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.

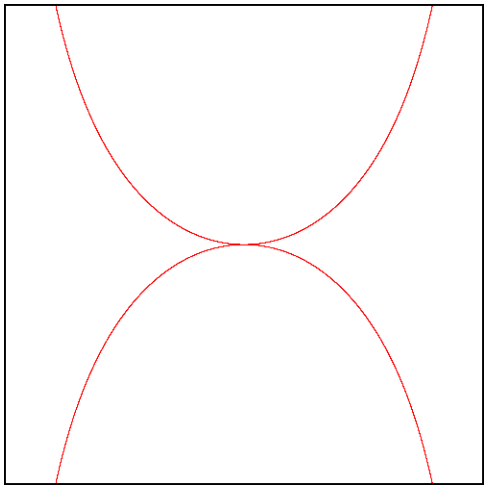
My primary purpose in assembling this guide was to help my own efforts in understanding the bewildering array of variations available for flame fractals. I can't promise I got everything correct, especially for the variations where no source code was available. I've learned a lot putting this together.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

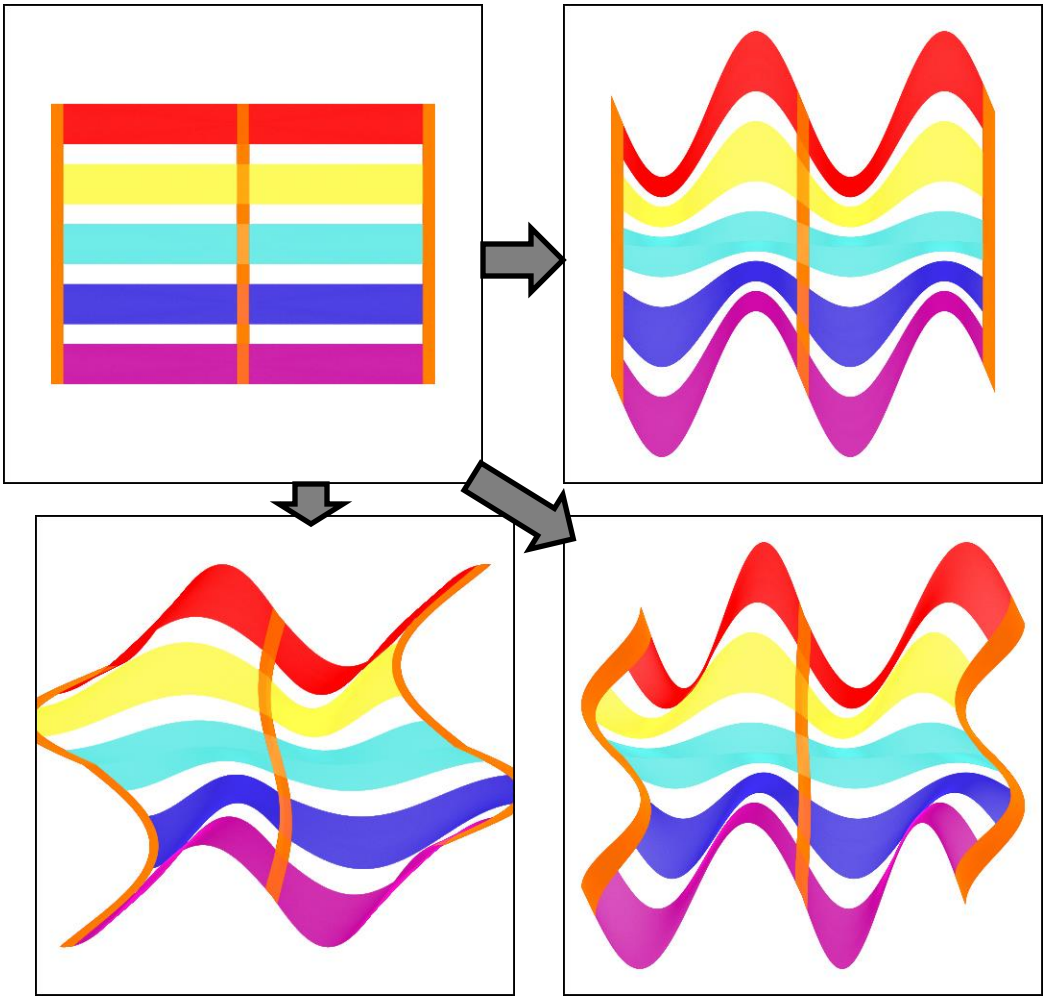
Top right: *sym* = 0 (so x is not affected), *weight* = 0.5, *freq* = 4, *scale* = 0.1

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.

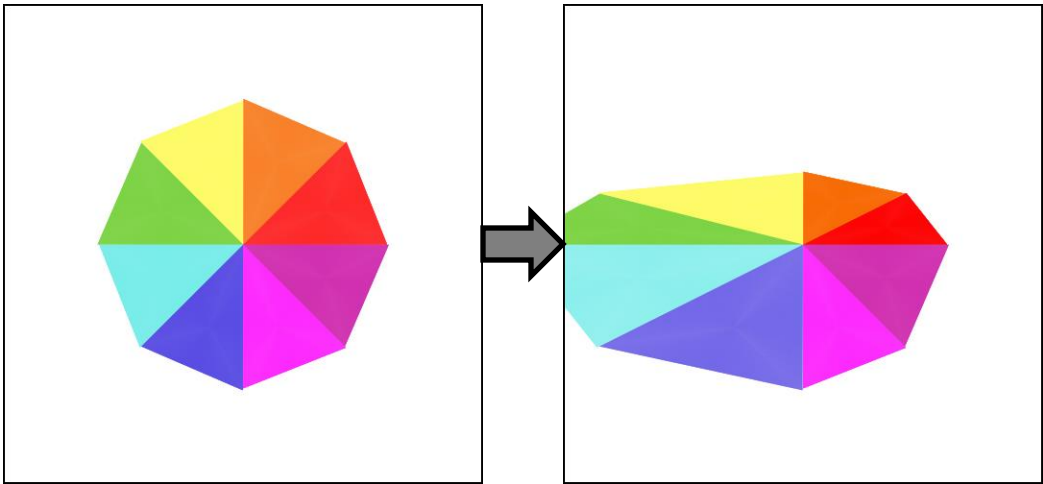


bent (2D)

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.

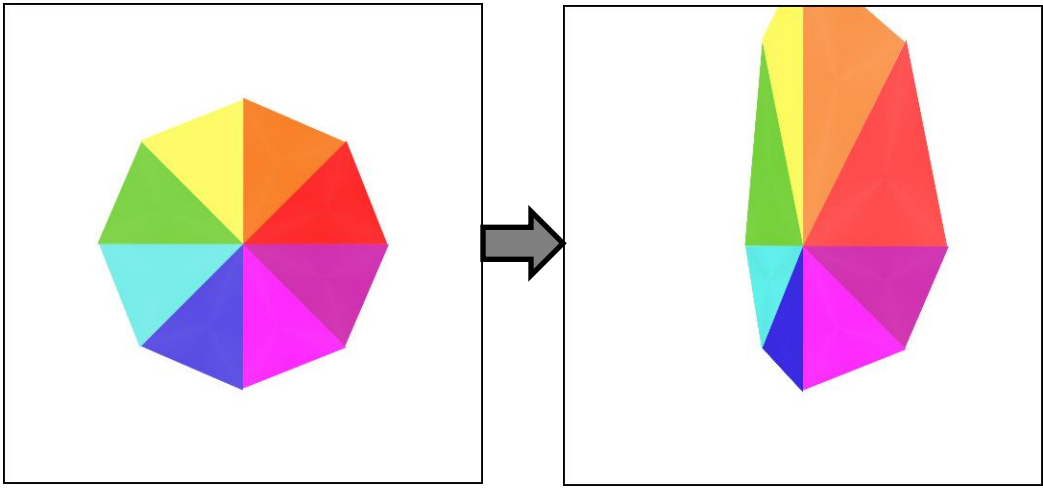


bent2 (2D)

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.

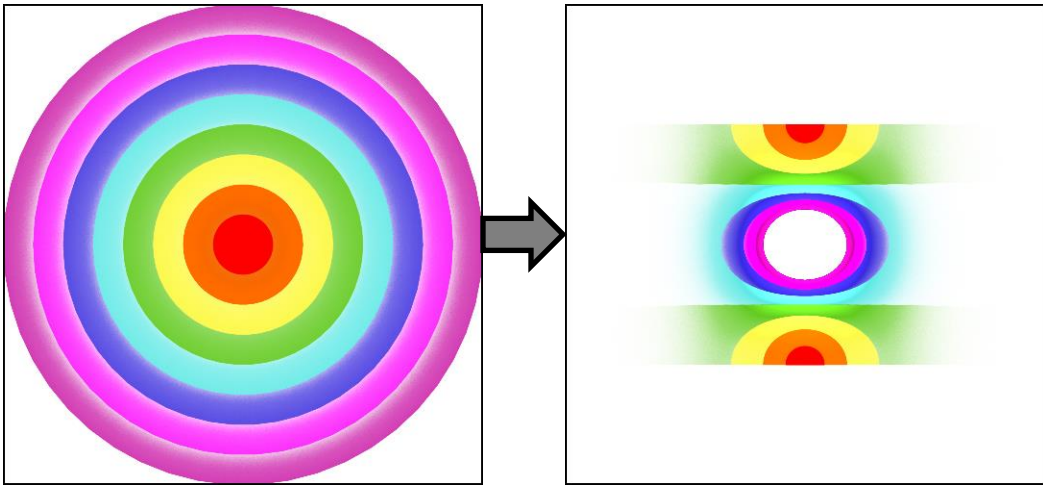


bipolar (2D)

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

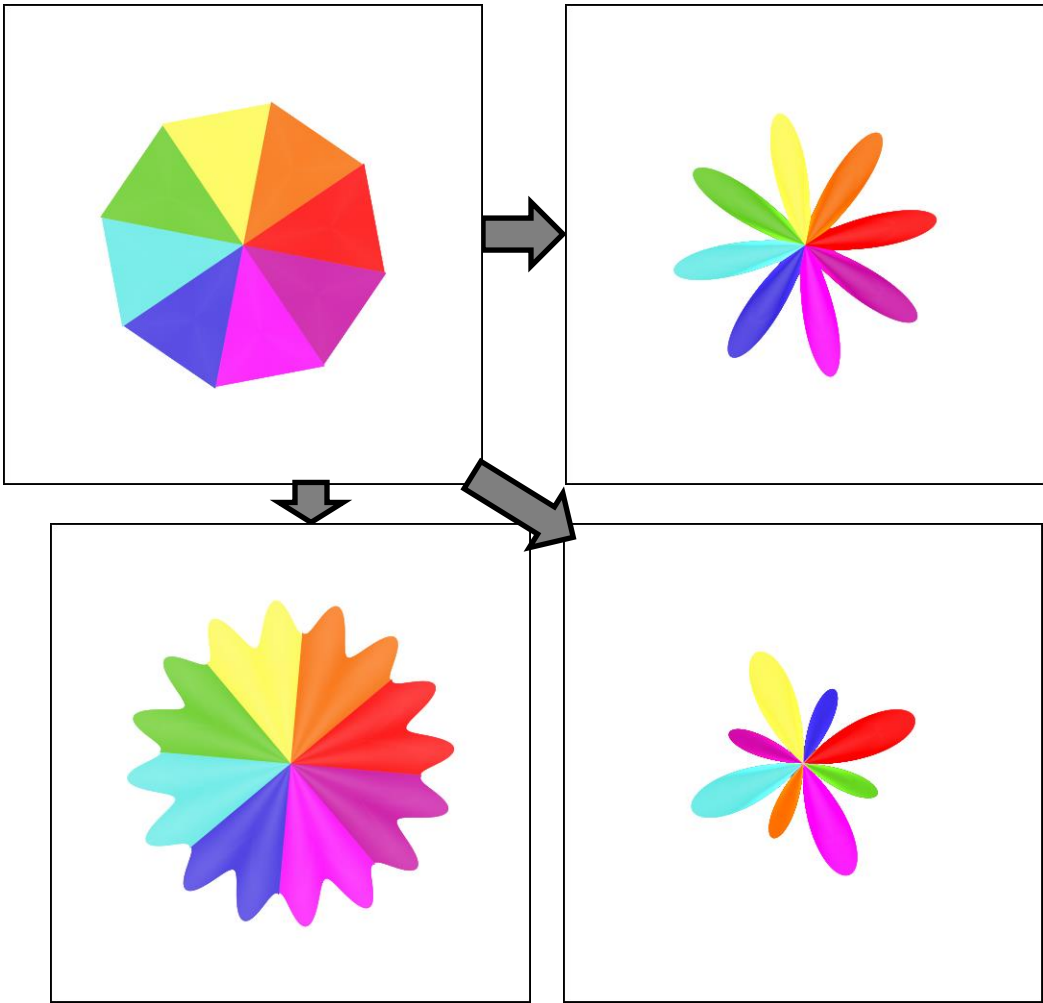
Pinches the plane according to the variables: *waves* is the number of pinches, and the height varies between *low* and *high*. The built-in version in Apo 2.09 requires an integer value for *waves*; the others do not.

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.

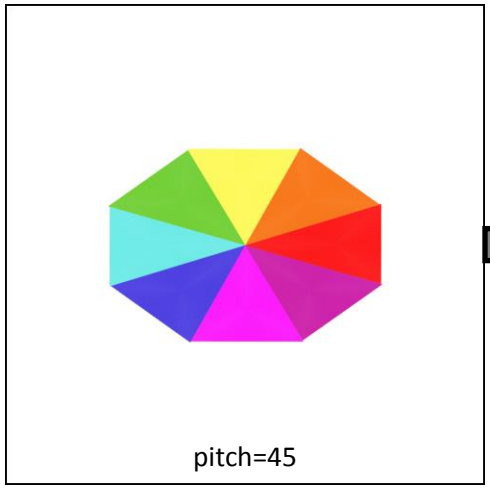


blob3D (3D, sets z)

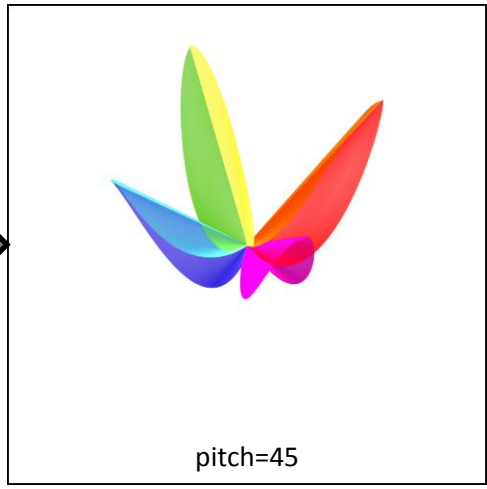
2.09	7X15B	7X16	jwf	ch
no	no	no	yes	no

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



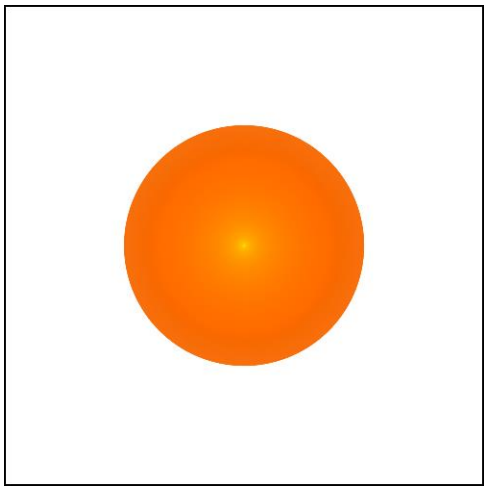
pitch=45

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)

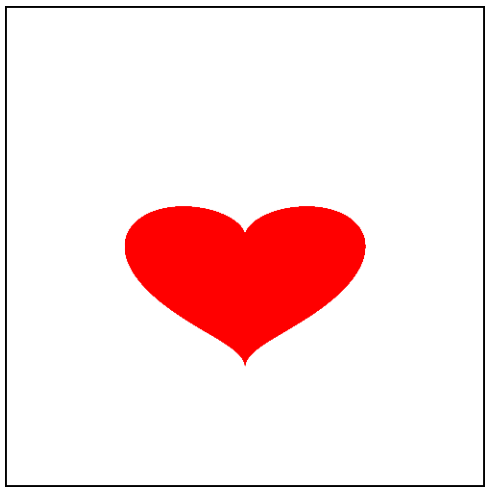
2.09	7X15B	7X16	jwf	ch
dll	dll	dll	no	dll

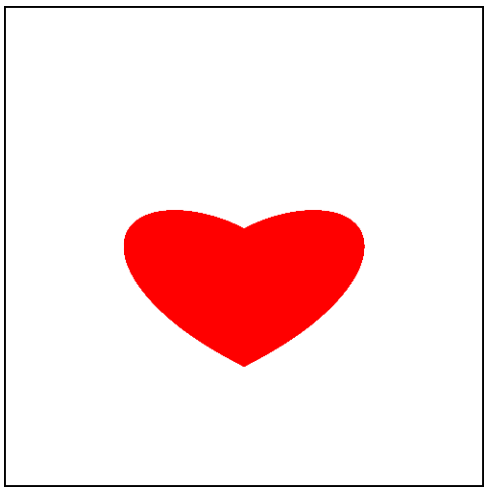
blur_heart.dll

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





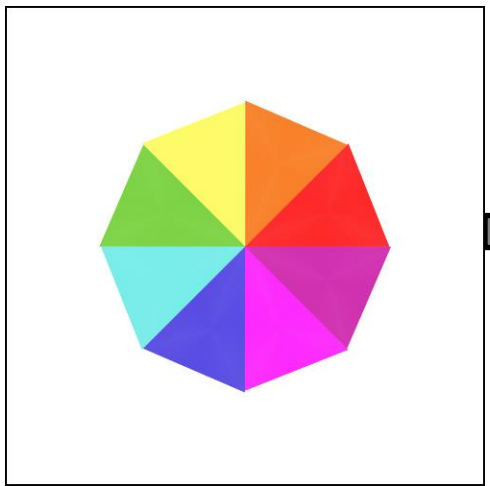
blur_linear (2D)

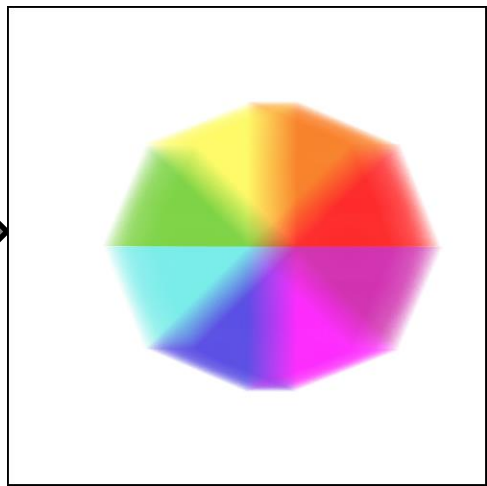
2.09	7X15B	7X16	jwf	ch
dll	dll	dll	no	yes

blur_linear_jf.dll

Creates a motion blur effect.

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





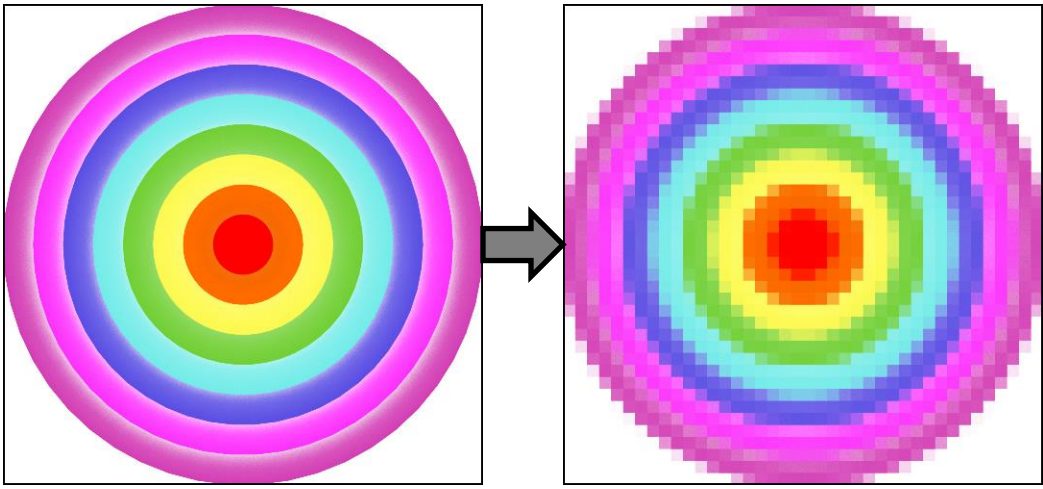
blur_pixelize (2D)

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)

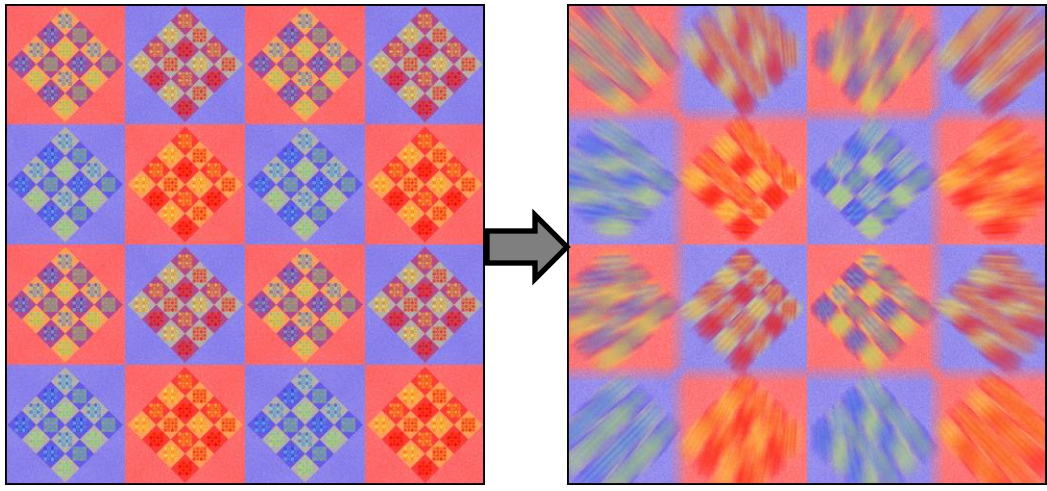
2.09	7X15B	7X16	jwf	ch
dll	dll	yes	yes	yes

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

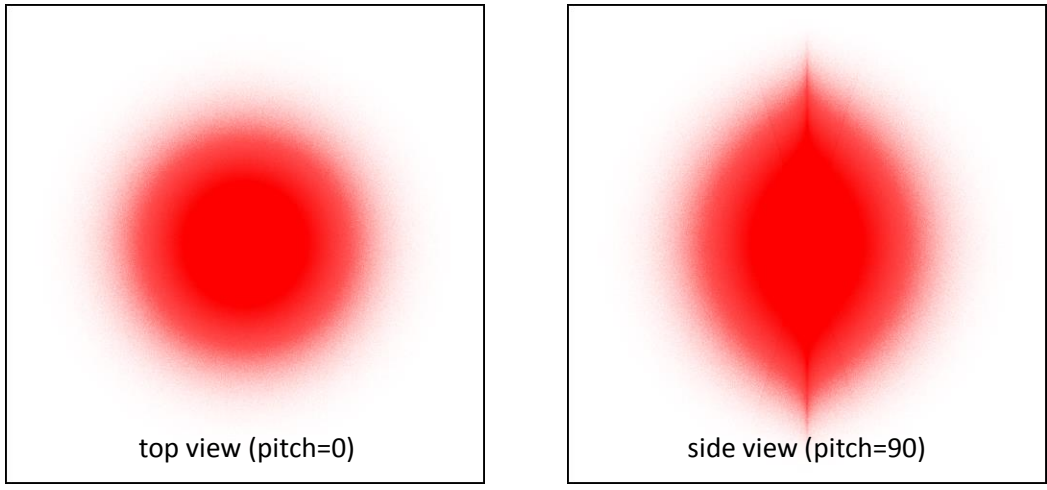
See falloff2 with *type* = 1.



blur3D (3D blur)

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

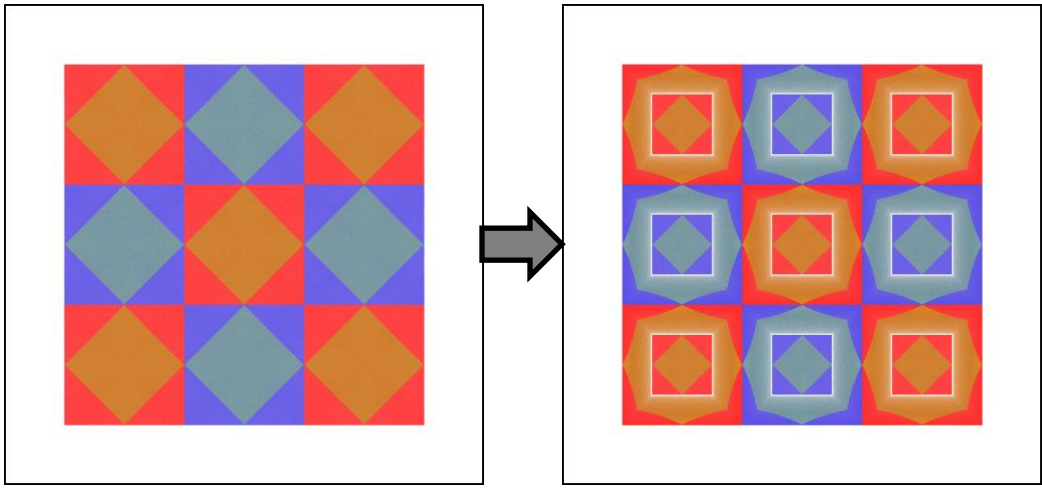
Three dimensional Gaussian blur



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.



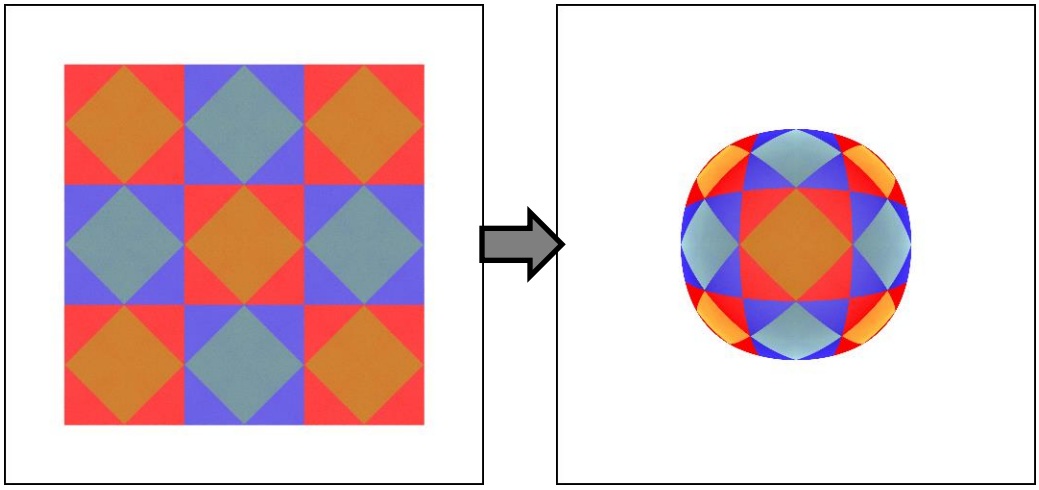
bubble (2D/3D, sets z)

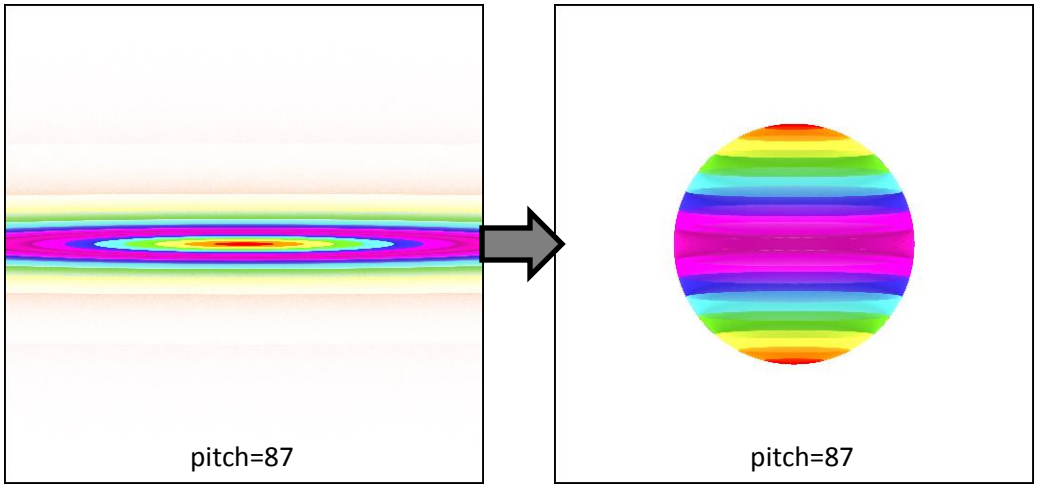
2.09	7X15B	7X16	jwf	ch
yes	yes	yes	yes	yes

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.





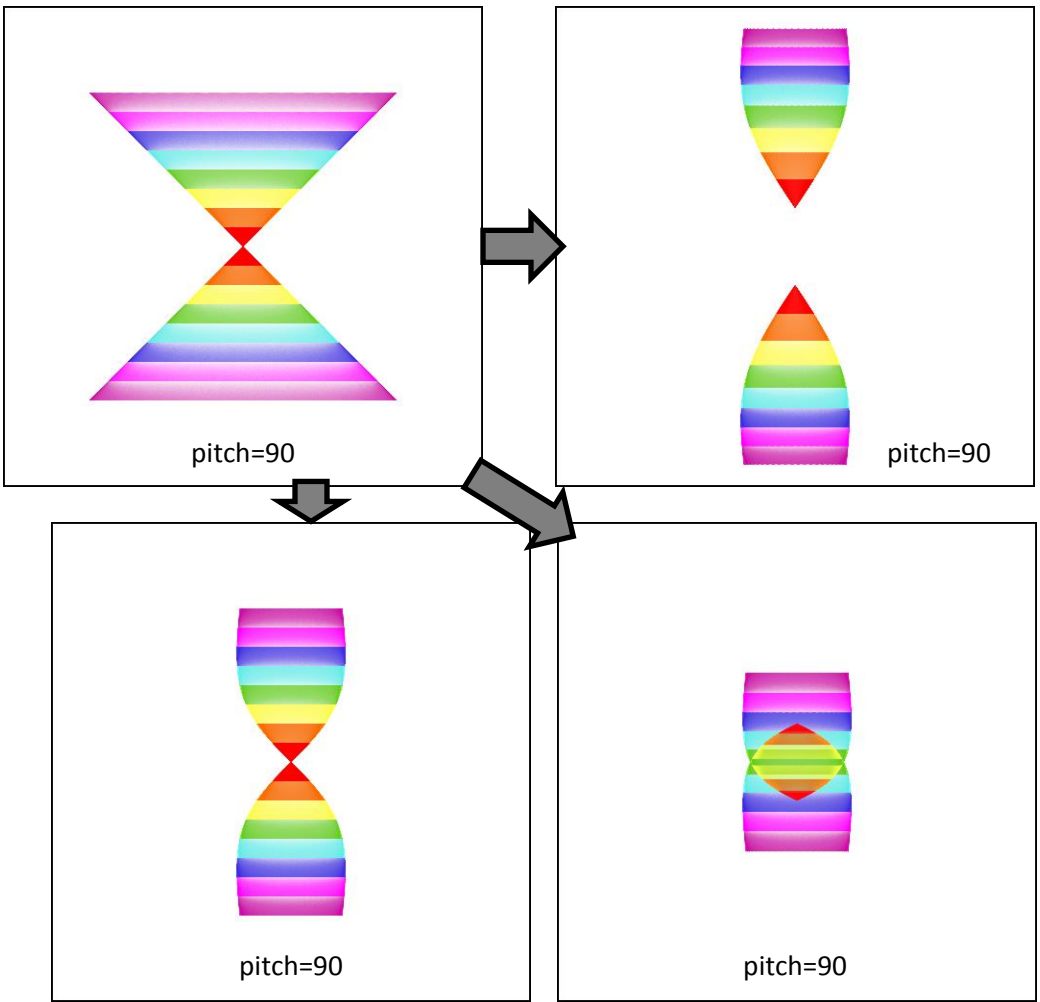
bubble2 (3D, transforms z)

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.

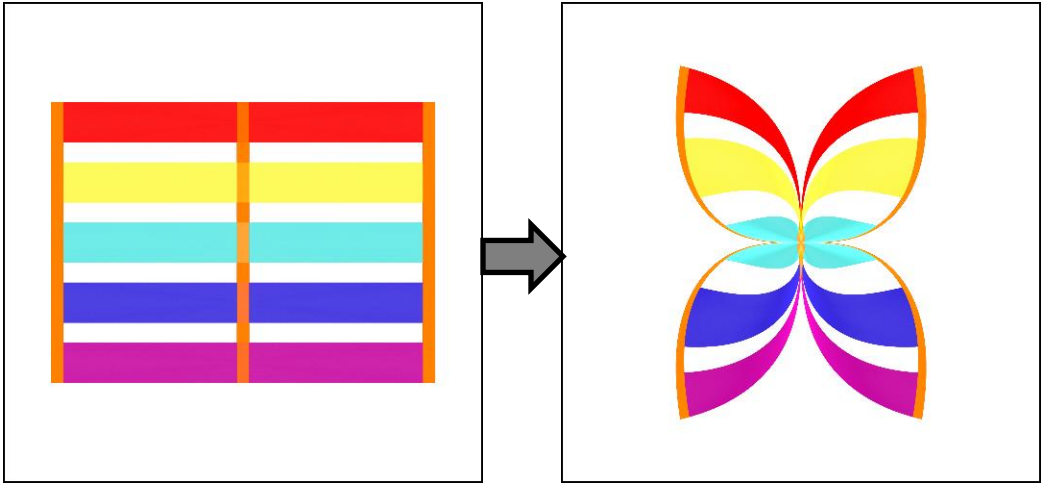


butterfly (2D)

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

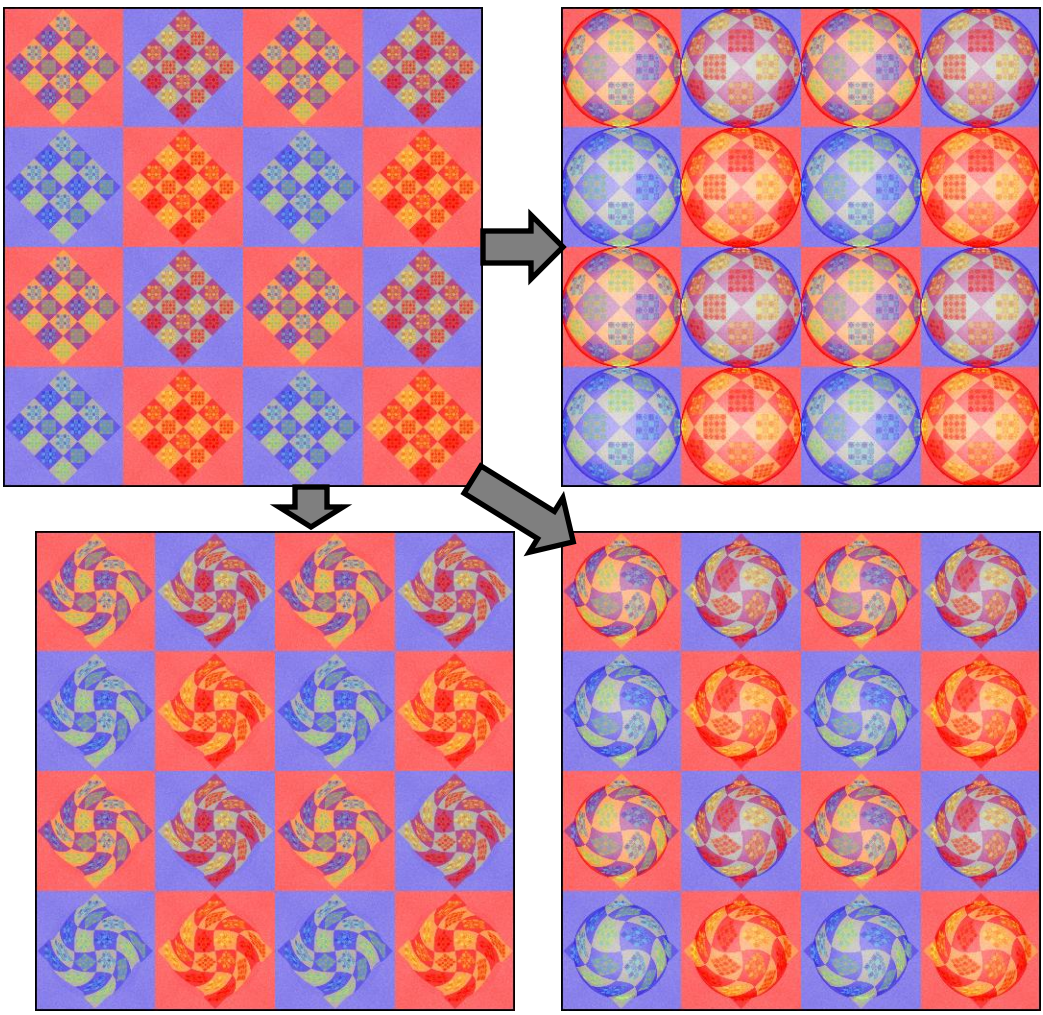
Overlays the flame on a grid of bubbles, like bubble wrap. Note this is 2D only; z is ignored.

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.



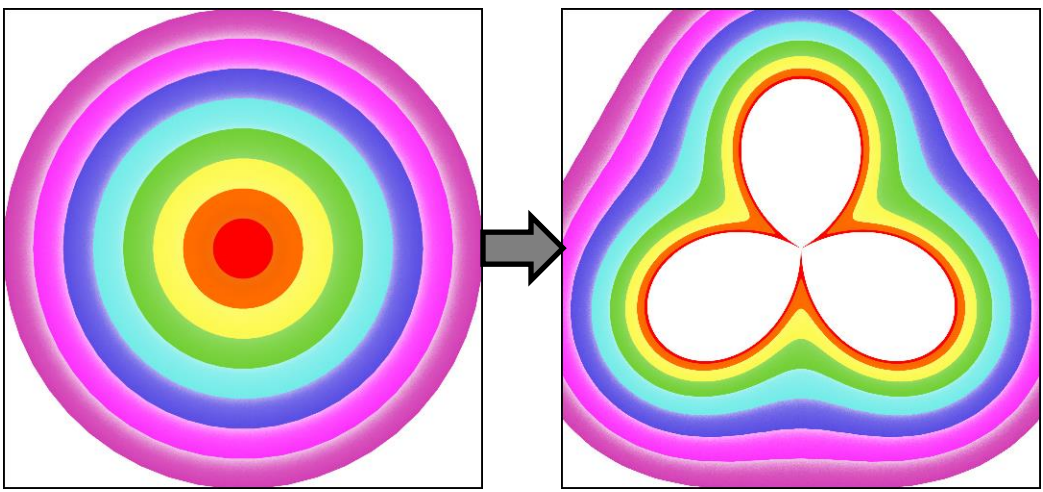
cardiod (2D)

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

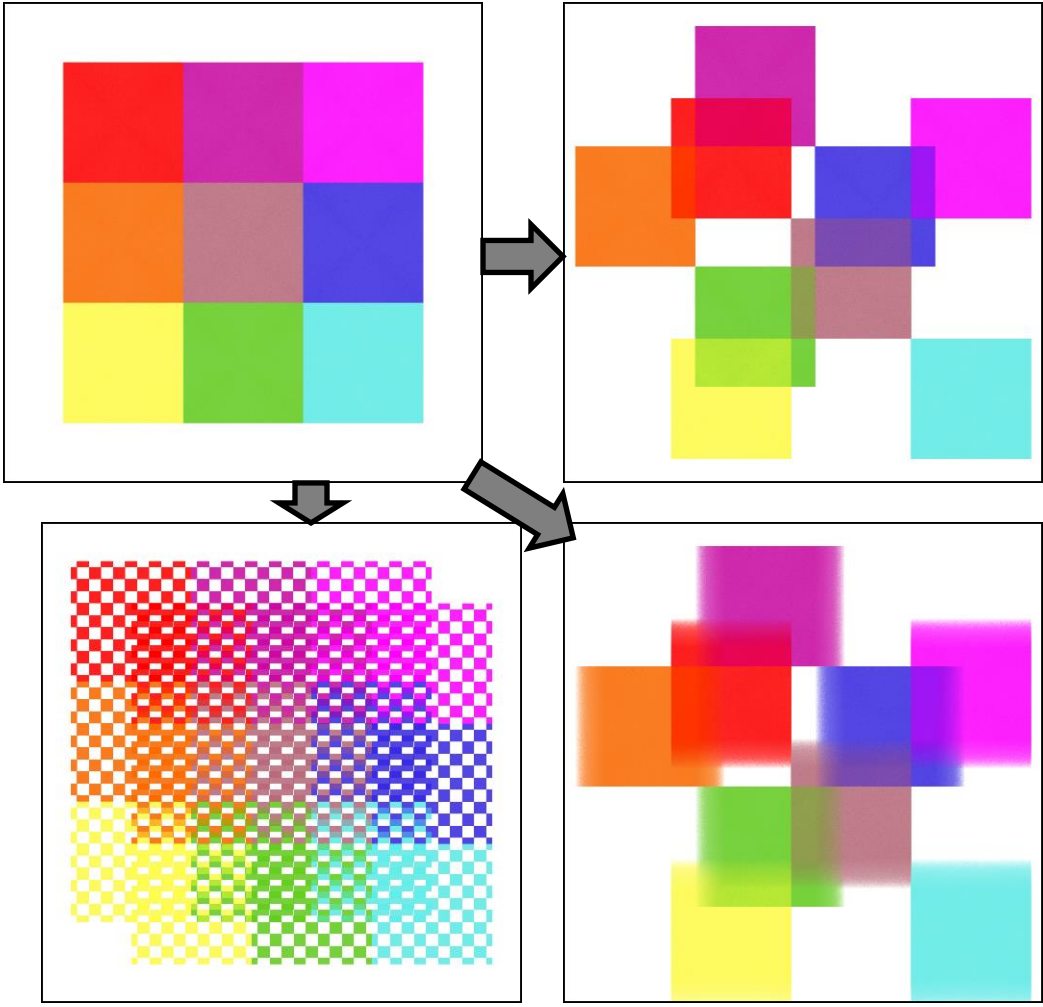
checks.dll

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.



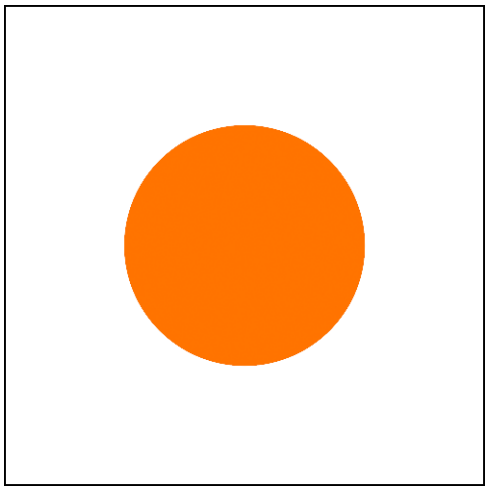
circleblur (2D blur)

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.

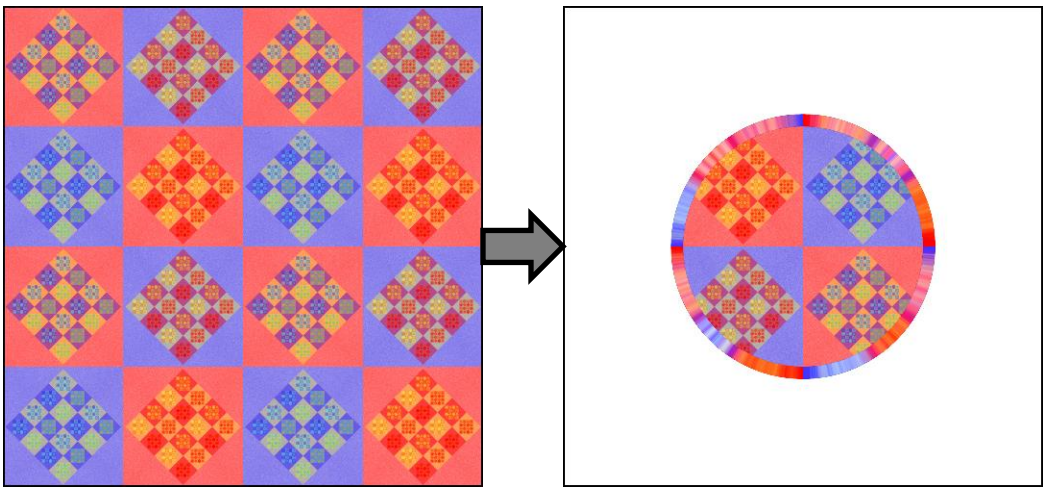


circlecrop, pre_circlecrop, post_circlecrop (2D, passes z)

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.



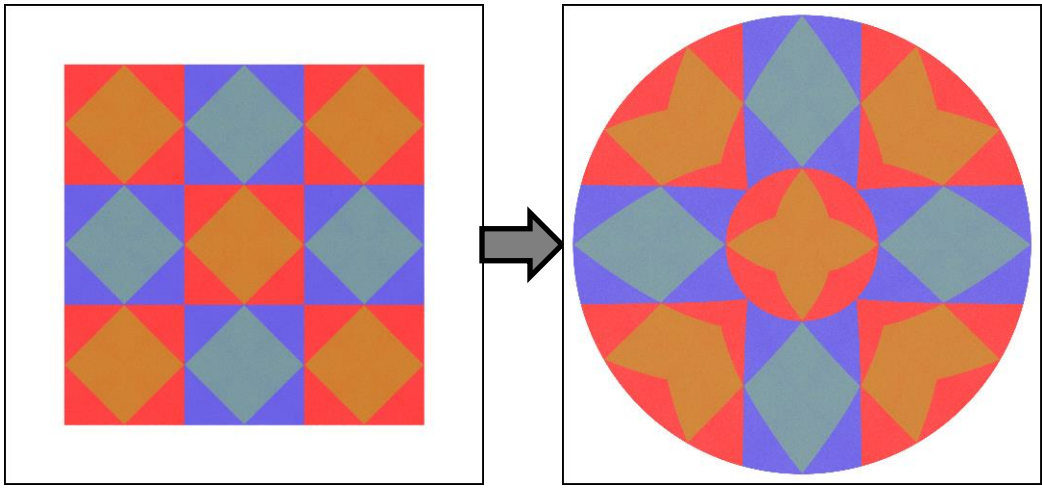
circlize (2D)

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.

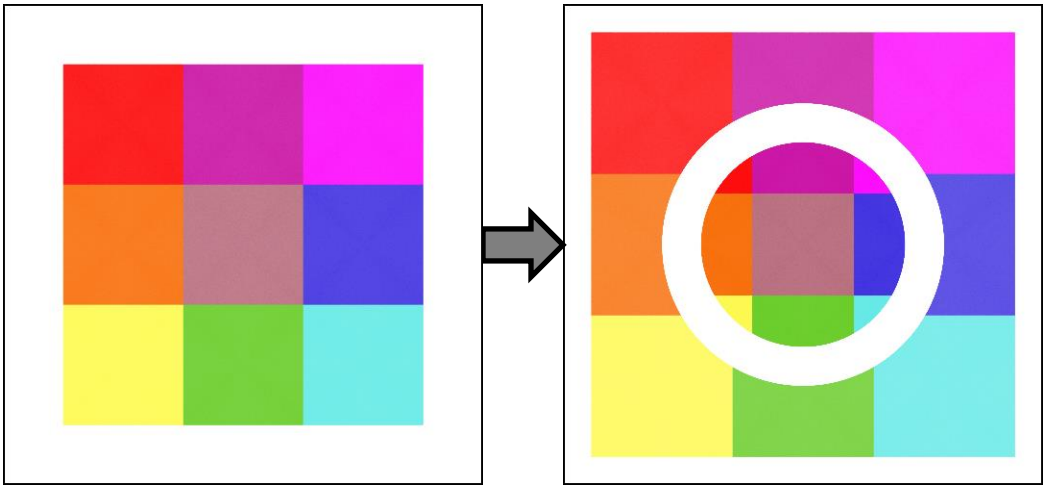
See squarize, which does the opposite.



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.



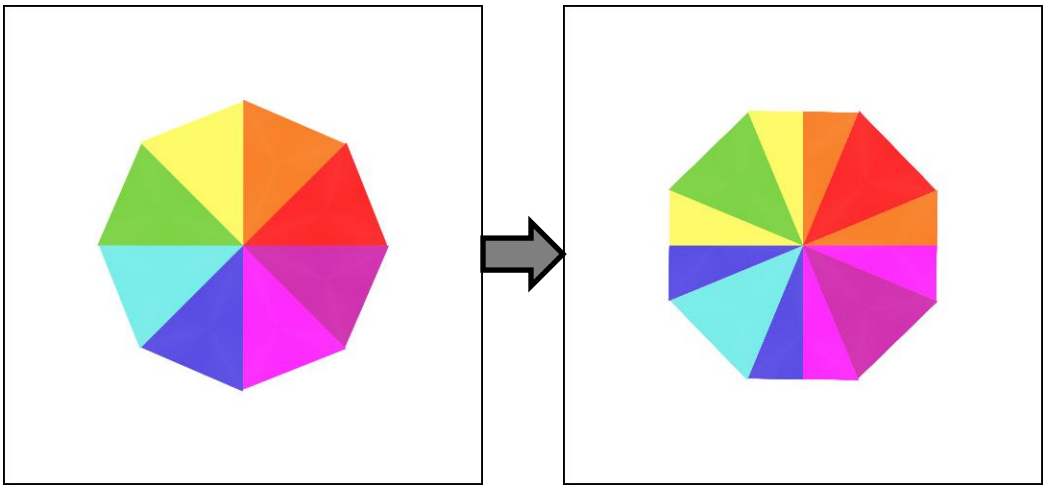
collideoscope (2D)

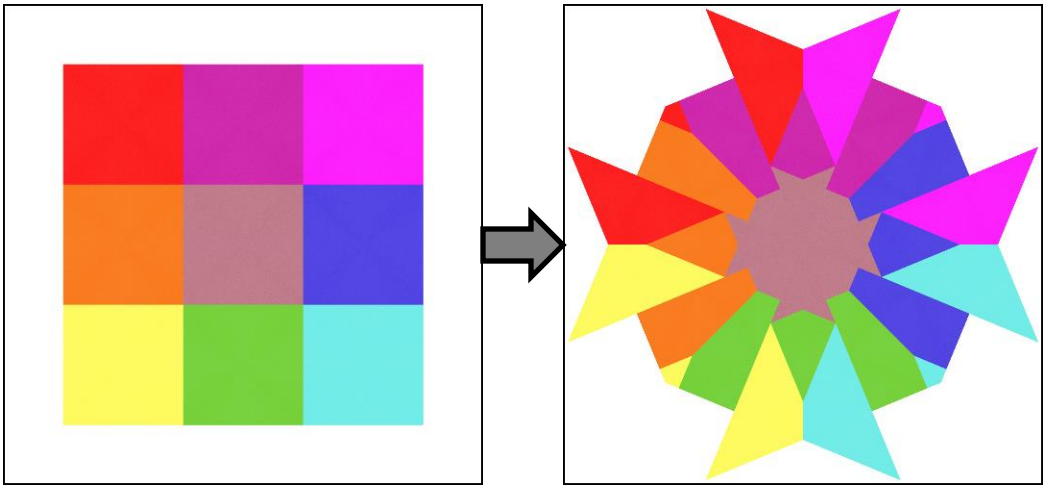
2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	yes

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)

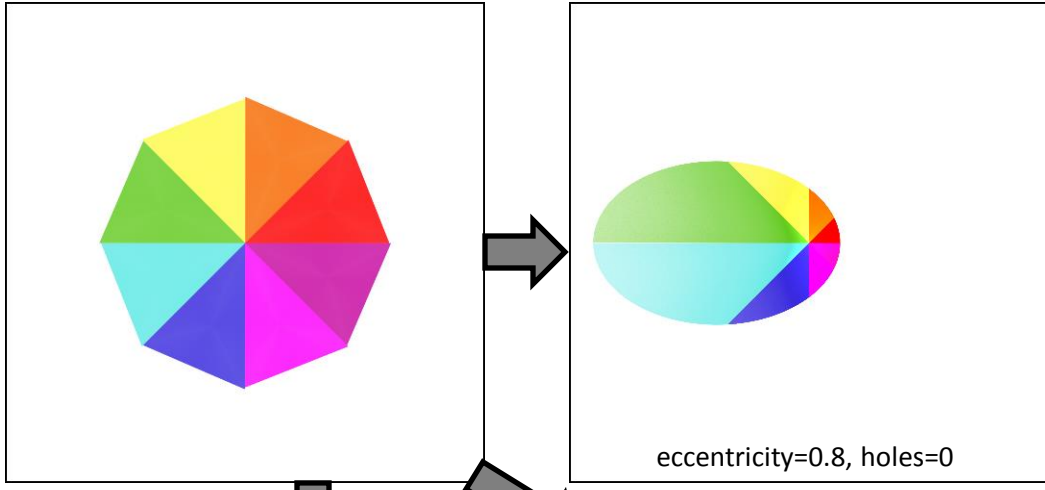
2.09	7X15B	7X16	jwf	ch
dll	dll	dll	no	dll

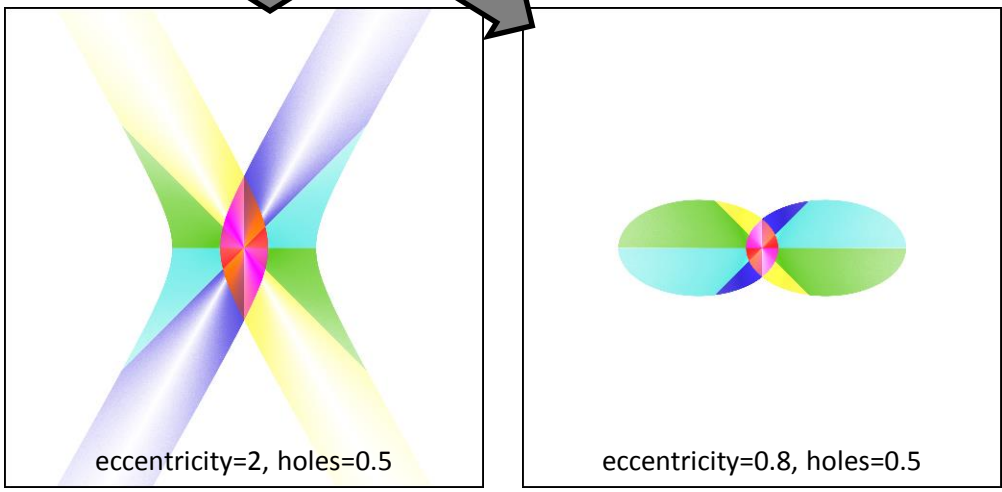
conic2.dll
Makes a conic section shape (ellipse, parabola, or hyperbola), with the focal point at the origin.

