 0, ***ratioHole* = 0.75, *circumCircle* = 1**, and *equalBlur* = 1









ngon (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	yes	yes	yes

ngon.dll

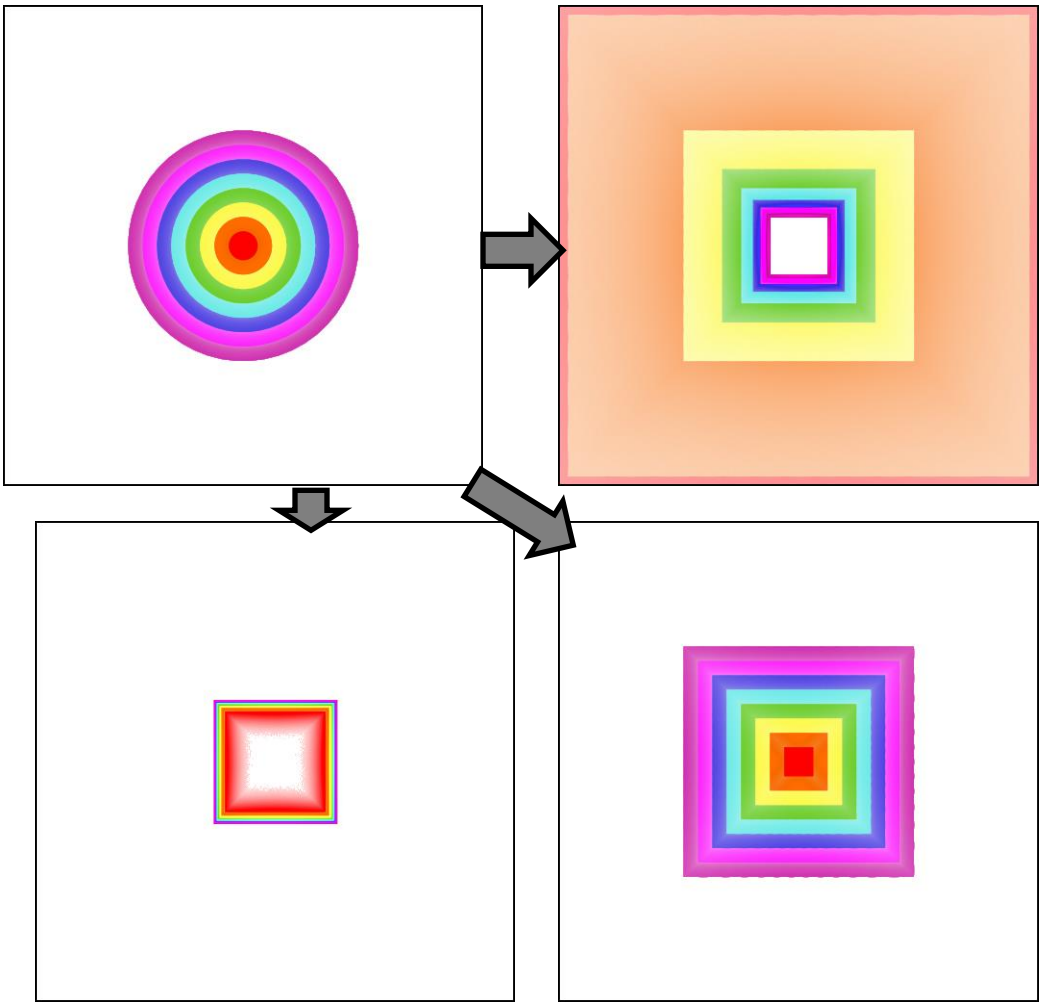
Stretches circles into polygons, with the option to turn them inside-out (like spherical).

Variables:
sides – Number of sides for the polygon. The examples here all use 4.

power – Controls the expansion of the plane. When 0 (bottom right), it is normal size. Decreasing it expands the outside and shrinks the middle (not shown). Increasing it does the opposite (bottom left, *power* = 0.9). When 1, it degenerates into an outline, and continuing turns the plane inside-out (when 2, top right, it is the same size as spherical).

circle – Rounds the sides of the polygon. When 1 (all examples here), the sides are straight.

corners – Defines the shape of the corners; 1 is normal (all examples here).

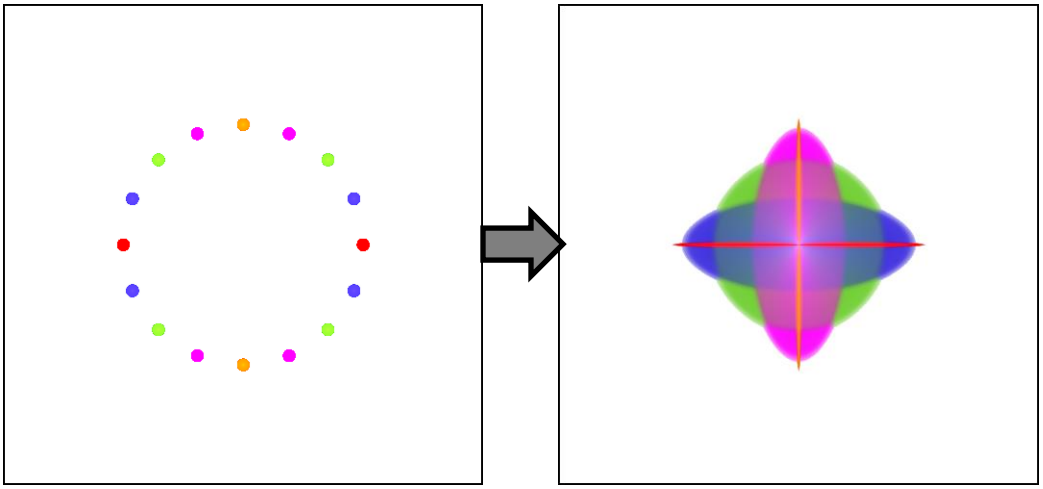


noise (2D half blur)