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*.

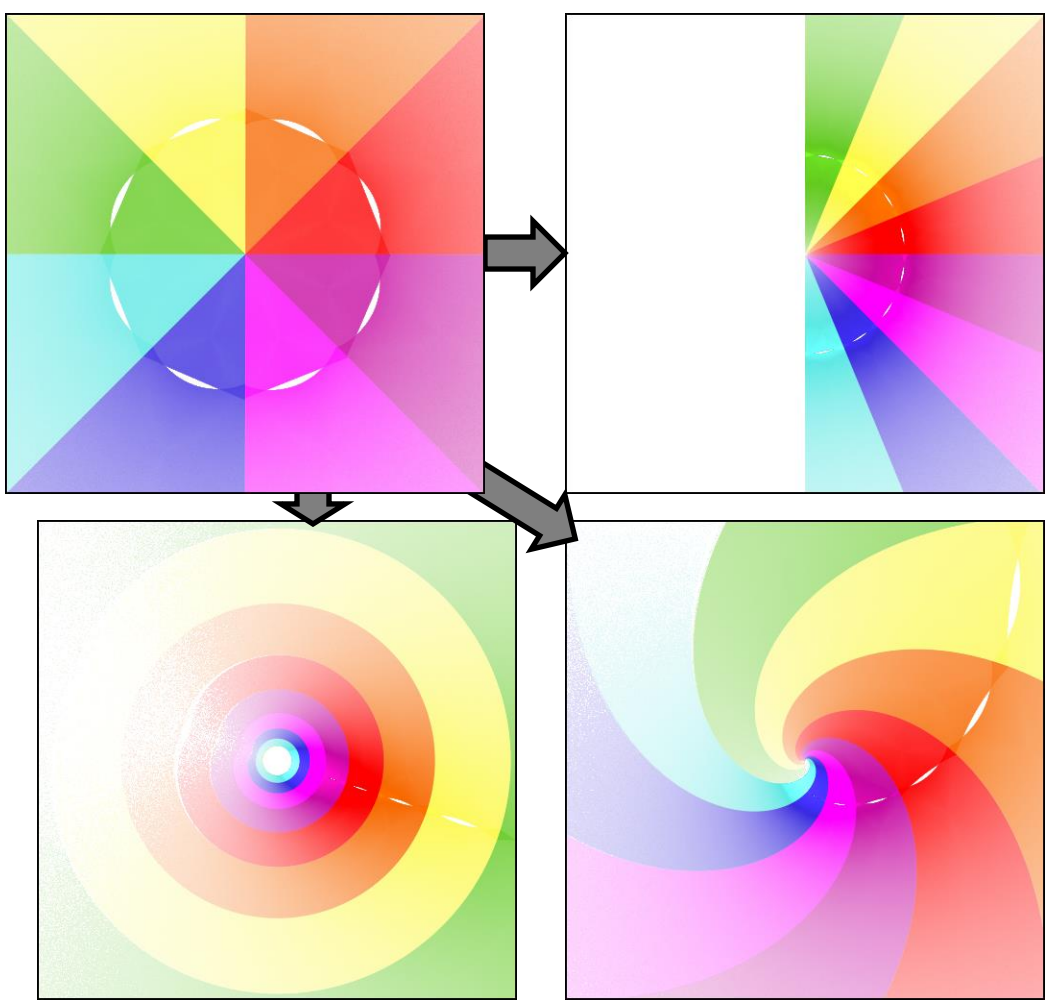
With *i* = 0, decreasing *r* from its default of 1 splits the plane along the negative x axis and rotates the top and bottom away from each other, much like folding a Japanese fan (top right, where *r* = 0.5 and *i* = 0). Increasing *r* does the opposite, causing an overlap.

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.

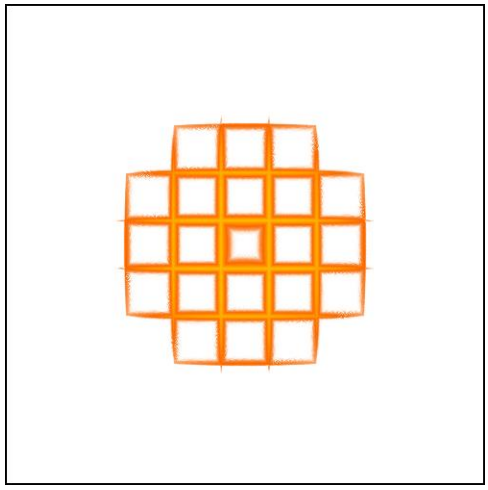
Top left: *cellsize* = 0.8, *power* = 0.1, *distort* = 0, *scale* = 1, *z* = 0

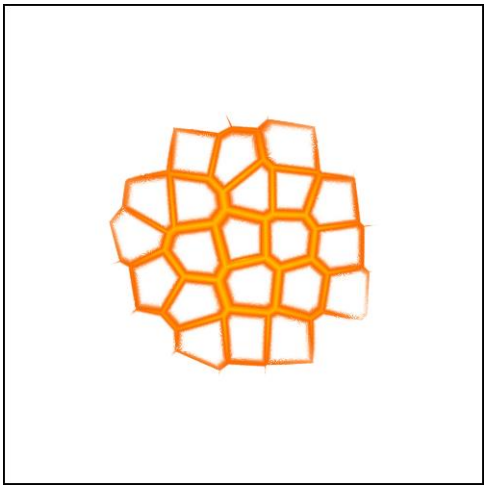
Top right: *cellsize* = 0.8, *power* = 0.1, *distort*= 0.4, *scale* = 1, *z* = 0

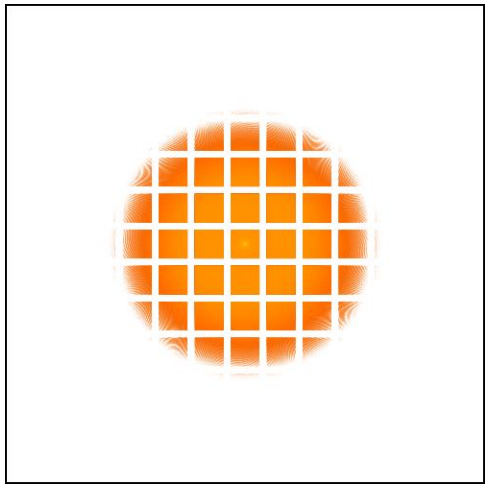
Bottom left: *cellsize* = 0.6, *power* = 1, *distort* = 0, *scale* = 0.8, *z* = 0

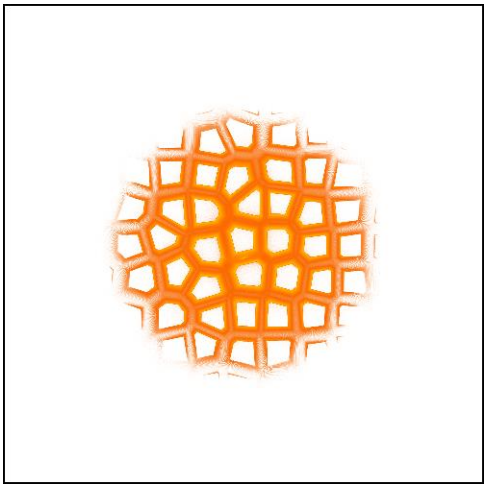
Bottom right: *cellsize* = 0.6, *power* = -0.3, *distort* = 0.4, *scale* = 0.6, *z* = 0

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







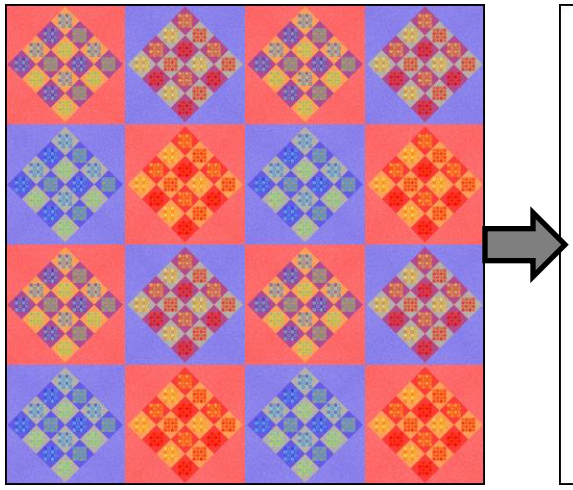


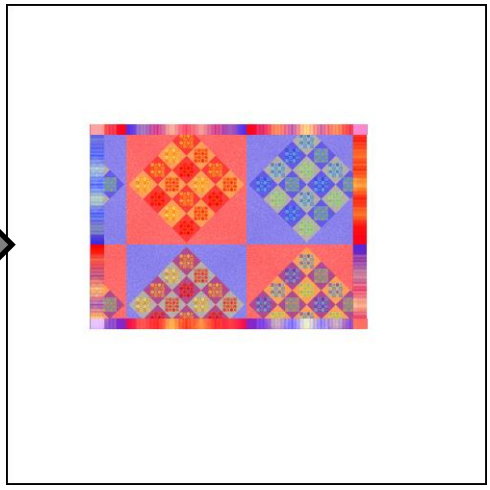
crop, pre_crop, post_crop (2D, passes z)

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.





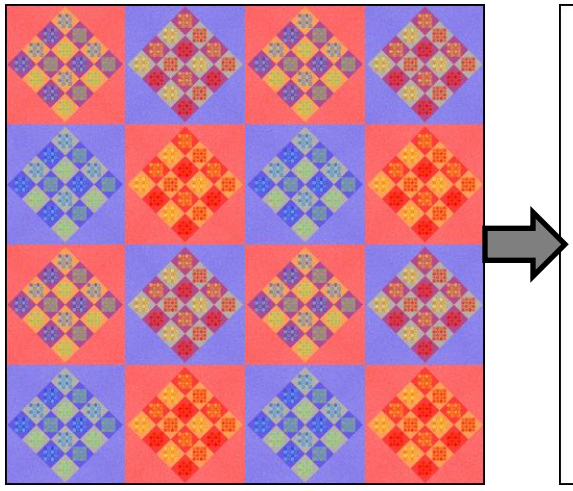
cropn (2D, passes z)

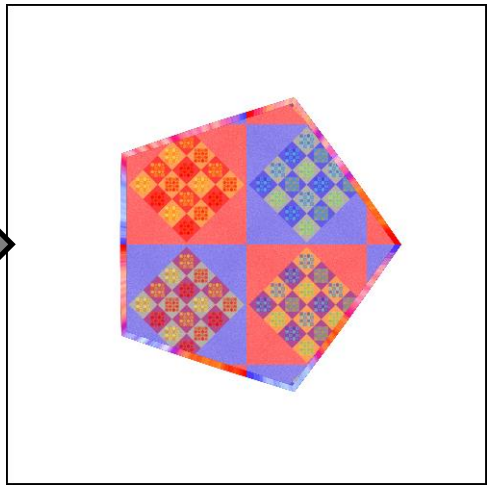
2.09	7X15B	7X16	jwf	ch
dll	dll	dll	no	yes

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.





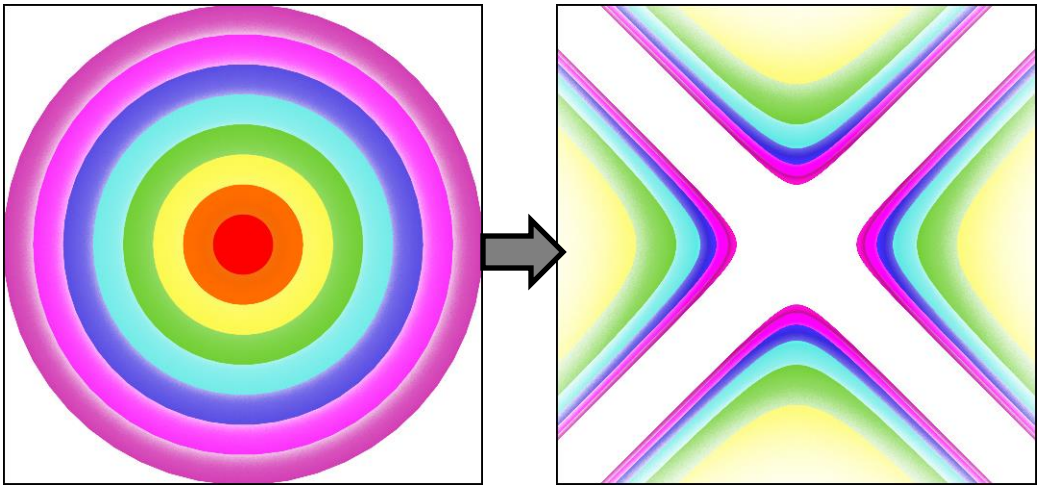
cross (2D)

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

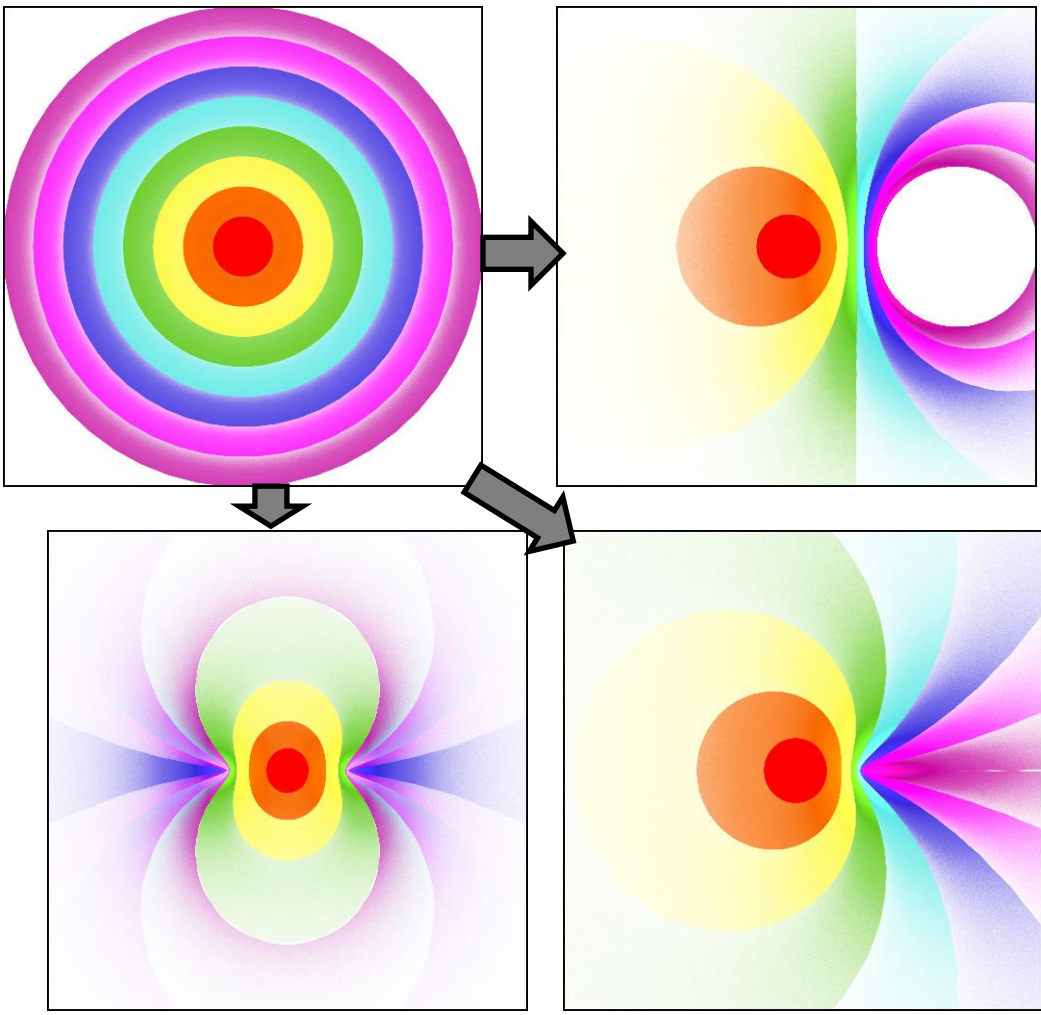
Think of the plane as a series of coincentric rings, as shown here but extending out to infinity.

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.

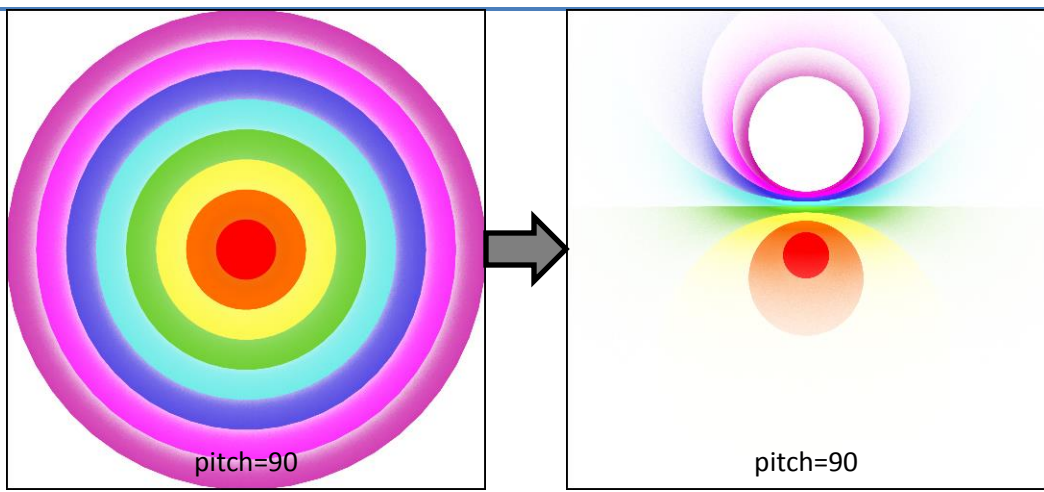


curl3D (3D, transforms z)

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

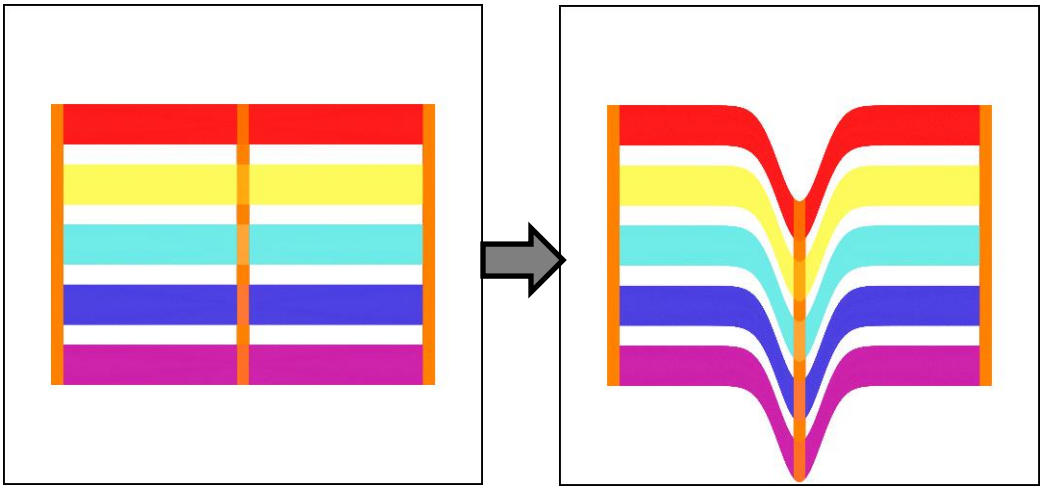


curve (2D)

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.

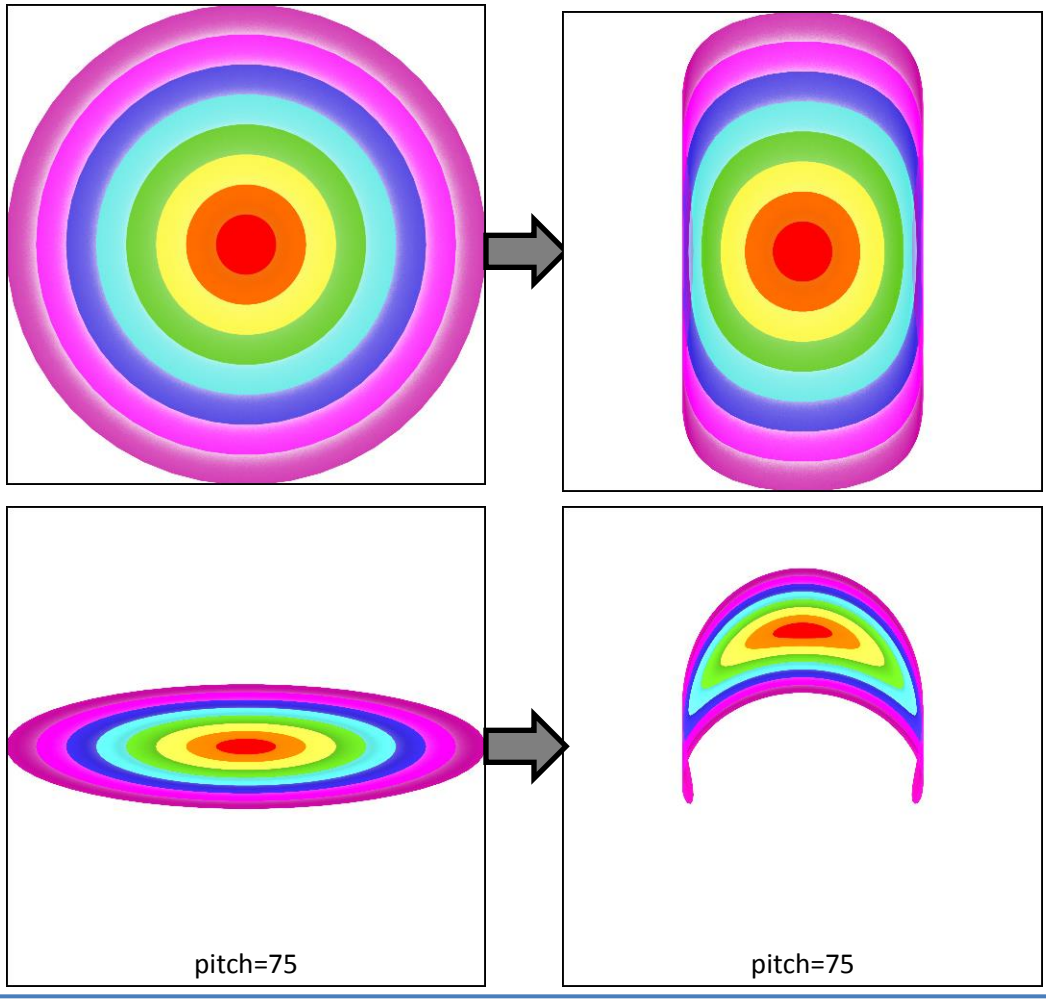
Example has *xamp* = 0, *yamp* = 0.8, *xlength* = 1, and *ylength*= 0.25



cylinder (2D/3D, sets z)

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

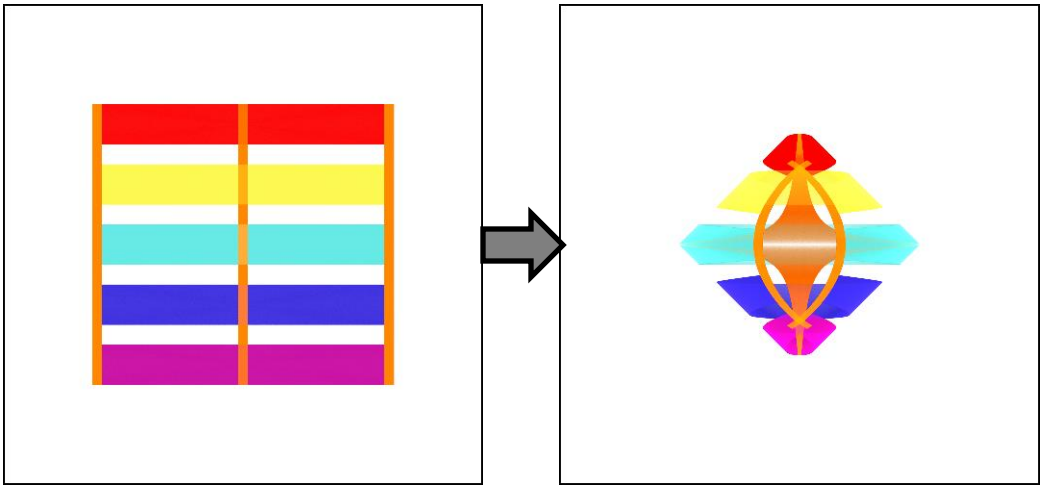
Curves the flame into a vertical cylinder. In 3D versions, it sets z (ignoring any previous value).



diamond (2D)

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

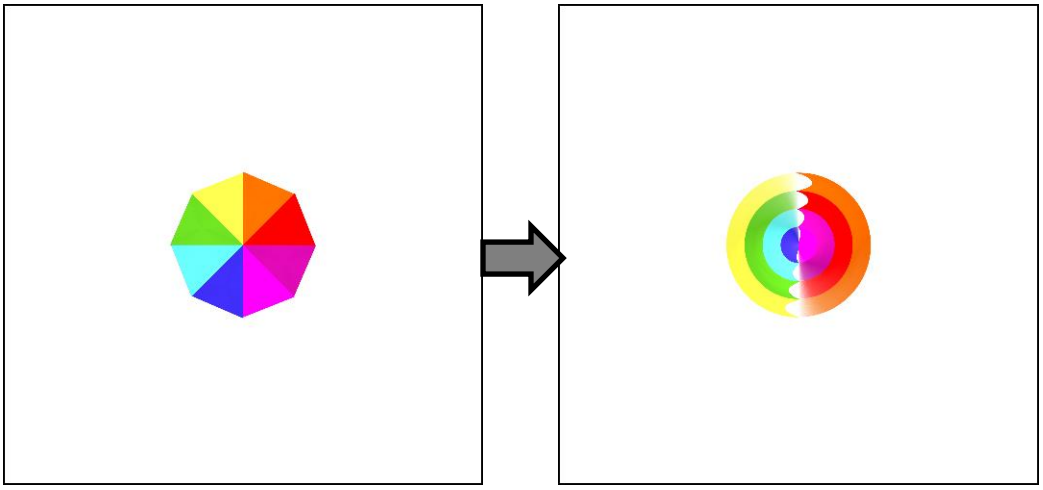
Warps flame into a diamond shape that fits in a unit circle. The example is carefully chosen to have minimal overlap, but as the original flame gets larger, diamond will warp it to fit the diamond shape.



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.

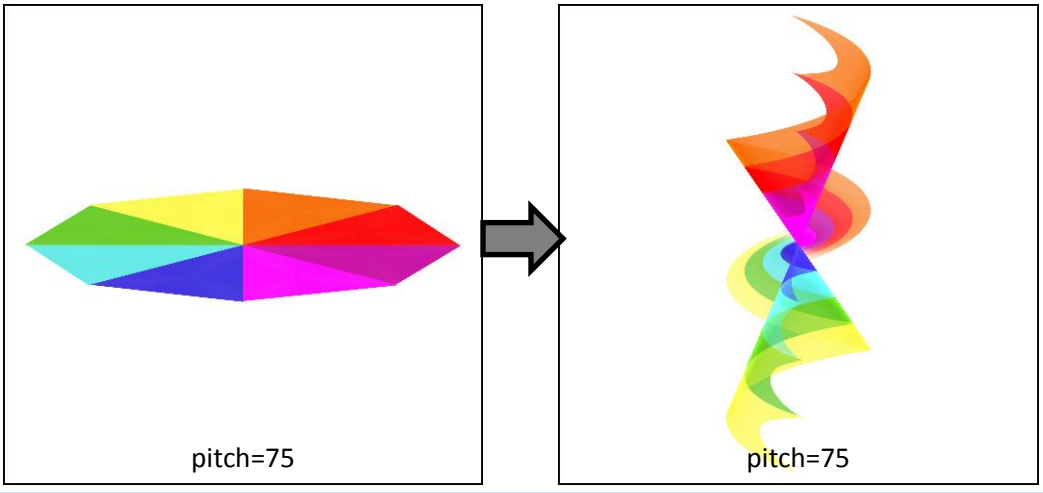


disc3d (3D, transforms z)

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

disc3d.dll

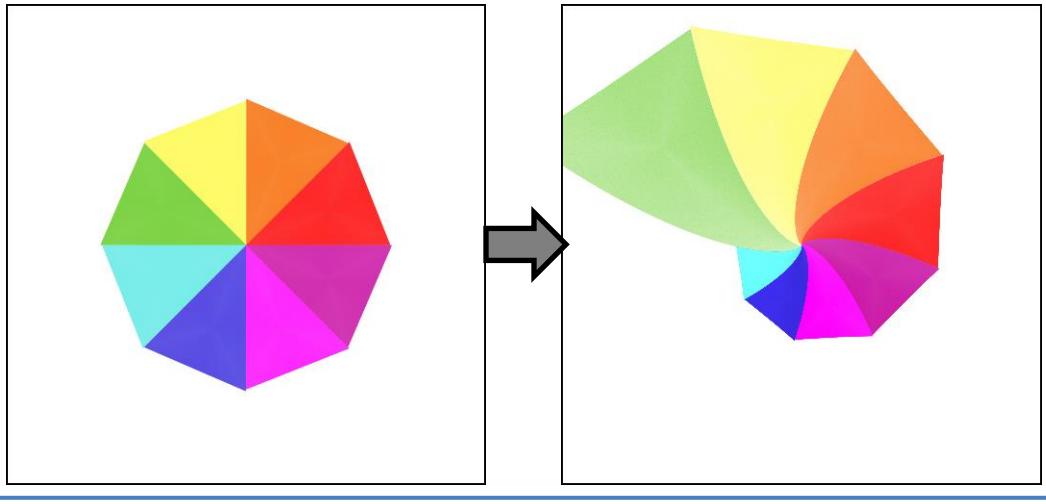
Changes x and y just like disc, but also transforms z according to the distance from the origin. The example shows the effect on a flat flame.



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.



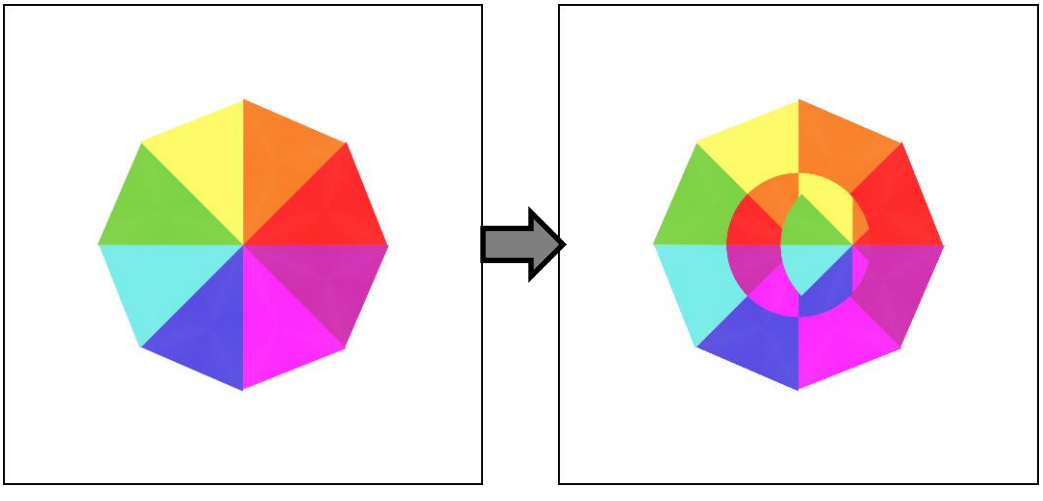
eclipse (2D)

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.

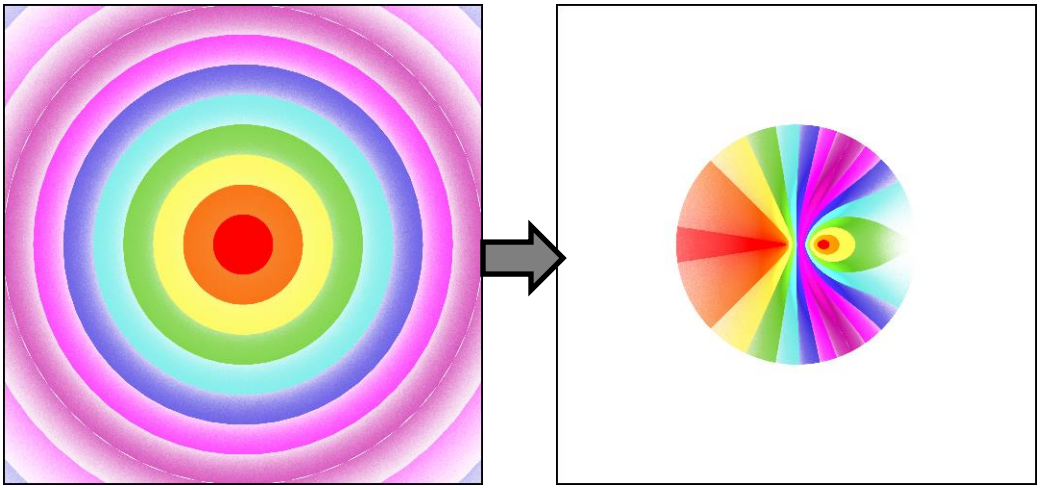


edisc (2D)

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.).

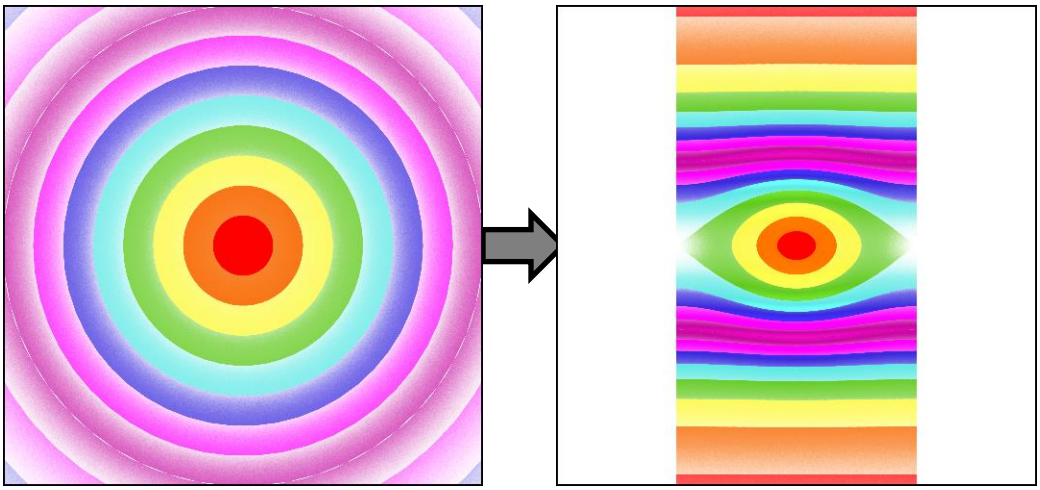


elliptic (2D)

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.



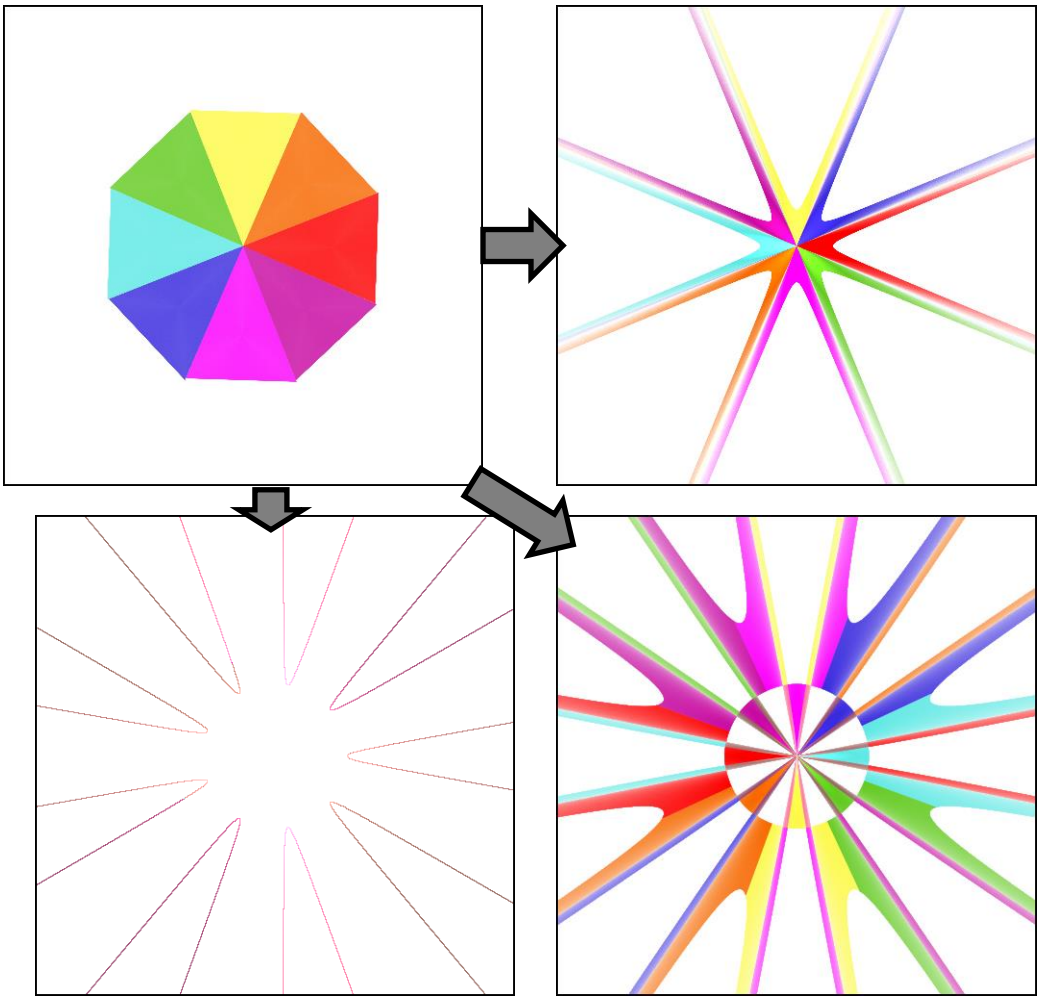
Epispiral (2D half blur)

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

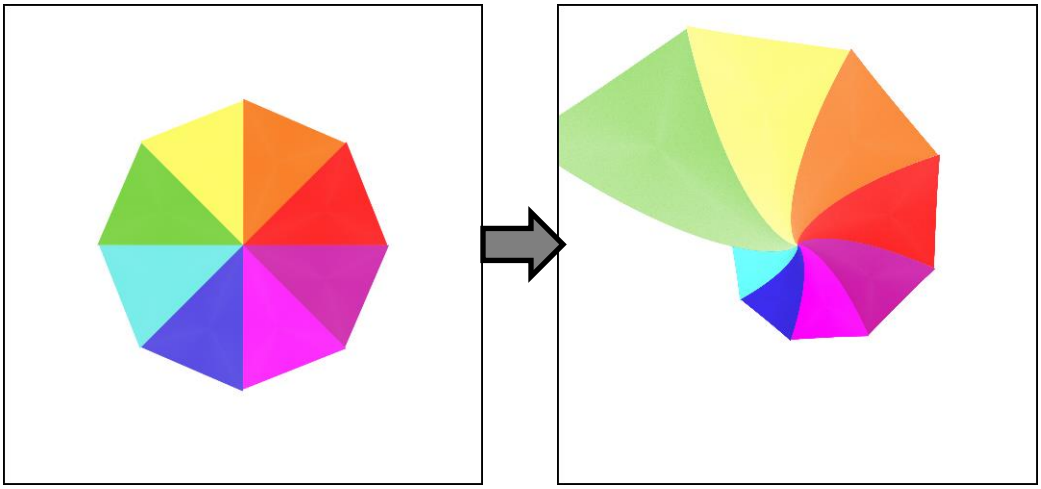


escher (2D)

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).

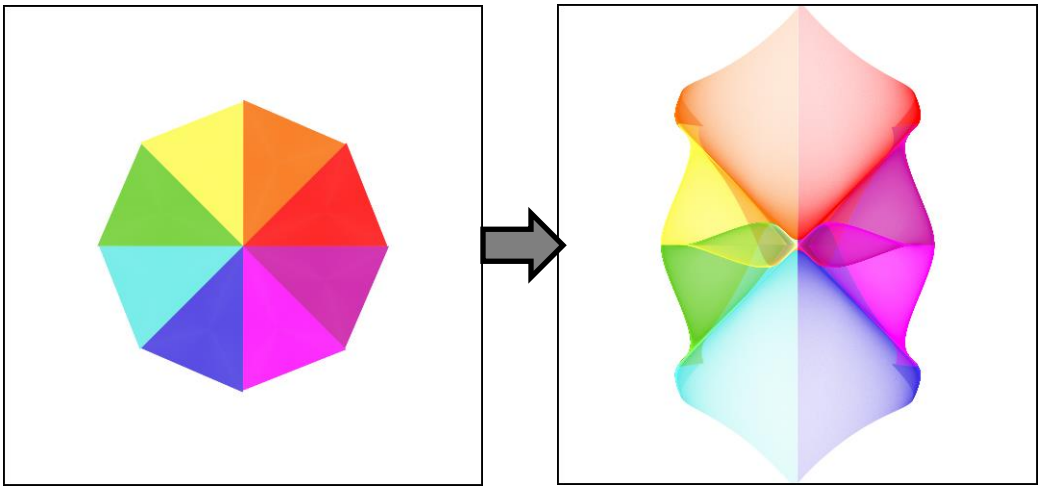


ex (2D)

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.

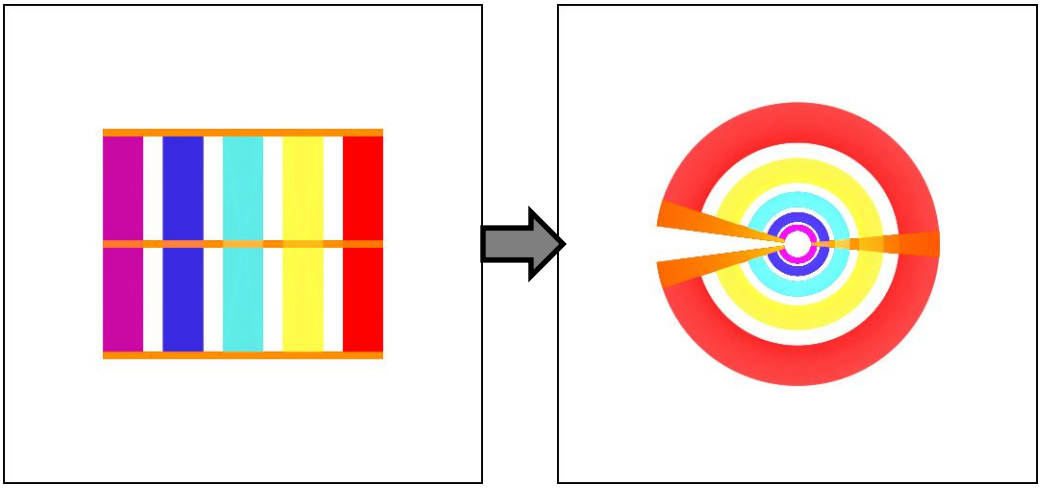


exponential (2D)

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.

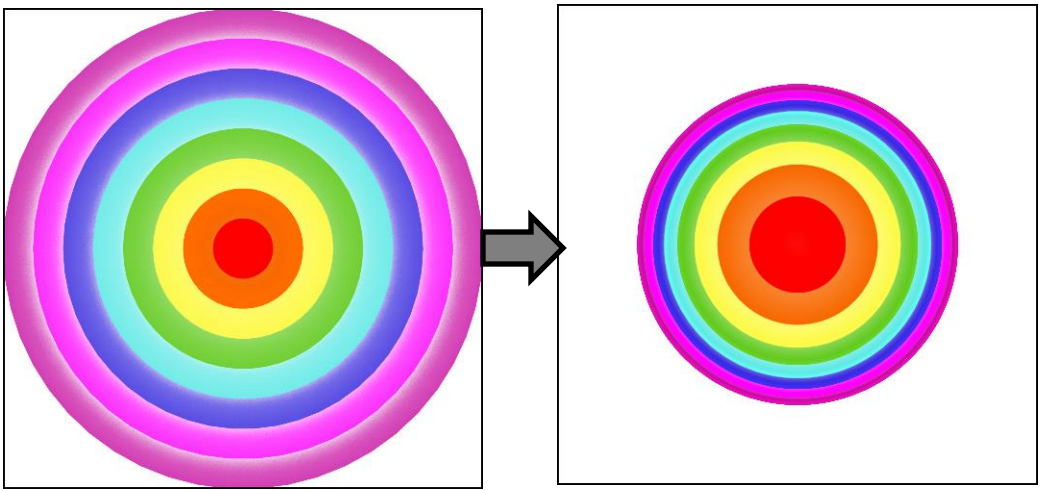


eyefish (2D)

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

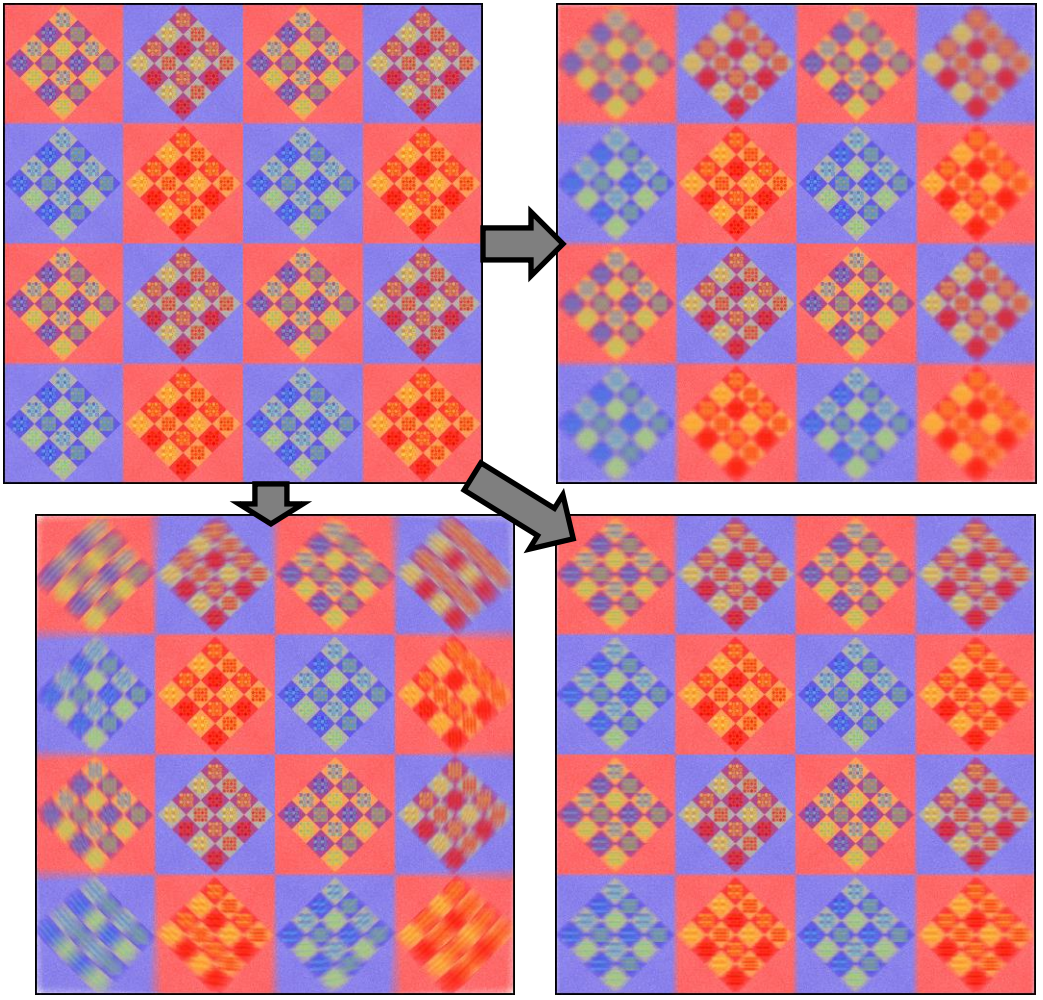
falloff2.dll, post_falloff2.dll, pre_falloff2.dll

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

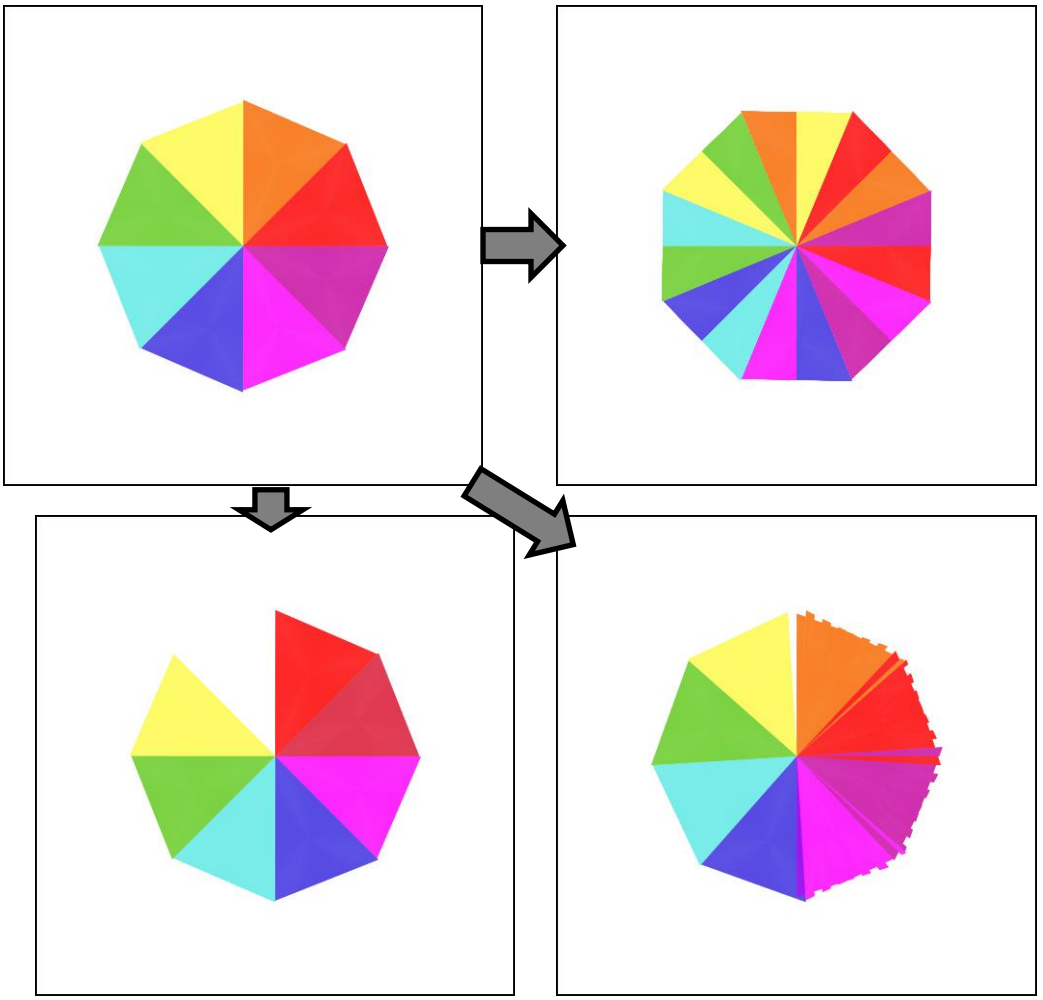
Divides the plane into wedges, then rotates them right or left. Variable *x* sets the size of the wedges. When *y* is 0, wedges on the left side are rotated left and wedges on the right are rotated alternately. Increasing *y* makes more wedges rotate alternately and decreasing *y* makes more rotate the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

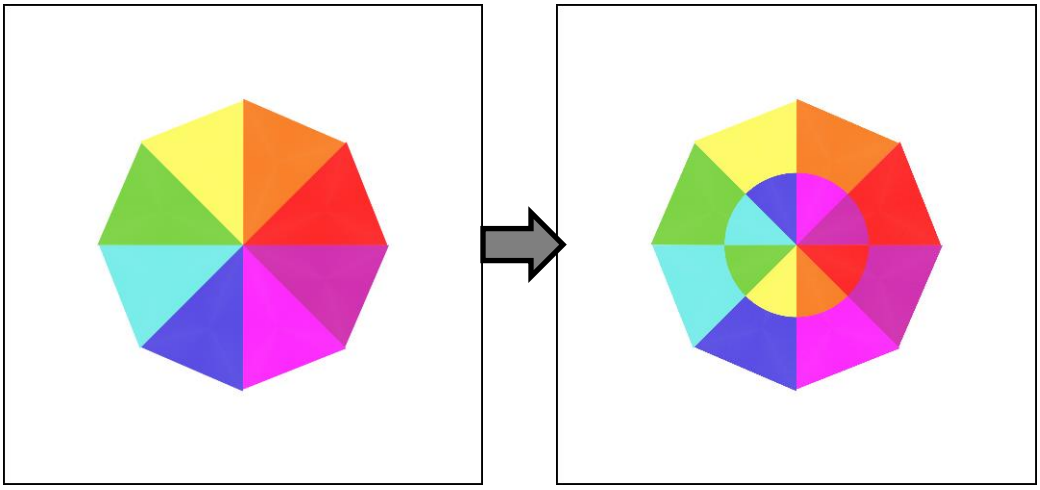


flipcircle (2D)

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.

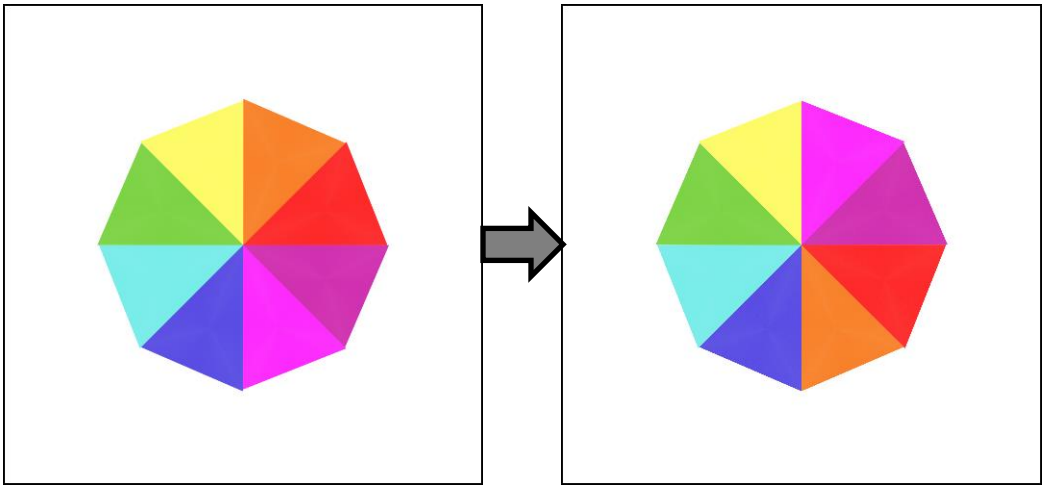


flipy (2D)

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.



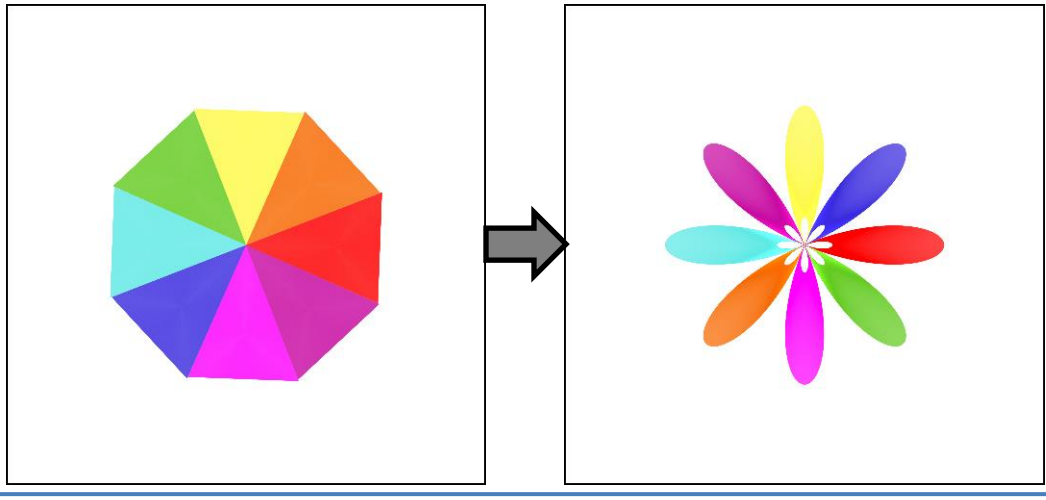
flower (2D half blur)

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.



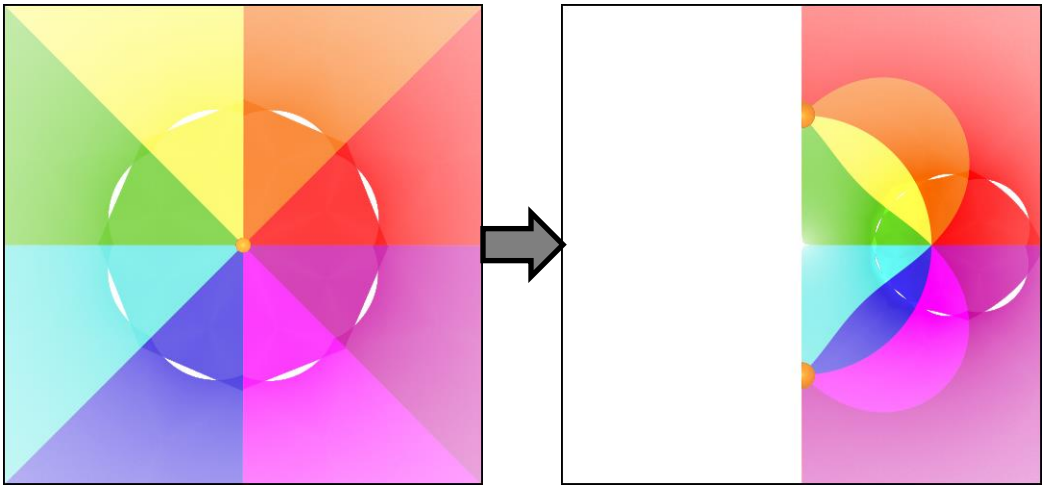
flux (2D)

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

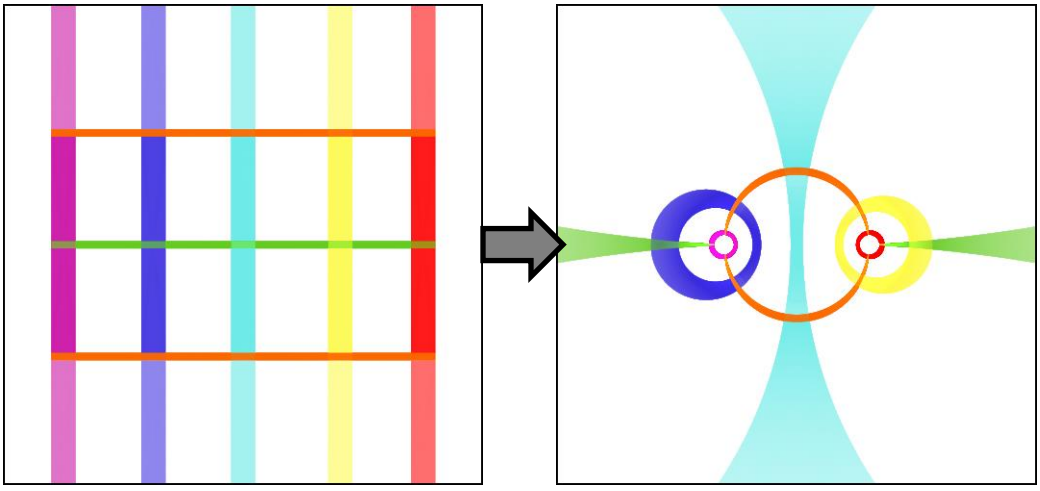
foci.dll

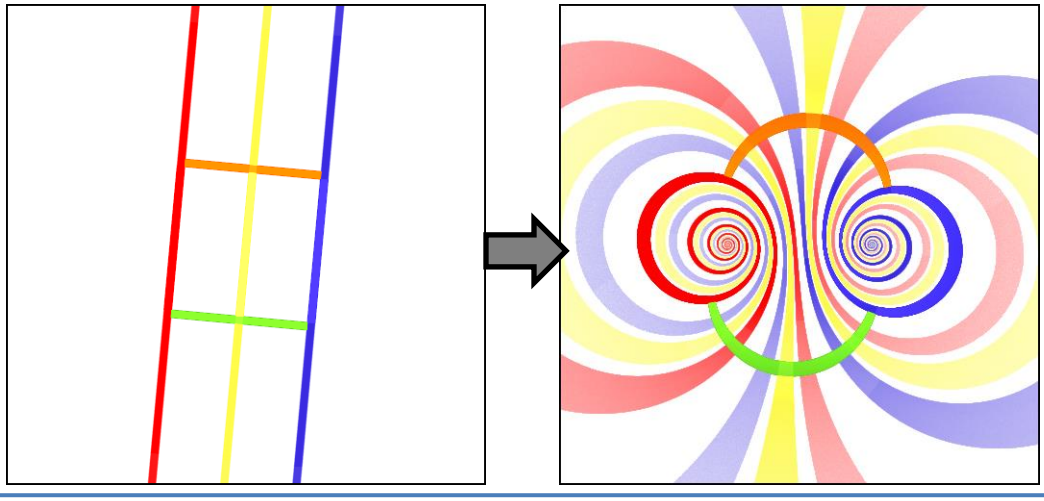
Wraps the plane around a horizontal cylinder, then tapers the ends and bends them away to form a U, looking up from the bottom.

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.



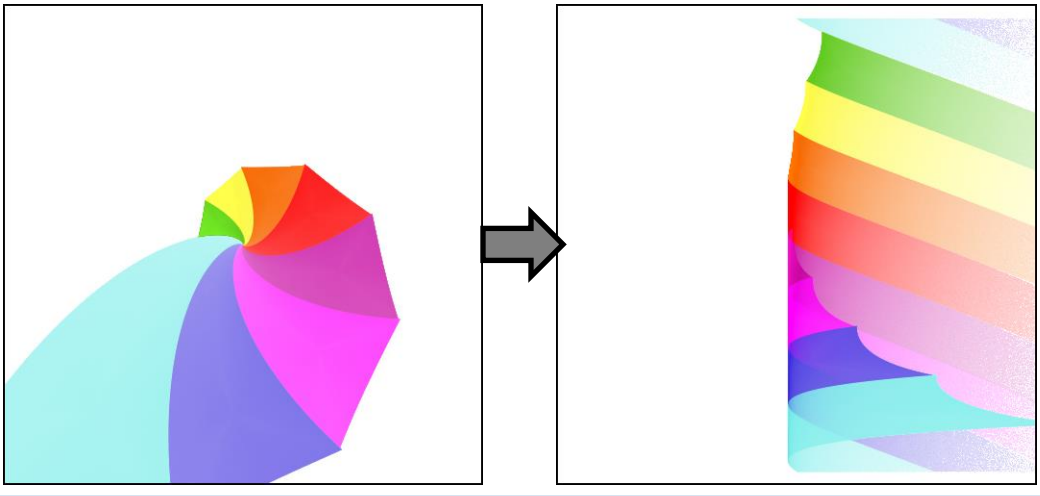


gamma (2D)

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.



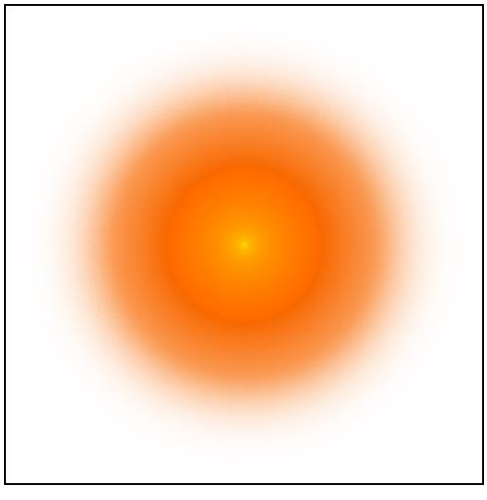
gaussian_blur, pre_blur (2D blur)

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

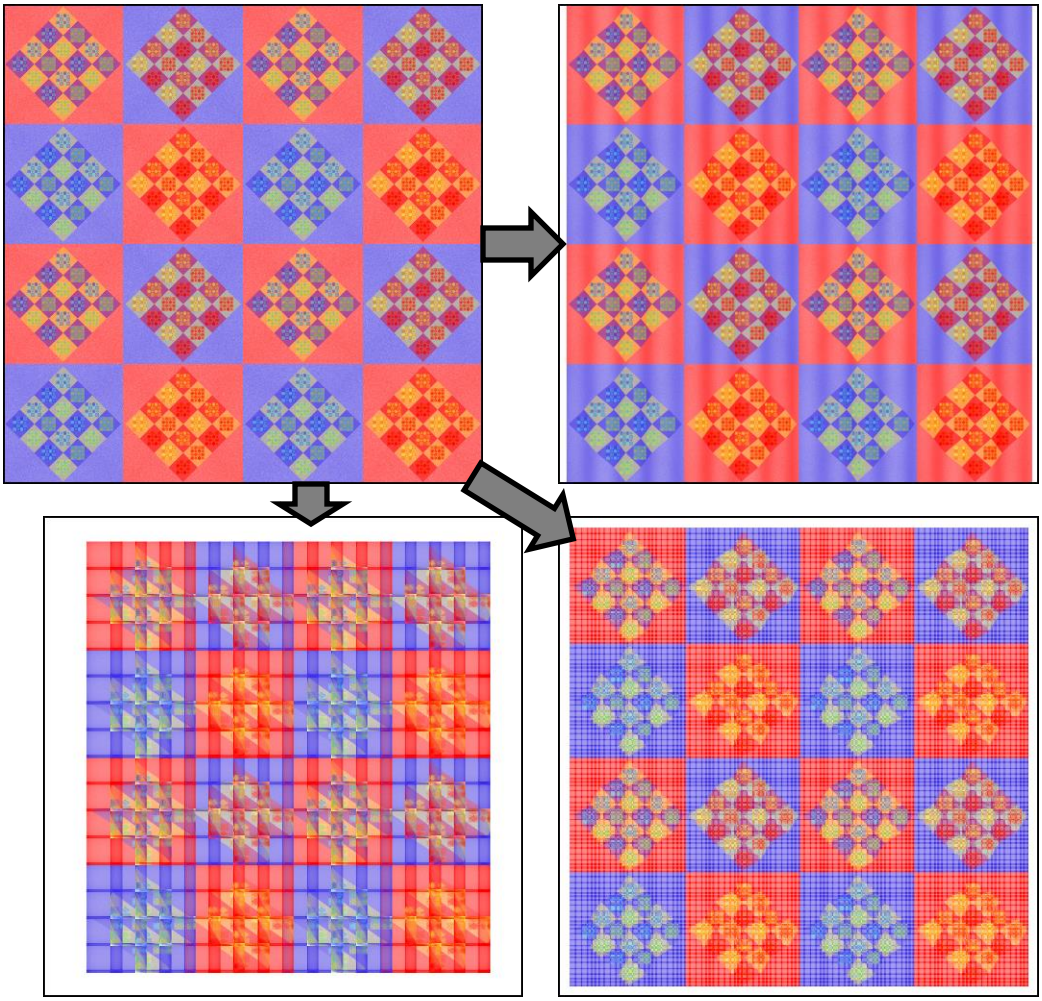
gdoffs.dll

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.

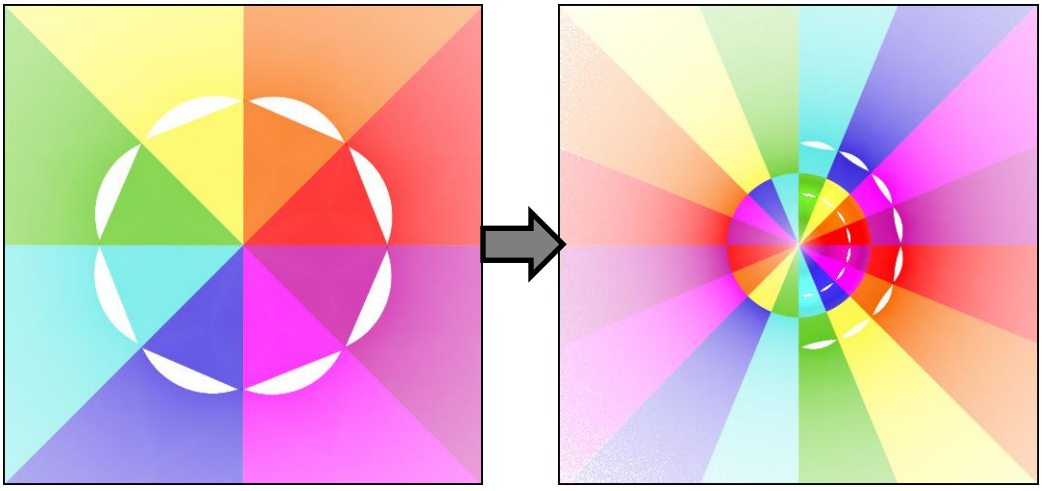


glynnia (2D)

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.

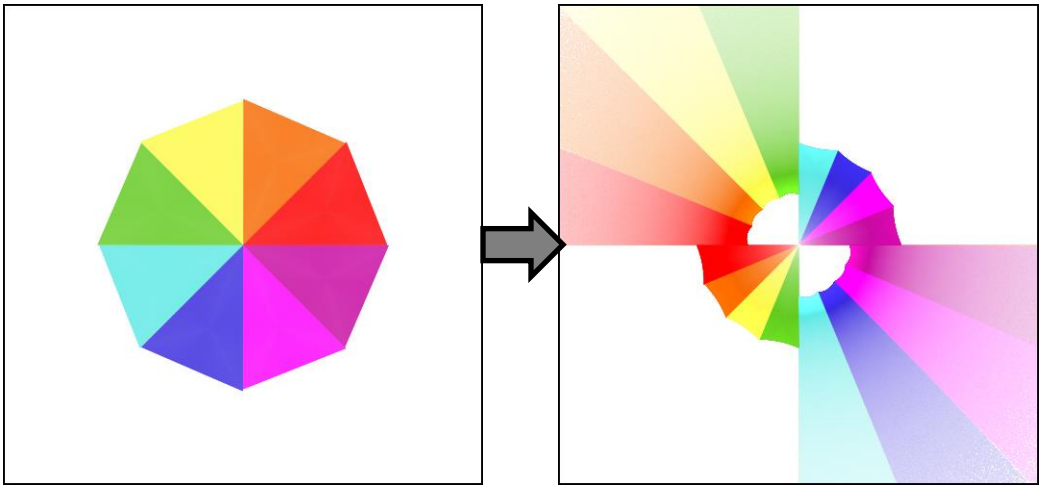


glynnia2 (2D)

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

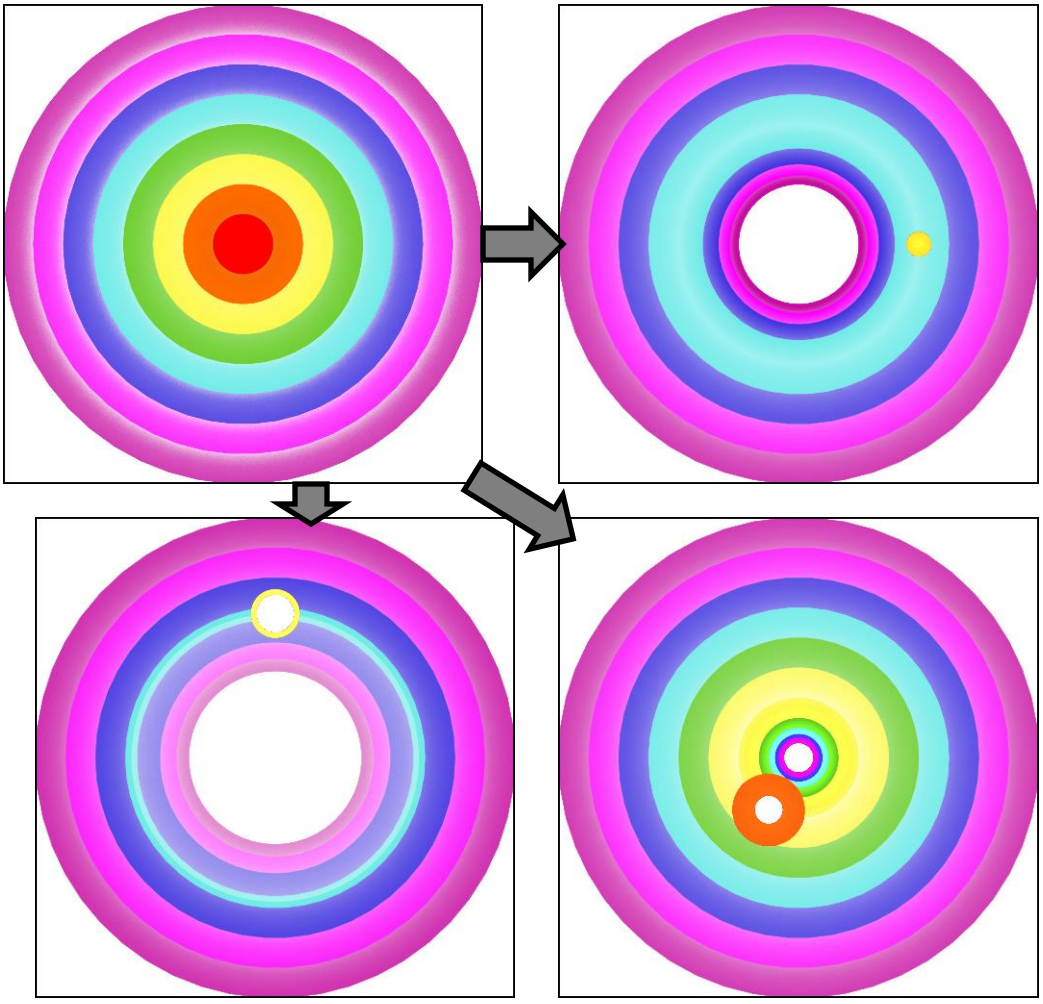
GlynnSim1.dll
The GlynnSim variations pass the part of the plane outside *radius* intact, and replace the inside with an everted copy of the outside. Variables *contrast* and *pow* control the density balance between inside and outside. Compare with spher.

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.



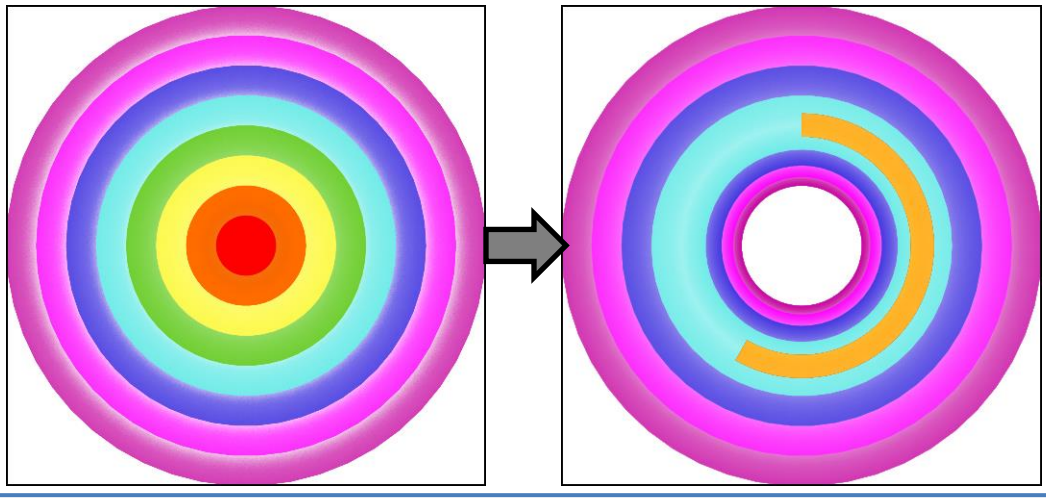
GlynnSim2 (2D)

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.



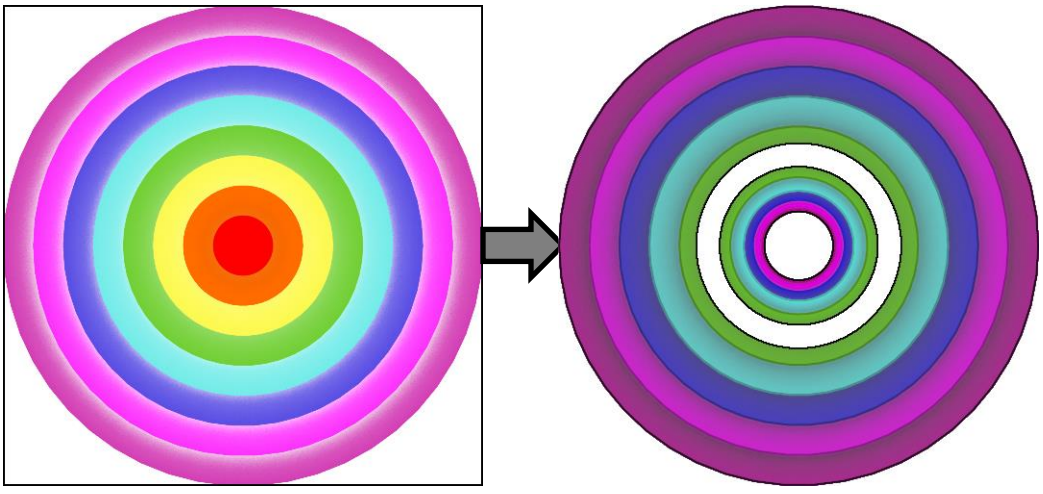
GlynnSim3 (2D)

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.



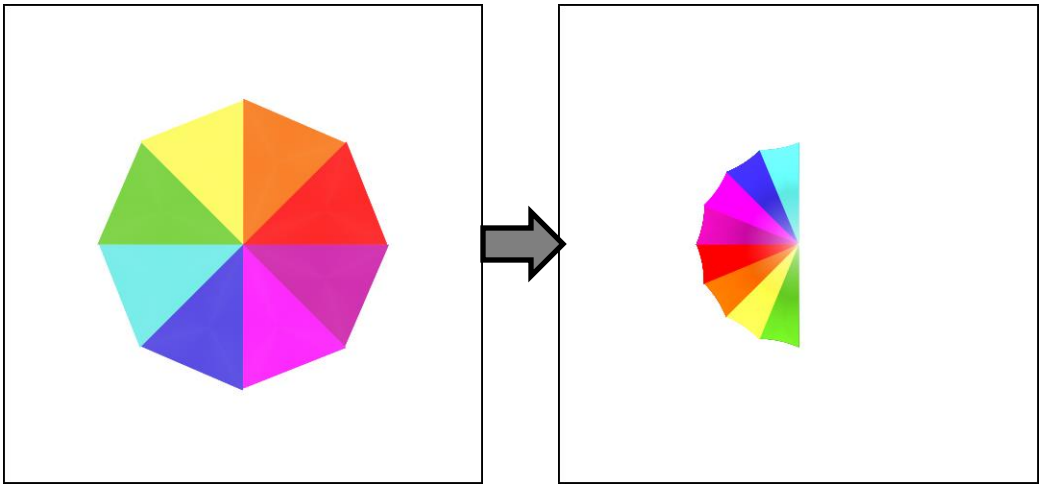
Half_Julian (2D)

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*.

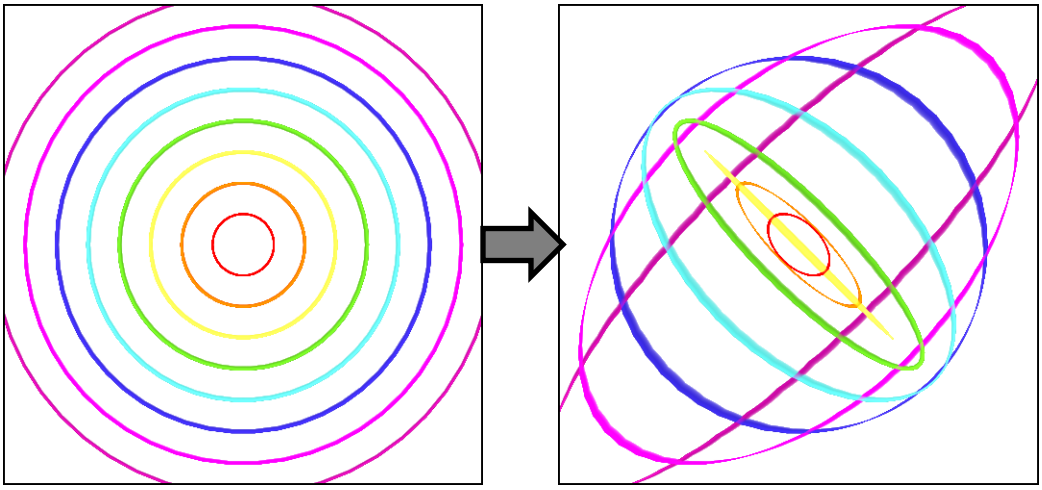


handkerchief (2D)

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

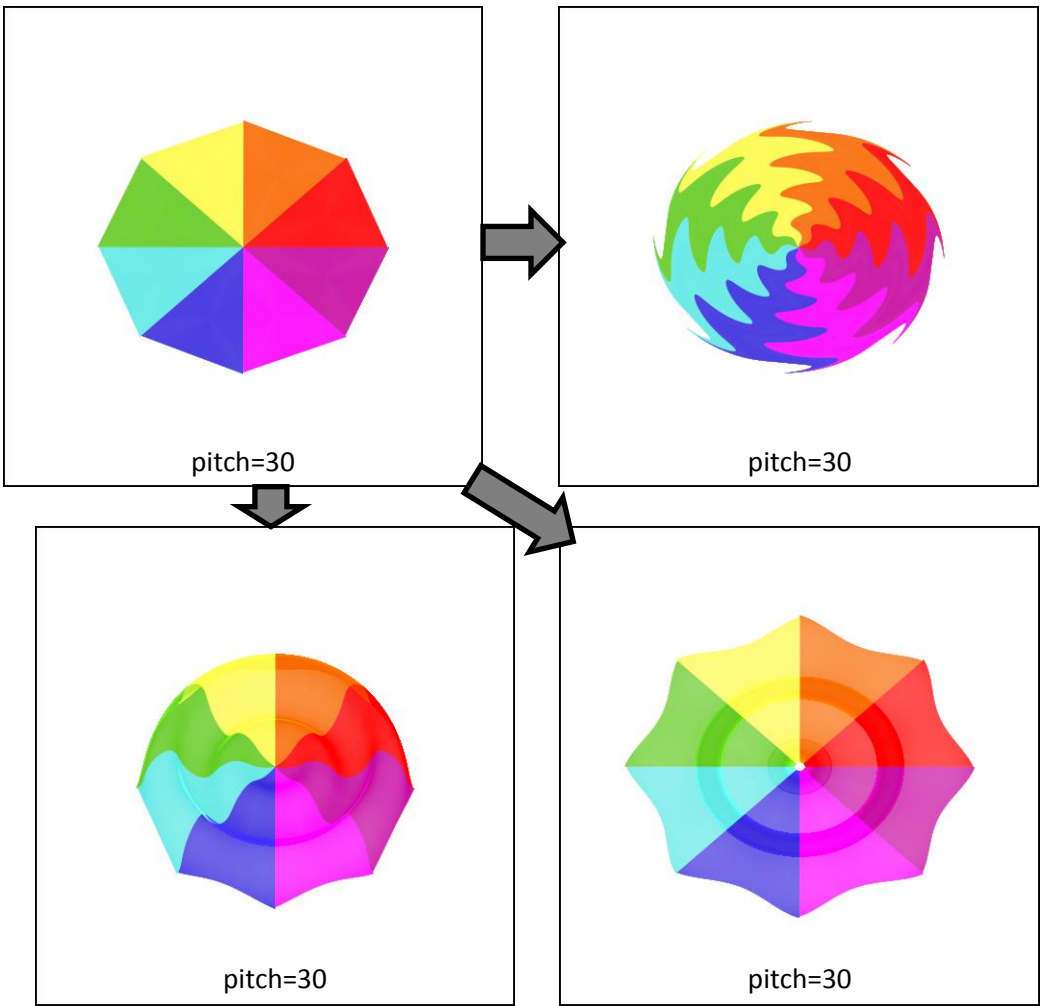
Applies a combination of three wave functions: theta is circular around the origin, phi is up and down (z-axis), and r is in and out from the origin (which usually shows up as a density variation). The examples show them individually for clarity.

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

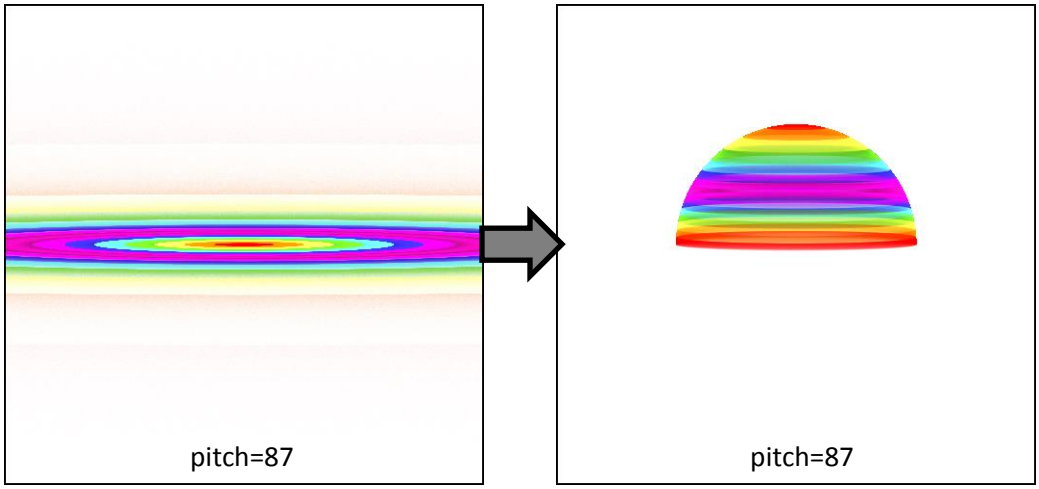


hemisphere (3D, sets z)

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

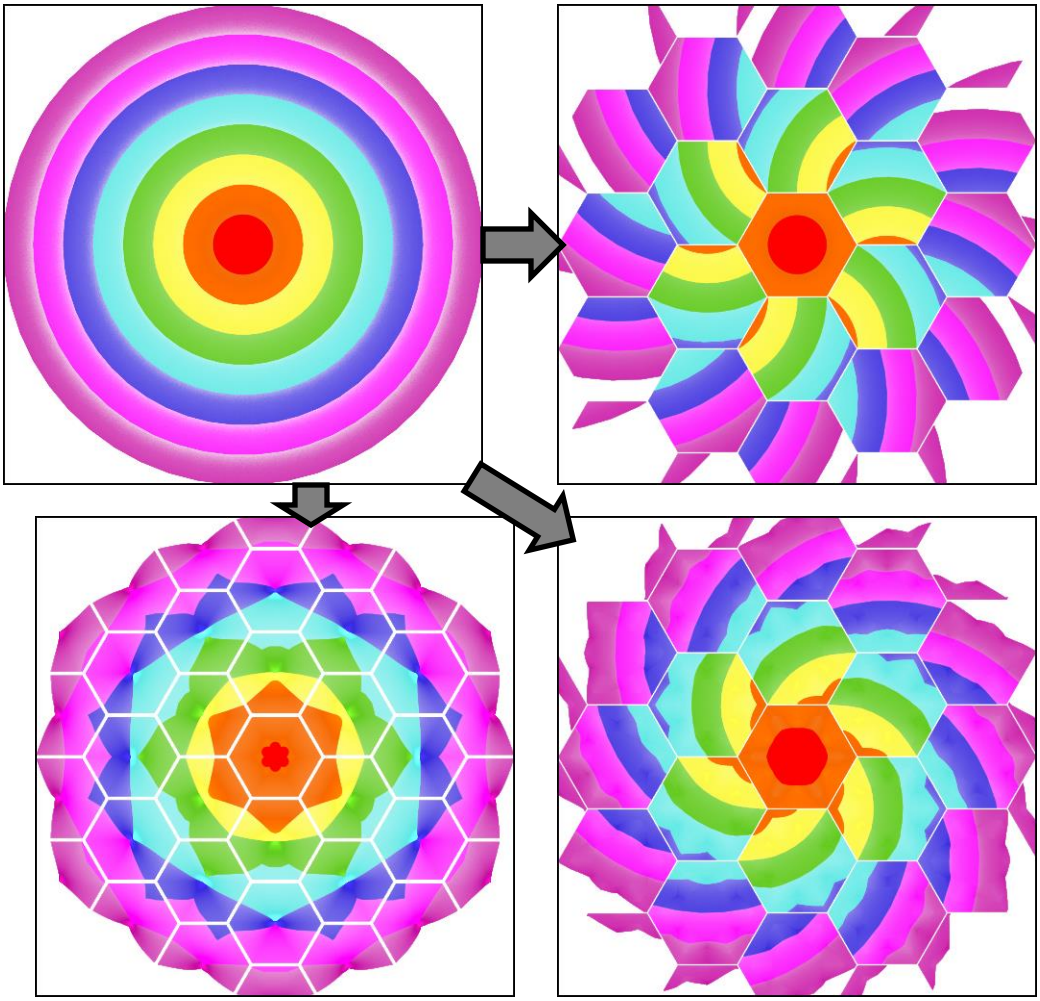
Divides the plane into hexagon tiles and rotates and scales each in place.

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*)

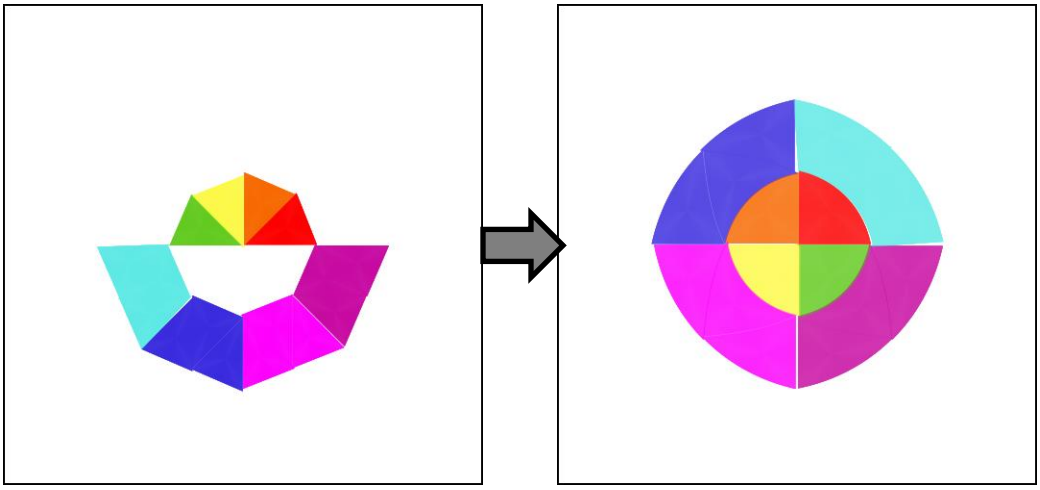


horseshoe (2D)

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.

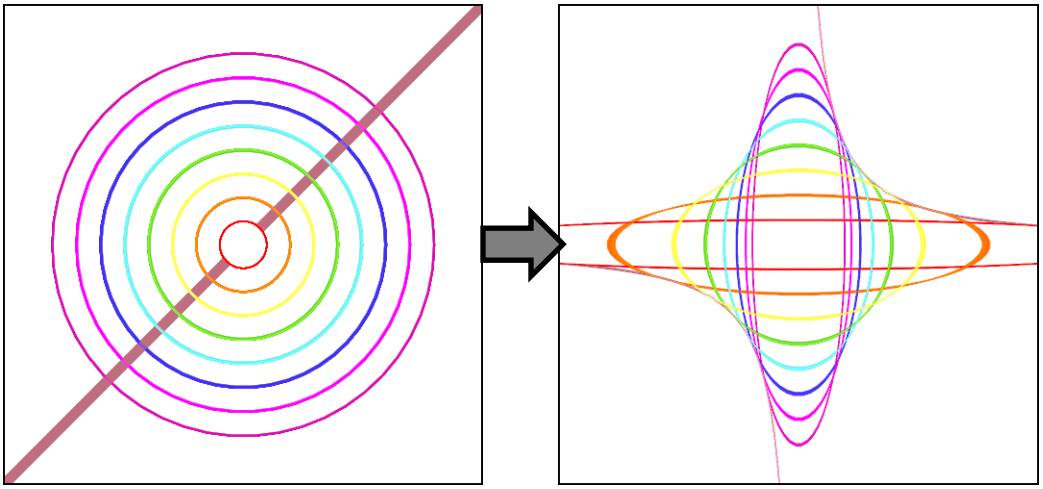
The name horseshoe may come because it transforms straight lines into U shapes.



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.



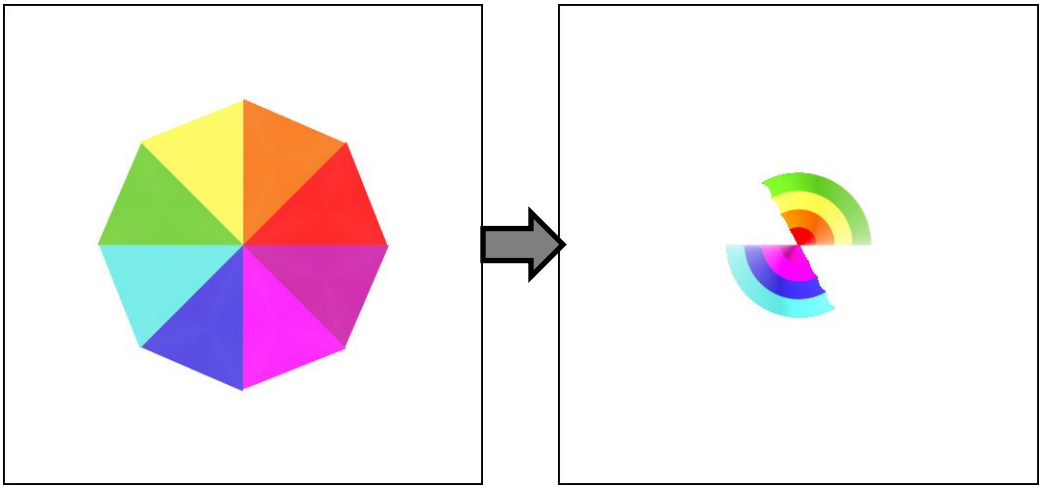
idisc (2D)

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.

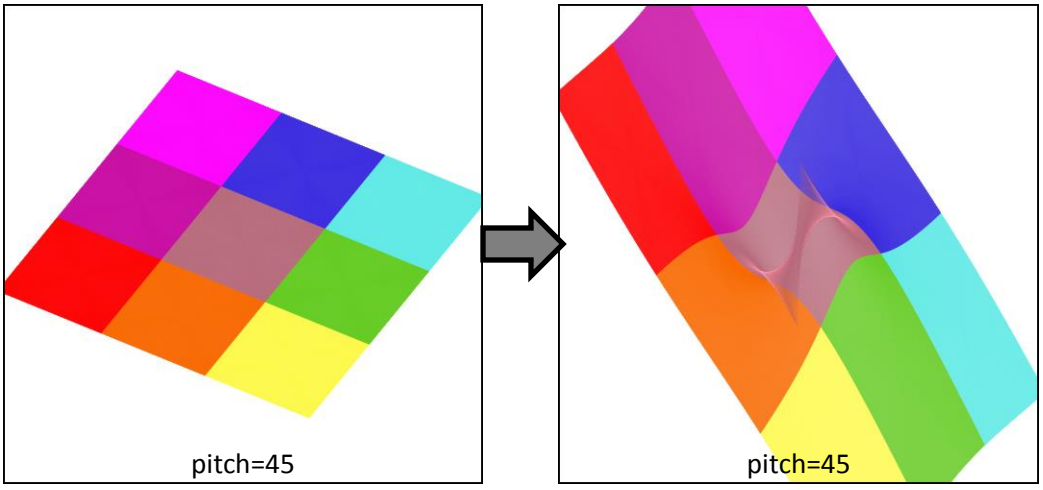


inflateZ_1 (3D, transforms z only)

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.

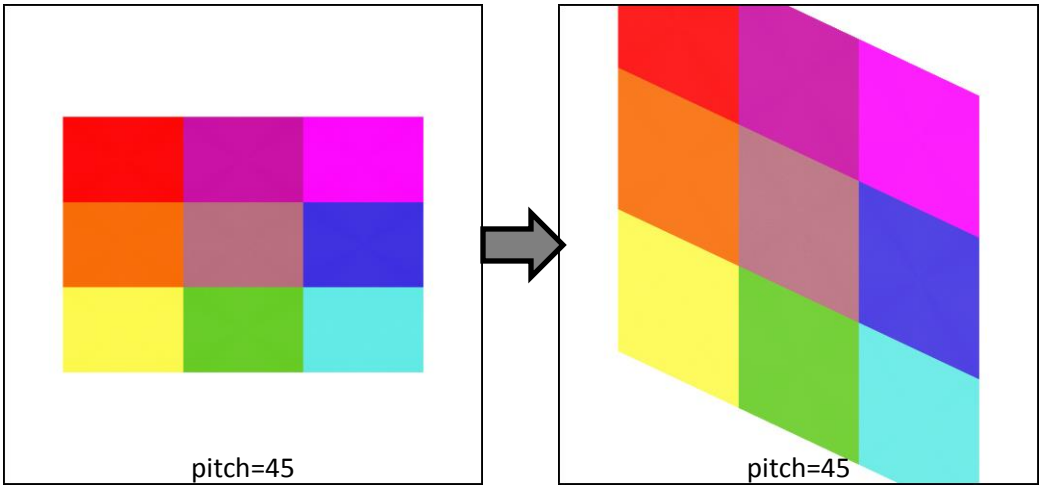


inflateZ_2 (3D, transforms z only)

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.

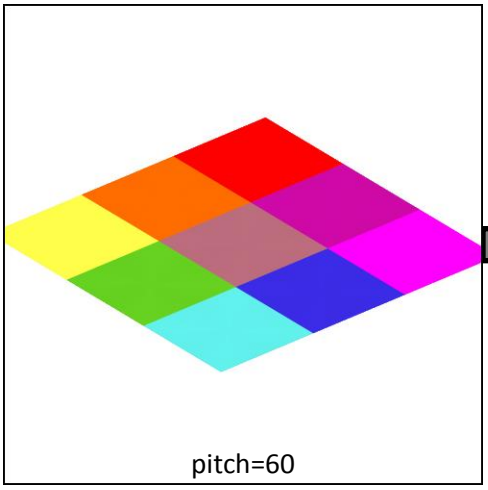


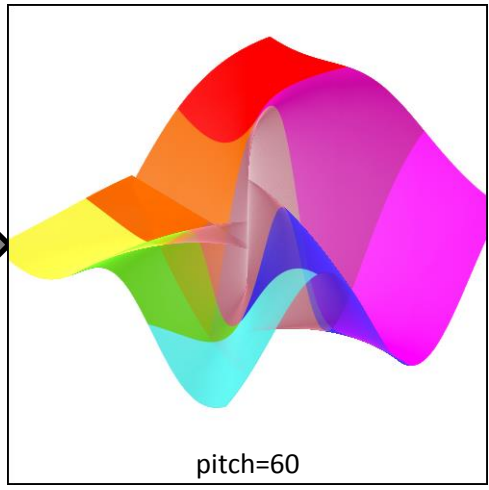
inflateZ_3 (3D, transforms z only)

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).



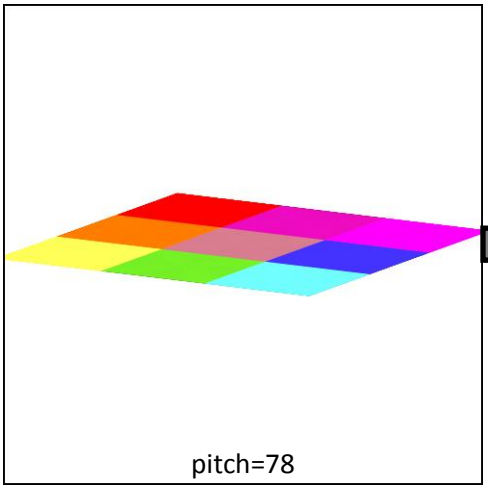


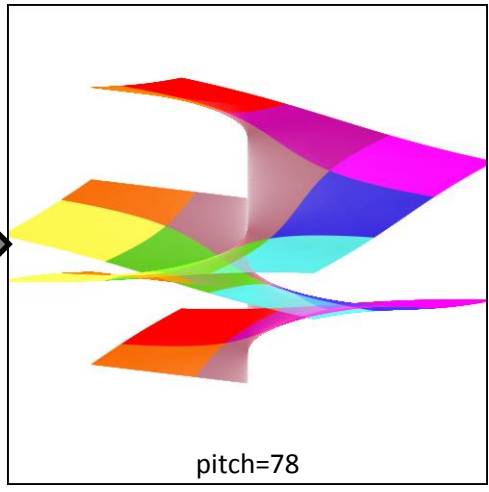
inflateZ_4 (3D, transforms z only)

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).