2.09	7X15B	7X16	jwf	ch
yes	yes	yes	yes	yes

Maps each point on the plane to an ellipse. Points on the main diagonals map to circles; points on the x and y axes map to horizontal or vertical line segments.

In the example, all the circles of the same color map to the same ellipse. For example the four green circles on the main diagonals map to the green circle in the result.



npolar (2D)

2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	dll

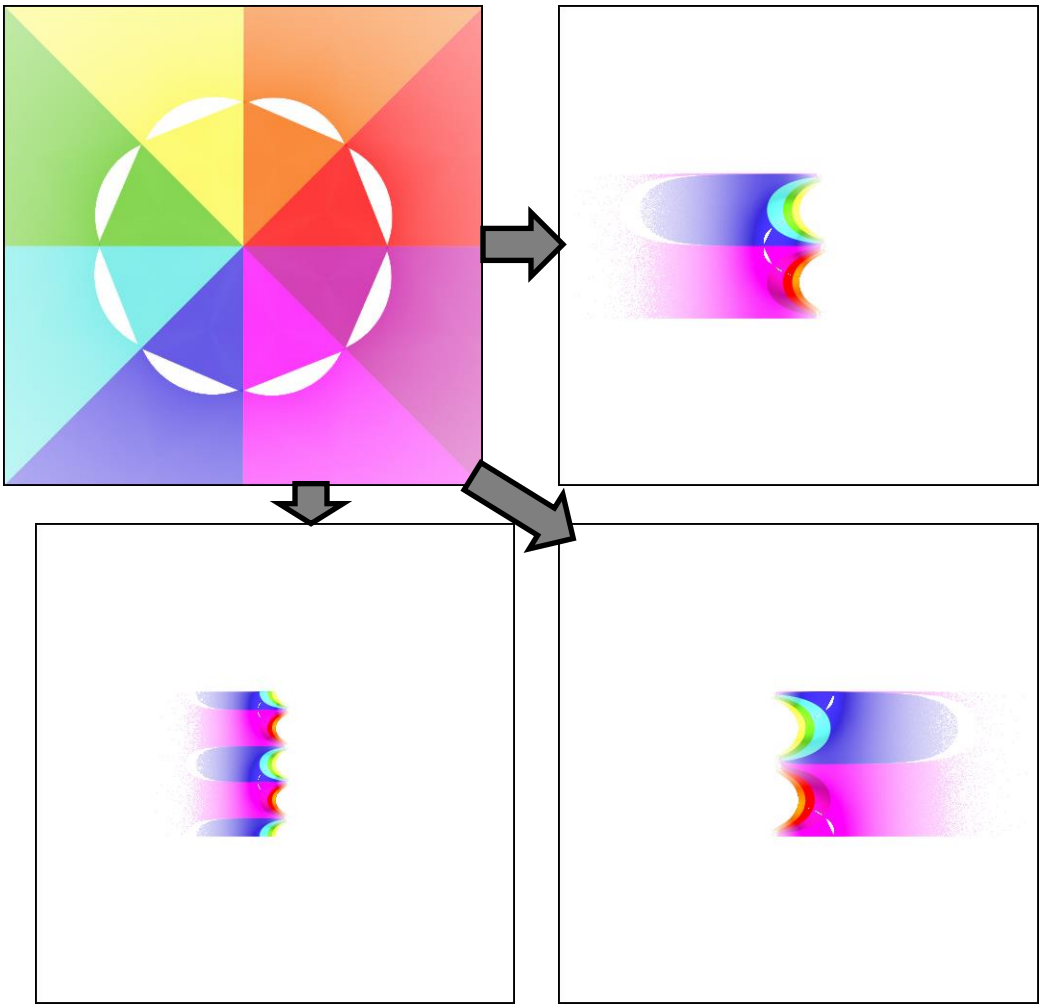
npolar.dll

When *parity* is 1, this is like julian (not shown). When 0, it does a polar conversion (setting x to the angle and y to the distance), followed by a julian followed by another polar conversion (this time setting x to the distance and y to the angle).

With *n* = 1 (top right), wedges are converted to arcs. The outer part of the original goes to the middle part of the result, and the left part of the original goes to the top part of the result.

With *n* = -1 (bottom right), the top and bottom halves of the result are mirrored both left to right and top to bottom.

With *n* = 2 (bottom left), the result is repeated twice and made smaller. Note how the outer unit is split between the top and bottom.



onion (3D, transforms z)

2.09	7X15B	7X16	jwf	ch
no	no	no	yes	no

Maps the plane to a 3D onion shape; a sphere with an exponential on top. The radius of the sphere is the value of the transform, and the pinch point in the result is twice that.

The two variables *centre_x* and *centre_y* set the center; the example has them both at 0 (the default).