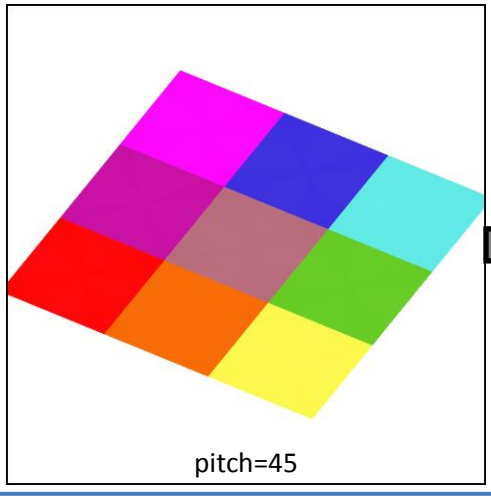


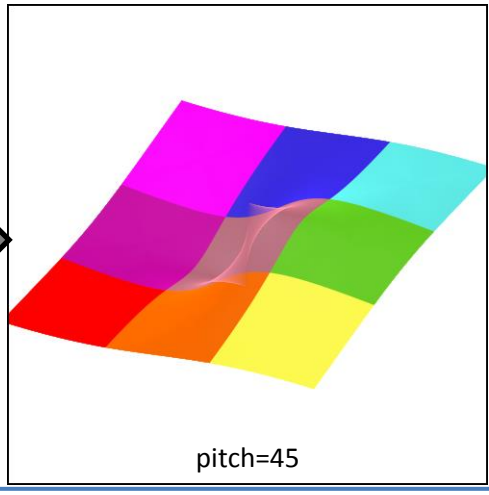
inflateZ_5 (3D, transforms z only)

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.



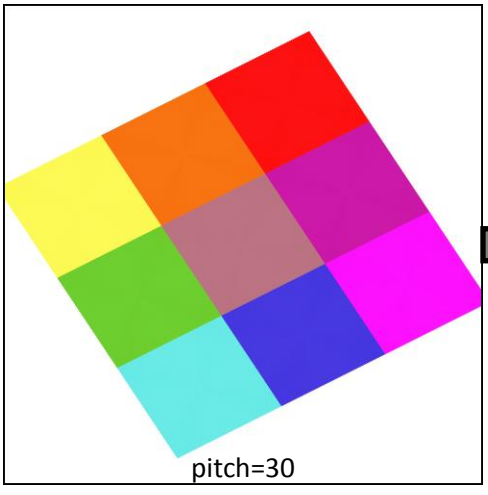


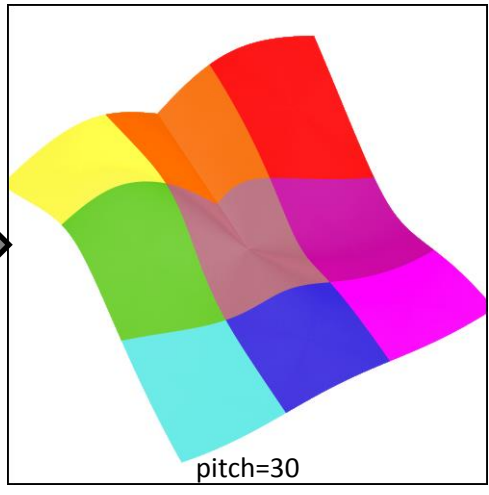
inflateZ_6 (3D, transforms z only)

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)

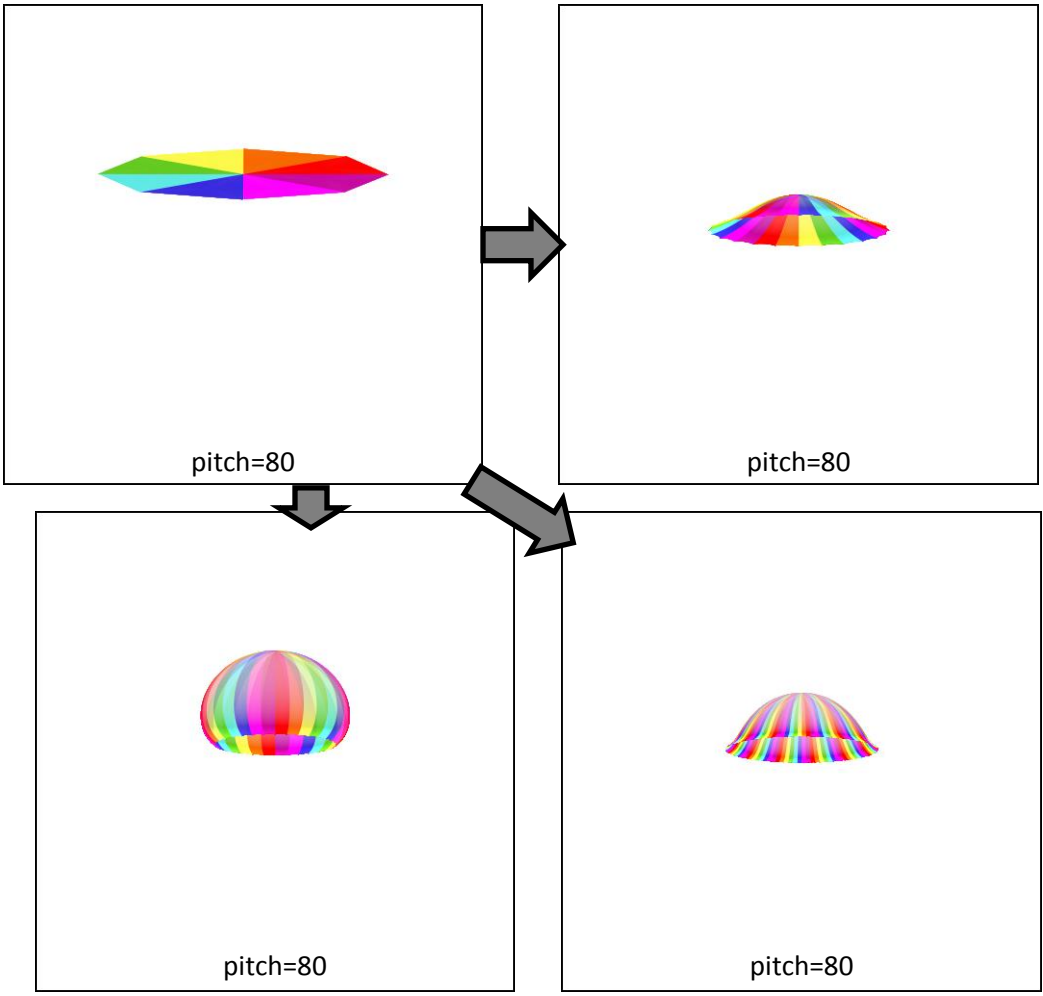
2.09	7X15B	7X16	jwf	ch
no	yes	yes	yes	no

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)

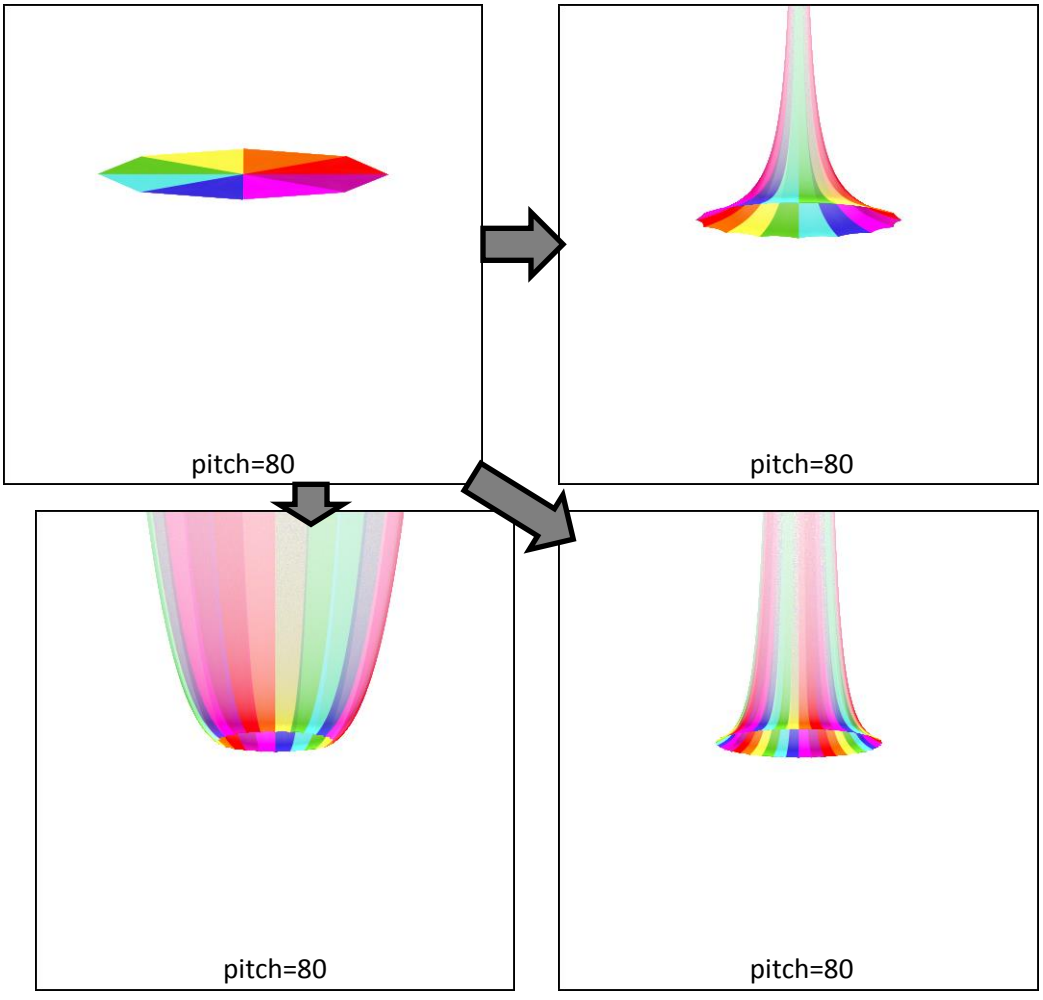
2.09	7X15B	7X16	jwf	ch
no	yes	yes	yes	no

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

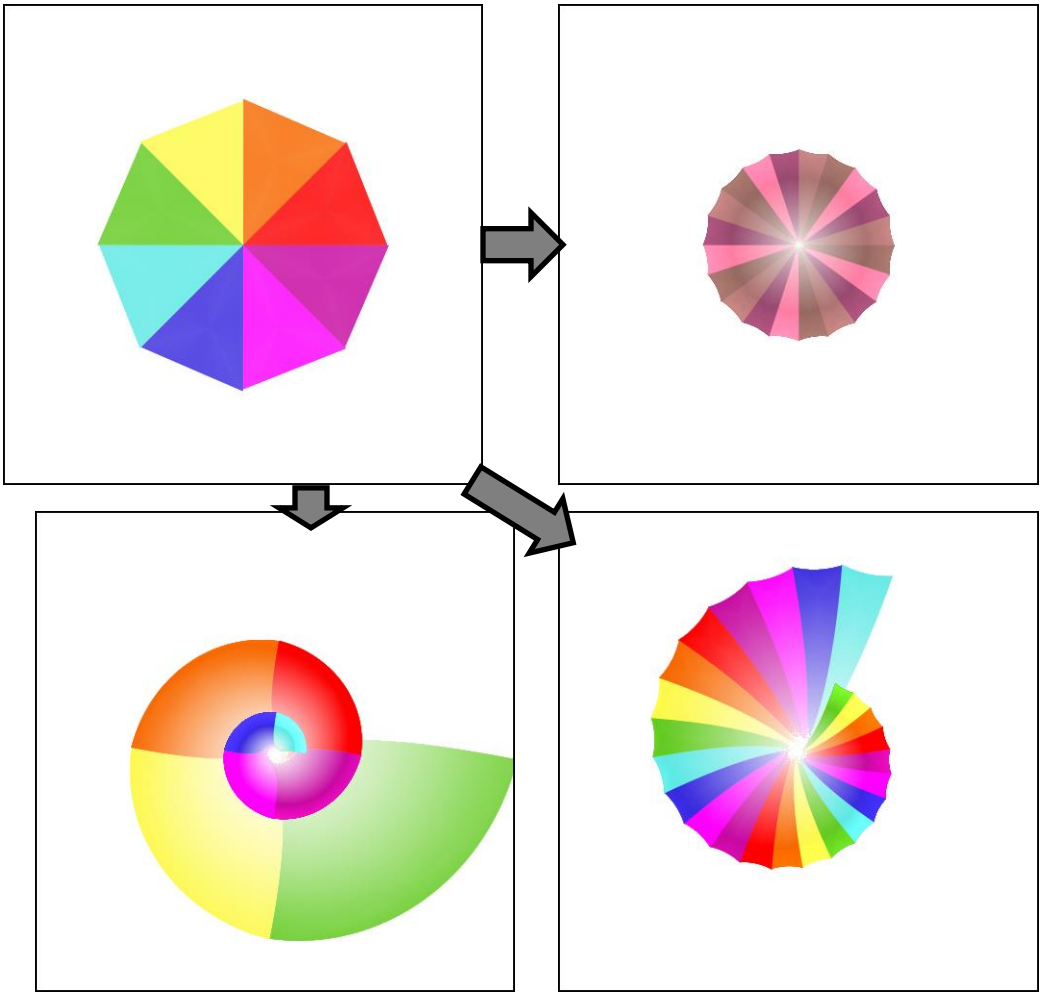
Like julian, but the *julian_power* variable is a complex number with real and imaginary parts (variables *re* and *im*) *Re* can't be zero; it doesn't need to be an integer, but the repetitions will overlap if it isn't, as shown in the top right example.

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

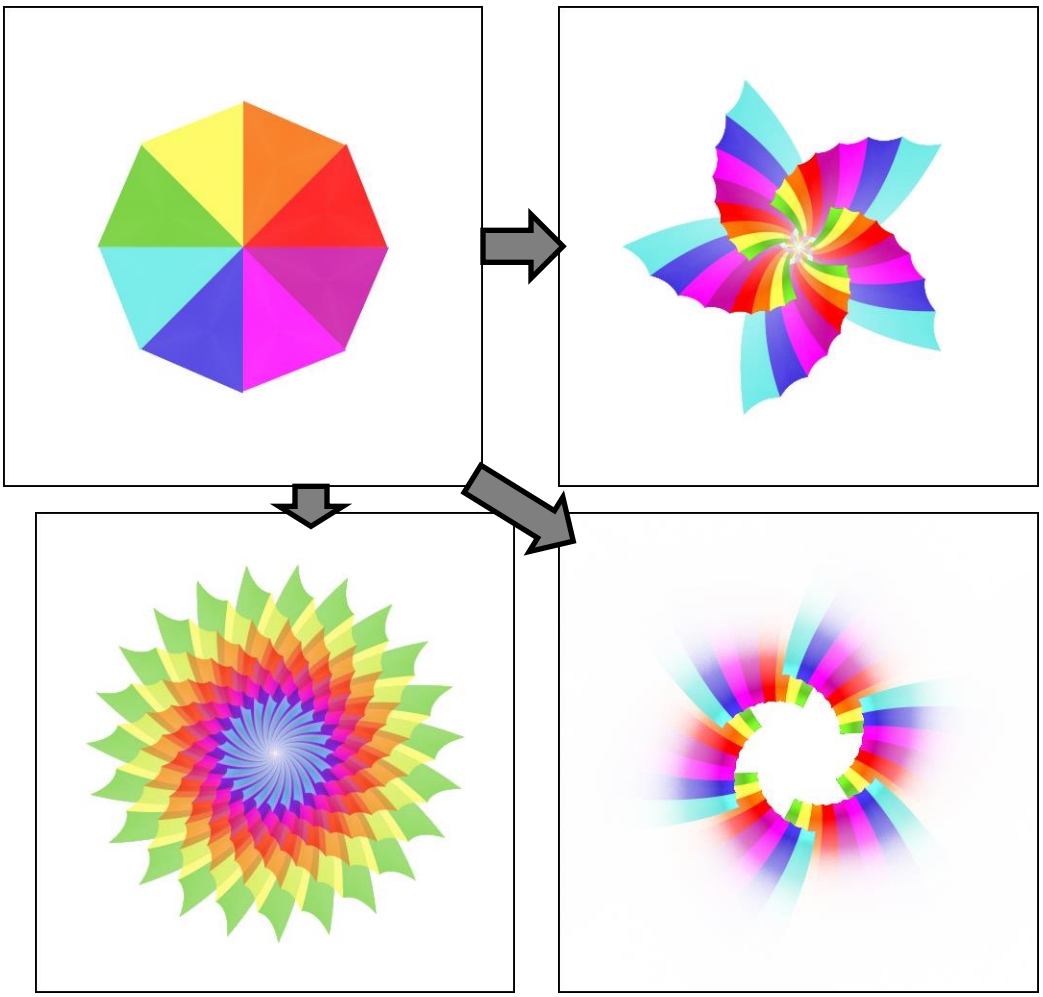
juliacomplex.dll

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.



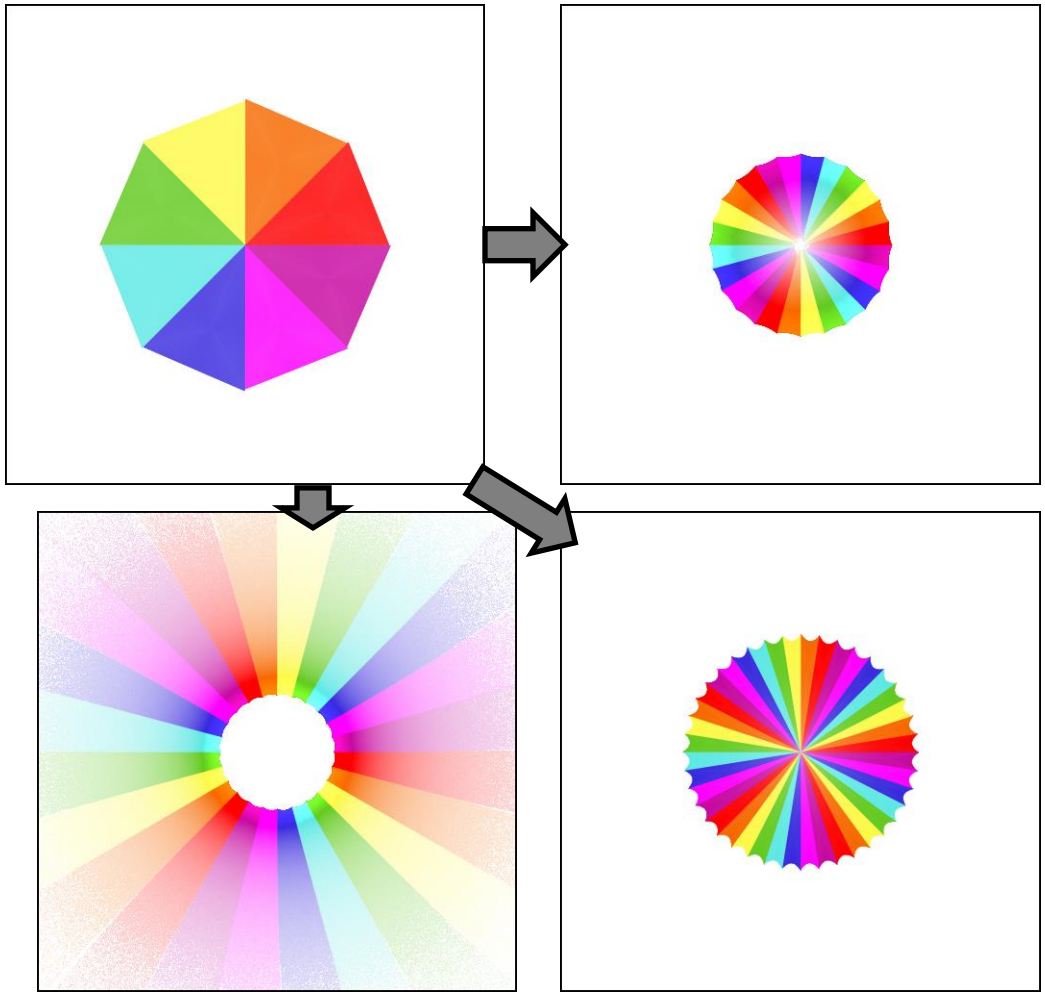
julian (2D)

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*.

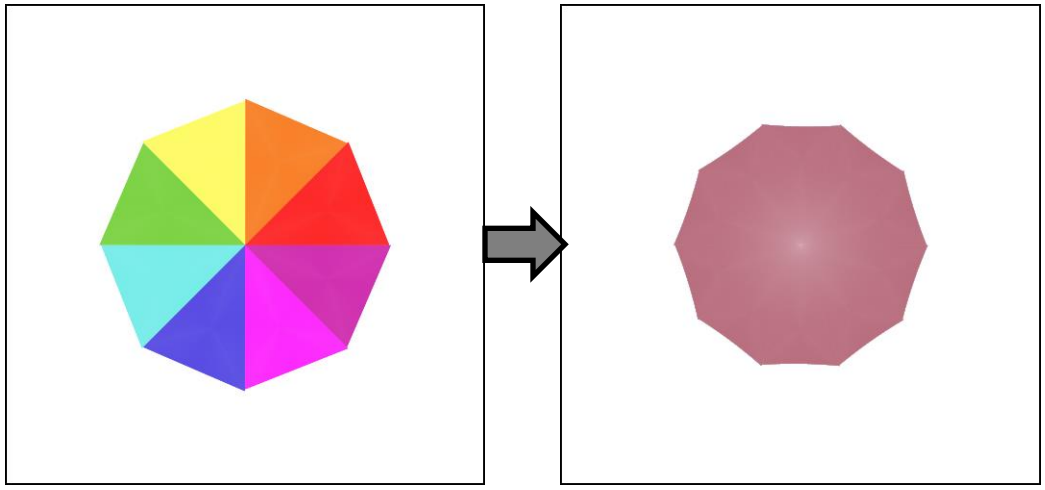


juliaq (2D)

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).

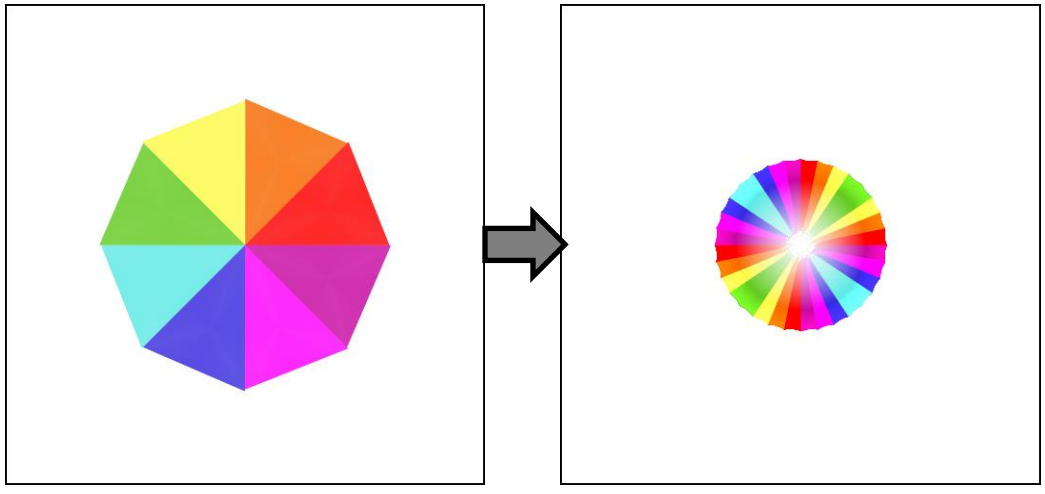


juliascope (2D)

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

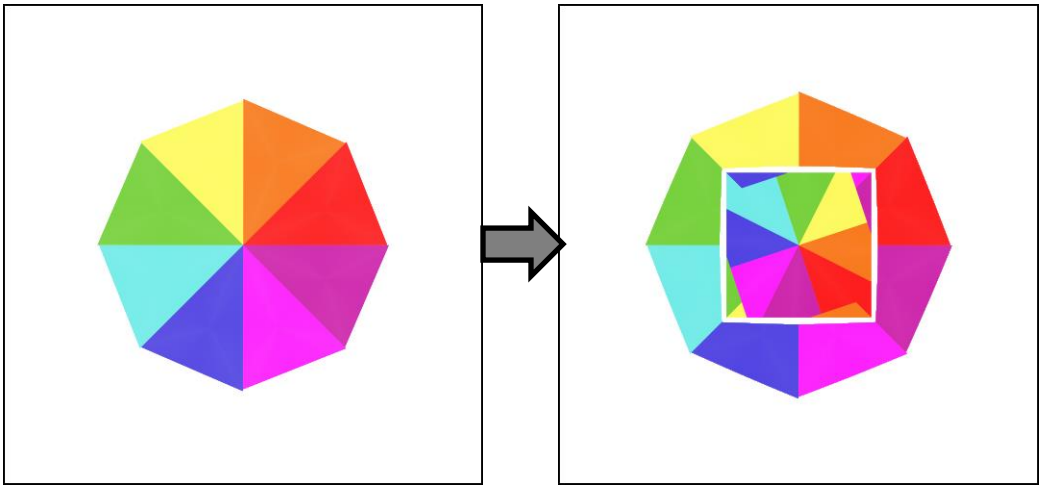


lazyjess (2D)

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)

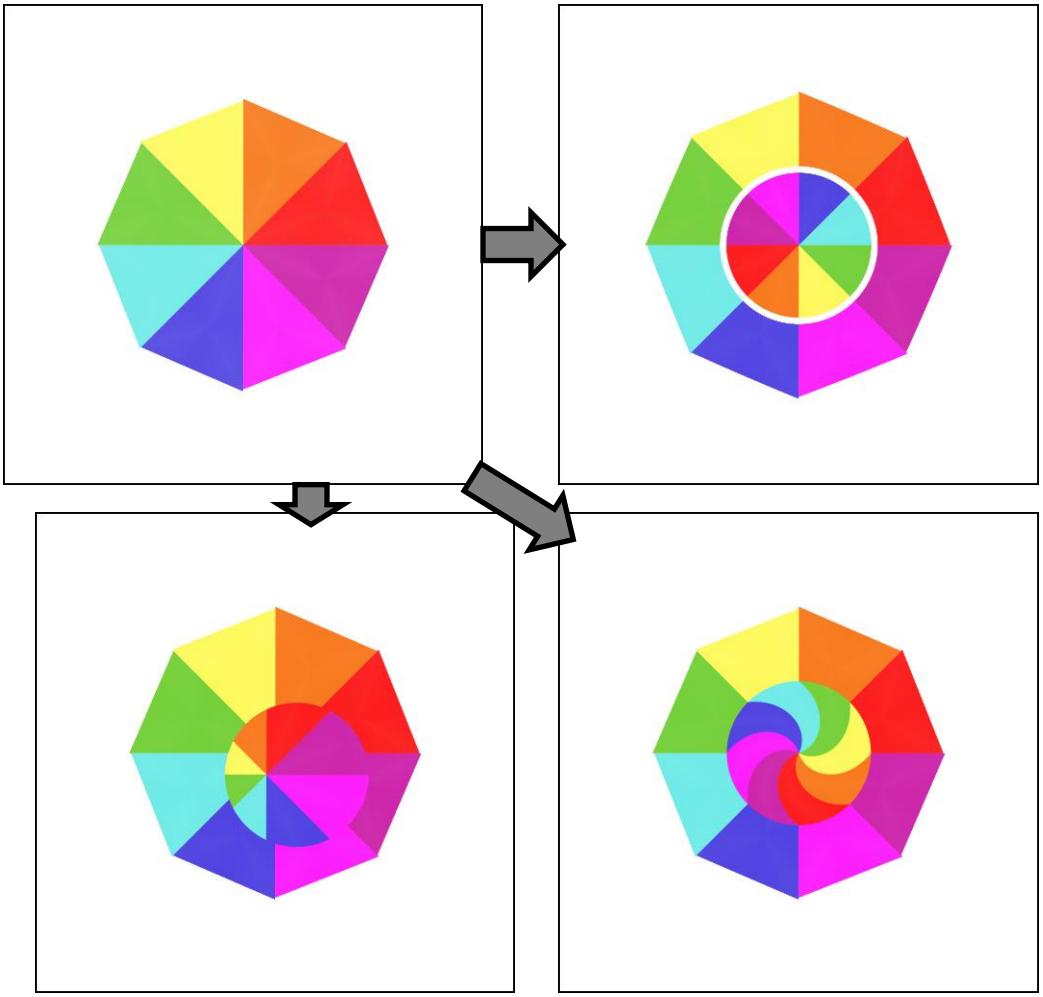
2.09	7X15B	7X16	jwf	ch
dll	dll	yes	yes	yes

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

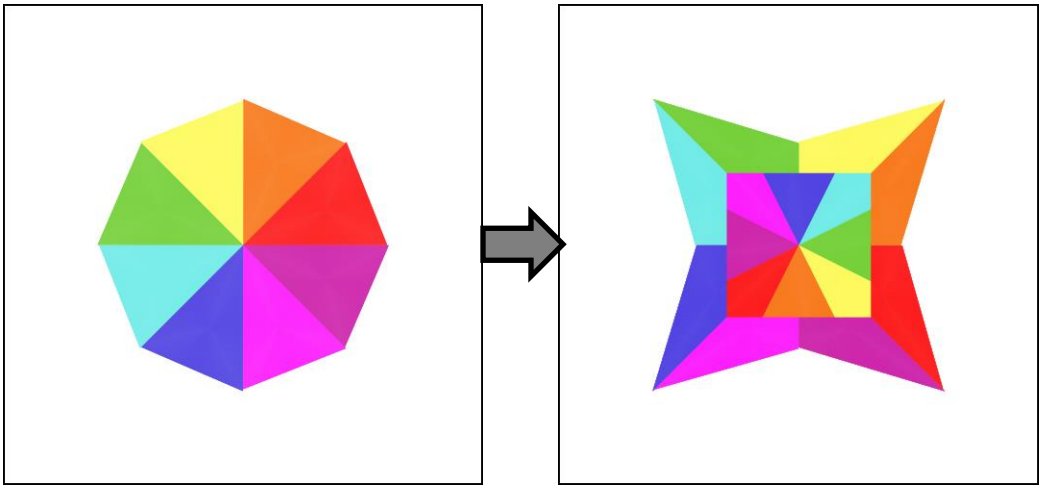


lazyTravis (2D)

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



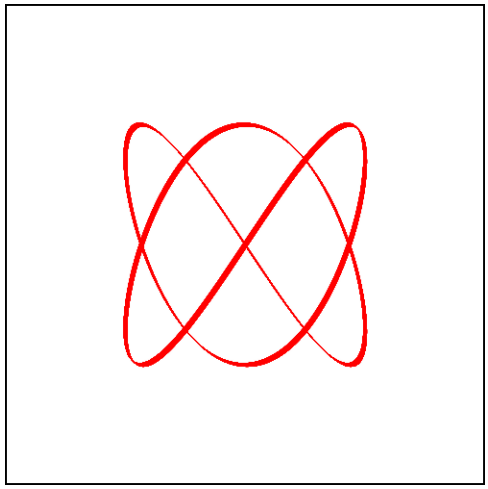
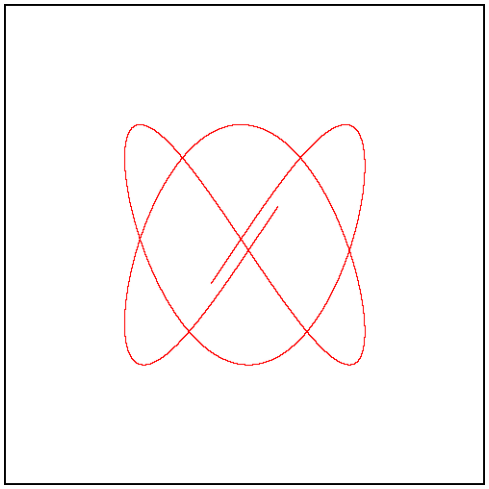
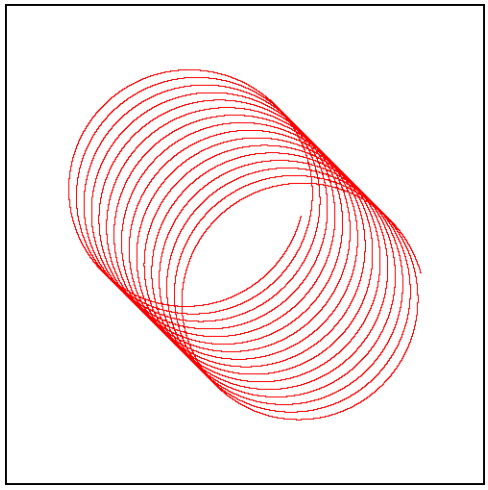
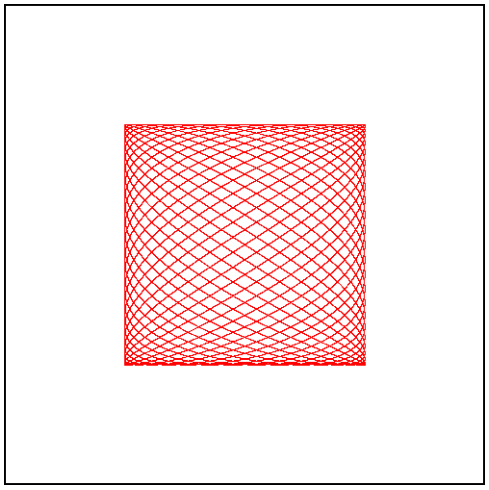
Lissajous (2D blur)

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.

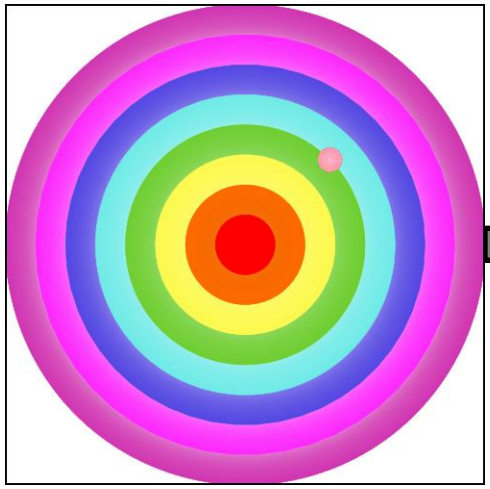
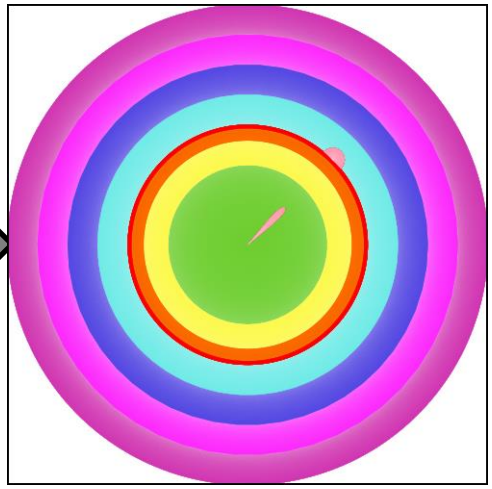
Top left: *tmin* = -3.14159, *tmax* = 3.14159, *a* = 2, *b* = 3, *c* = 0, *d* = 0, *e* = 0.03
Top right: *tmin* = -3.25, *tmax* = 3.25, *a*= 2.02, *b* = 3, *c* = 0, *d* = 0, *e* = 0
Bottom right: *tmin* = -30, *tmax* = 30, *a* = 3.125, *b* = 2, *c* = 0, *d* = 0, *e* = 0
Bottom left: *tmin* = -25, *tmax*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

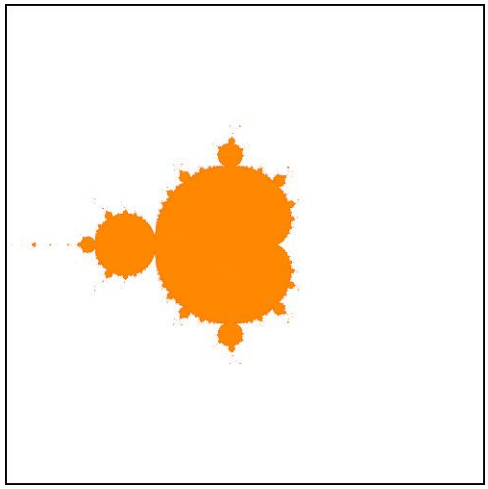
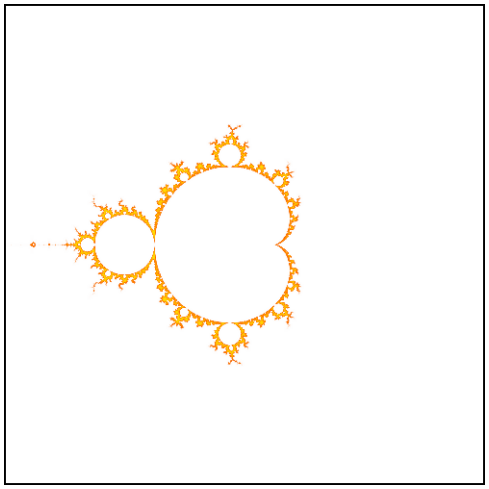



Mandelbrot (2D blur)

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.

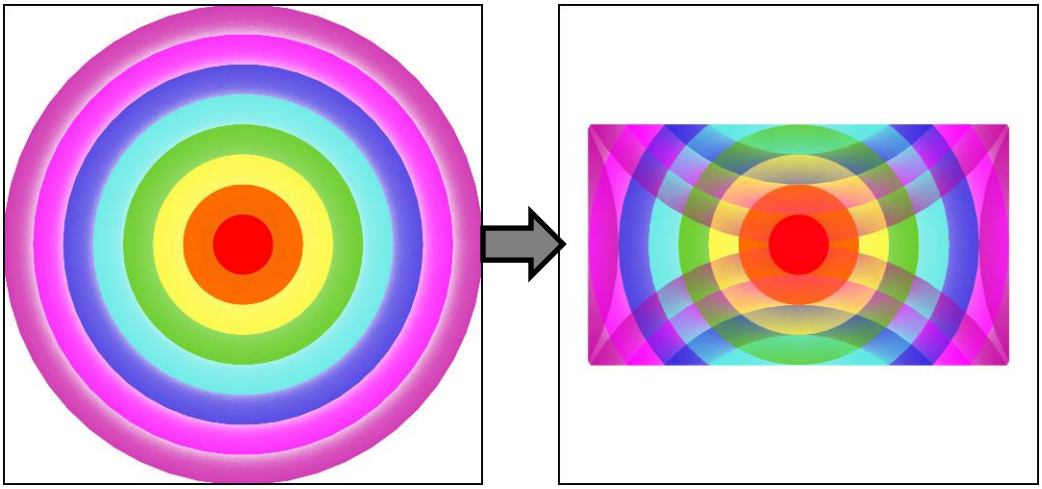
modulus (2D)

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

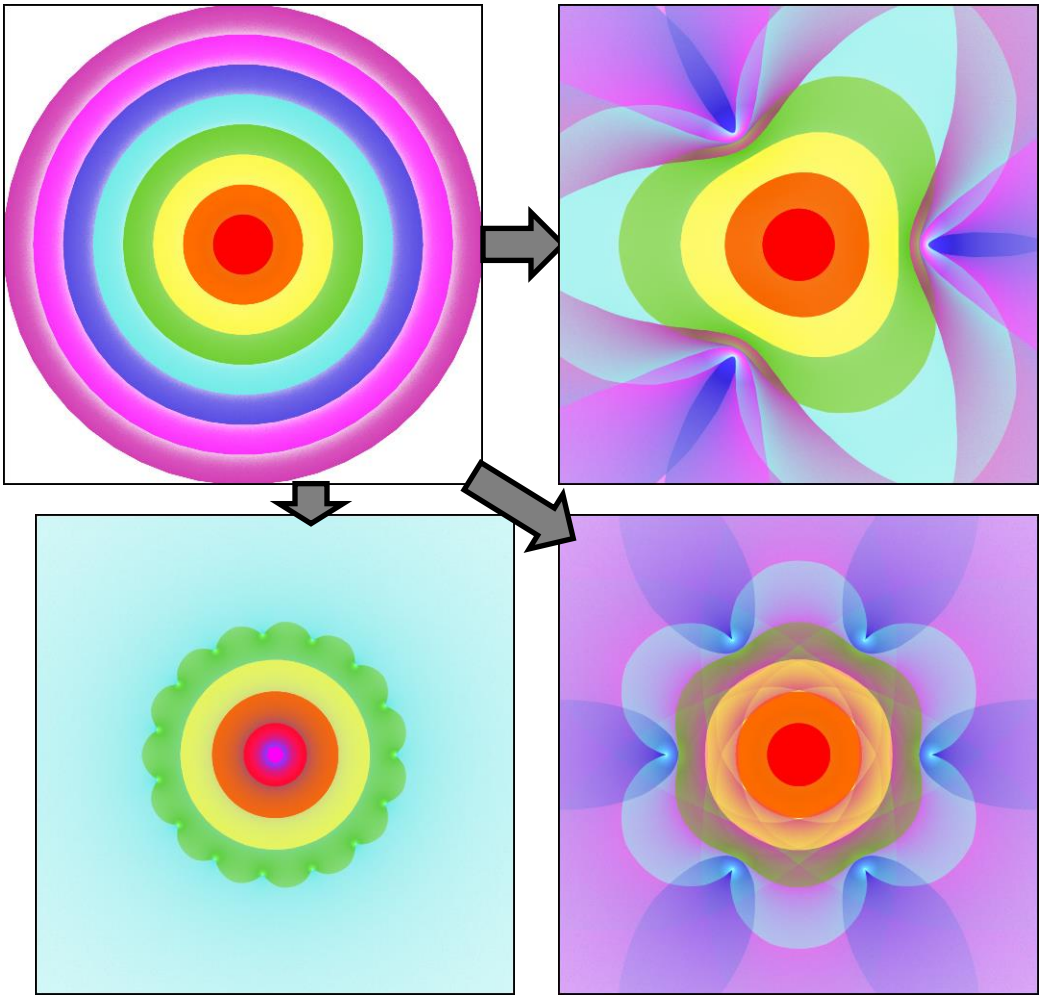
murl.dll
Based on curl. Think of the plane as a series of coincentric rings, divided into *power* lobes. As *c* increases, the spaces between the lobes pinch in and the lobes stretch out and rotate around until they cross and converge in the center.

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.

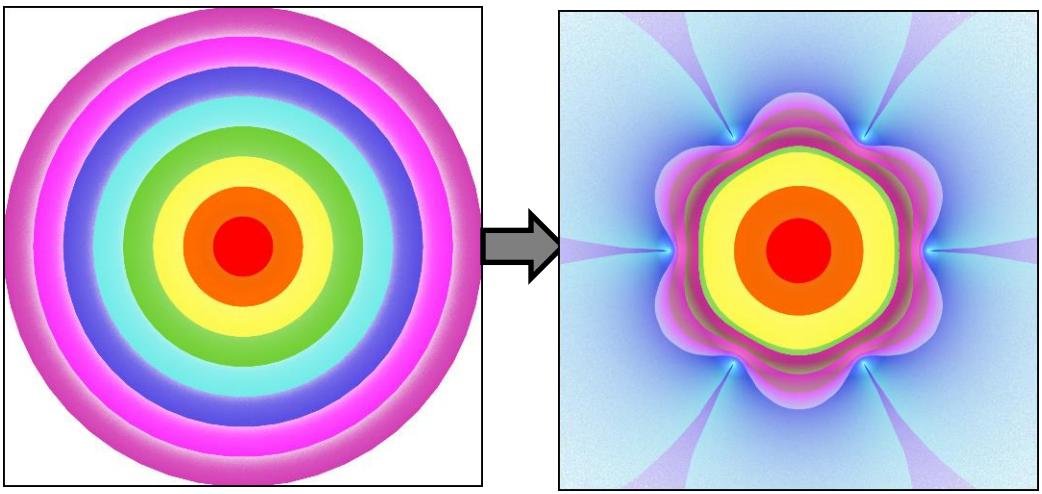


murl2 (2D)

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

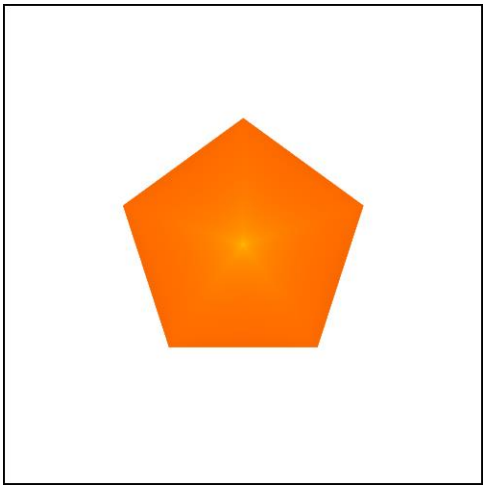
Generates regular polygons with quite a few variables to control various options such as stripes and a hole in the center. The examples show some of the possibilities; the particular variables each exemplifies are in bold.

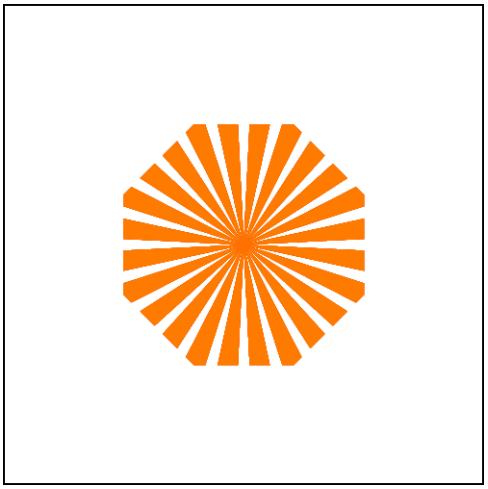
Top left: *numEdges* = 5, *numStripes* = 0, *ratioHole* = 0, *circumCircle* = 0, and ***equalBlur* = 0**

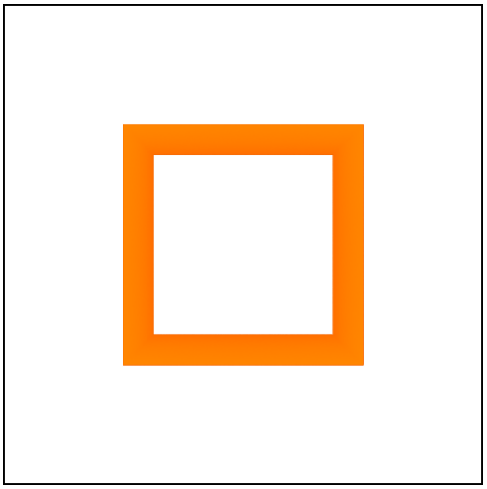
Top right: *numEdges* = 8, ***numStripes* = 3, *ratioStripes* = 1.25**, *ratioHole* = 0, *circumCircle* = 0, and *equalBlur* = 1

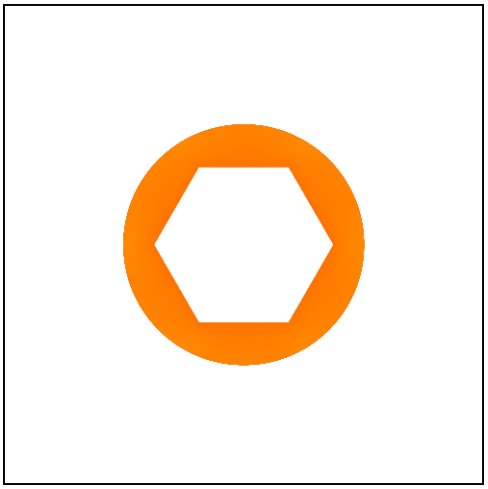
Bottom left: *numEdges* = 4, *numStripes* = 0, ***ratioHole* = 0.75**, *circumCircle* = 0, and *equalBlur* = 1

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

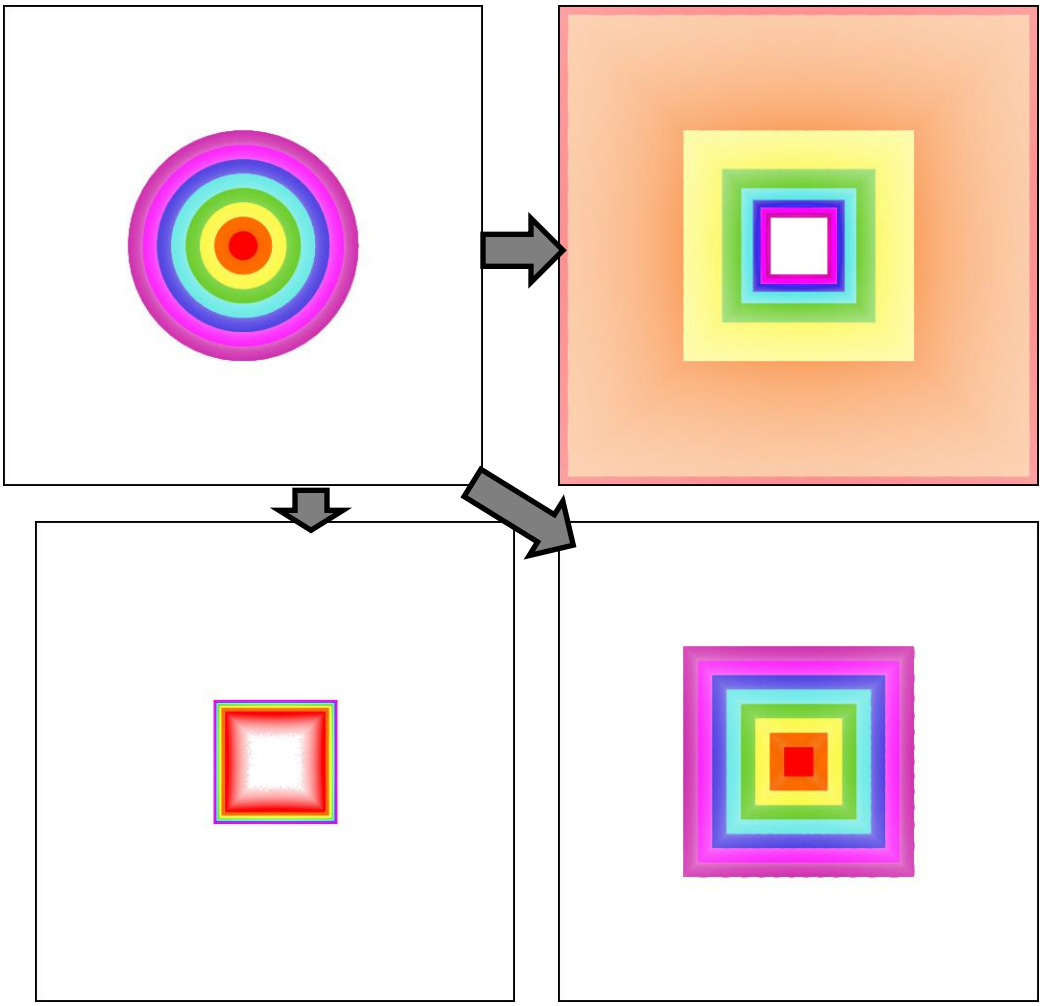
Stretches circles into polygons, with the option to turn them inside-out (like spherical).

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).

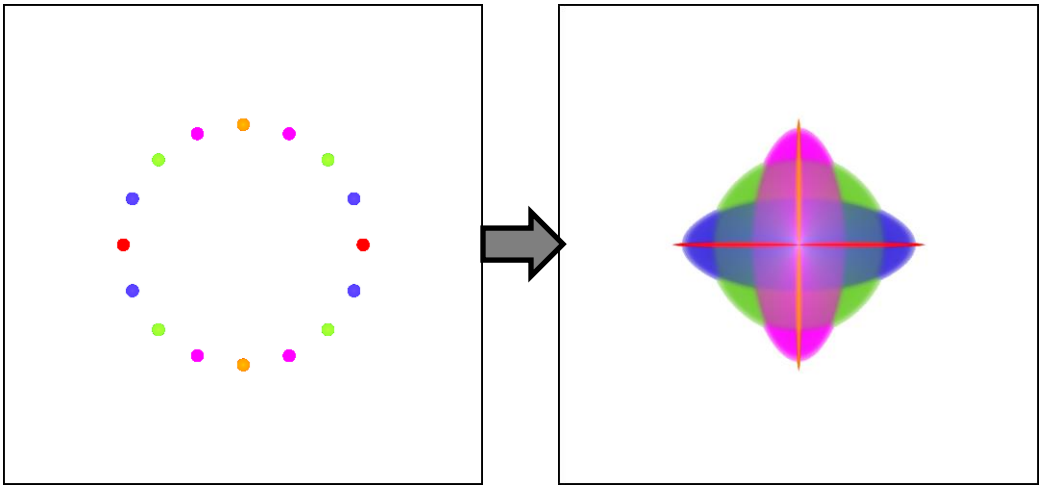


noise (2D half blur)

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

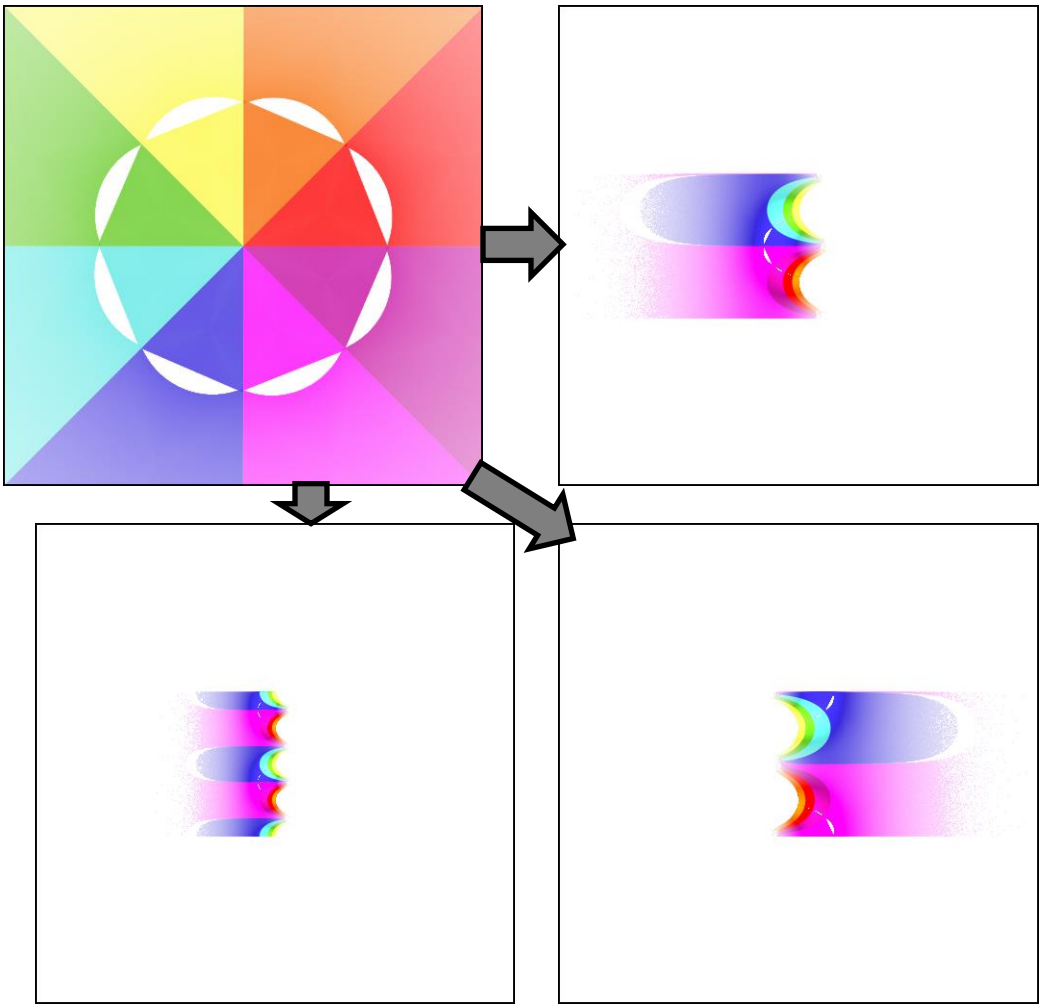
npolar.dll

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.

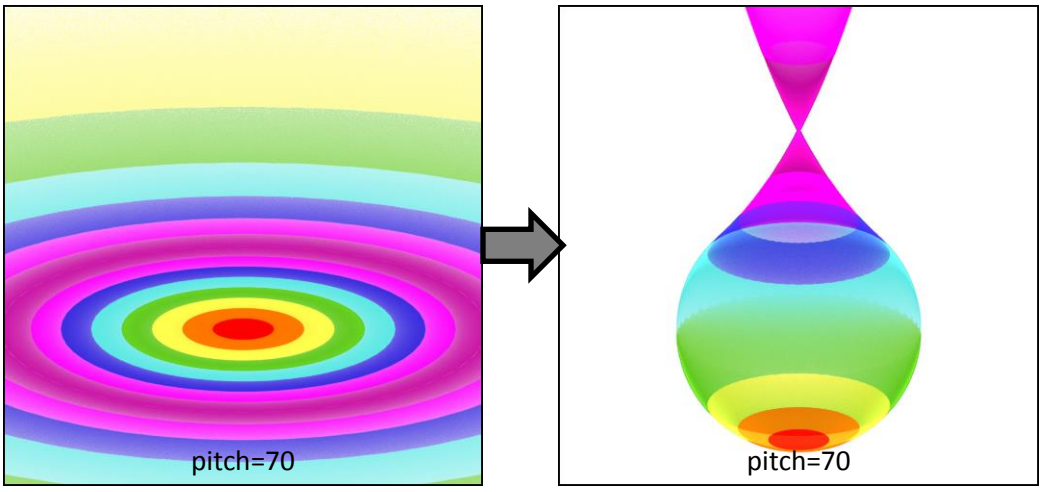


onion (3D, transforms z)

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).



ortho (2D)

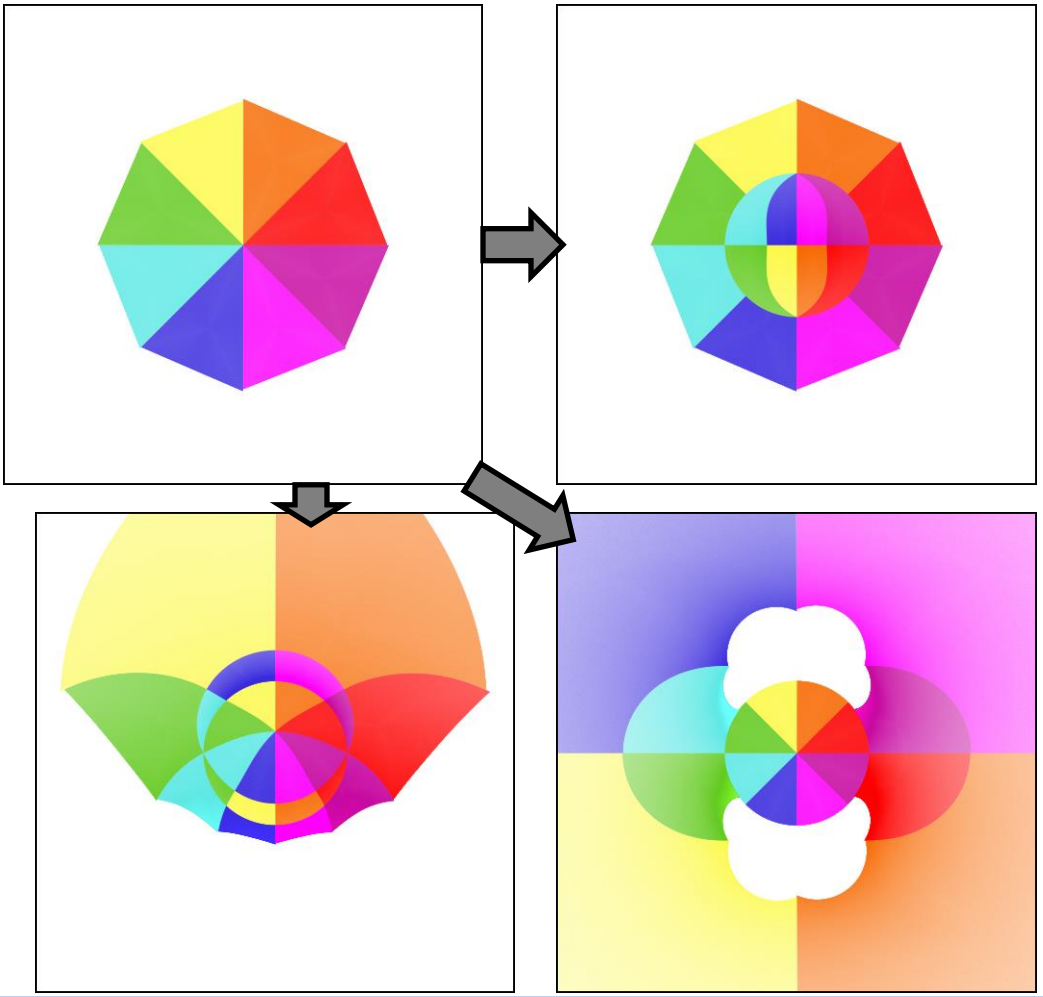
2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	yes

ortho.dll

Variable *in* ranges from 0 to 2, and controls shifting of the unit circle. When 0 or 2, there is no effect. As it increases from 0, the circle shifts down, with the bottom looping back around to the top. When it decreases from 2, the circle shifts up. A value of 1 (top right) gives maximum effect.

Variable *out* does the same to the rest of the plane, but in opposite direction, and as it shifts it turns inside-out. Again, maximum effect is at 1 (bottom right).

Top right: *in* = 1, *out* = 0
Bottom right: *in* = 0, *out* = 1
Bottom left: *in* = 1.5, *out* = 0.5



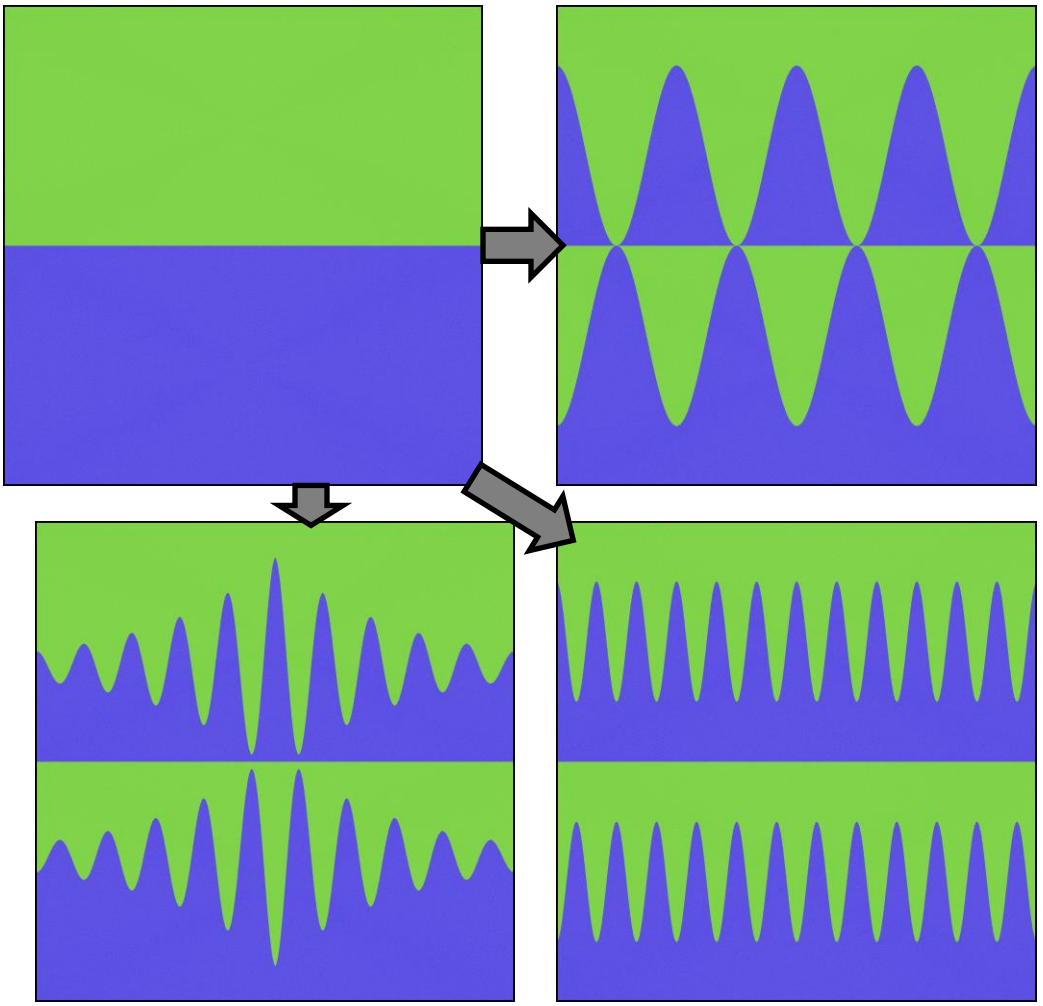
oscilloscope (2D)

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

oscilloscope.dll
Flips the plane in a sine wave shape across the x axis.

frequency is the frequency of the wave; higher values give more waves.
amplitude specifies the height of the wave.
separation specifies how far the top and bottom parts are separated from each other. When equal to *amplitude* (top right), the top and bottom halves barely touch. When higher (bottom right), there is space between the two halves.
damping, when 1, damps the wave on both sides (bottom left).

Top right: *separation* = 0.75, *frequency* = 1, *amplitude* = 0.75, *damping* = 0
Bottom right: *separation* = 1, *frequency* = 3, *amplitude* = 0.5, *damping* = 0
Bottom left: *separation* = 0.8, *frequency* = 2, *amplitude* = 0.9, *damping* = 1

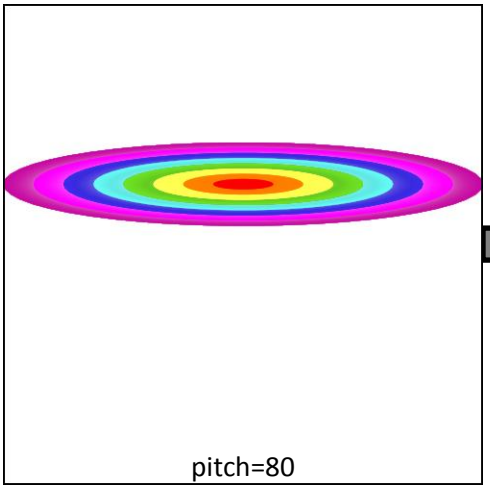


ovoid3d (3D, transforms z)

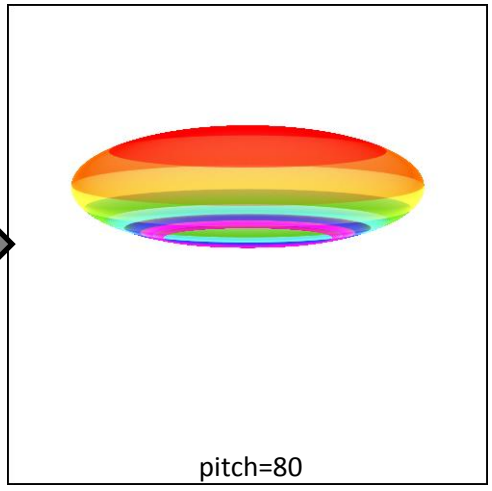
2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	no

ovoid3d.dll

Like spherical3d, but has variables to scale x, y, and z. The example is raised so that z is 0.5, and has variables *x* = 1.5, *y* = 1, and *z* = 0.5.



pitch=80

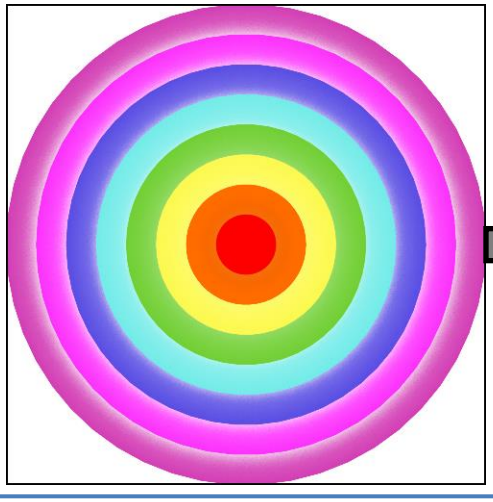


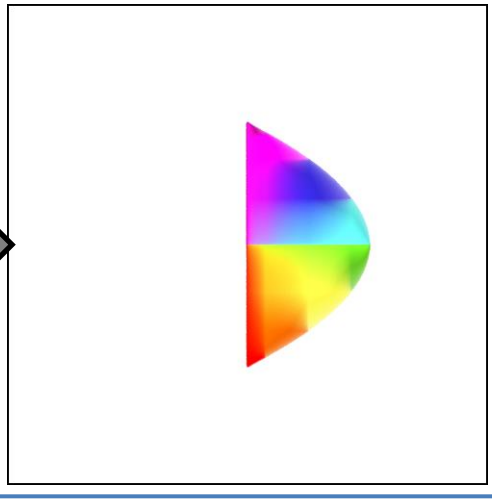
pitch=80

parabola, parabola2 (2D half blur)

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

parabola2.dll
Generates a parabola shape, using the distance from the center of existing points as a parameter. To show the whole parabola, the input was scaled to set the outer edge of the circle to π. Variables *width* and *height* are both 1 for this example.
The built-in version is named parabola; the plugin version is named parabola2.

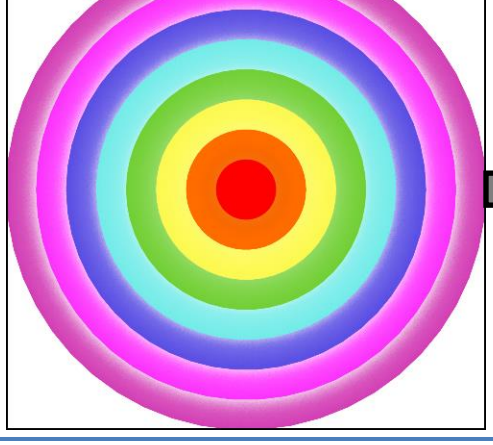


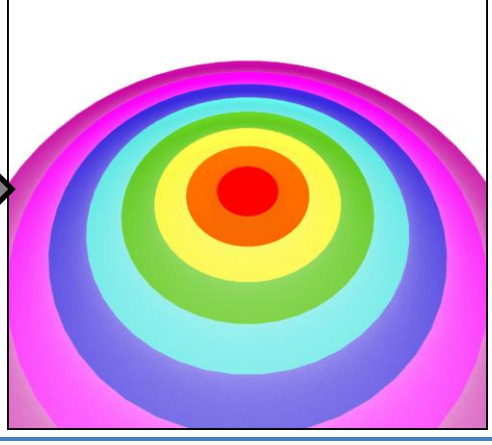


perspective (2D)

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

A 2D variation that transforms the plane as if viewed from a 3D vantage point. The variable *angle* controls the viewing angle, from 0 (face on) to 1 (edge on).



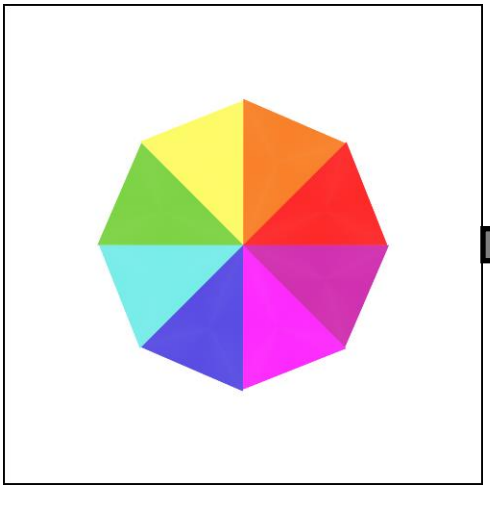


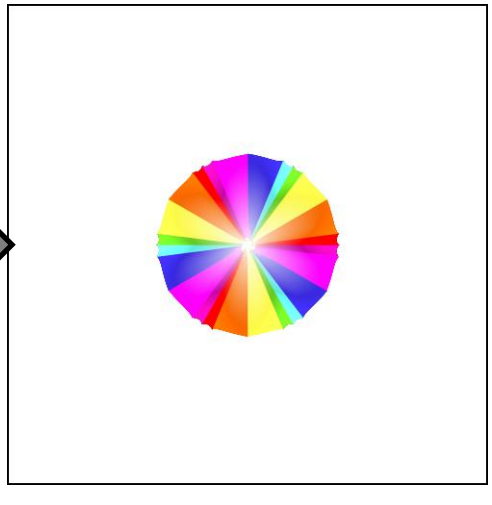
phoenix_julia (2D)

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

PJulia.dll

Works the same as julian, except there are two variables, *x_distort* and *y_distort* that change the proportion of different wedges with relation to each other. The example shown here has *x_distort* = 1.5 and *y_distort* = 0.

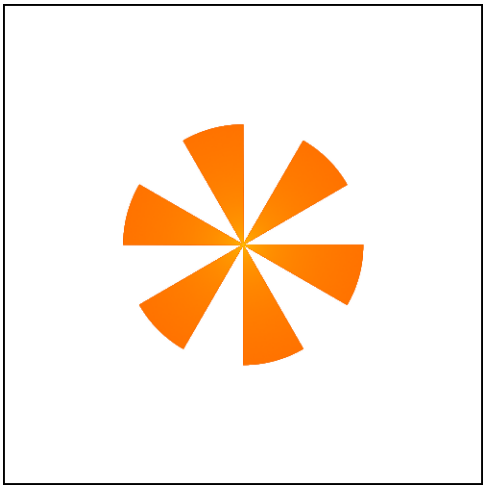


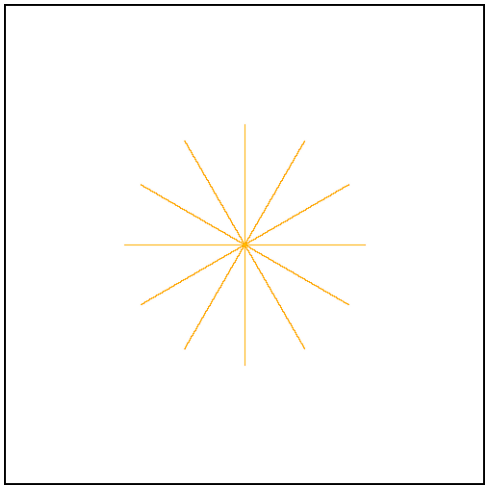


pie, pie_fl (2D blur)

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

pie.dll, pie_fl.dll
Generates a figure with *slices* wedges separate by spaces (pie.dll requires *slices* to be an integer; the others do not). The proportion of wedge to space is set by *thickness*, which can range from 0 (lines, right) to 1 (solid circle, same as blur); left example is 0.5. Variable *rotation* (in radians) allows the figure to be rotated.





pie3D (3D blur)

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

Generates a figure like pie, but with the wedges curved . Same as pie when pitch = 0.

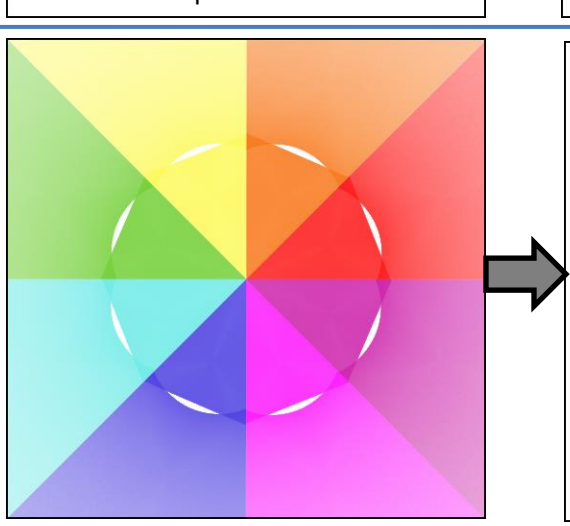
The examples both have *slices* = 8, *thickness* = 0.75, and *rotation* = 0. The one on the left has a variation value of 1; the one on the right has a value of 3 to show the undulating 3D shape.

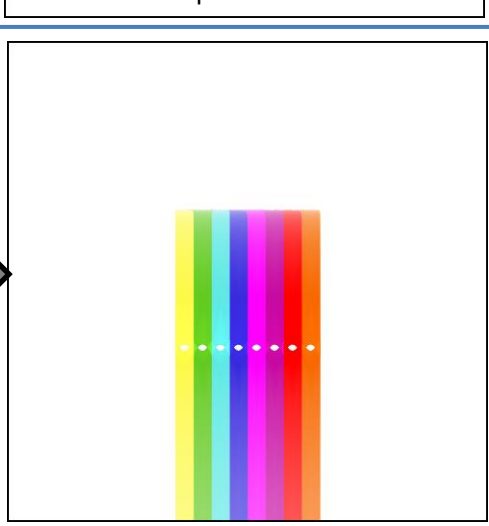
polar (2D)

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

Converts each point of the plane to polar coordinates (angle and distance), then treats them as rectangular coordinates, mapping the angle to x and the distance to y.

Wedges are thus converted to vertical stripes (as shown), and rings are converted to horizontal stripes (not shown).



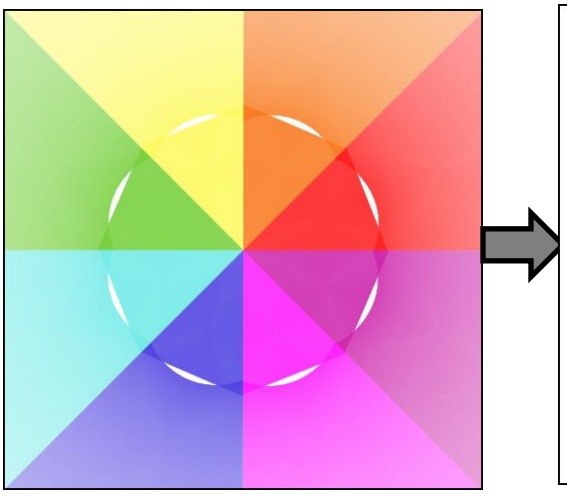


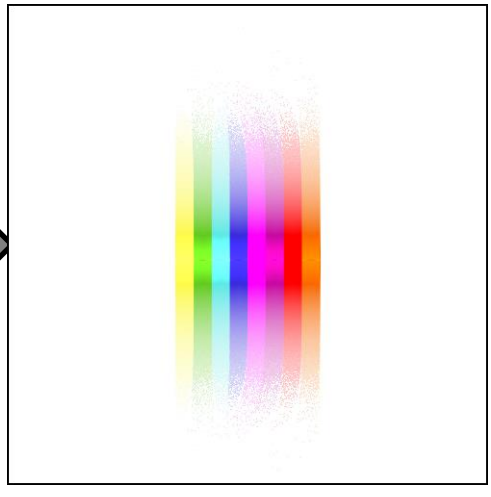
polar2 (2D)

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

polar2.dll

Similar to polar, but uses slightly different math to make the vertical extant symmetrical . (Specifically, the log of the distance is used for y instead of the raw distance.)





popcorn2 (2D)

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

popcorn2.dll

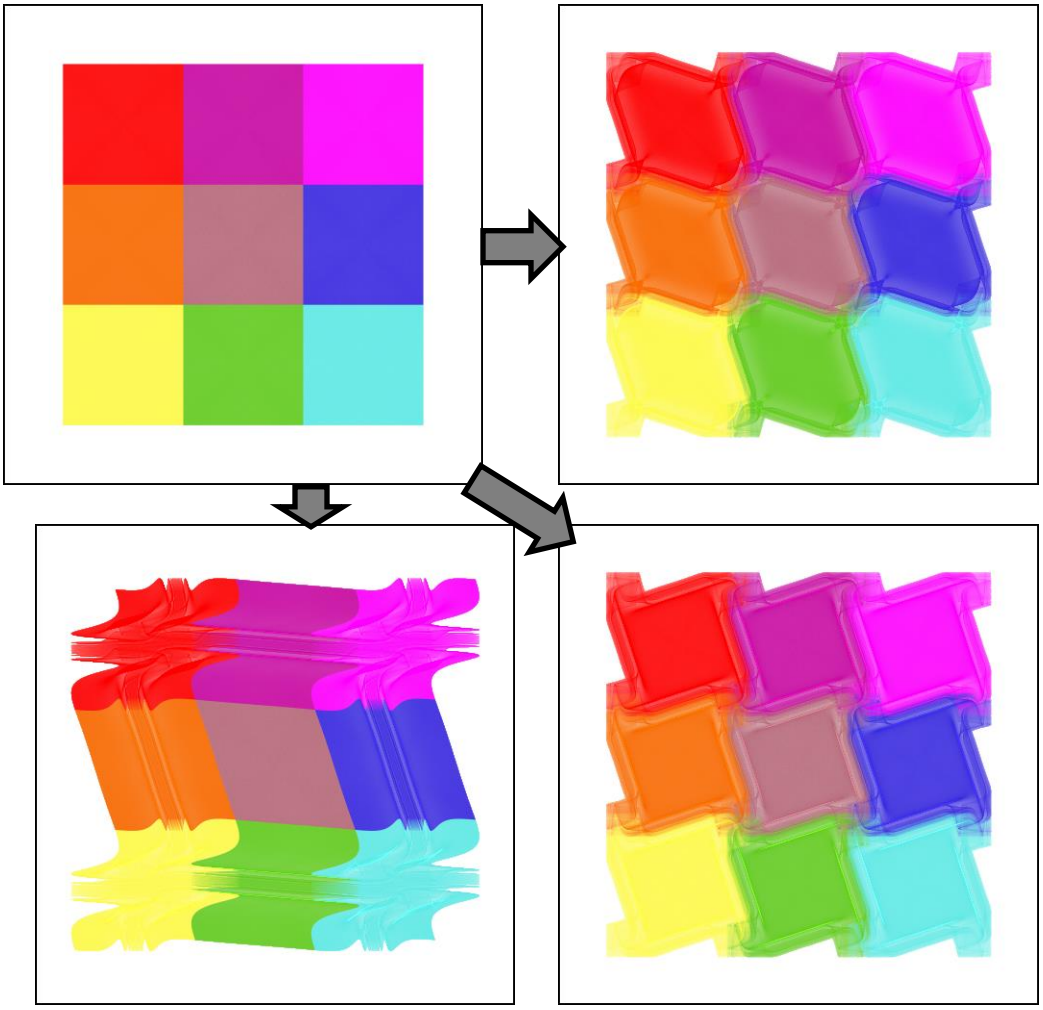
Puts horizontal and vertical jags at the sides of each square in the plane, creating a unique shape. Variables *x* and *y* control the size of the jags; larger values create larger jags. Variable *c* controls the size of the squares; larger values work with smaller squares.

Two types of shapes can be made; depending on whether *x* and *y* have the same sign (top right) or different signs (bottom right).

The right examples have $c = \pi$ to specify squares of size 1. The bottom left has $c = \pi/2$ to specify squares of size 2, putting the distortions in the middle of the outer squares where they are more visible.

Top right: $x = 0.1$, $y = 0.1$, $c = 3.142$
Bottom right: $x = 0.1$, $y = -0.1$, $c = 3.142$
Bottom left: $x = 0.2$, $y = 0.05$, $c = 1.571$

Supercedes popcorn.



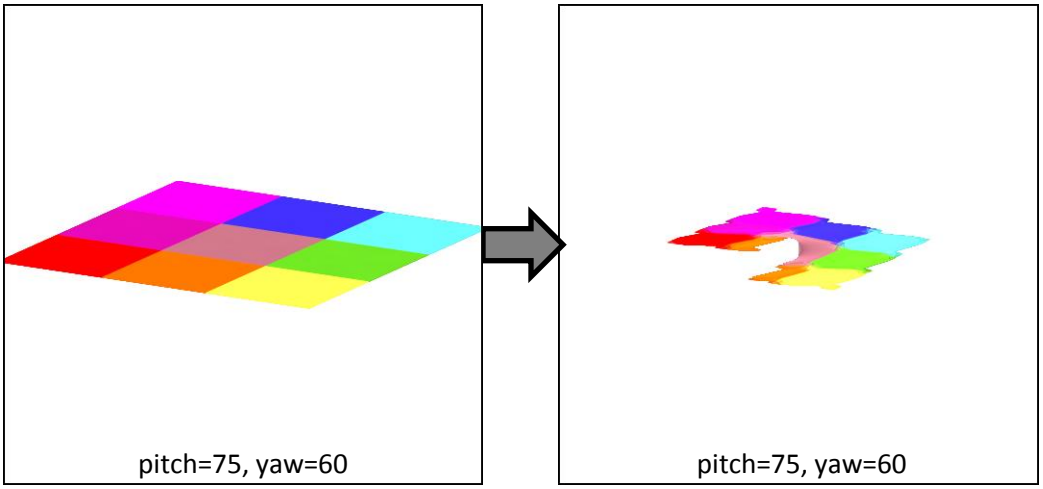
popcorn2_3D (3D, sets z)

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

popcorn2_3D.dll

A variant of popcorn2, it halves the x-y size and adds a z component by splitting the plane along the negative x axis (turned here for clarity) and moving half up and half down.

The example uses $x = 0.1$, $y = 0.1$, $z = 0.1$, and $c = 3.1$.

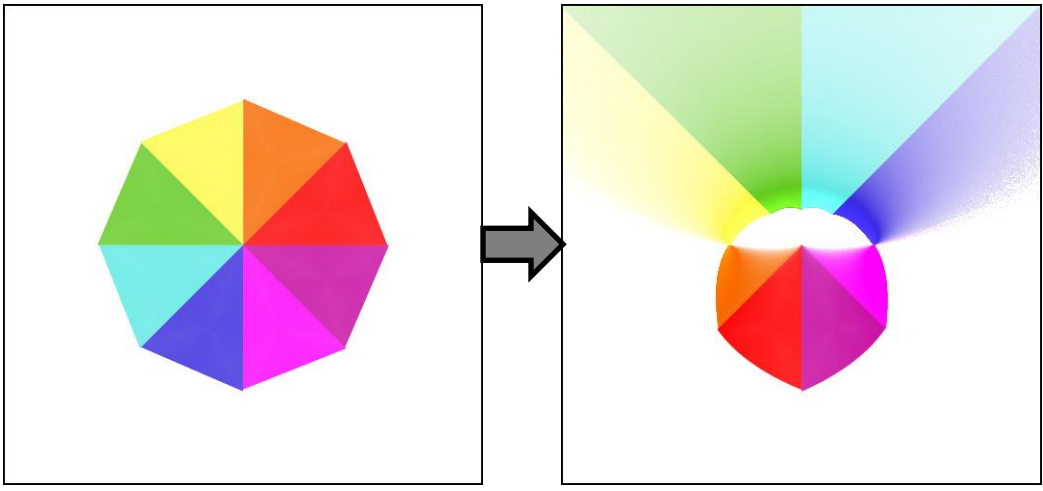


power (2D)

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

power.dll

Rotates the plane 90° clockwise, distorts it, and turns the original left half (now the top half) inside out.

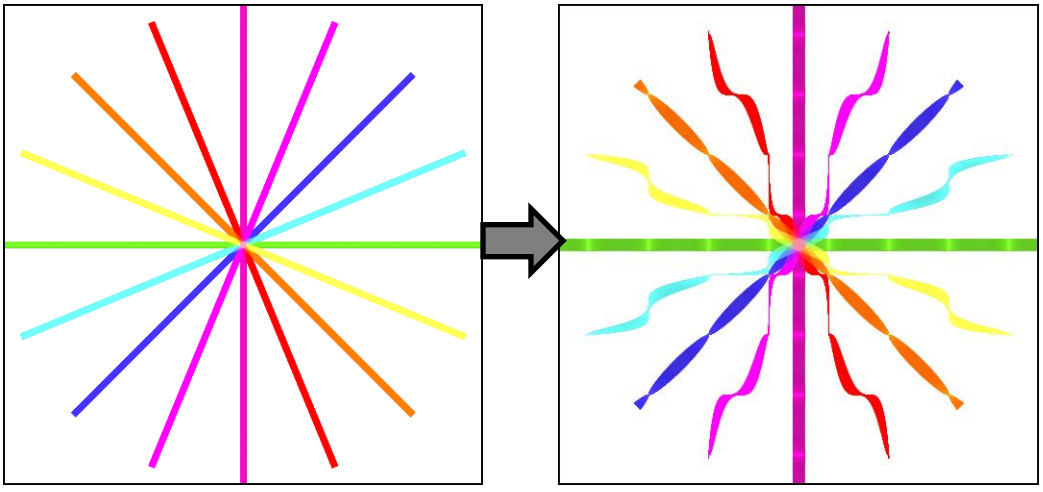


pressure_wave (2D)

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

pressure_wave.dll

Imagine the plane is a rubber sheet, and scrunch it together vertically at regular intervals (like a curtain, but stay in two dimensions). Do the same horizontally. The example here has *x_freq* = 2 and *y_freq* = 2.



pRose3D (3D half blur)

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

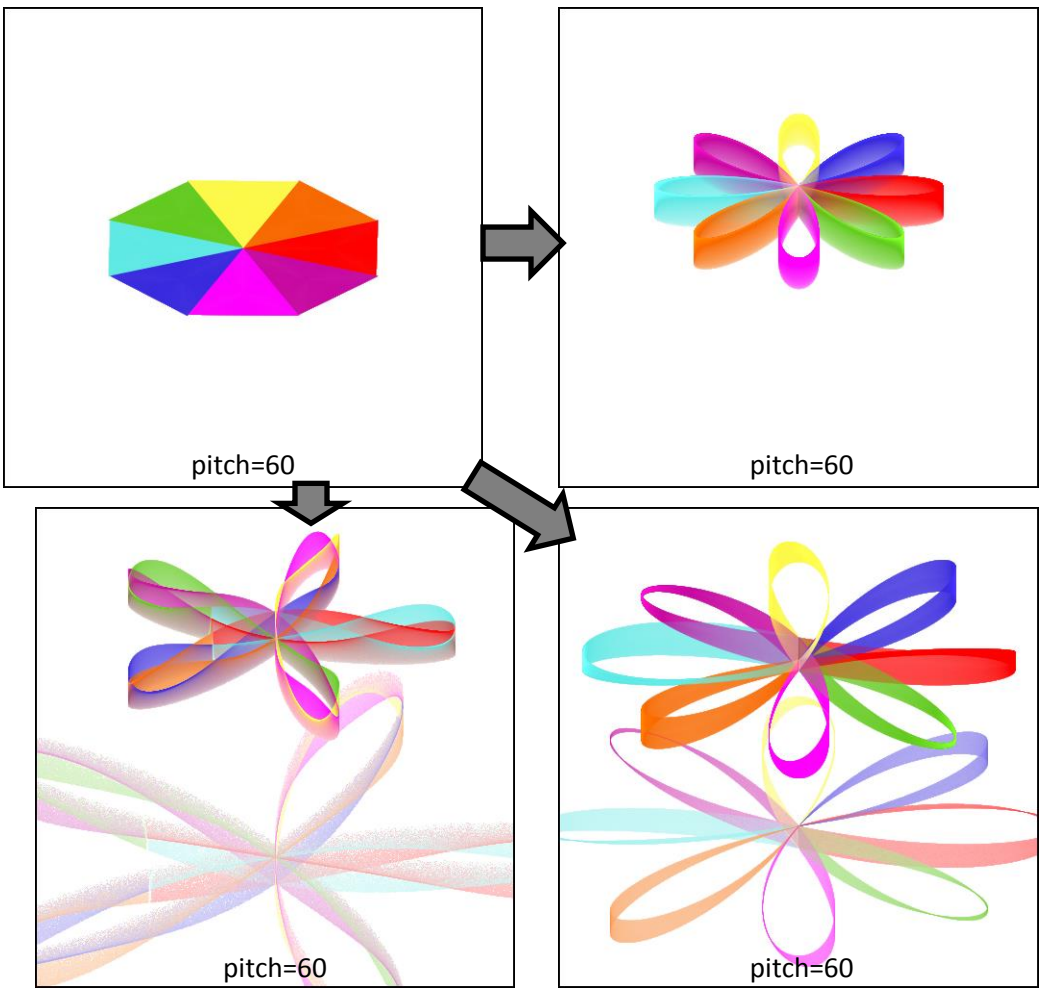
pRose3D.dll

A 3D version of the polar rose curve. There are lots of variables! No attempt is made to explain them here, but a few examples are shown.

Top right: *l* = 4, *k* = 4, *c* = 0, *z1* = 1, *z2* = 1, *refSc* = 1, *opt* = 1, *optSc* = 1, *opt3* = 0, *transp* = 0.5, *dist* = 1, *wagsc* = 0, *crvsc* = 0, *f* = 3, *wigsc* = 0, *offset* = 0

Bottom right: *l* = 6, *k* = 4, *c* = 0, *z1* = 2, *z2* = 1, *refSc* = 1.2, *opt* = -2, *optSc* = 1, *opt3* = 0, *transp* = 0.2, *dist* = 2.5, *wagsc* = 0, *crvsc* = 0, *f* = 3, *wigsc* = 1, *offset* = 0

Bottom left: *l* = 5, *k* = 5, *c* = 0, *z1* = 0.5, *z2* = 0.5, *refSc* = 2, *opt* = -1, *optSc* = 1, *opt3* = 0, *transp* = 0.125, *dist* = 3, *wagsc* = 2, *crvsc* = -3, *f* = 6, *wigsc* = 1, *offset* = 0

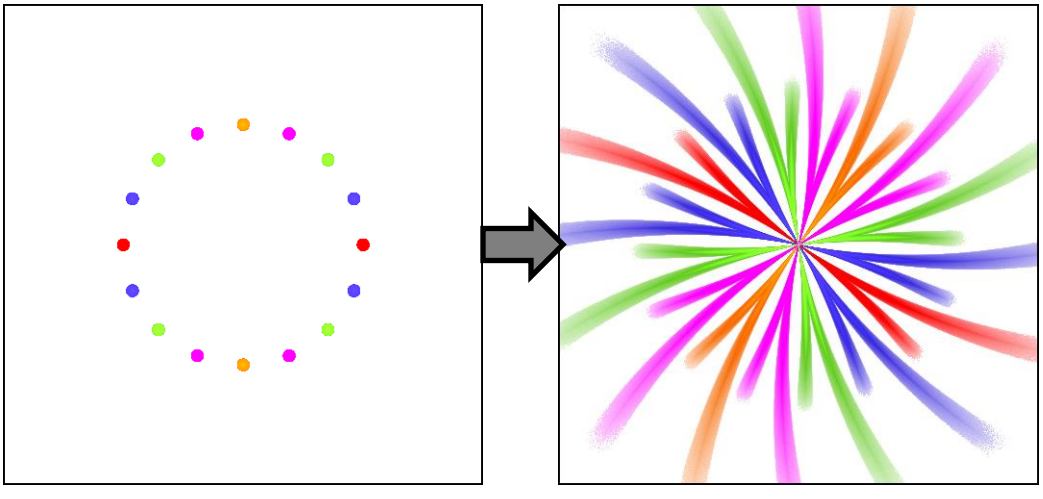


radial_blur (2D half blur)

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

Maps each point on the plane to a "swoosh" that goes to the origin and beyond, curved according to the variable *angle*, which has effective values between -1 and 1, with 0 being no curvature. (Other values are allowed but repeat this range.)

The example is contrived to show how the variation works. It has *angle* = 0.4.

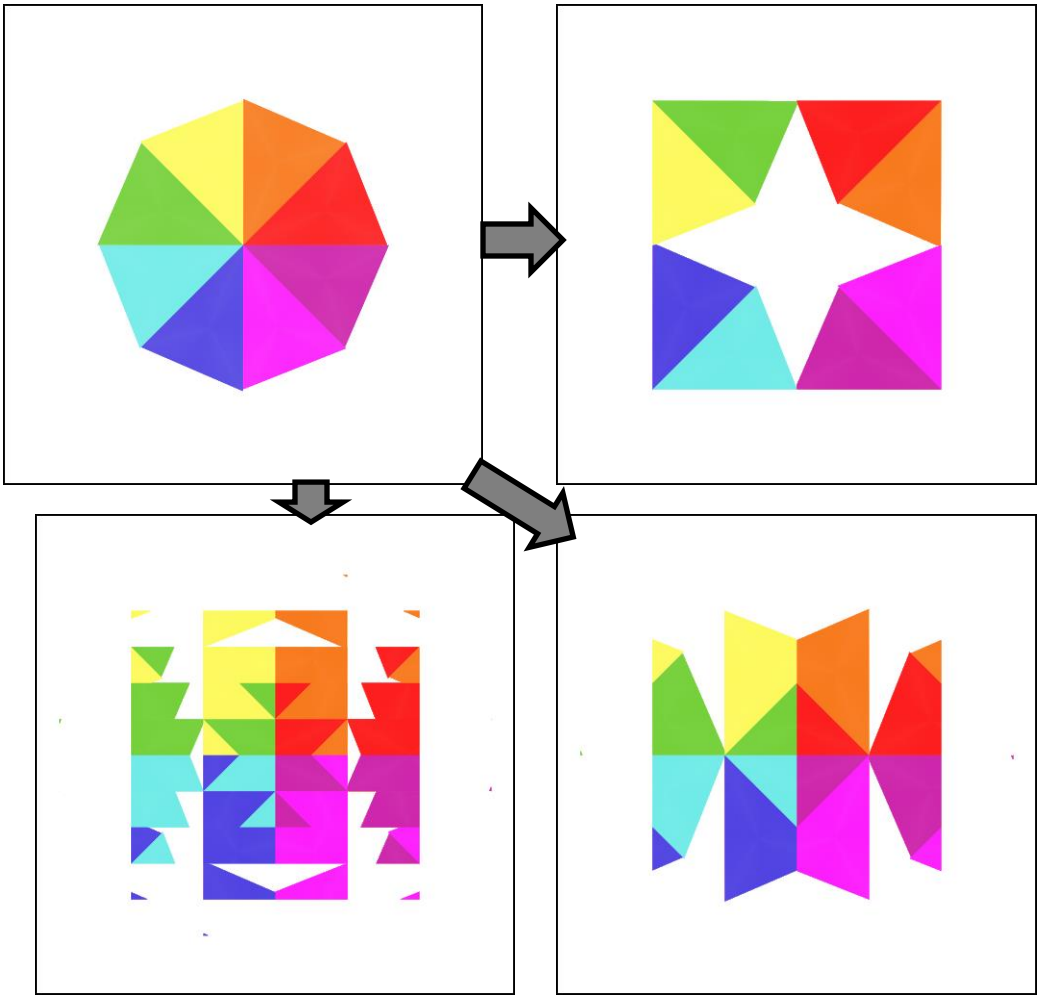


rectangles (2D)

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

Divides the plane into *x* by *y* rectangles and flips each both horizontally and vertically (equivalent to rotating by 180°). If *x* or *y* is 0, the plane is divided in to vertical or horizontal stripes and flipped only one direction. (If both are 0, there is no effect.)

Top right: *x* = 2, *y* = 2
Bottom right: *x* = 1, *y*= 0
Bottom left: *x* = 1, *y* = 0.5



rings2 (2D)

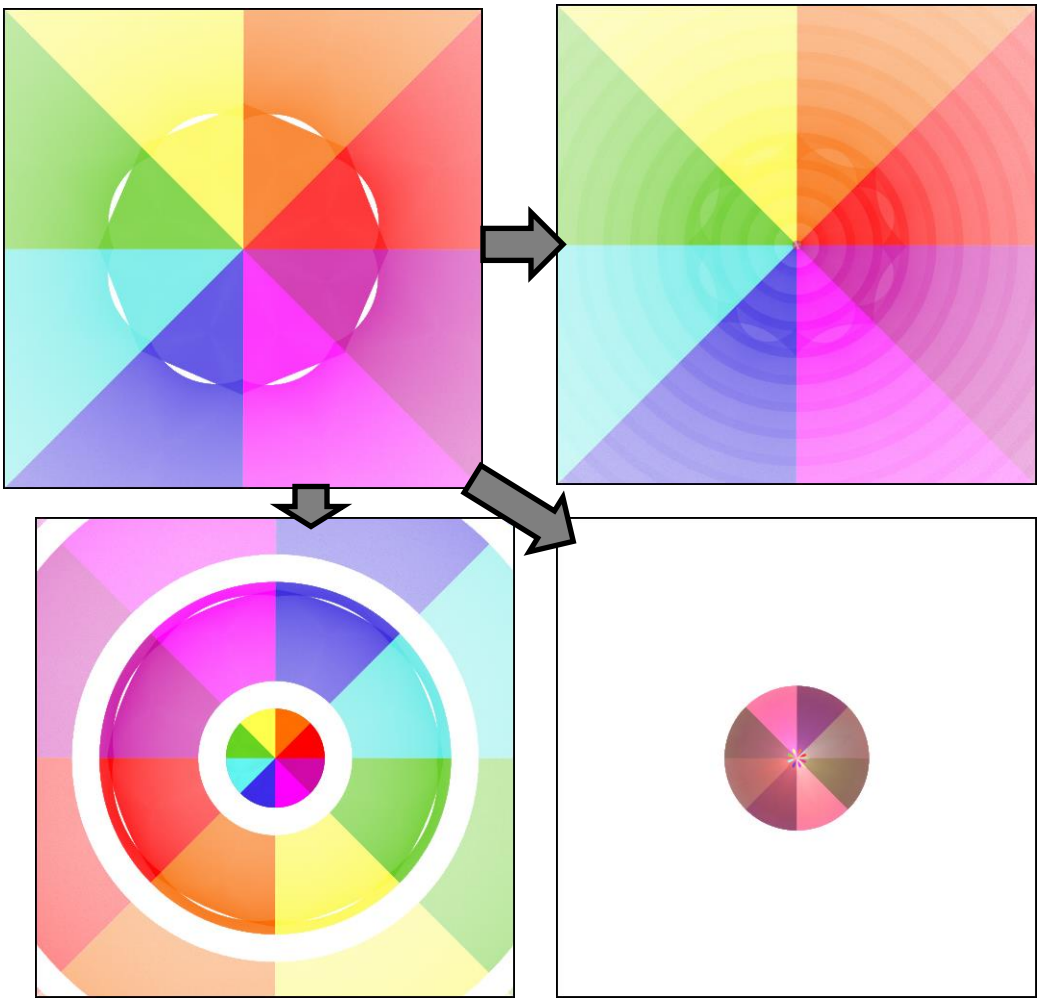
2.09	7X15B	7X16	jwf	ch
yes	yes	yes	yes	yes

Divides the plane into rings whose size and width depends on the variable *val*. When *val* < 1, each ring will be shrunk but the width will increase, making the rings overlap. When *val* > 1, the radius of each ring will go negative (reflecting the ring across the origin and turning it inside-out) and the width will decrease.

Special cases:
val = 1: the width of each ring is the same, but the size decreases to 0, turning the rings into superimposed circles.
val = 1.2247: the width of each ring matches the size of its neighbors so the rings touch without overlapping.
val = 1.4142: degenerate case where the width of each ring is 0

Top right: *val* = 0.5
Bottom right: *val* = 1
Bottom left: *val* = 1.25 (larger than 1.2247, so there are gaps between the rings)

Supercedes **rings**.



ripple (2D)

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

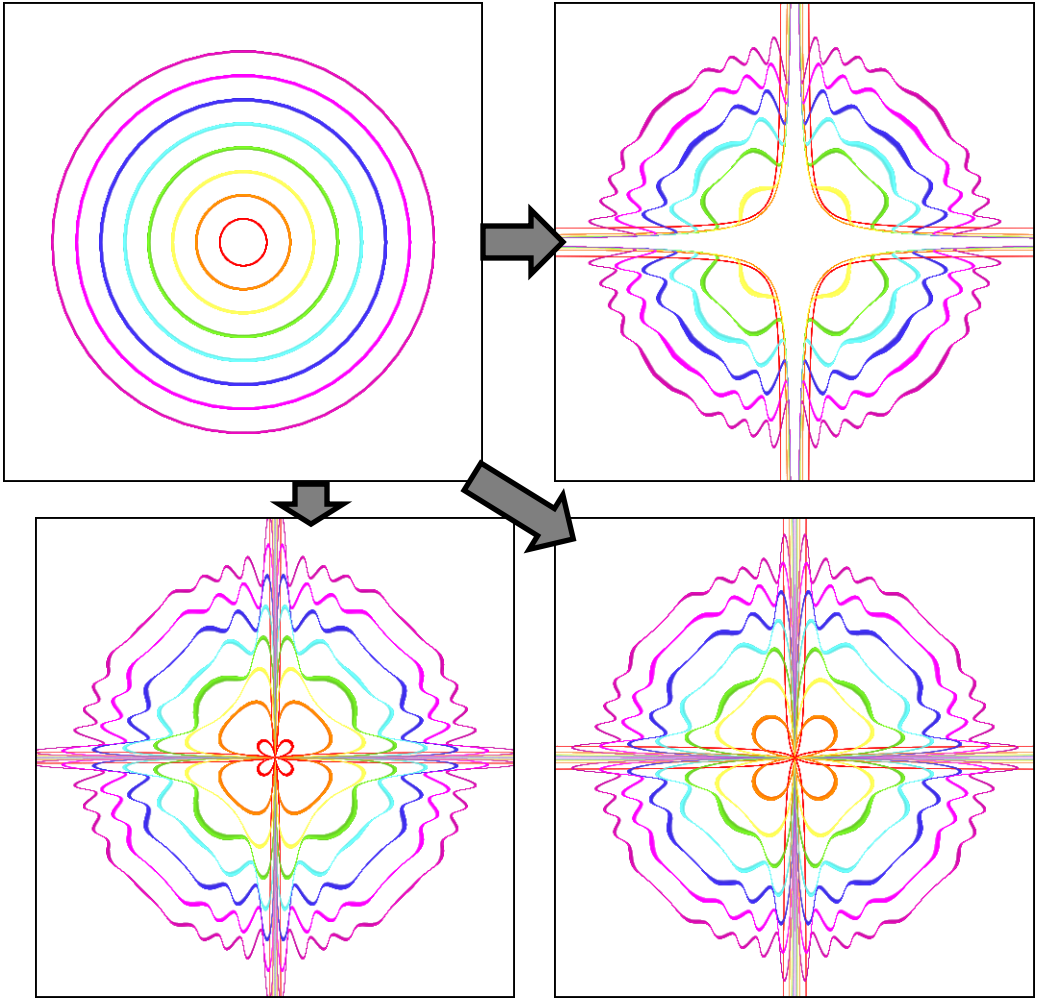
ripple.dll

Modulates the plane using two radial cosine waves. The wave frequencies are determined by *frequency* and the distance from the center (adjusted by *centerx*, *centery*, and *scale*). The amplitudes are determined by *amplitude* and the angle from the center. The modulation is determined by *velocity* and *phase*, and is extrapolated by *phase*.

Top right: *frequency* = 1, *velocity* = 1, *amplitude* = 0.3, *centerx* = 0, *centery* = 0, *phase* = 0, *scale* = 1.75

Bottom right: shows the effect of *velocity*; parameters are the same as top right except *velocity* = 0

Bottom left: shows the effect of *phase*; parameters are the same as top right except *phase* = 1.3333



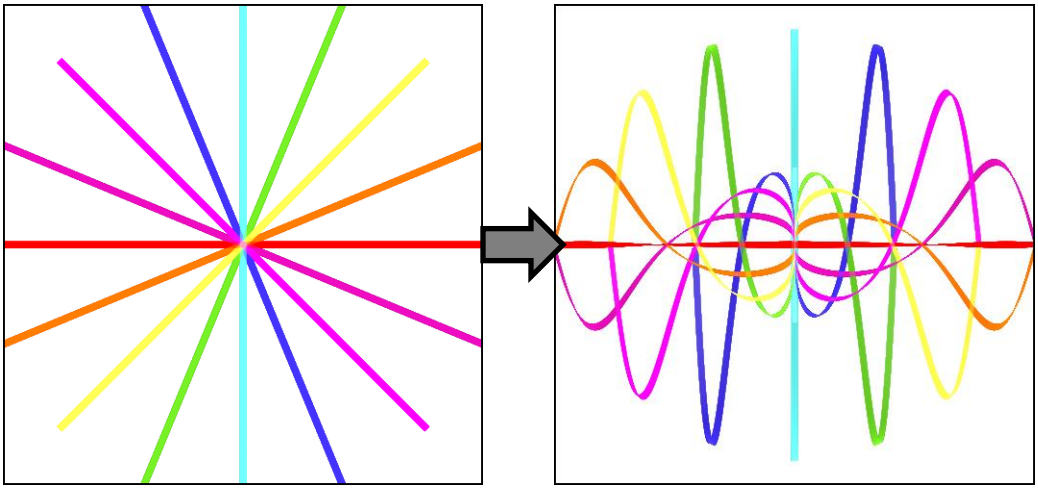
rippled (2D)

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

rippled.dll

Converts lines to ripples by manipulating the y value (x is not changed).

A pre_ version is also available as a plugin.



rose (2D)

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

rose.dll

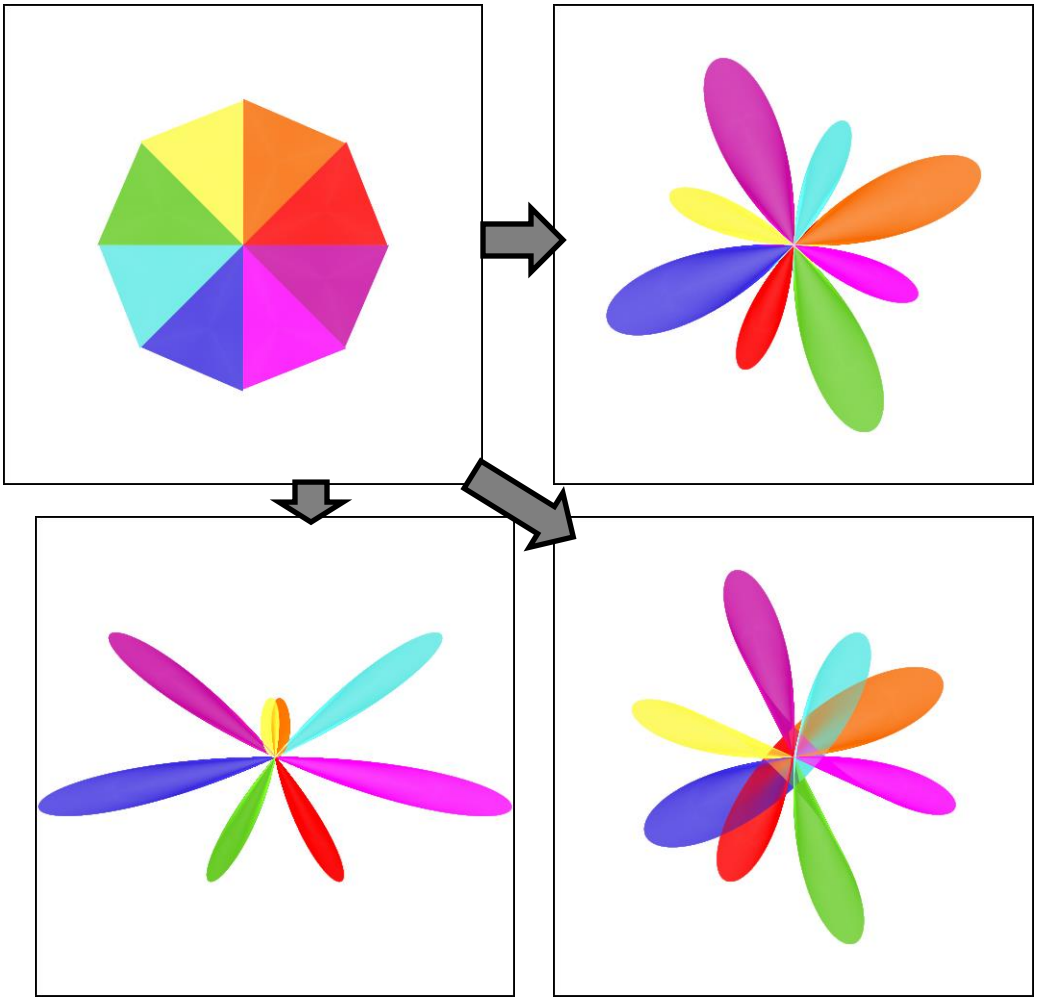
Tranforms the plane using the rose curve. Note this is a true transform, not a blur like flower and pRose3d. Some distortion can also be done, controlled by the variables.

The number of petals is controlled by *numer/denom*; when an integer, there are twice as many petals; half overlap when it is odd, but the other variables treat the overlapping petals separately so they can be distinguished. The examples here all have *numer* = 4 and *denom* = 1, giving 8 petals.

The top right example has *strength* = 1.25 (when 1, all petals are the same size).

The bottom right example also has *strength* = 1.25, but has *shear_x* = 0.3.

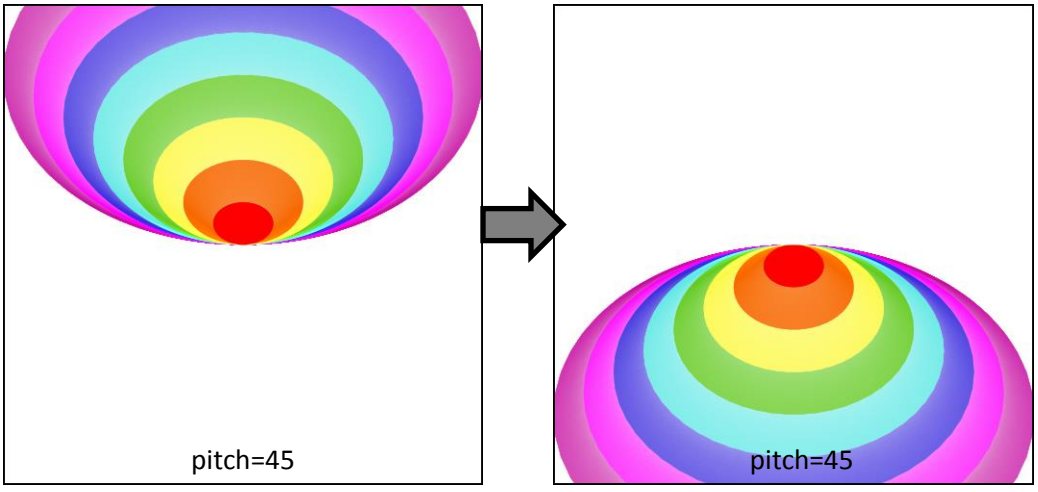
The bottom left example has *strength* = 1 but has *turn_x* = 1.



pre_rotate_x, post_rotate_x (3D)

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

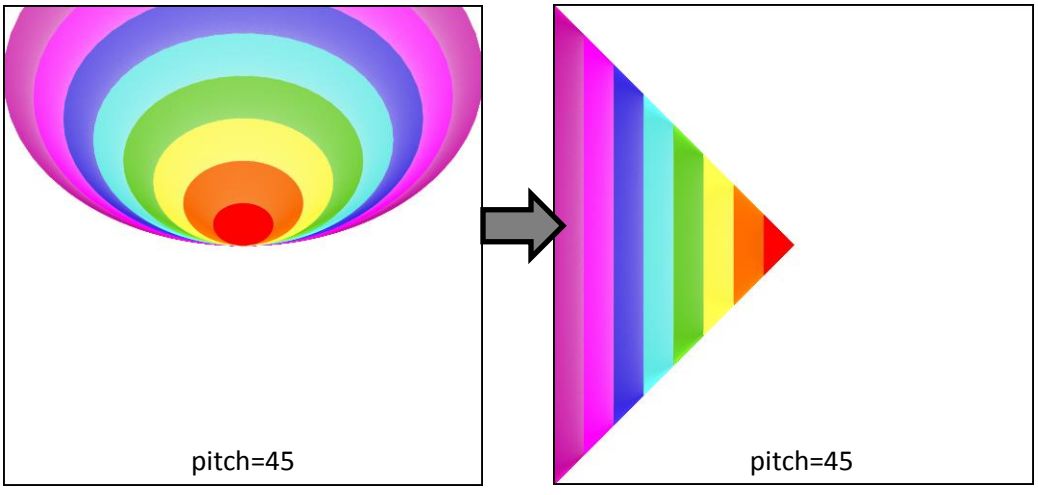
Rotates the 3D space around the x axis. The distance is controlled by the variation value; 1 (shown here) means 90°.



pre_rotate_y, post_rotate_y (3D)

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

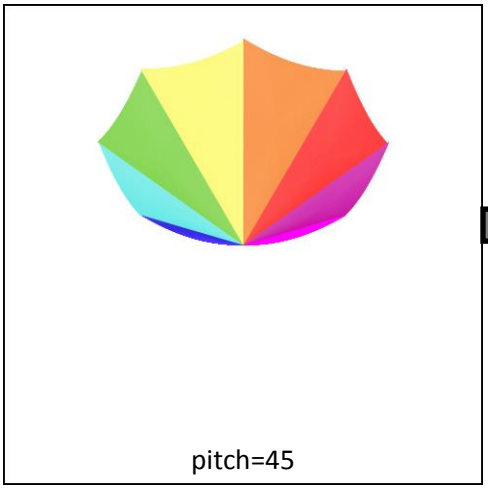
Rotates the 3D space around the y axis. The distance is controlled by the variation value; 1 (shown here) means 90°.

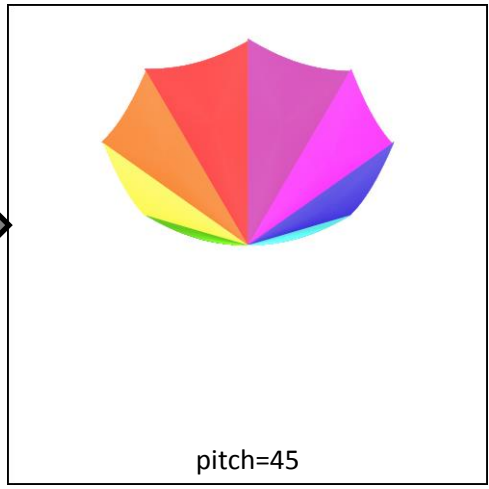


pre_rotate_z, post_rotate_z (3D)

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

Rotates the 3D space around the z axis. The distance is controlled by the variation value; 1 (shown here) means 90°.



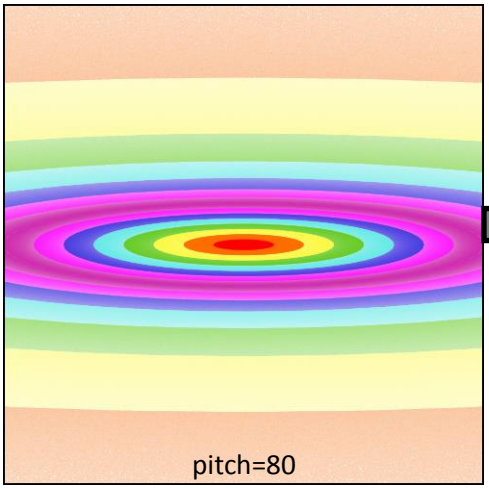


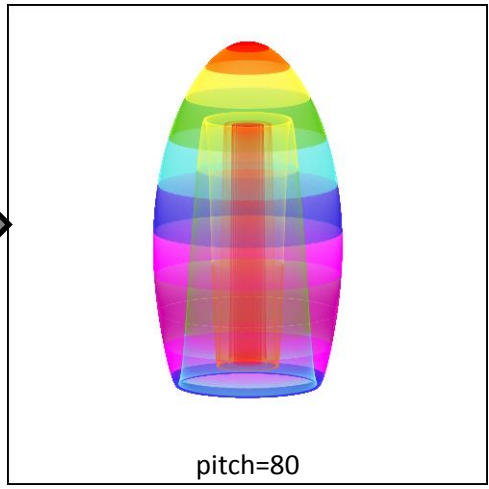
roundspher3D (3D, transforms z)

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

roundspher3D.dll

Transforms a plane to a 3D shape as shown. The inner part of the original goes to the outer port of the 3D shape. But it also takes existing z values into consideration, so the shape will be different if the original isn't a flat plane.





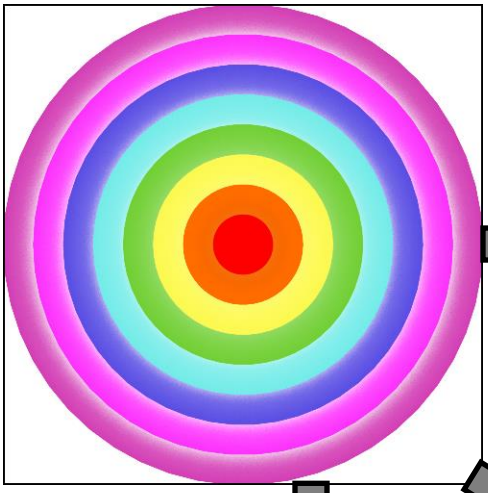
scry (2D)

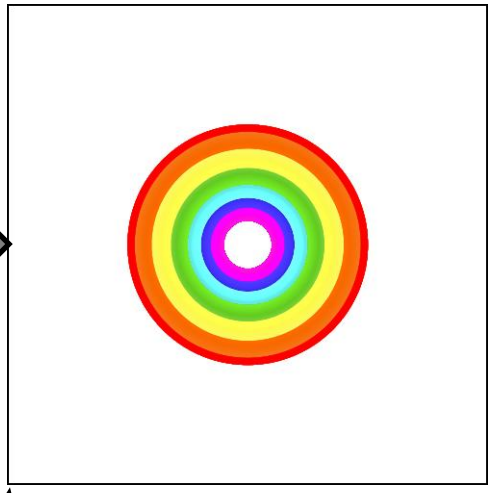
2.09	7X15B	7X16	jwf	ch
dll	dll	yes	yes	yes

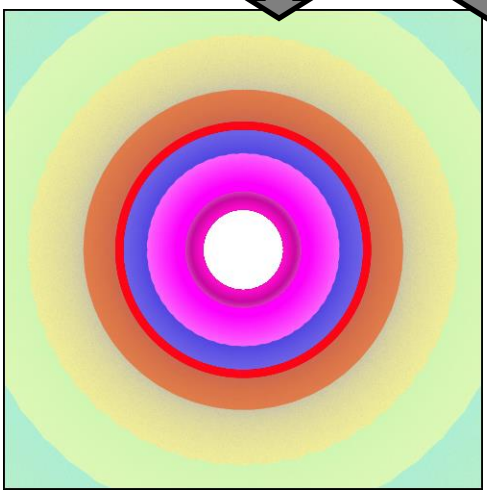
scry.dll

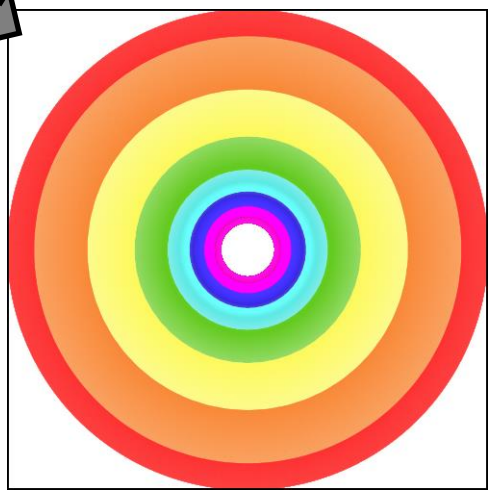
When the variation value is positive, it turns the plane inside out, limiting it at the value. So a line going out from the origin would transform to a line starting at the variation value distance and going the other direction (towards the origin). This is shown on the right examples with values 1 (top) and 2 (bottom).

When the variation value is negative, it's the same in principle but looks quite different. A line going out from the origin would transform to a line starting at the value distance and going the other direction, but it starts opposite where it would start if positive, so it initially goes away from the origin. When it reaches the point originally the value distance from the origin, it is at infinity, and continues coming in the other side, this time really going towards the origin. The bottom left example shows this with a value of -1.









scry_3D (3D, transforms z)

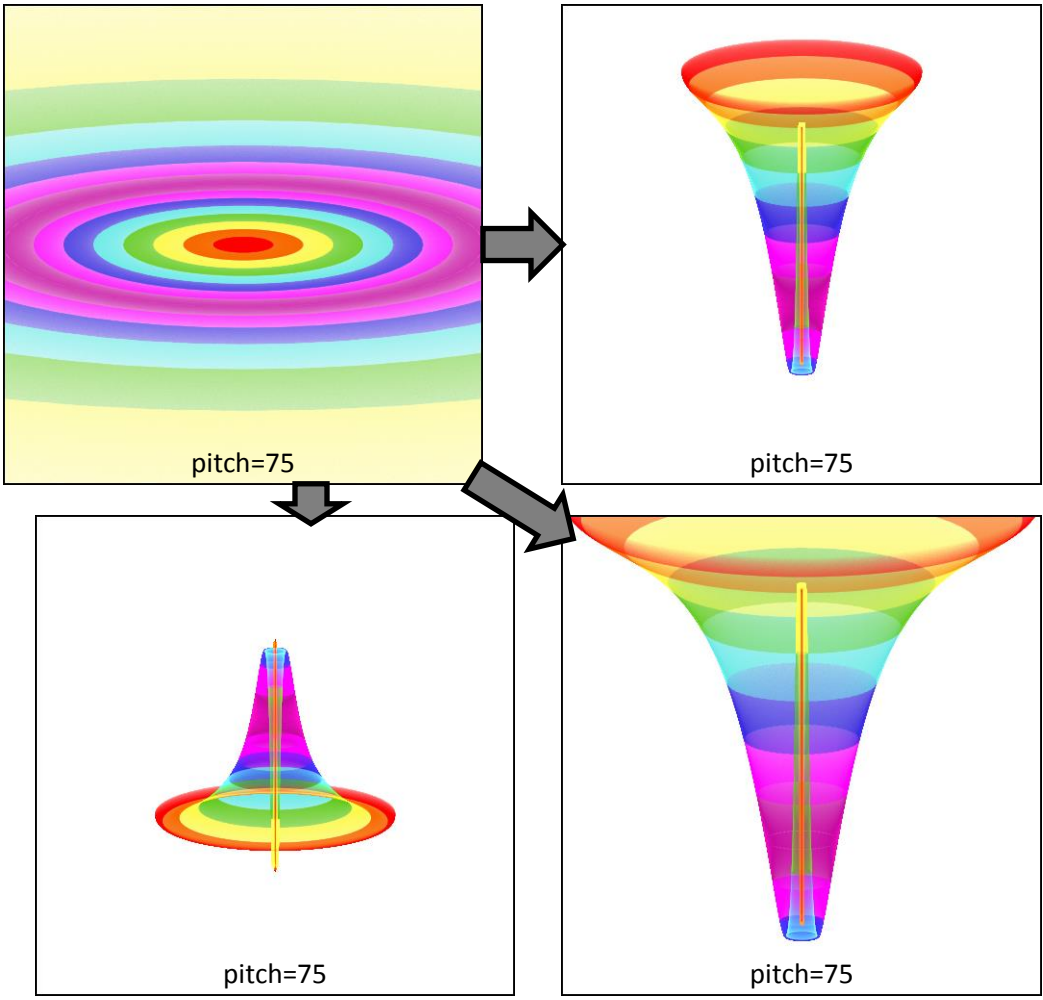
2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	no

scry_3D.dll

A 3D version of scry, which transforms z if it is set and sets it if it is 0 (shown in the examples).

Viewed face-on (pitch=0), scry_3D is the same as scry only when the variation value is 1. Negative values invert the 3D shape, but not the 2D shape like scry does.

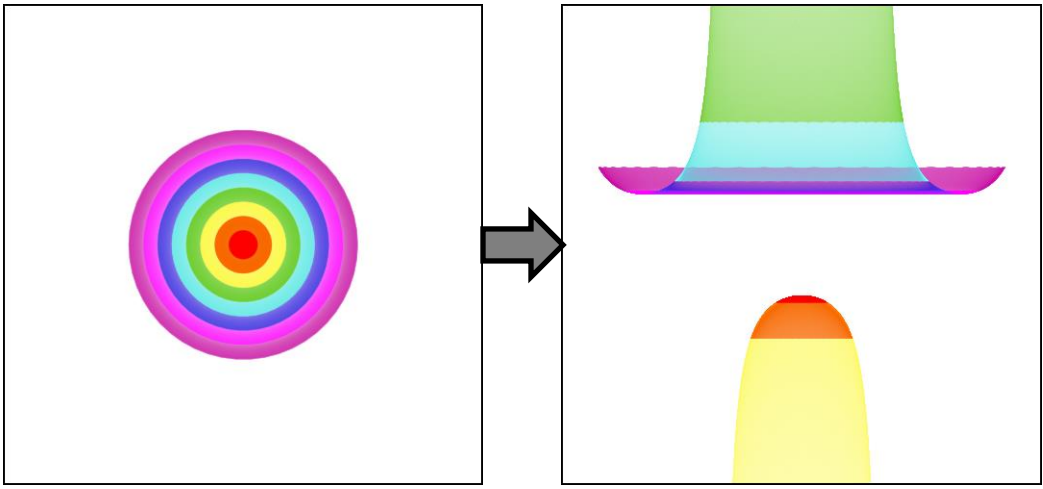
Top right: variation value = 1
Bottom right: variation value = 1.414
Bottom left: variation value = -1



secant (2D)

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

SecantVariationPlugin.dll
Multiplies x by the variation value (2 in the example), but otherwise leaves it unchanged. Uses the trignometric secant function to compute the y value. Since secant will never return a value between 1 and -1, there is always a gap in the middle. The example is scaled down from normal to show the variation effect.

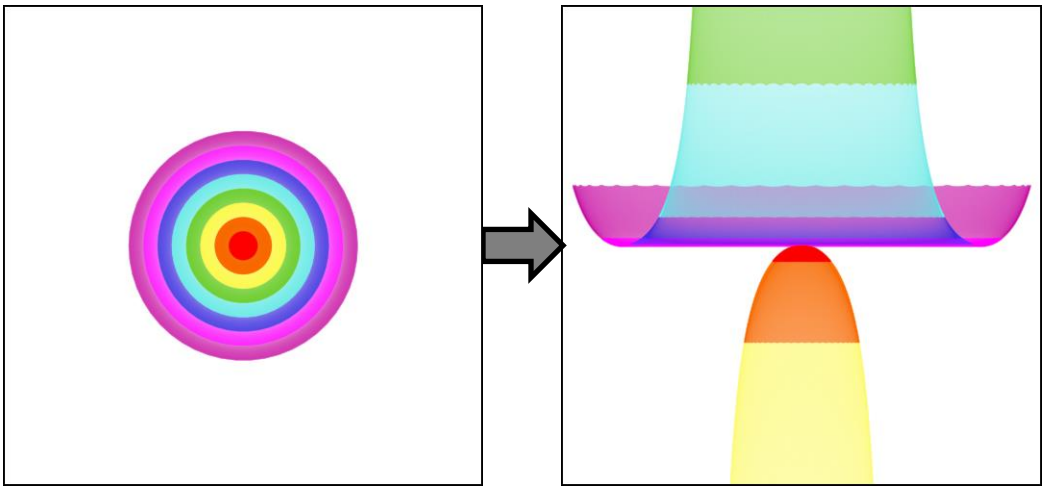


secant2 (2D)

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

An alternative to secant that moves the top and bottom halves towards each other to eliminate the gap inherent in secant.

The example uses the same scale as for secant for comparison.



separation (2D)

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

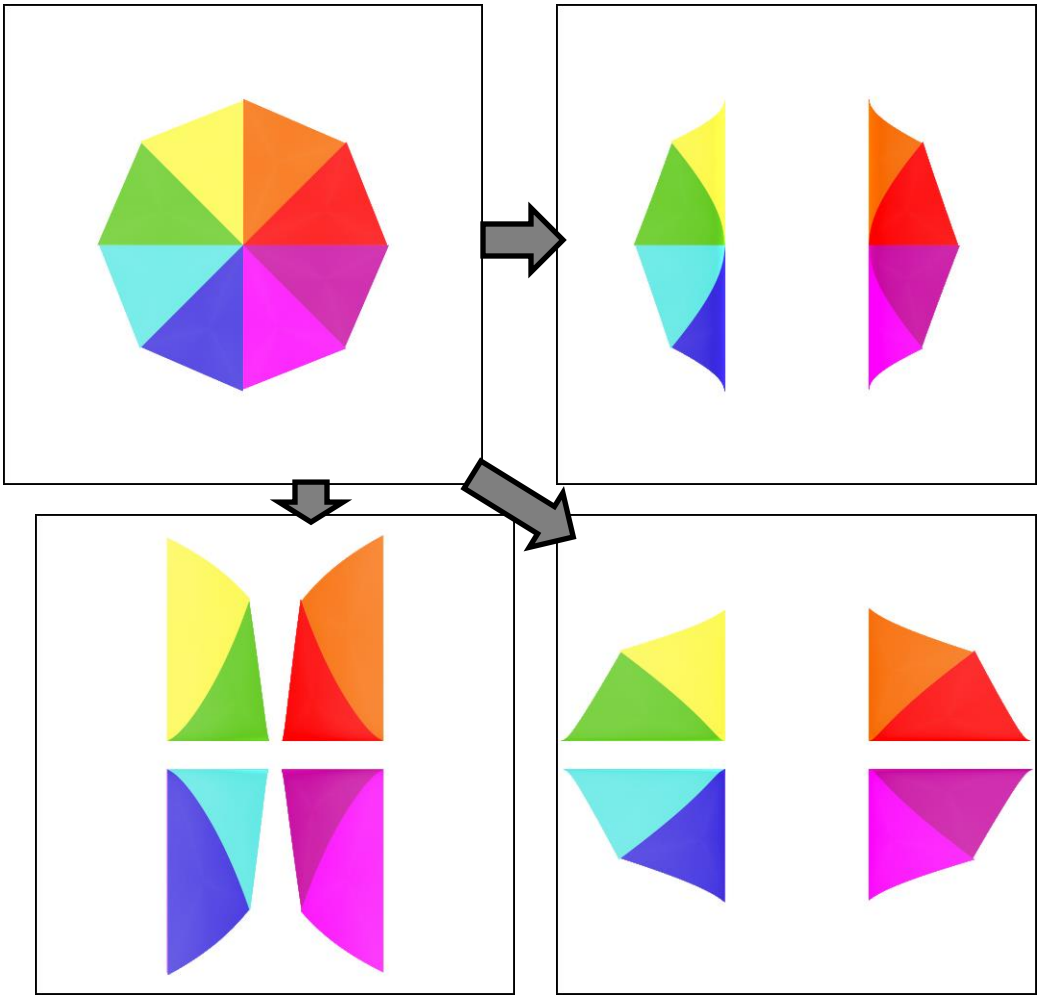
separation.dll

Splits the plane down the y axis, moving each half outwards *x* units, but not evenly; movement of points further from the axis is controlled by *xinside*, which ranges from mild squeezing when 0, to stretching with a large enough negative value, and mirroring with a large enough positive value.

The same things happens vertically, controlled with the *y* and *yinside* variables.

top right: *x* = 1, *y* = 0, *xinside* = 0, *yinside* = 0
bottom right: *x* = 1, *y* = 0.2, *xinside* = -0.5, *yinside* = 0
bottom left: *x* = 1.5, *y* = 0.2, *xinside* = 1.2, *yinside* = -0.5

See splits, which does does the splitting but moves the entire half planes evenly.



shape (2D half blur)

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

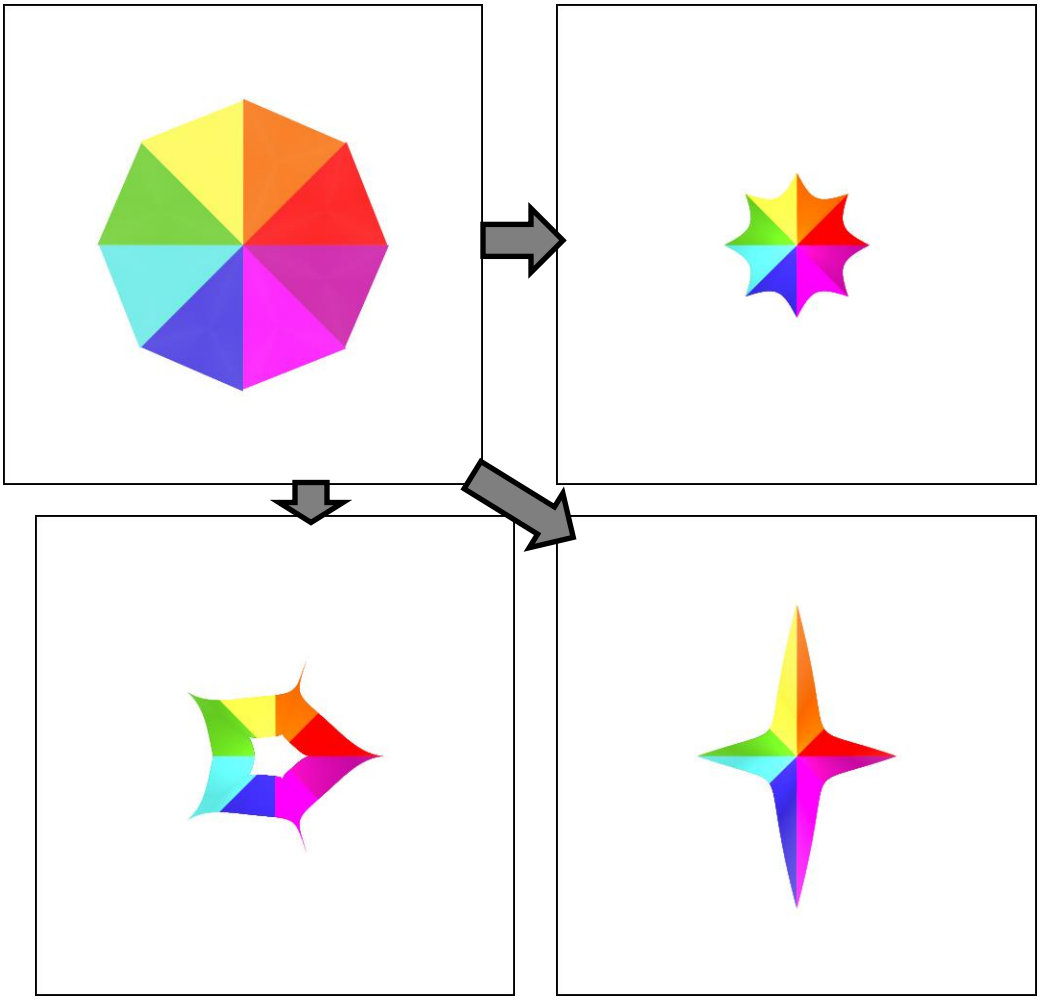
ShapeVariationPlugin.dll

A 2D half blur implementation of the superformula. Some examples are shown.

Variable *m* is the number of corners, *n1* is the main shaping variable, *n2* and *n3* are other shaping variables, *a* and *b* stretch or contract the shape, and *holes* puts a hole in the center if less than 0. (Although the default for *holes* is 1, it works best to use 0 for no hole and a negative value to add a hole. Positive values will reverse the colors.)

Top right: *m* = 8, *n1* = 1, *n2* = 1, *n3* = 1, *a* = 1, *b* = 1, *holes* = 0
Bottom right: *m* = 4, *n1* = 0.3, *n2* = 1, *n3* = 1, *a* = 1.1, *b* = 1.25, *holes* = 0
Bottom left: *m* = 5, *n1* = 1, *n2* = 0.2, *n3* = 0.6, *a* = 1, *b* = 1, *holes* = -0.5

See SuperShape3d and super_shape.



shredlin (2D)

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

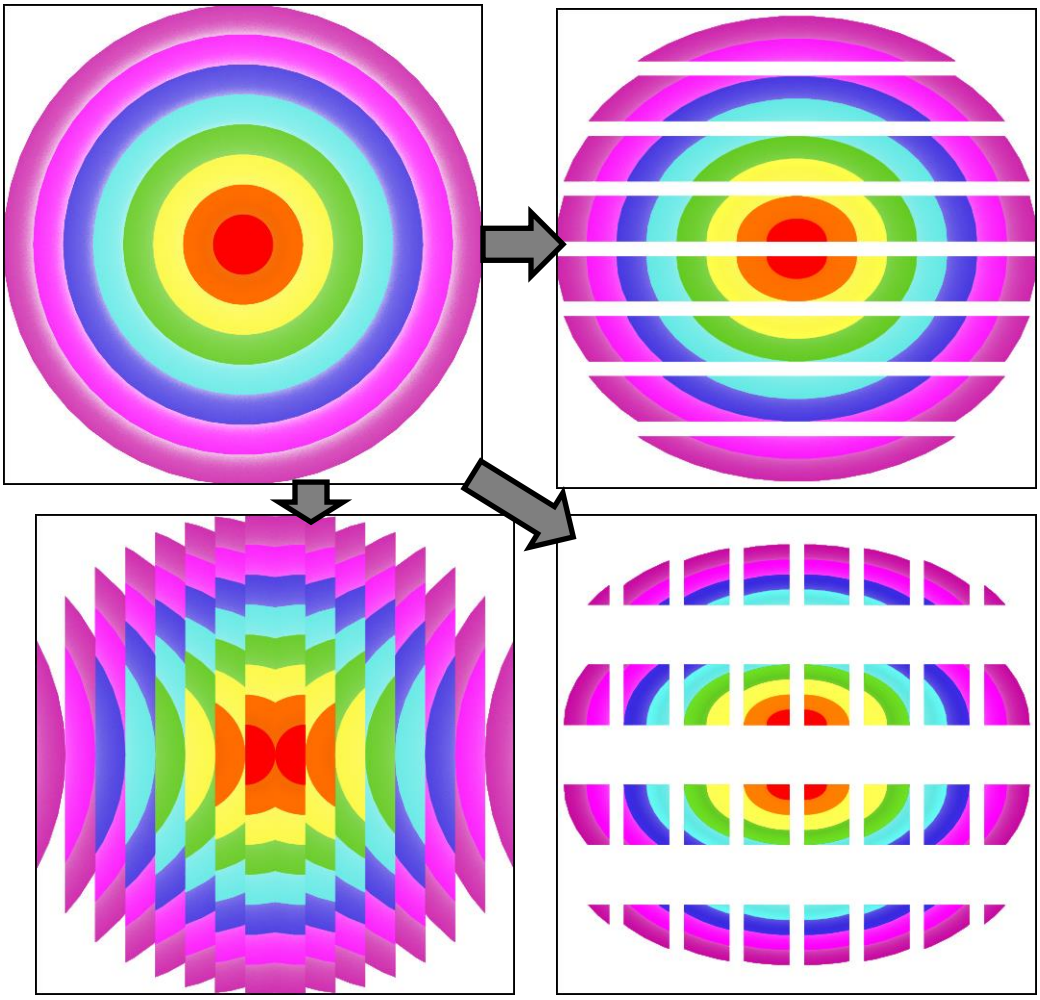
shredlin.dll

Divides the plane into horizontal and vertical strips. The width of each strip is specified by *xdistance* and *ydistance*. Each strip is then shrunk according to *xwidth* and *ywidth*, which specifies the proportion each strip will have (1 leaves it unaffected, 0.5 halves the size, leaving space between them). Negative values work the same, but reverse each strip.

Top right: *xdistance* = 1, *xwidth* = 1, *ydistance* = 0.5, *ywidth* = 0.75

Bottom right: *xdistance* = 0.5, *xwidth* = 0.75, *ydistance* = 1, *ywidth* = 0.5

Bottom left: *xdistance* = 0.25, *xwidth* = -1, *ydistance* = 1, *ywidth* = 1



shredrad (2D)

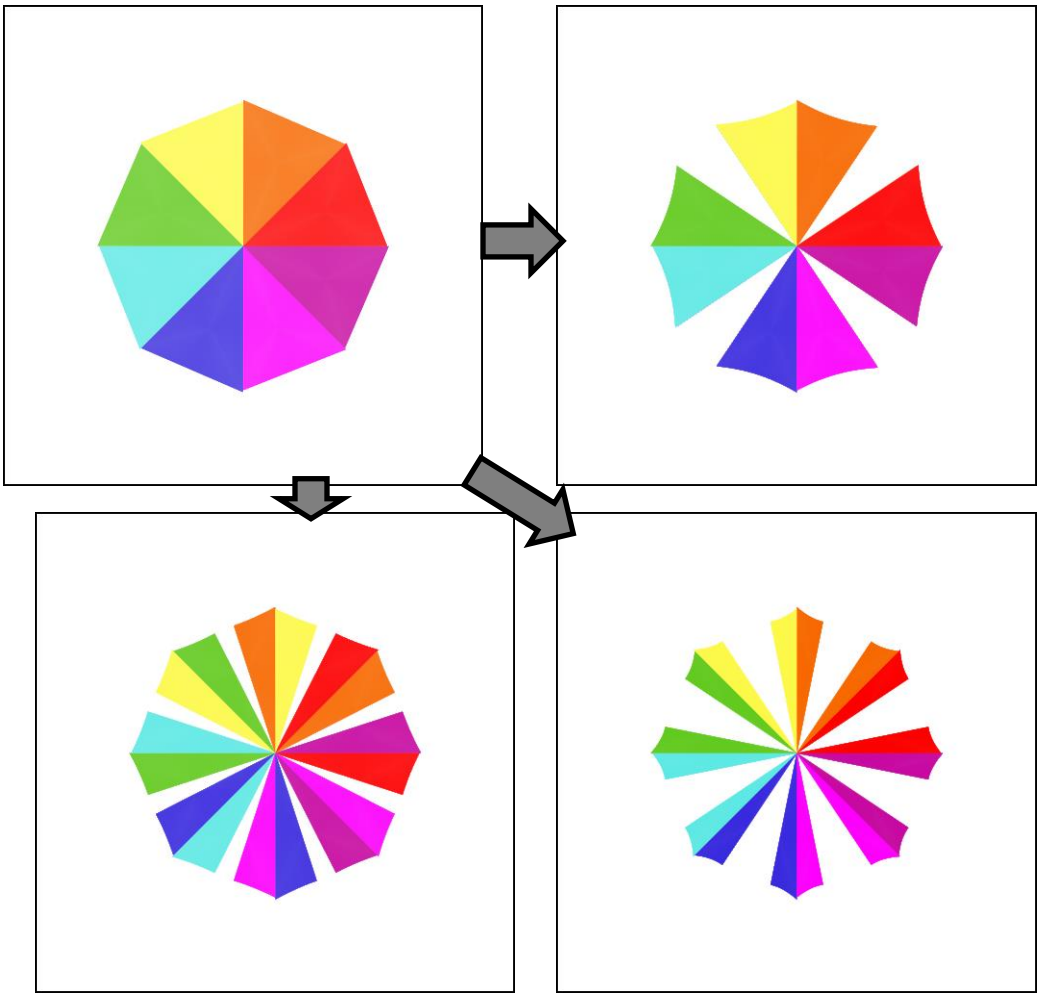
2.09	7X15B	7X16	jwf	ch
dll	dll	dll	no	dll

shredrad.dll

Divides the plane into *n* wedges, and shrinks each according to *width*, which specifies the proportion each wedge will have (1 leaves it unaffected, 0.5 halves the size, leaving space between them). Negative values work the same, but reverse each wedge.

Top right: *n* = 4, *width* = 0.75
Bottom right: *n* = 8, *width* = 0.5
Bottom left: *n* = 8, *width* = -0.8

Compare wedge.



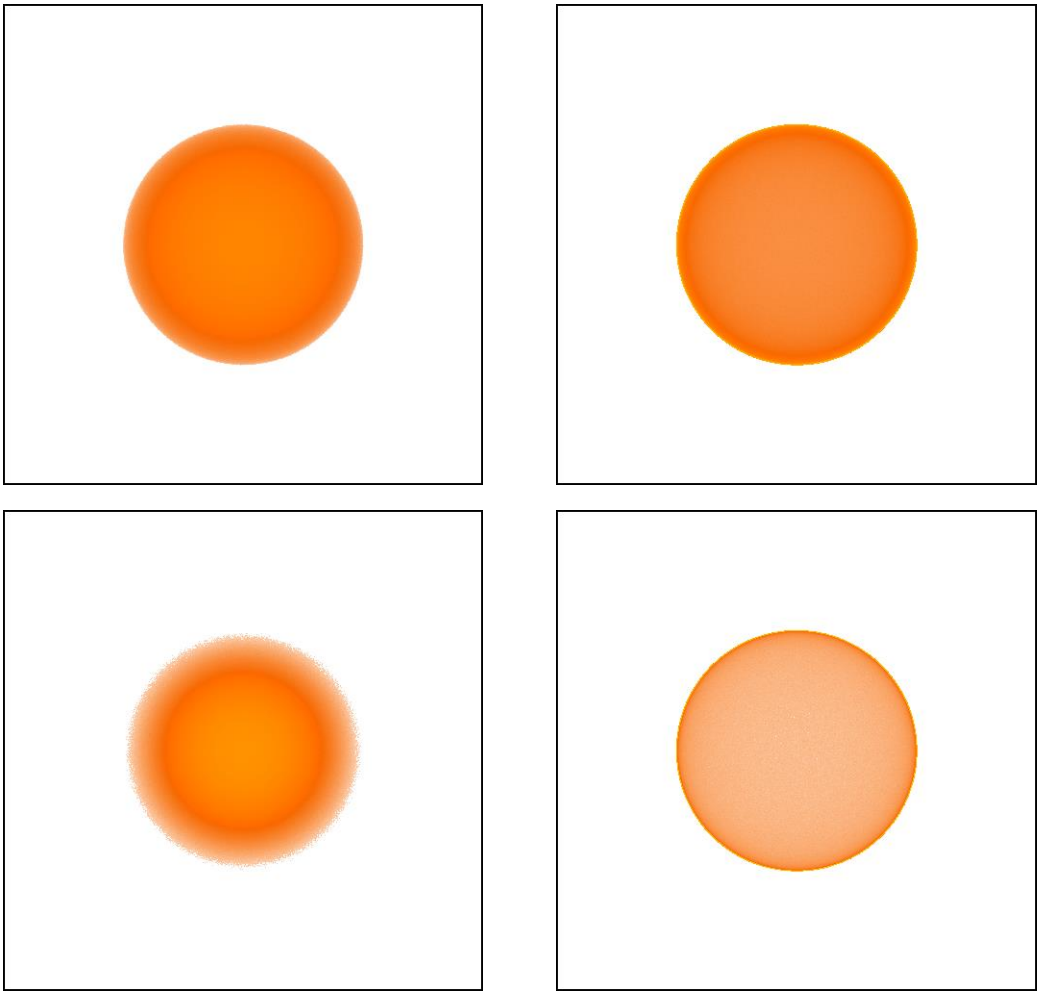
sineblur (2D blur)

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

sineblur.dll

A circle, like blur, but with a shading effect controlled by variable *power*. The appearance is greatly influenced by the background; the default of 1 works well for dark backgrounds, but a somewhat higher value (like 20, top right) is better for light backgrounds. Very high values will have low density in the center. Values less than 1 will produce a smaller circle with a fuzzy edge.

Top left: *power* = 1
Top right: *power* = 20
Bottom left: *power* = 0.5
Bottom right: *power* = 250



sinusgrid (2D, passes z)

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

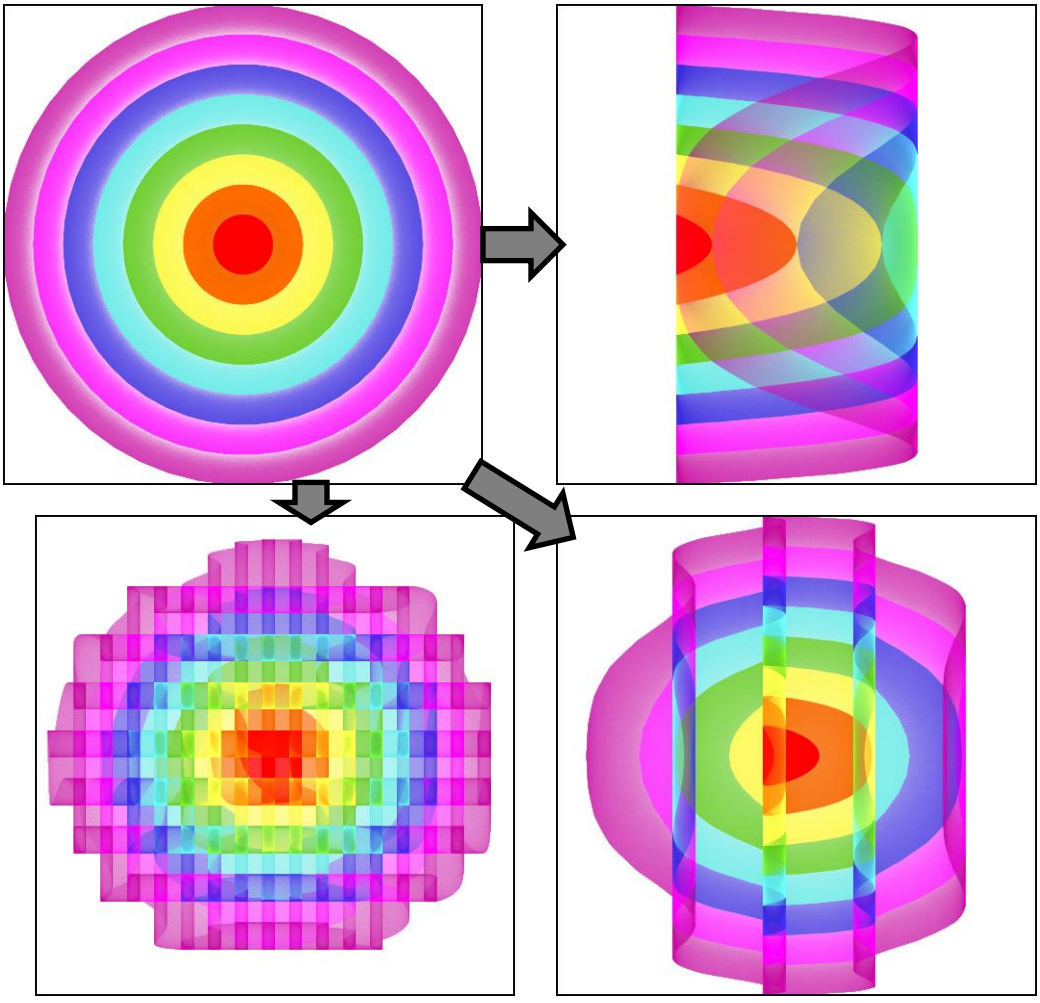
sinusgrid.dll

First it maps the plane to a unit square using a negative cosine function based on the values of *freqx* (for horizontal) and *freqy* (for vertical). Then it interpolates between the original and the mapped versions based on the values of *ampx* and *ampy*, 0 meaning use the original and 1 meaning use the mapped.

The top right example has *freqx* = 0.5, which folds the visible image on itself four times (the left and right halves look the same), with *ampx* = 1 to show the mapped version horizontally; *ampy* is 0 so no vertical mapping is done.

The bottom right example has *freqx* = 1, so doubles the frequency, but *ampx* = 0.25 so it only has a partial effect; *ampy* is again 0.

The bottom left example has large frequencies but small amplitudes: *ampx* = 0.1, *ampy* = 0.2, *freqx* = 4, *freqy* = 2.

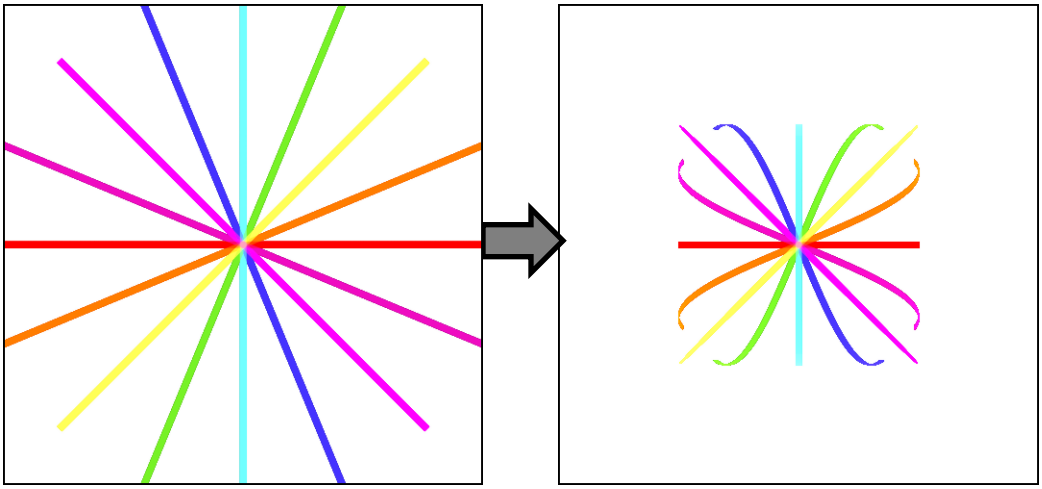


sinusoidal (2D)

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

Uses the sine function to map the plane to a unit square, folding it on itself to do so. The example uses lines to show how it works: horizontal and vertical lines fold on themselves; main diagonal lines become pointy, and other diagonal lines curve.

A pre_ version is available as a plugin.

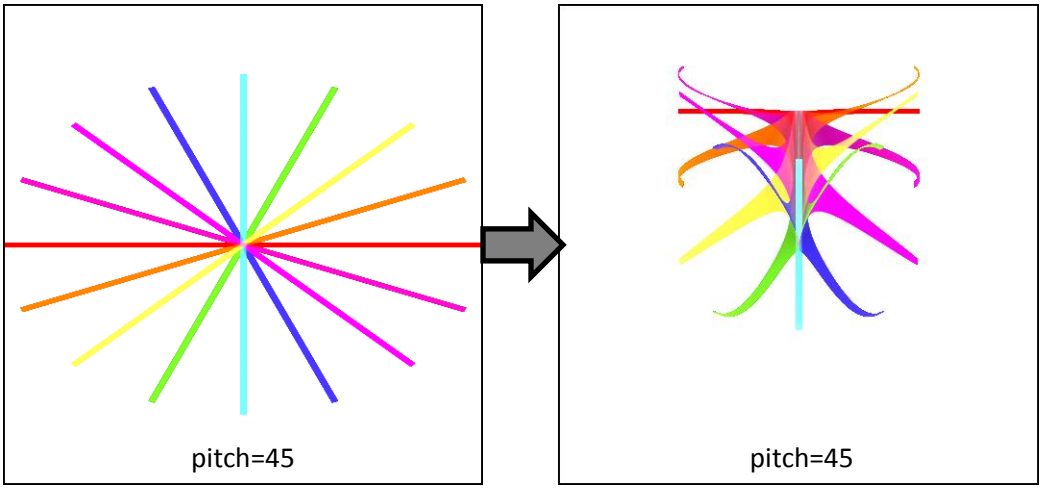


sinusoidal3d, pre_sinusoidal3d (3D, transforms z)

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

sinusoidal3d.dll

A 3D version of sinusoidal.

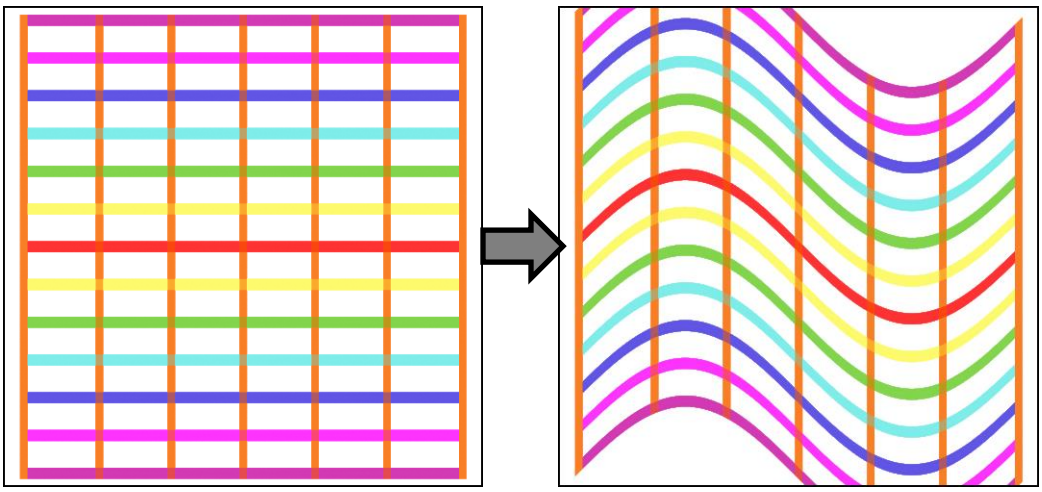


sinusoidal_linear (2D)

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

sinusoidal_linear.dll

Adds a vertical sine wave.

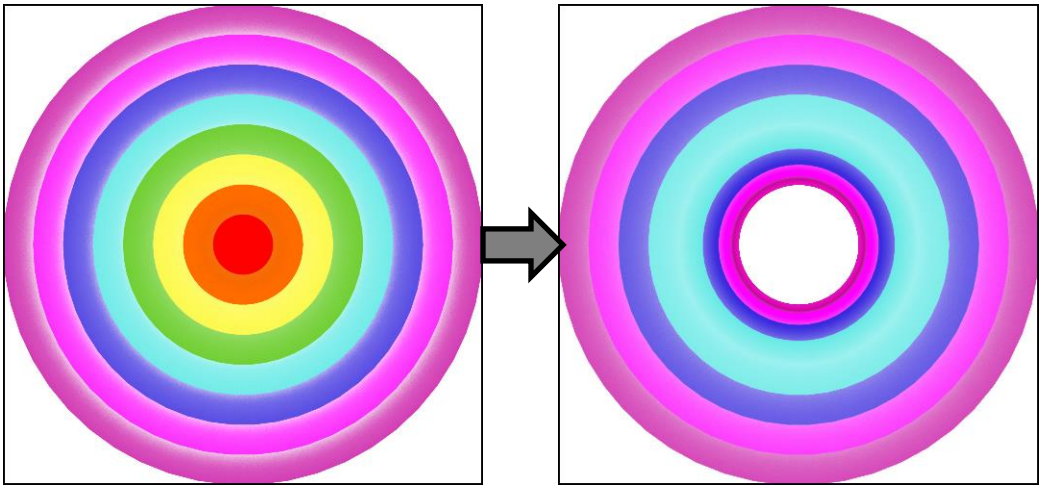


spher (2D)

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

Spher.dll
A cross of linear and spherical; acts like linear outside a circle the size of the variation value and spherical inside it. The example uses a variation value of 1, so the circle in this case is the boundary between the green and cyan rings.

Compare Glynnsim1, Glynnsim2, Glynnsim3.

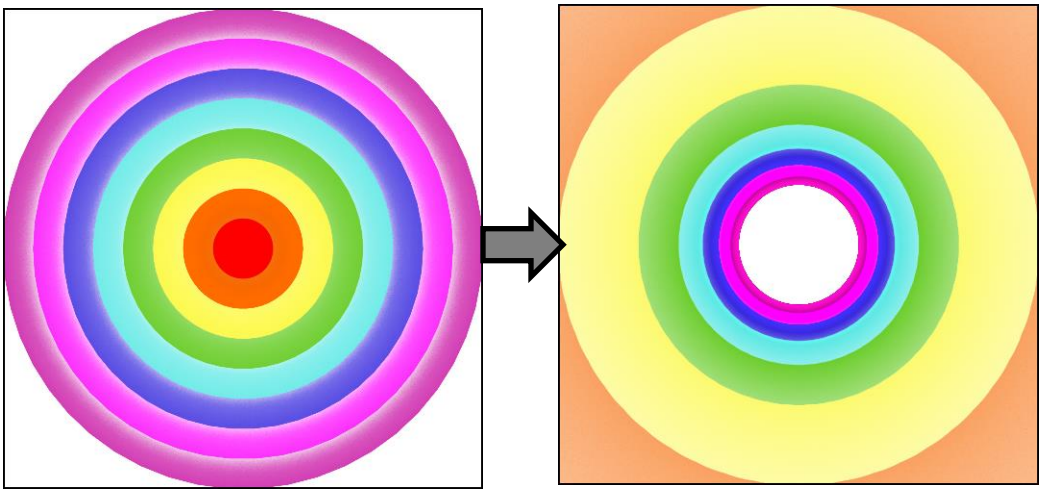


spherical (2D)

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

Turns the plane inside-out. Points near the center are moved away and points far away are moved towards the center.

Pre_ and post_ versions are available as plugins.

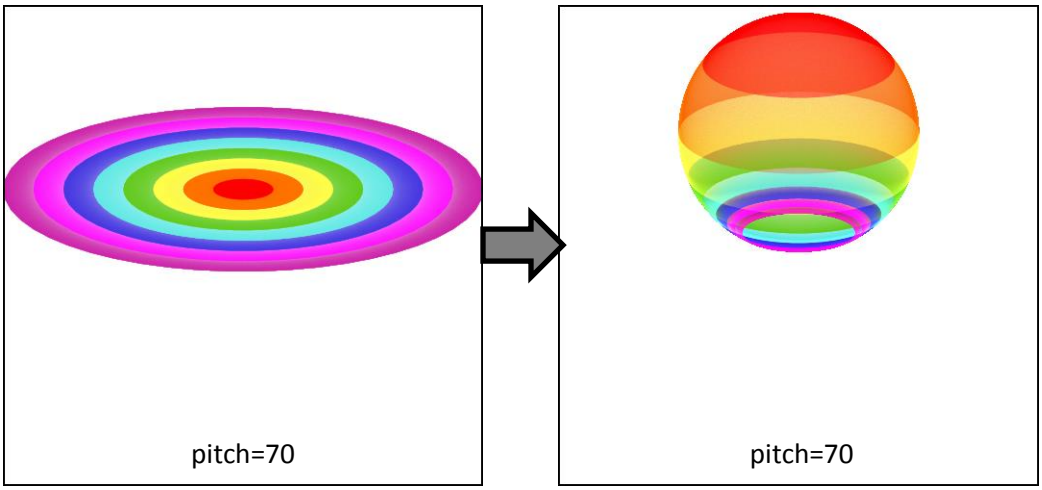


Spherical3D (3D, transforms z)

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

Spherical3D.dll

A 3D version of spherical that sets z based on the original z value and the distance from the center. In the example, the original is a plane with z = 0.5.

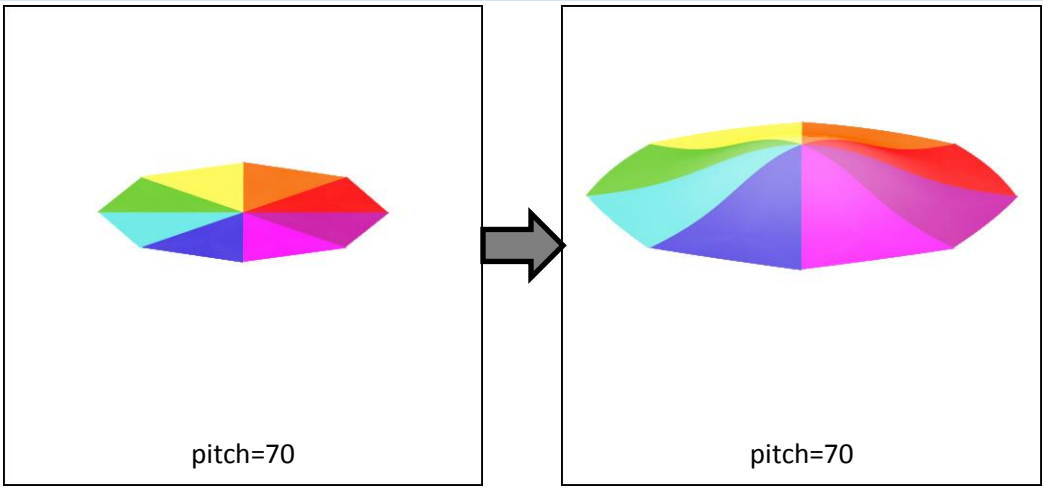


spherivoid (3D, transforms z)

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

spherivoid.dll

Inflates the 3D space by putting a spherical “hole” in the middle with size specified by *radius*. In the example, the original is a plane with z = 0.5 and *radius* = 1. (When z = 0, the middle is blank.)



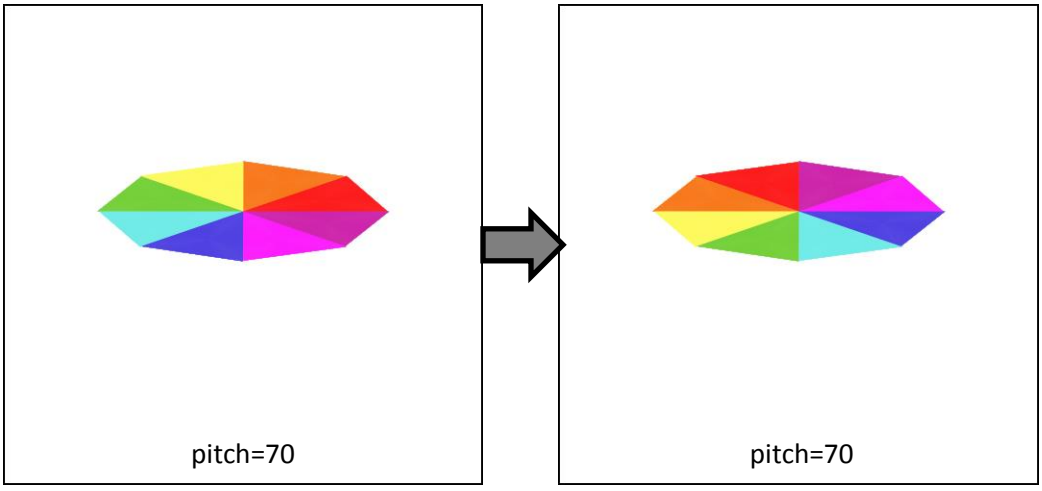
post/pre_spin_z (2D)

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

post_spin_z.dll, pre_spin_z.dll

Rotates the 3D space around the z axis (like yaw, but only for a specific variation). The variation value specifies how much to rotate; effective values are from -2 to 2 (corresponding to -180° to 180°). In the example, the value is 1 (90°).

Does NOT pass z.

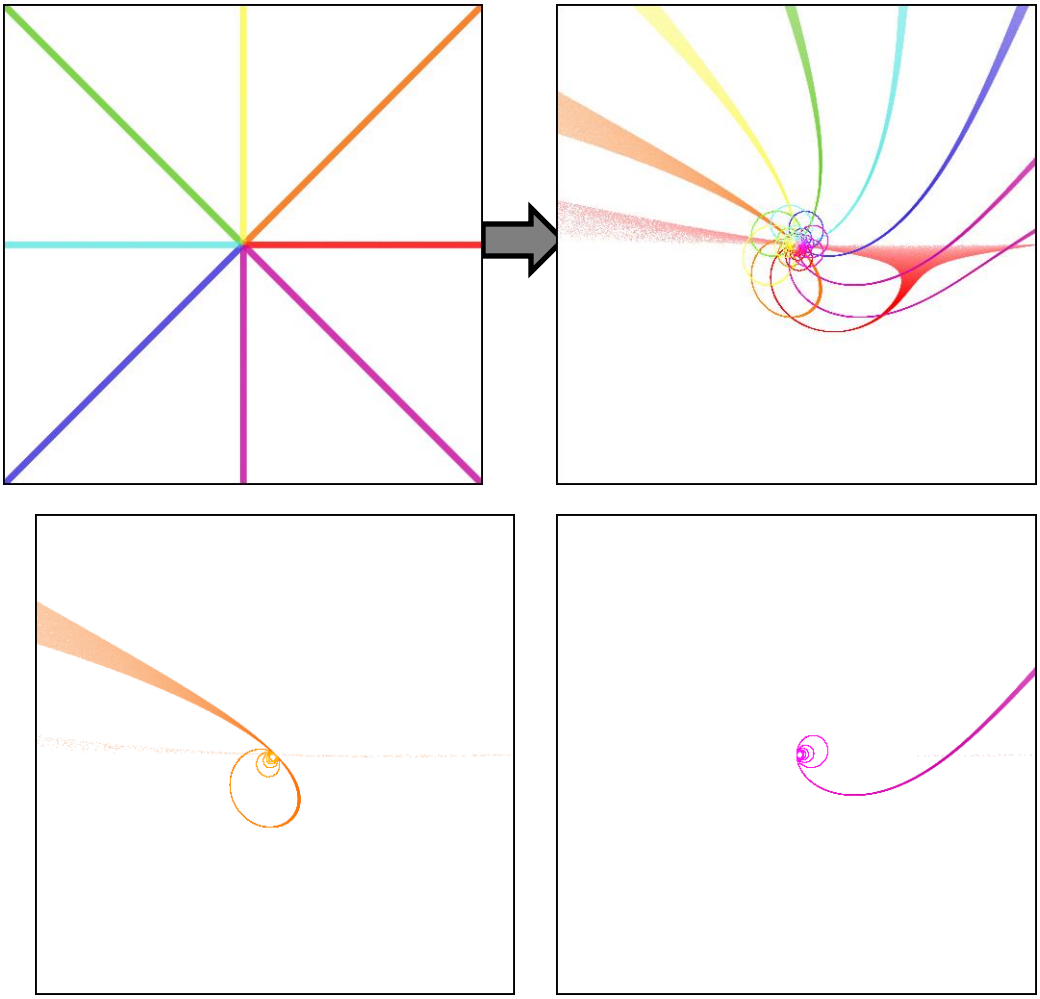


spiral (2D)

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

Maps rays extending from the center into spirals. The example shows an overlay of eight rays going different directions. For clarity, the bottom examples show two of them separately to show the spiral shape. Points closest to the center in the original are furthest from the center after the transform.

The red line in the example looks different from the others because it straddles the x axis. The part above the axis goes to the left, and the part below the axis goes to the right.



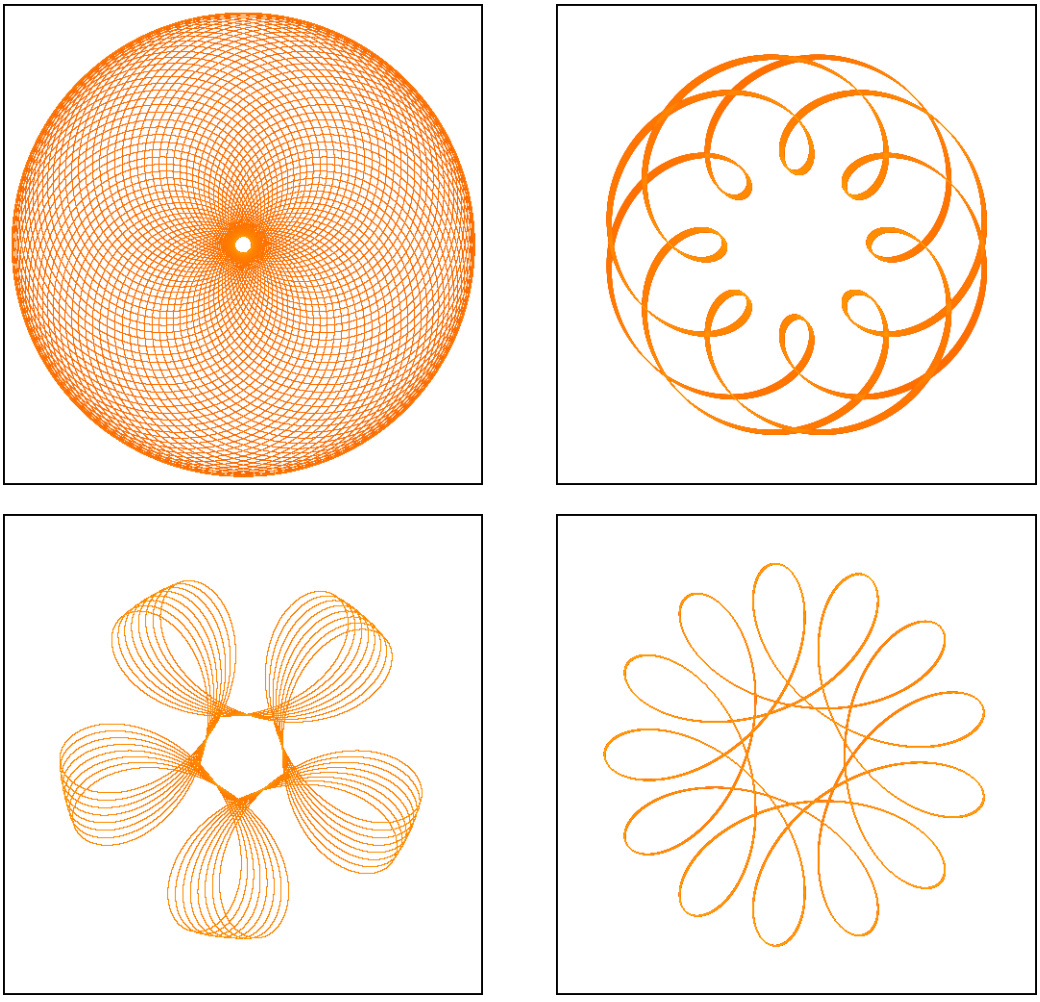
Spirograph (2D blur)

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

SpirographVariationPlugin.dll

Plots an epitrochoid, a figure made by placing a pen a distance *c* from the center of a circle of size *b* rotating around a fixed circle of size *a*, where *c* is actually two variables *c1* and *c2*. Make them equal for an epitrochoid. Make one negative to plot a hypotrochoid, where the moving circle rotates around the inside of the fixed circle instead of the outside. Variables *tmin* and *tmax* control the path length, and *ymin* and *ymax* control the pen thickness.

Top left: *a* = 0.7, *b* = 0.23, *c1* = 1, *c2* = 1, *tmin* = 0, *tmax* = 1000, *ymin* = 0, *ymax* = 0
Top right: *a* = 0.8, *b* = 0.3, *c1* = 0.5, *c2* = 0.5, *tmin* = 0, *tmax* = 50, *ymin* = -0.02, *ymax* = 0.02
Bottom right: *a* = 0.7, *b* = 0.3, *c1* = 0.6, *c2* = -0.6, *tmin* = 0, *tmax* = 50, *ymin* = -0.005, *ymax* = 0.005
Bottom left: *a* = 0.7, *b* = 0.23, *c1* = 0.6, *c2* = -0.6, *tmin* = 0, *tmax* = 49.75, *ymin* = 0, *ymax* = 0



split (2D)

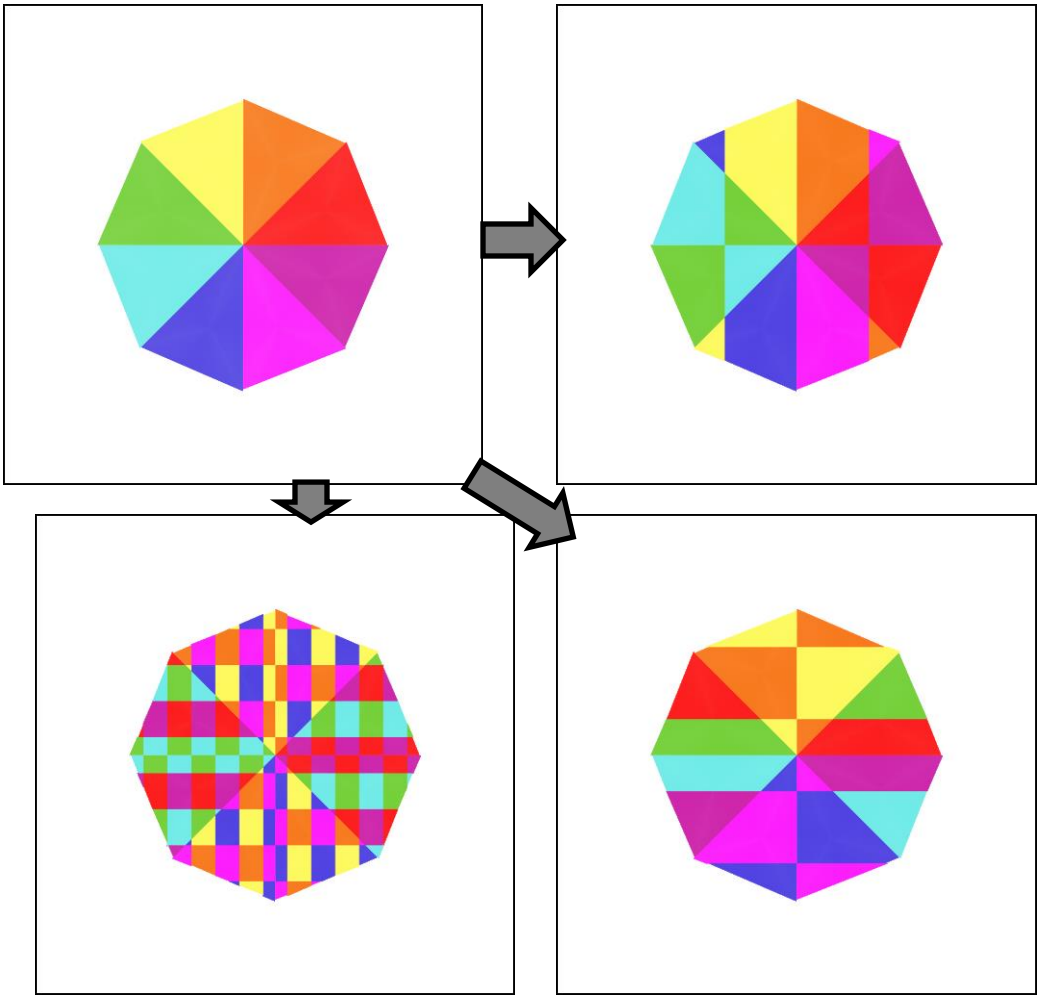
2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	dll

split.dll

Splits the plane into vertical strips with a size defined by *xsize* (higher values give more strips), and flips every other strip across the x axis. Also does the same with horizontal strips defined by *ysize*.

Top left: *xsize* = 0.5, *ysize* = 0
Bottom left: *xsize* =0, *ysize* = 1
Bottom right: *xsize* = 3, *ysize* = 2

Don't confuse with similarly named variation splits.



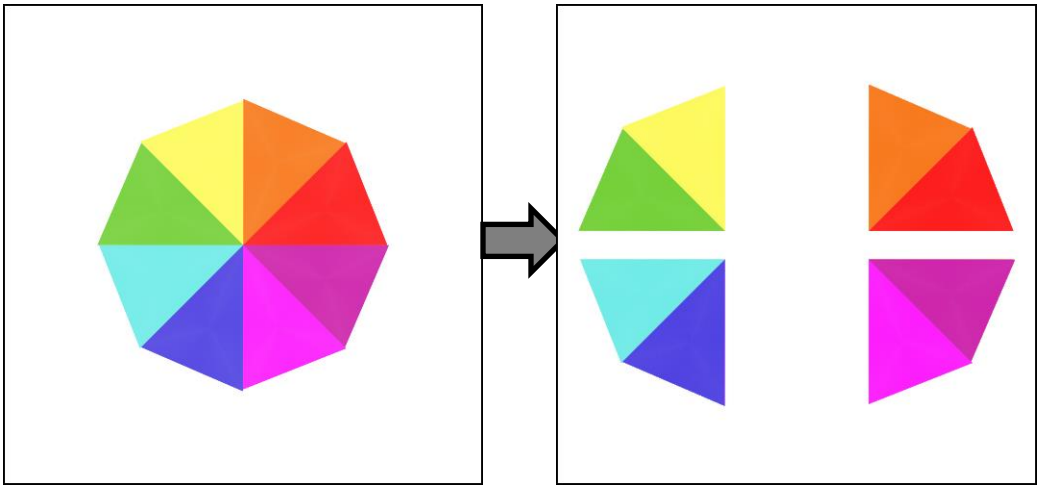
splits (2D)

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

splits.dll

Splits the plane along the x and y axes, shifting the quarters apart to leave a vertical space of size *x* and a horizontal space of size *y*. Example has *x* = 1 and *y* = 0.2.

Compare with separation. Don't confuse with similarly named variation split.

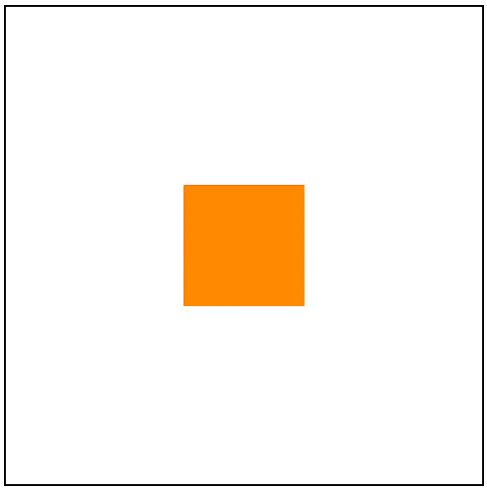


square (2D blur)

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

SquareBlurVariationPlugin.dll

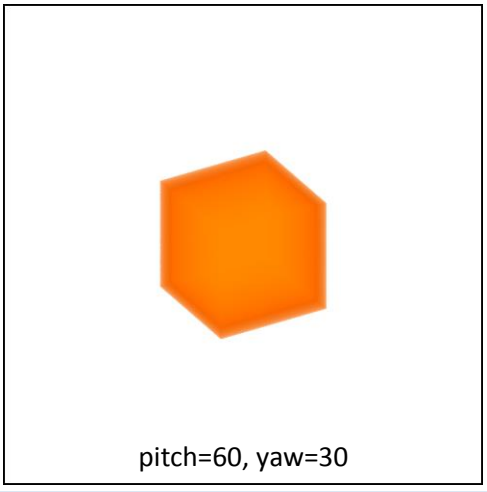
A square. The size is determined by the variation value (1 here).



square3D (3D blur)

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

A cube.

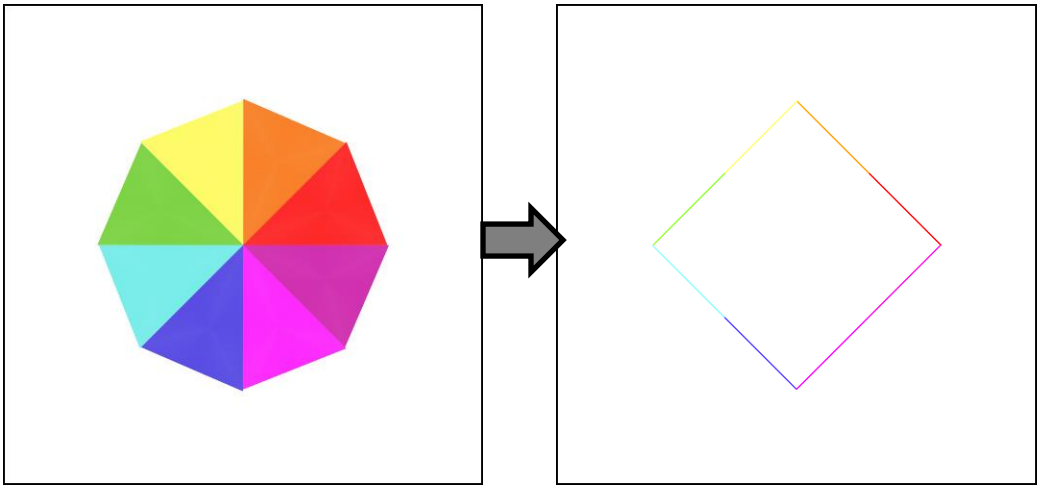


squarical (2D)

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

squarical.dll

Maps the plane into a square outline with corners at (2,0), (0,2), (-2,0), (0,-2) if the variation value is 1; using other values will make the result larger or smaller.



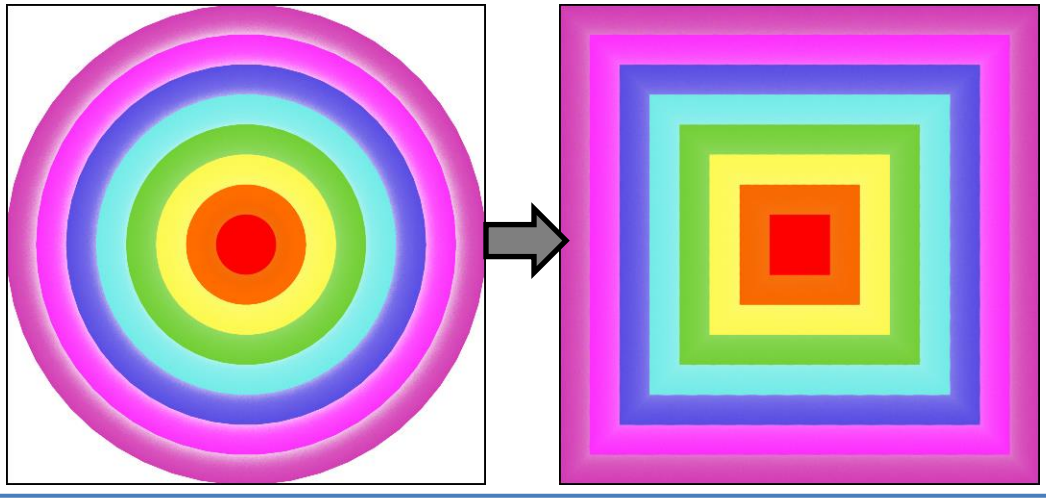
squarize (2D)

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

squarize.dll

Maps circles centered at the origin to squares.

See circlize, which does the opposite.



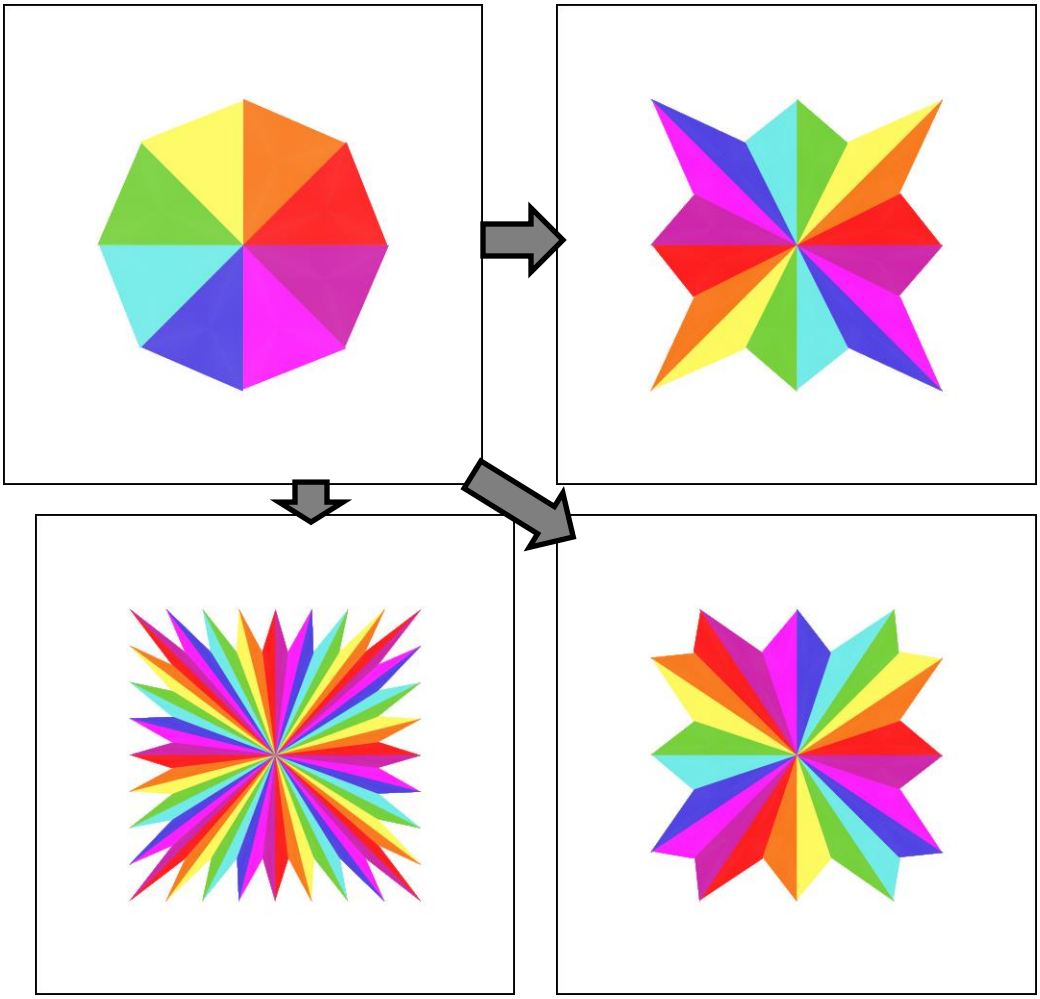
squish (2D)

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

squish.dll

Repeats the plane power times radially as julian does, then maps it to a distinctive square pointy shape.

Top right: *power* = 2
Bottom right: *power* = 3
Bottom left: *power* = 8



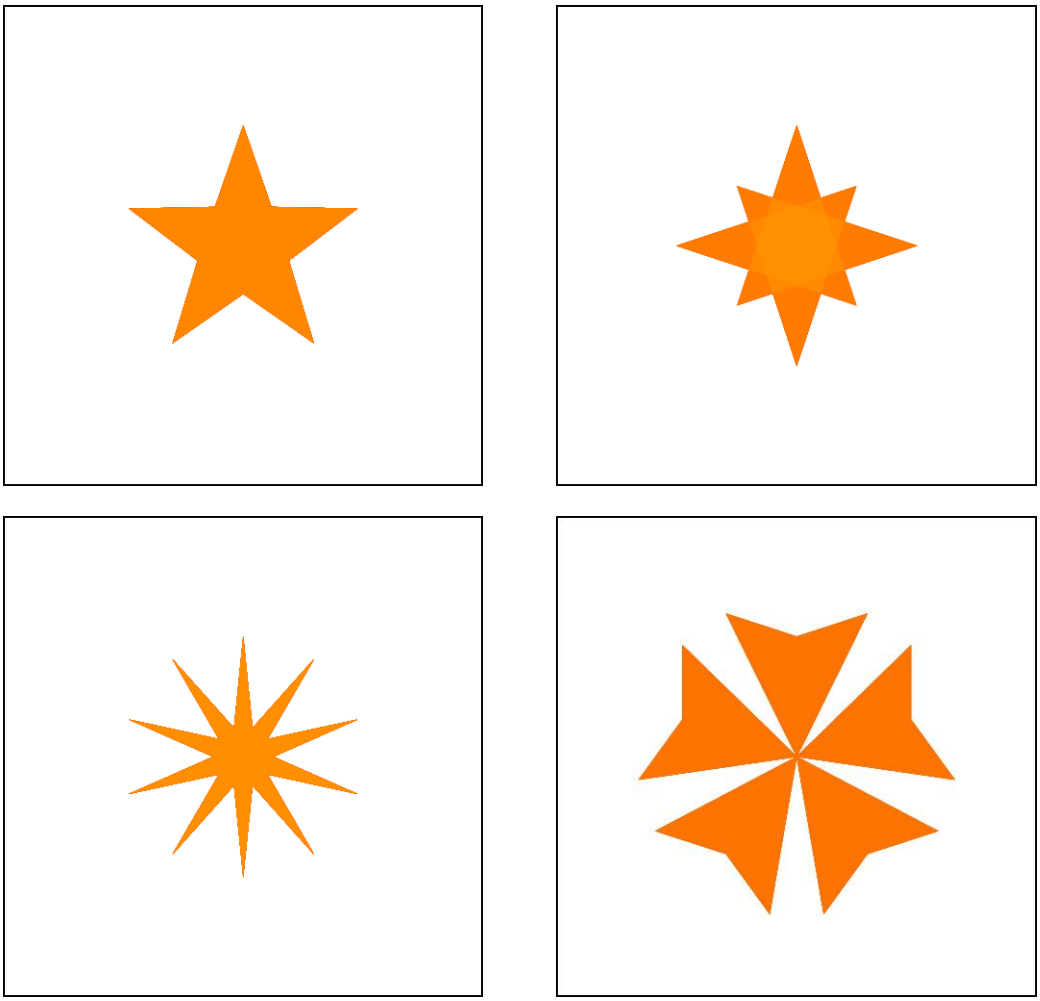
starblur (2D blur)

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

starblur.dll

A star, with variables to control the number of points (*power*) and proportional distance of the inner angles (*range*). Negative values are allowed, and can generate interesting shapes (see examples).

Top left: *power* = 5, *range* = 0.4
Top right: *power* = 4, *range* = -0.7
Bottom left: *power* = 10, *range* = 0.25
Bottom right: *power* = -5, *range* = 1



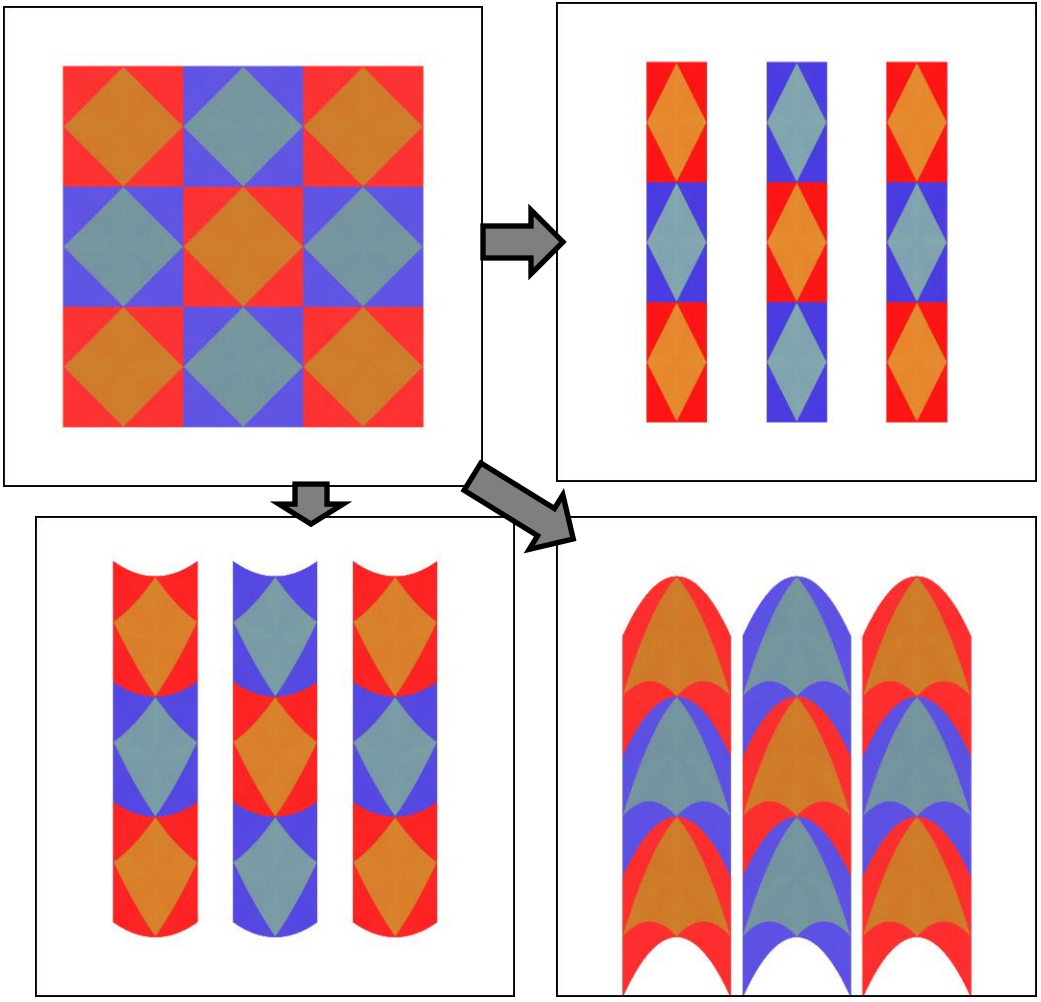
stripes (2D)

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

stripes.dll

Divides the plane into stripes with width 1, then shrinks each according to *space* (0 won't shrink at all, 1 will shrink the stripes to lines). Then warps each of the stripes based on *warp*.

Top right: *space* = 0.5, *warp* = 0
Bottom right: *space* = 0.1, *warp* = 2
Bottom left: *space* = 0.3, *warp* = -0.5



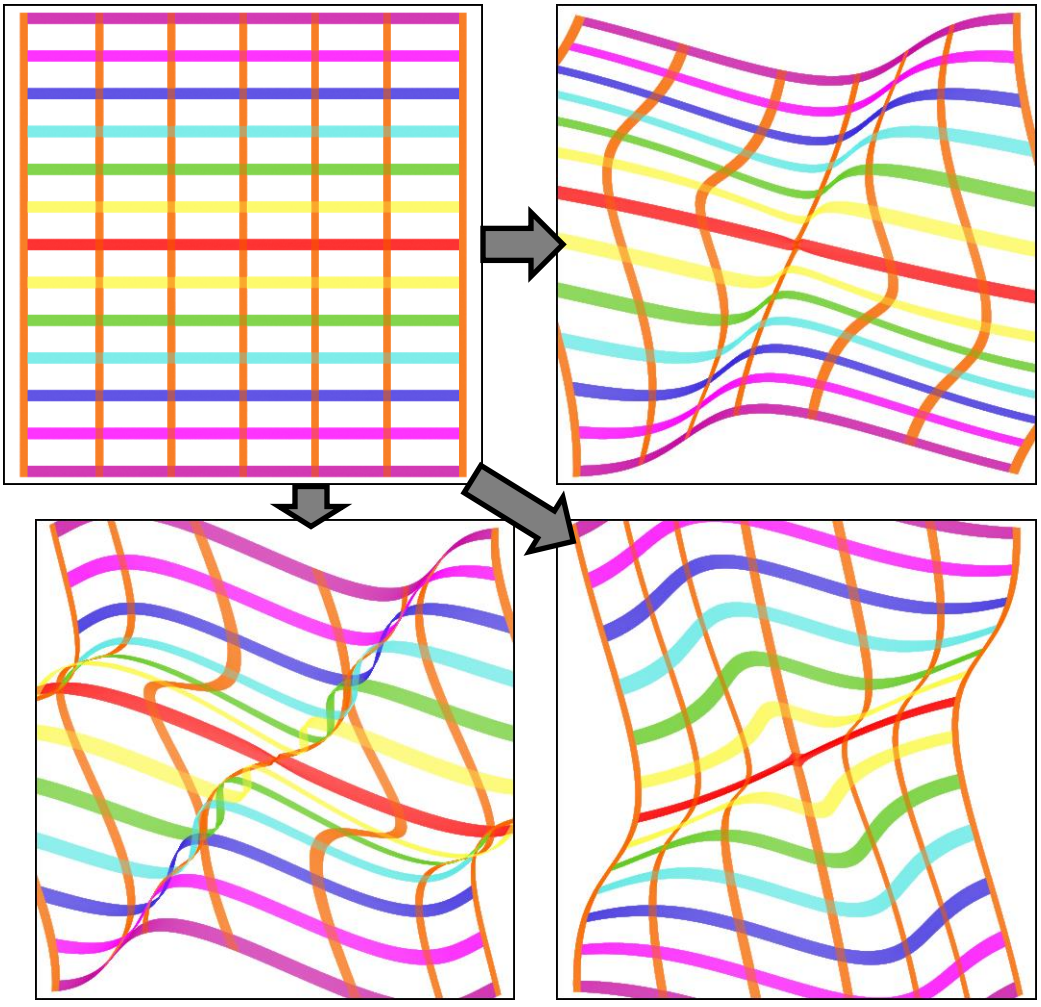
stwin (2D)

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

stwin.dll

Distorts the plane according to variable *distort*. It gives the appearance of warping the plane in the third dimension, but this is a 2D only variation.

Top right: *distort* = 1
Bottom right: *distort* = -1
Bottom left: *distort* = 2



super_shape (2D)

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

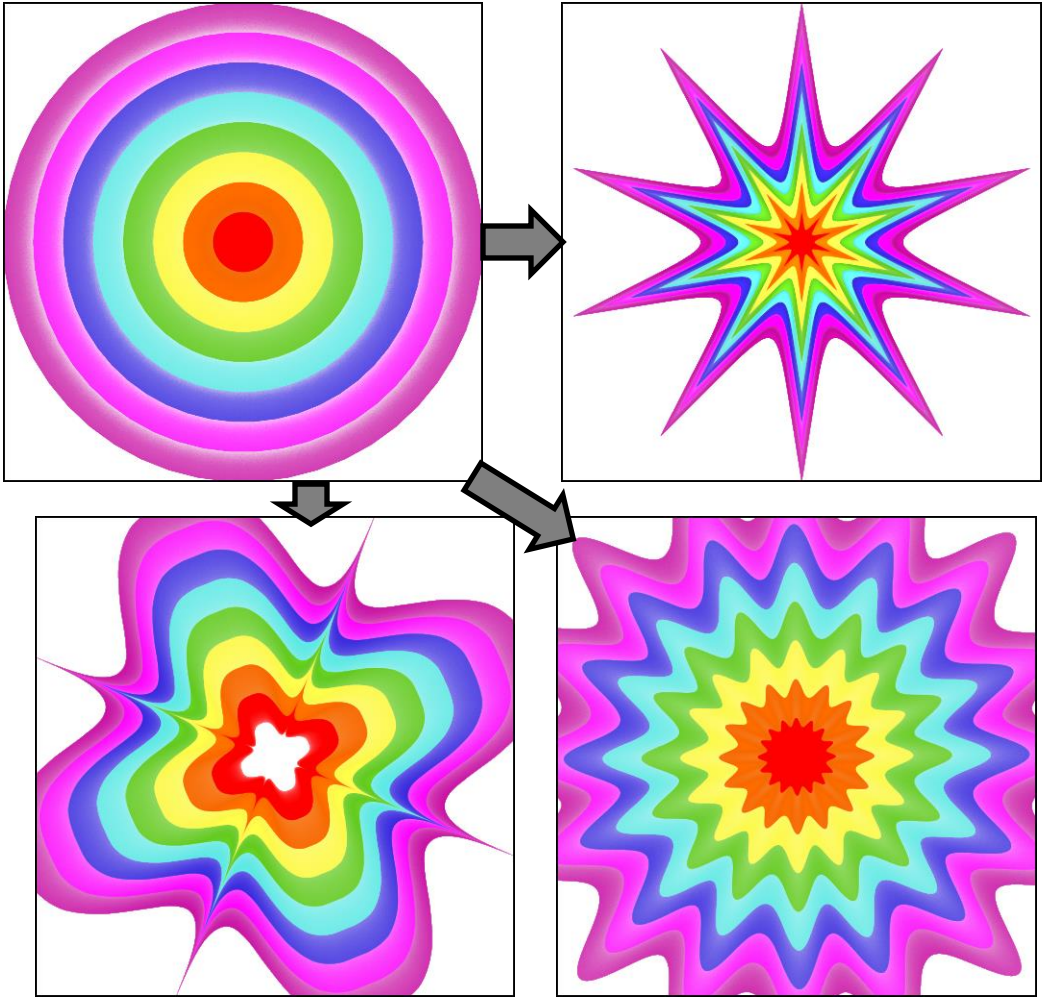
supershape.dll

Transforms the plane based on a formula developed by botanist Johan Gielis to describe shapes found in nature, known as the superformula. Variable *m* is the number of sections for the shape, and *n1*, *n2*, and *n3* control the shape.

The superformula has two other variables *a* and *b* that stretch or shrink the result. They are always set to 1 for this variation; there is no way to adjust them. But this variation does have two variables, *rnd* and *holes* that further modify the shape.

Top right: *m* = 10, *n1* = 0.4, *n2* = 1, *n3* = 1, *rnd* = 0, *holes* = 0
Bottom right: *m* = 16, *n1* = 2.5, *n2* = 5, *n3* = 3, *rnd* = 0, *holes* = 0
Bottom left: *m* = 8, *n1* = 1, *n2* = 0.3, *n3* = 3, *rnd* = 0, *holes* = -0.3

See shape and SuperShape3d.



SuperShape3d (3D blur)

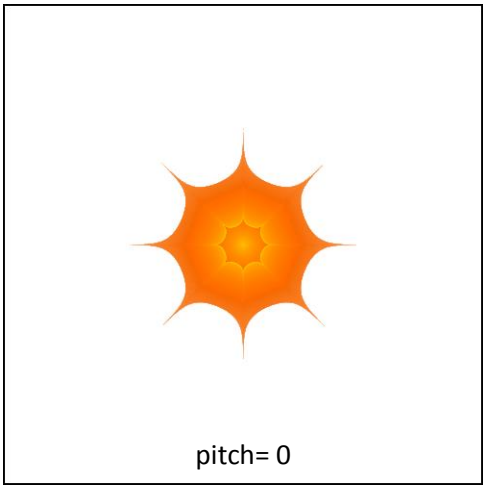
2.09	7X15B	7X16	jwf	ch
no	dll	dll	yes	no

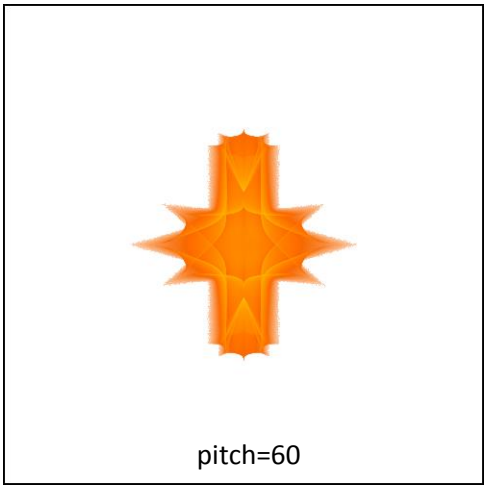
SuperShape3d.dll
A 3D shape based on two superformula instances (see super_shape for a basic description). The first is for the xy plane, and only generates an outline The second is for the yz plane, and generates the solid.

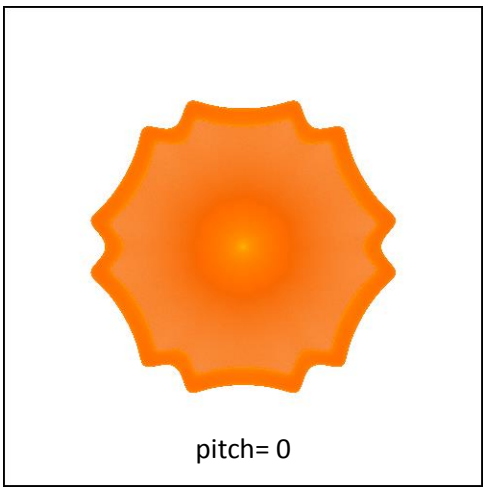
Variable *rho* sets the upper bound for xy drawing; when *m1* is an integer, the optimal value is $\pi^2 \approx 9.8696$. Increase when *m1* has a fractional part. Variable *phi* does the same for the yz drawing. A typical value is $\pi^2/4 \approx 2.4674$, which fills a quarter of the shape. Adjust as needed.

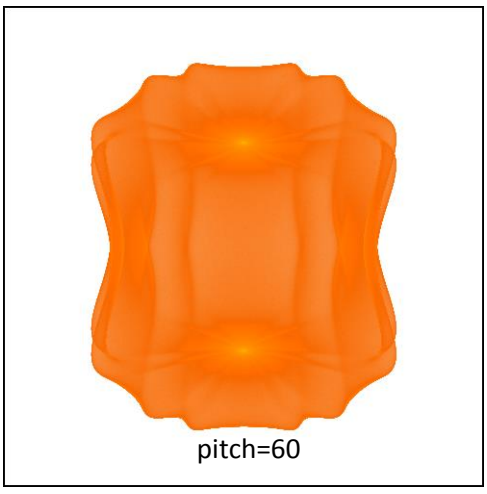
In all the examples, *rho* = 9.8696, *phi* = 2.4674, and *a1*, *a2*, *b1*, and *b2* = 1.
Top: *m1* = 8, *n1_1* = 1, *n2_1* = 0.3, *n3_1* = 0.3, *m2* = 5, *n1_2* = 0.4, *n2_2* = 1, *n2_3* = 1
Bottom: *m1* = 12, *n1_1* = 15, *n2_1* = 20, *n3_1* = 7, *m2* = 4, *n1_2* = 4, *n2_2* = 7, *n2_3* = 7

See shape and super_shape.







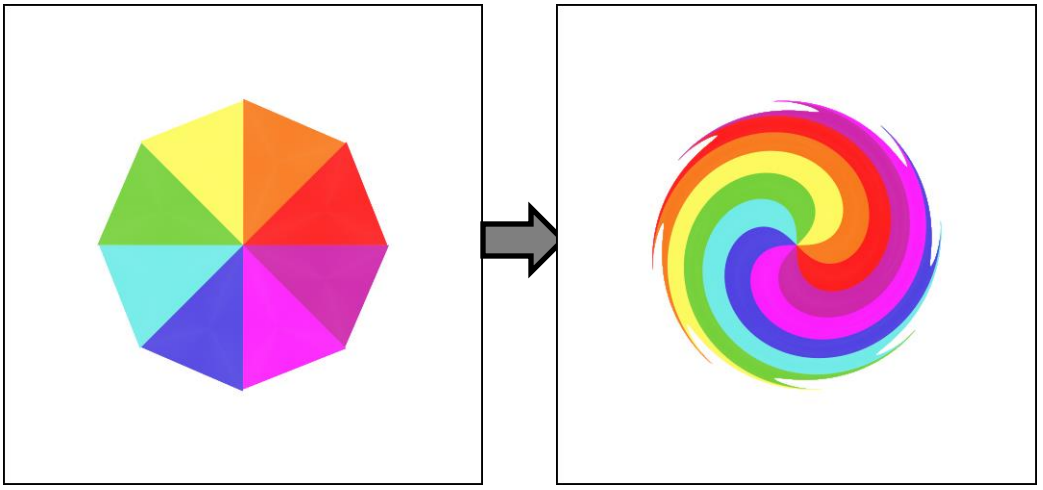


swirl (2D)

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

Twists the plane counter-clockwise, mapping wedges to spirals.

See swirl2.

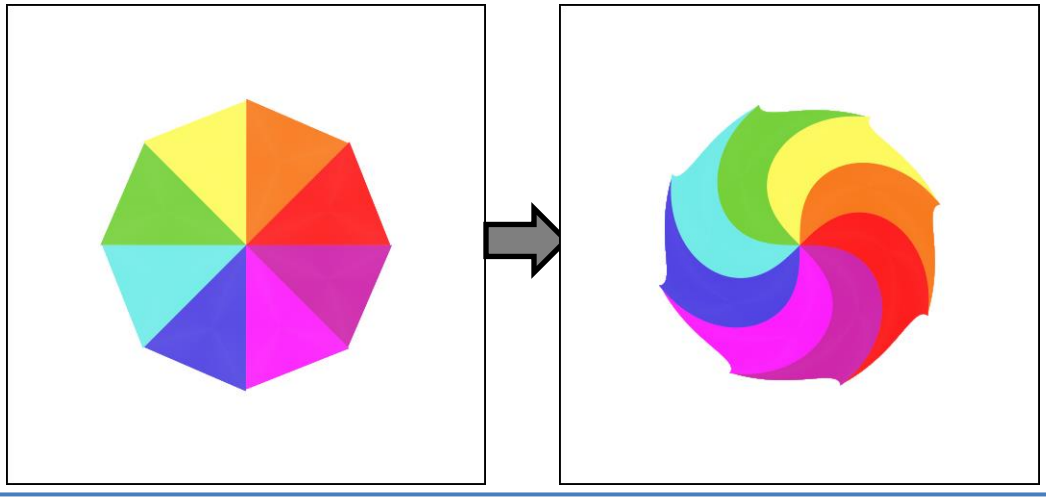


swirl2 (2D)

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

swirl2.dll

Twists the plane clockwise with variables to control the effect: *twistamount* is how much to twist; example uses 0.1, use 1/π=0.31831 for same amount as swirl (but opposite direction). *radius* specifies size; use 1 (shown) to keep original size. *centerx* and *centery* set the center of the effect.

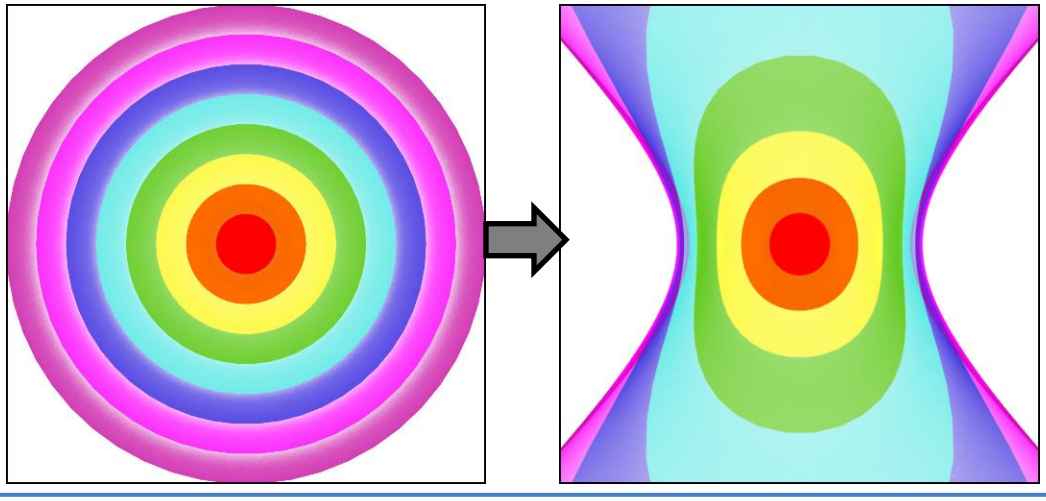


tangent (2D)

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

Maps the plane to a shape based on the tangent function, but it's probably easier to think of this as similar to sinusoidal, except it maps to an hourglass shape instead of a square.

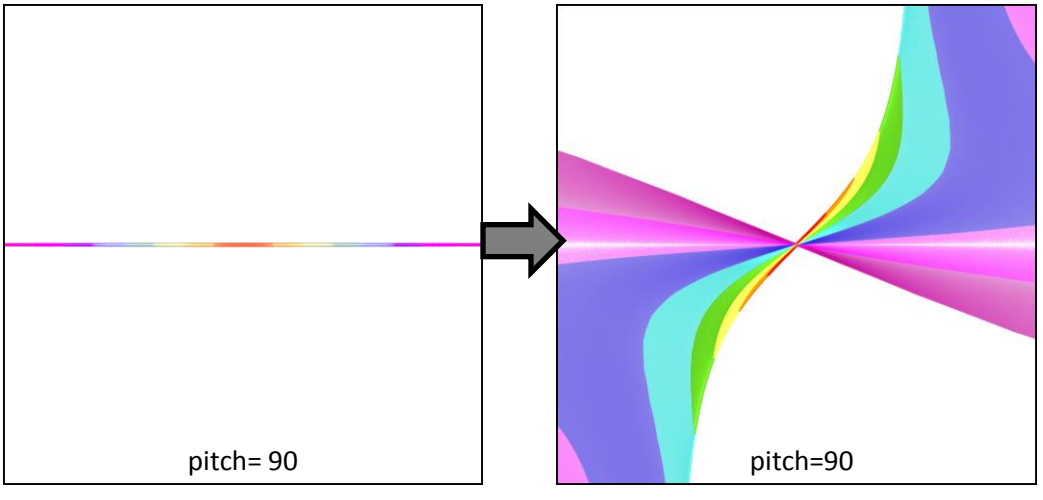
Although there is no tangent plugin, there is a **Z_tangent** plugin named Ztangent.dll. It has variables; set them all to 1 to mimic tangent.



tangent3D (3D, sets z)

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

Like tangent, but also sets z. The sample shows an edge-on view to show the 3D shape. With pitch = 0, it is the same as tangent.



text_wf (2D blur)

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

Adds text to a fractal. Variables allow specifying the text string, font, size, and offset (location).

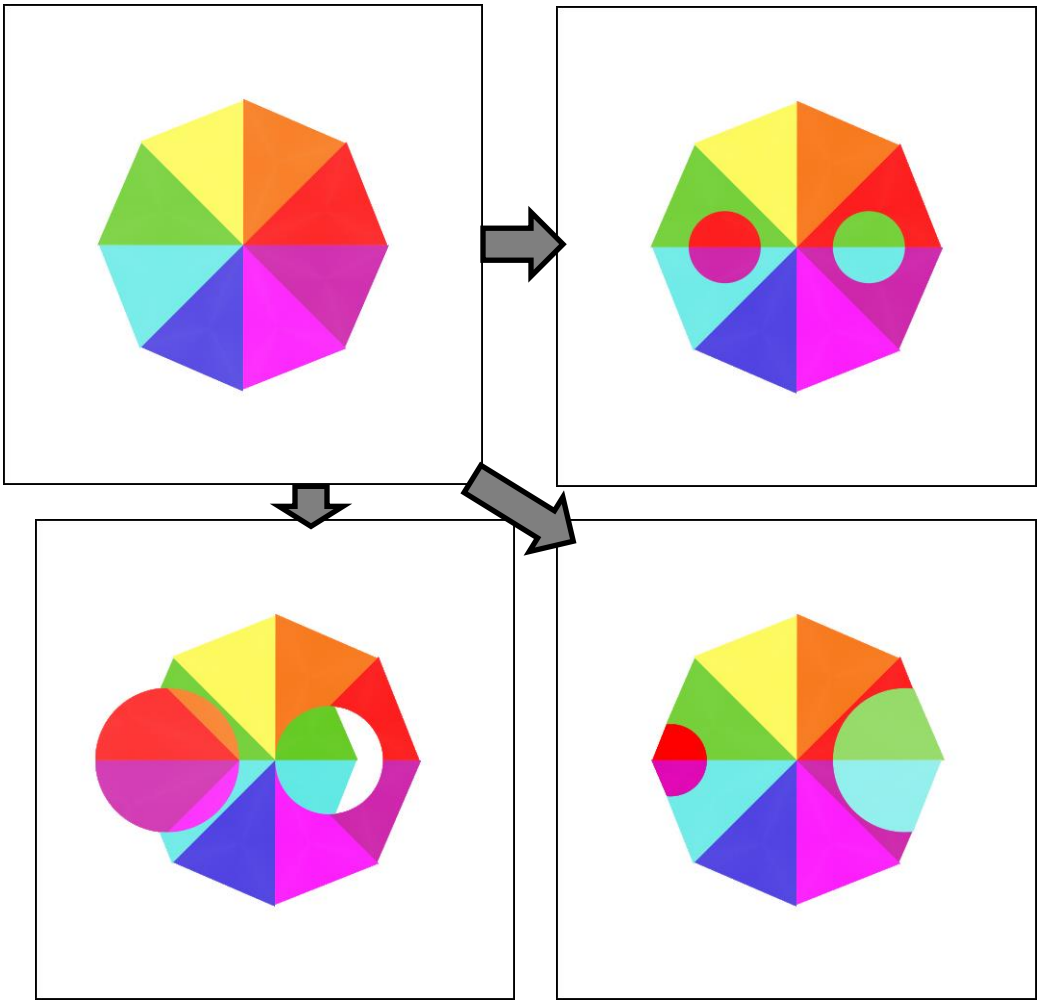
trade (2D)

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

trade.dll

Takes one circle of radius *r1* with its left edge distance *d1* to the right of the origin and another circle of radius *r2* with its right edge distance *d2* to the left of the origin, and trades them, scaling as needed, and flipping right and left.

Top right: *r1* = 0.5, *d1* = 0.5, *r2* = 0.5, *d2* = 0.5
Bottom right: *r1* = 1, *d1* = 0.5, *r2* = 0.5, *d2* = 1.25
Bottom left: *r1* = 0.75, *d1* = 0, *r2* = 1, *d2* = 0.5

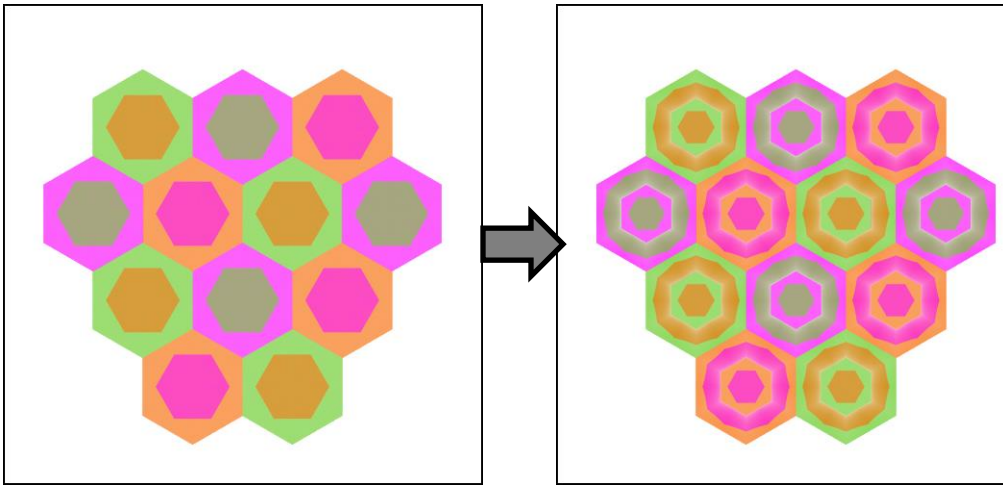


tri_boarders2 (2D)

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

Tri_boarders2.dll
Divide the flame into hexagons, and make a copy of each. Shrink one copy and keep in the middle. Poke a hexagonal hole in the other and expand it to make a frame around the first. The sample has *radius* = 0.5 and *width* = 0.5.

See boarders2 and xtrb.



Truchet (2D)

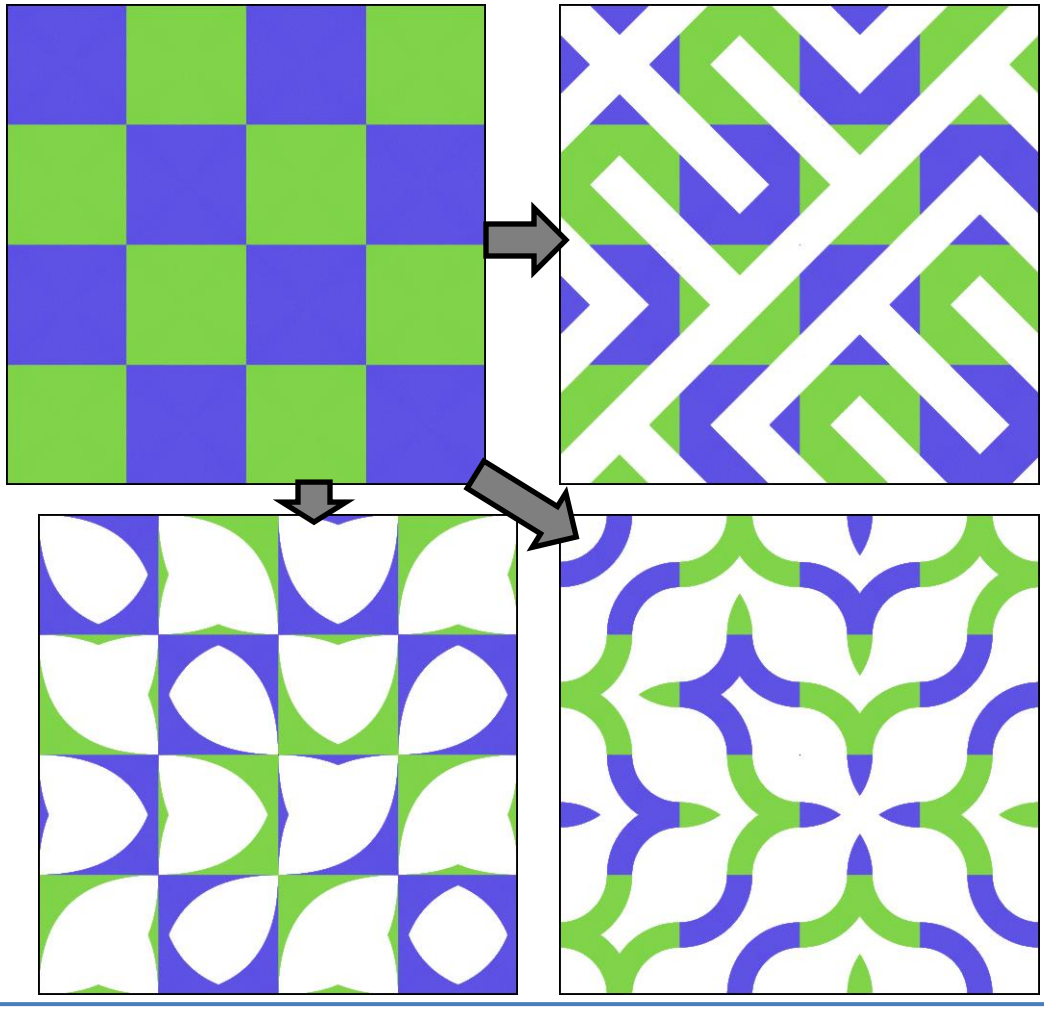
2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	dll

Truchet.dll

Crops the plane to a random Truchet tiling. The plane is divided into square tiles with each face either connected to one, or both adjacent faces or teminated in a point. The squares in the sample match the truchet squares. Set *exponent* to 1 for a straight line (top left), 2 for a quarter circle (bottom left), or other values between 0.008 and 2 for other shapes (bottom right uses 0.5).

The width of the lines is determined by *arc_width*, ranging from 0.001 (barely visible) to 1 (full size). Change *seed* to get a different random pattern.

Top left: *exponent* = 1, *arc_width* = 0.5, *seed* = 35
Bottom left: *exponent* = 2, *arc_width* = 0.5, *seed* = 70
Bottom right: *exponent* = 0.5, *arc_width* = 0.33, *seed* = 12.34

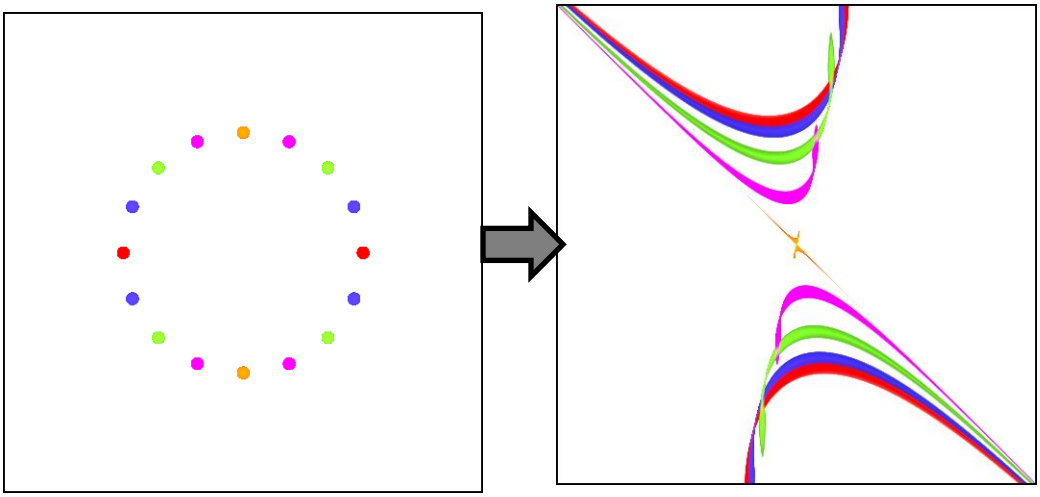


twintrian, twintrian2 (2D)

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

twintrian2.dll

The built-in version is twintrian; the plugin version is twintrian2. Both work the same. It maps dots to arcs, but not in a very predictable way. For points close to the center, the result tends to have two branches as shown. Points on the left map to the top; points on the right to the bottom.

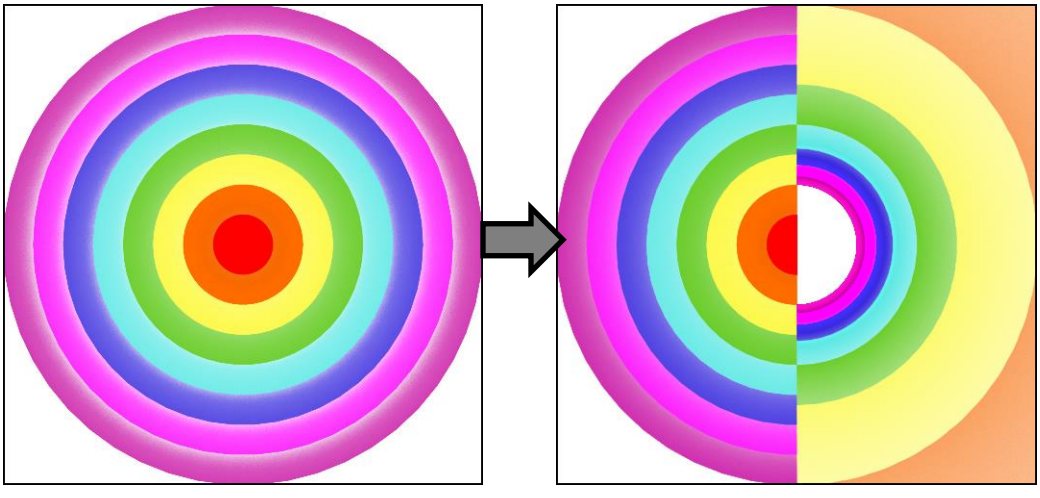


twoface (2D)

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

twoface.dll

A combination of linear on the left and spherical on the right.



unpolar (2D)

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

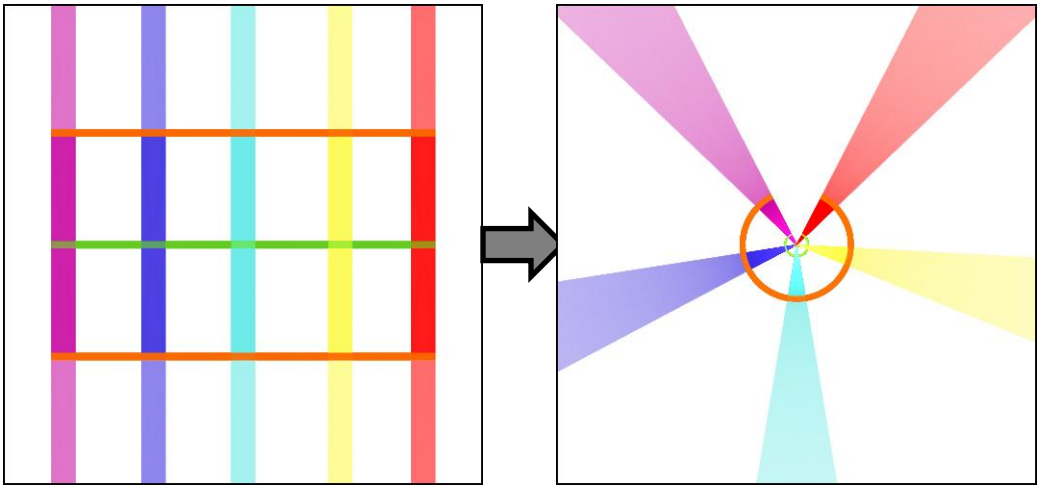
unpolar.dll

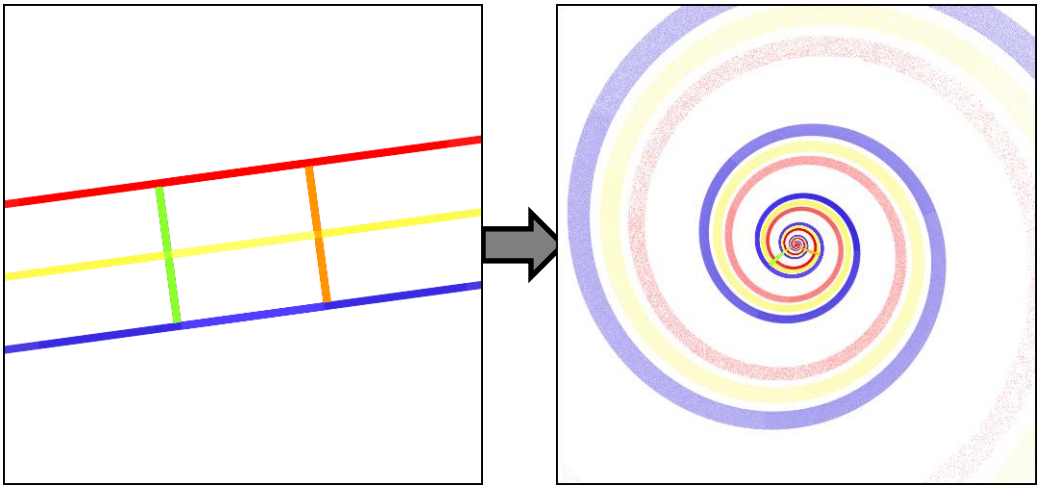
Wraps the plane around a horizontal cylinder, then views it through the end.

In the first example, the vertical lines are infinite and map to rays emanating from the center due to the wrapping. The top of the original maps to the center.

The second example is more complex, but better shows how unpolar works. The infinite horizontal lines are slanted slightly, so when wrapped around the cylinder they spiral around. These spirals can be seen when the cylinder is view through the end.

See foci.





voron (2D)

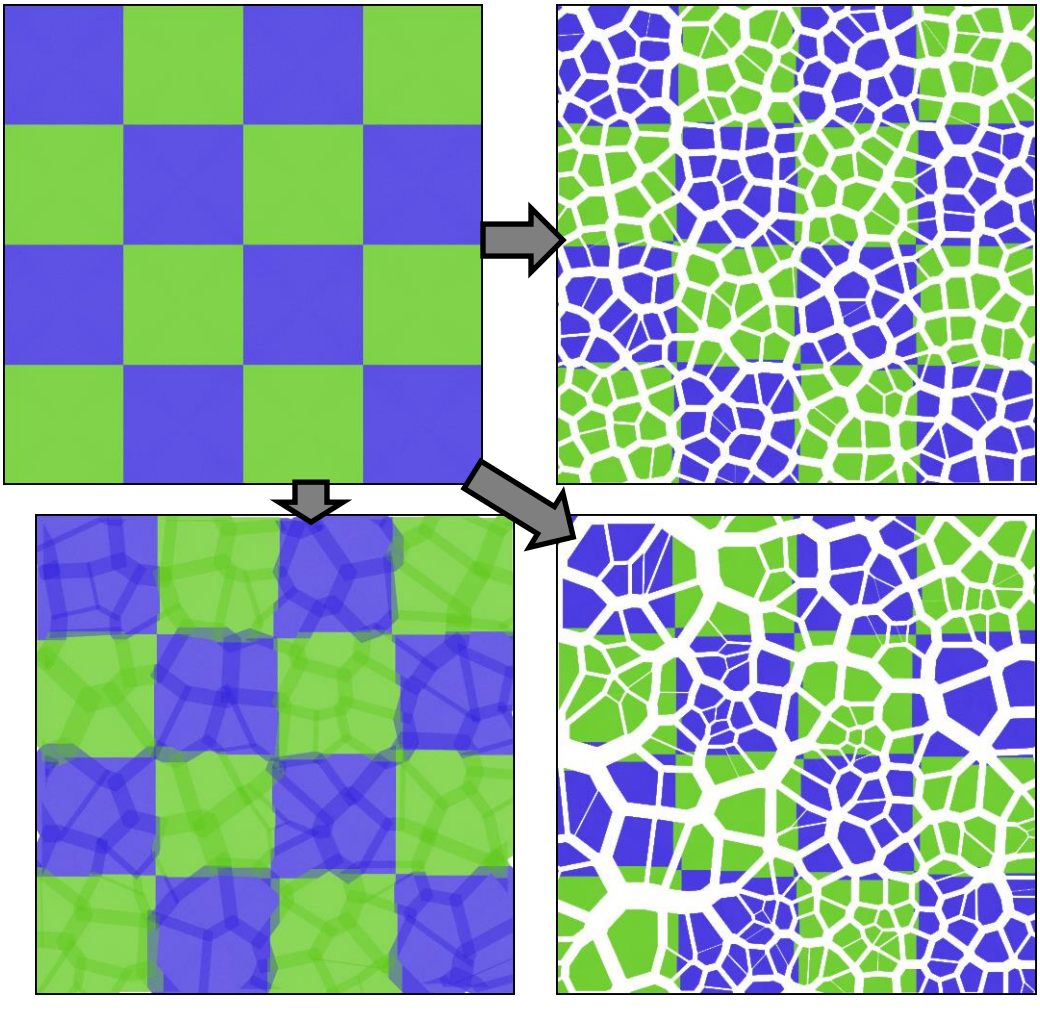
2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	dll

voron.dll

Maps the plane to a random voronoi tessellation. A set of random points is chosen, and the plane is divided into regions based on the nearest random point. Each region is scaled by *K*; if less than 1, boundaries between the regions are seen; if greater, the regions overlap.

Variable *Step* determines the spacing of the random points, and so the size of the regions. The process is repeated *Num* times, resulting in finer details (but much slower performance) as *Num* increases.

Top right: *K* = 0.75, *Step* = 0.25, *Num* = 1
Bottom right: *K* = 0.75, *Step* = 1, *Num* = 25
Bottom left: *K* = 1.2, *Step* = 0.5, *Num* = 2



w (2D)

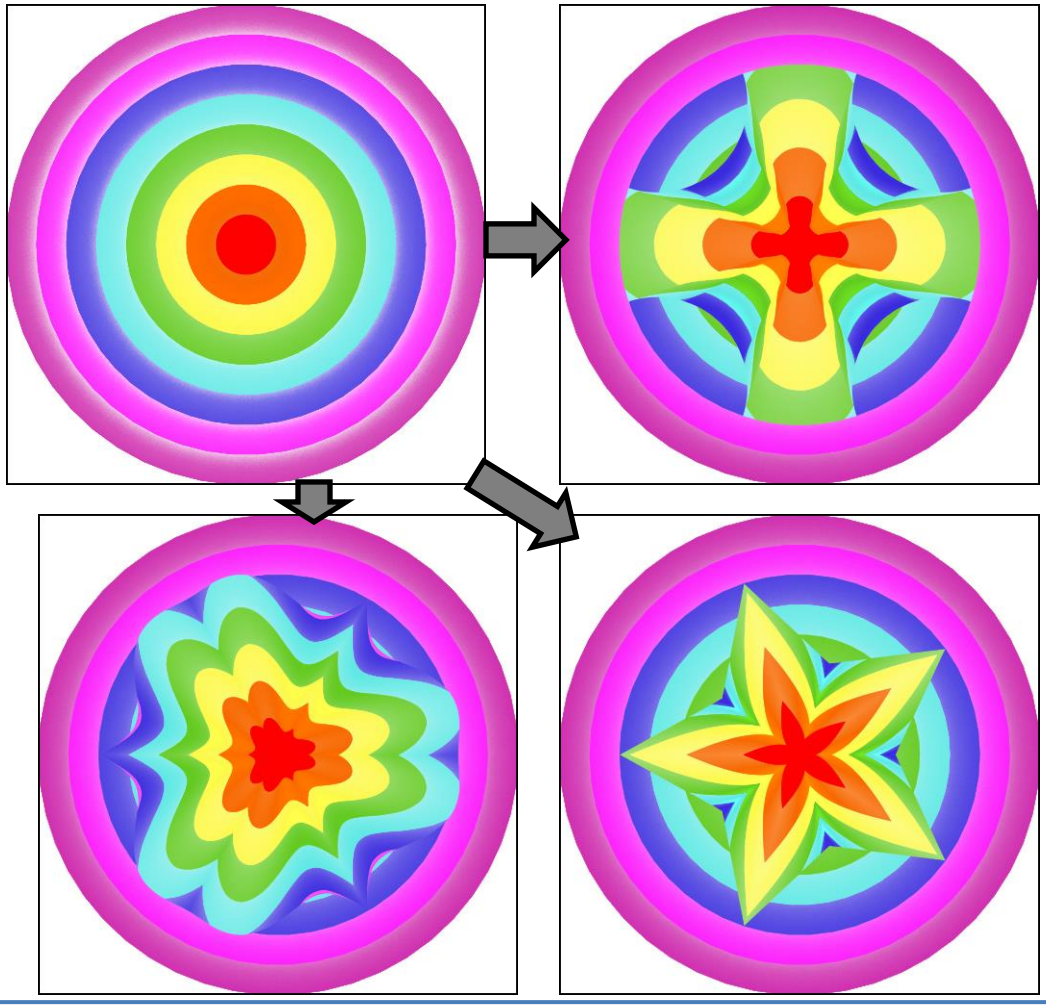
2.09	7X15B	7X16	jwf	ch
dll	dll	dll	no	dll

w.dll

Contorts the inner part of the plane to various shapes (hypergon, star, and supershape shown; lituus and combos not shown). The contorted part is rotated *angle* radians, though that is not apparent in the samples due to the starting point (chosen to be consistent with variations x, y, and z); *angle* also controls the amount of the effect; the values for the samples are chosen to maximize the effect.

Top right: *angle* = 0.785 (45°), *hypergon* = 1.5, *hypergon_n* = 4, *hypergon_r* = 0.5
Bottom right: *angle* = 0.628 (36°), *star* = 1.5, *star_n* = 5, *star_slope* = 0.5
Bottom left: *angle* = 1.05 (60°), *super* = 1.5, *super_m* = 6, *super_n1* = 0.75, *super_n2* = 1, *super_n3* = 3

See also x, y, and z.



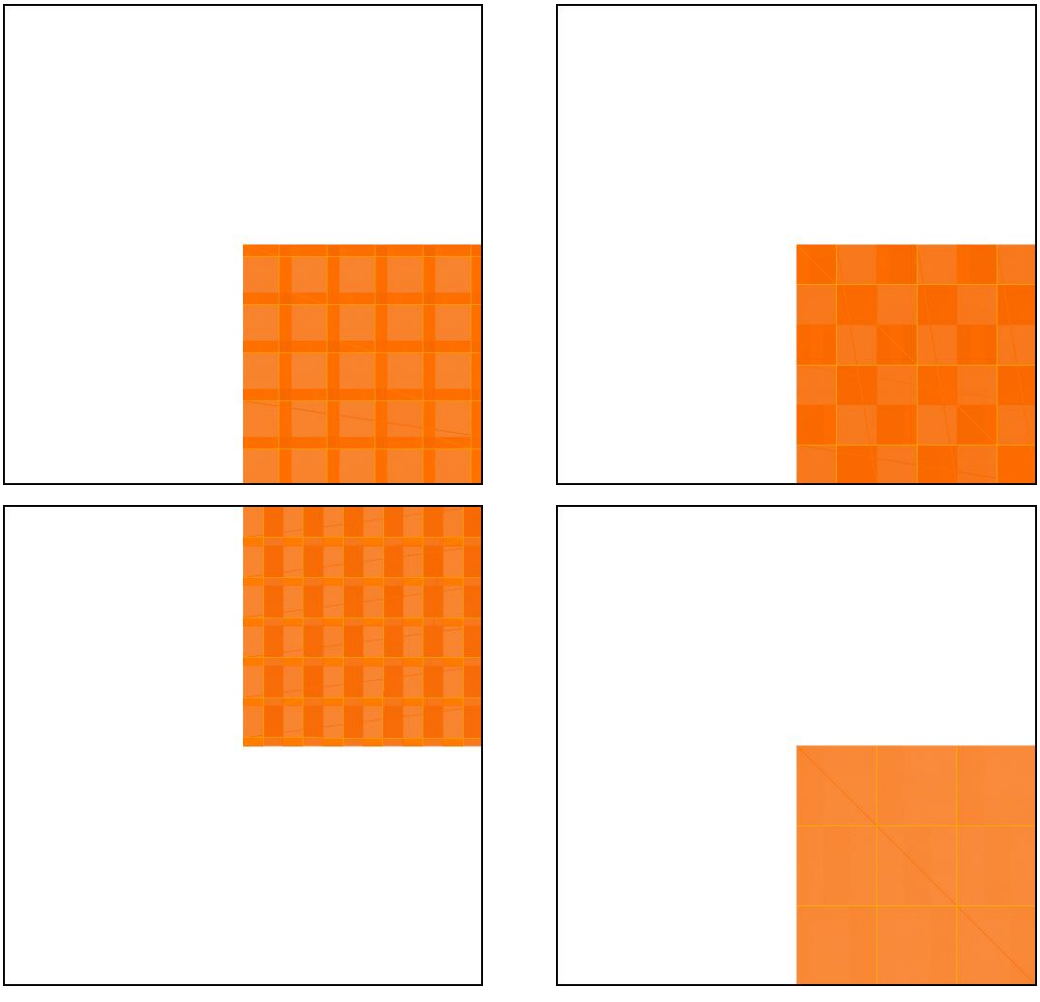
waffle (2D blur)

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

WaffleVariationPlugin.dll

A square shape with cross-hatching to give the appearance of a waffle. Note that the cross-hatching is not apparent with all colors. Variables control the number of divisions and thickness of the lines.

Top left: *slices* = 5, *xthickness* = 0.75, *ythickness* = 0.25, *rotation* = 0
Top right: *slices* = 3, *xthickness* = 0.5, *ythickness* = 0.5, *rotation* = 0
Bottom right: *slices* = 3, *xthickness* = 1, *ythickness* = 1, *rotation* = 0
Bottom left: *slices* = 6, *xthickness* = 0.2, *ythickness* = 0.5, *rotation* = 1.57



waves2 (2D, but 7X16 is 3D)

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

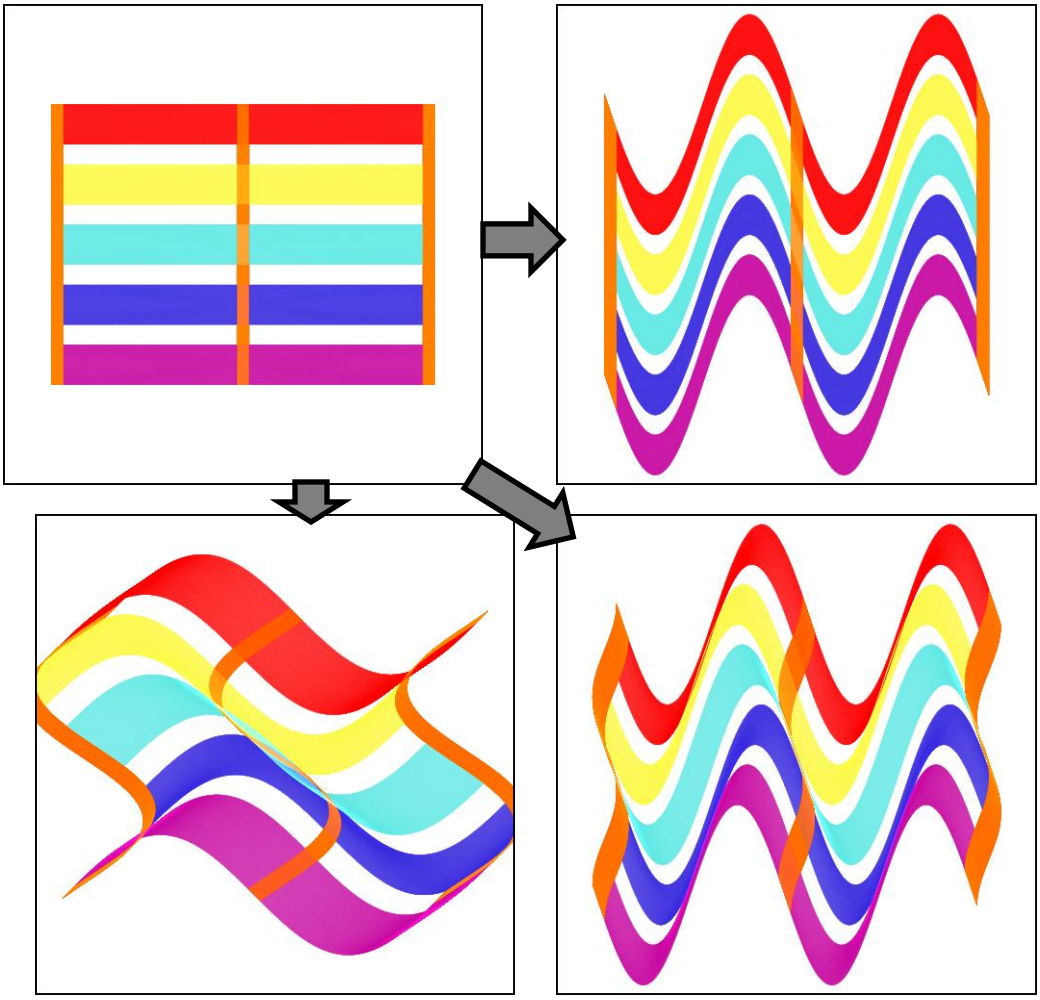
waves2.dll

Creates a wave effect independently on x and y applied equally through the plane. Variables *freqx* and *scalex* specify the frequency and scale for x, and *freqy* and *scaley* specify the frequency and scale for y.

Top right: *freqx* = 0, *freqy* = 4, *scaley* = 0.25
Bottom right: *freqx* = 4, *scalex* = 0.1, *freqy* = 4, *scaley* = 0.25
Bottom left: *freqx* = 3, *scalex* = 0.5, *freqy* = 2, *scaley* = 0.5

The version built in to 7X16 has additional *freqz* and *scalez* variables.

Supercedes **waves**.
Compare auger, waves2b, wavesn.



waves2b (2D)

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

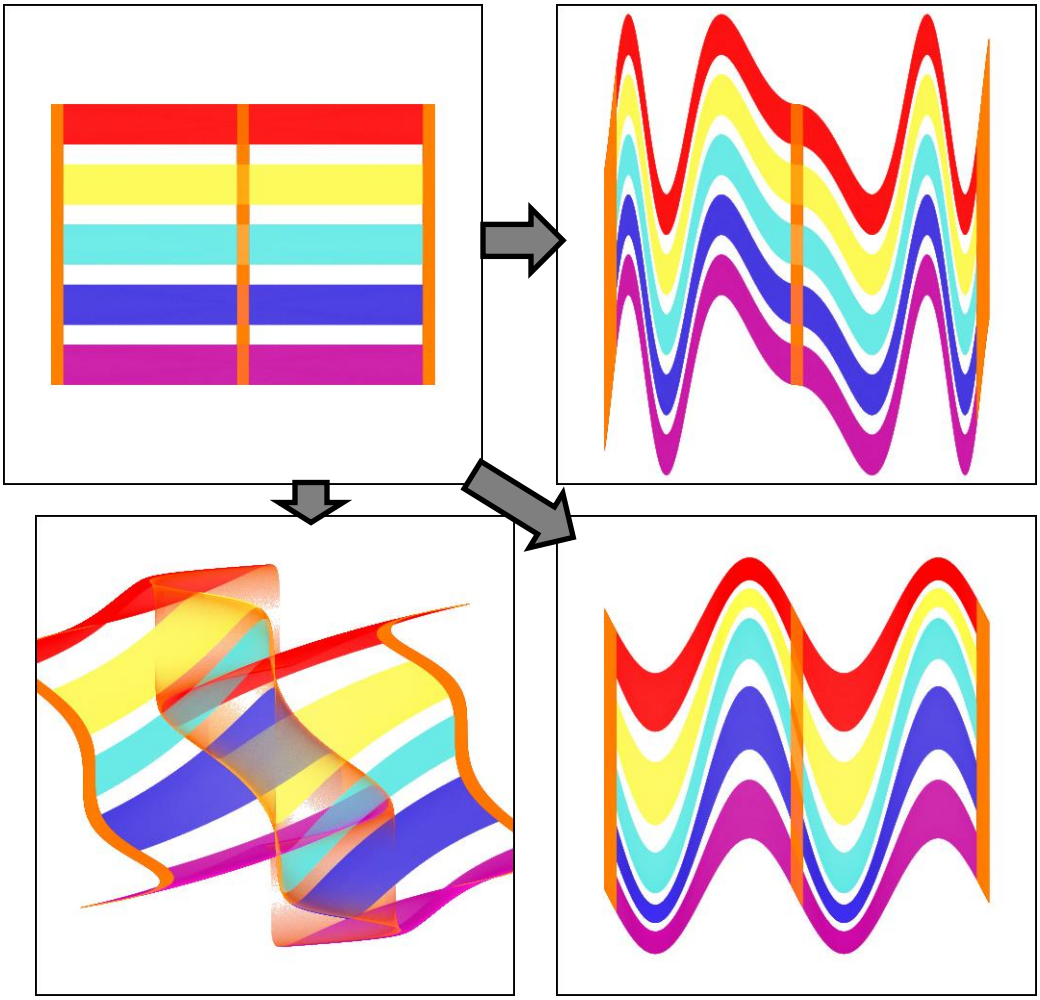
waves2b.dll

Creates a wave effect independently on x and y like waves2, but with more variables. The wave is raised to the *pwx* and *pwy* power, distorting the wave (top right). The scale can be different at the center (*scalex* and *scaley*) and edges (*scaleinfx* and *scaleinfy*); the bottom right example decreases the scale at the edges (opposite of auger). Compare top and bottom right with waves2, top right.

Bottom left example mixes all the options.

Top right: *freqx* = 0, *freqy* = 4, *pwy* = 2, *scaley* = 0.75, *scaleinfy* = 0.75
Bottom right: *freqx* = 0, *freqy* = 4, *pwy* = 1, *scaley* = 1, *scaleinfy* = 0.1
Bottom left: *freqx* = 2, *freqy* = 3, *pwx* = 3, *pwy* = 0.25, *scalex* = 1, *scaleinfx* = 0.5, *scaley* = 0.5, *scaleinfy* = 0

Compare auger, waves2, wavesn



wavesn (2D)

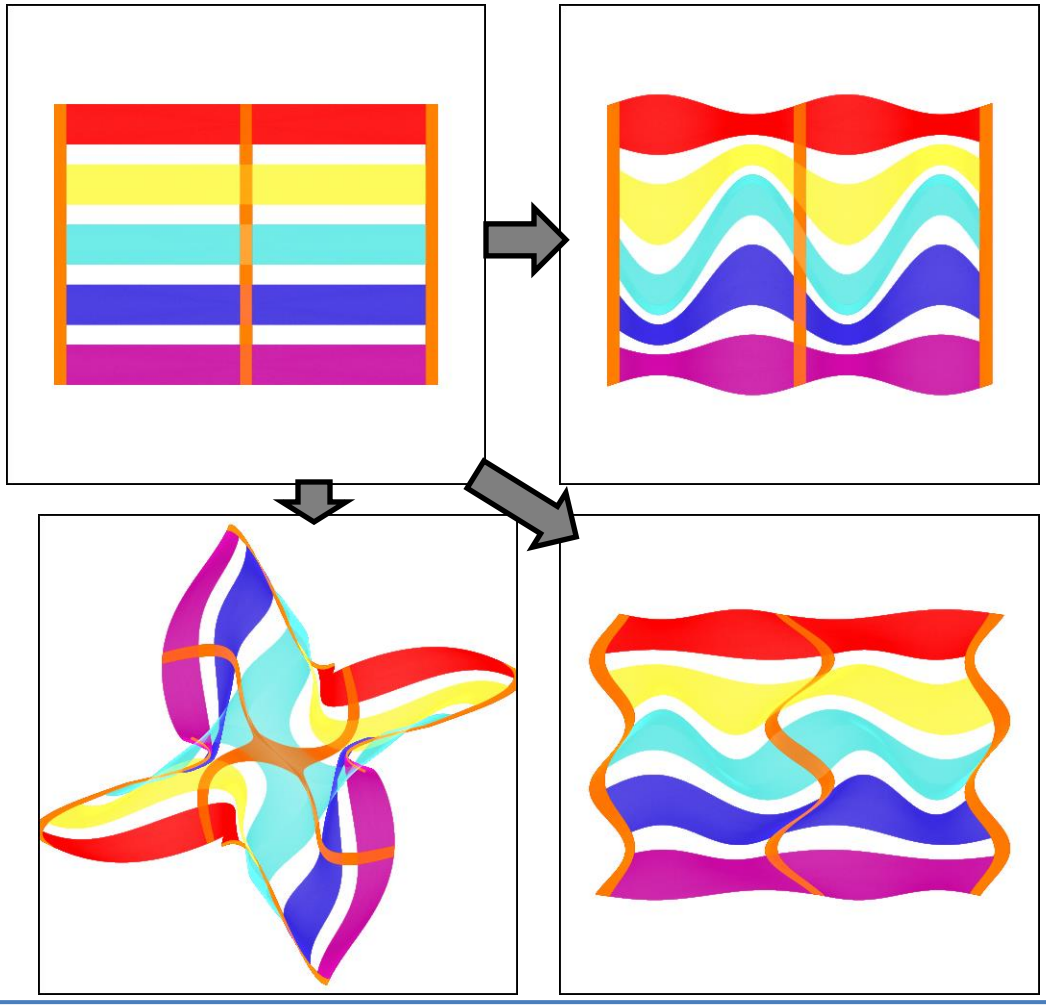
2.09	7X15B	7X16	jwf	ch
dll	dll	dll	no	dll

wavesN.dll

First does an optional radial replication like julian (variable *power*), then adds a wave effect independently on x and y with frequencies *freqx* and *freqy*, starting with scales *scalex* and *scaley* at the center and changing outward by *incx* and *incy*.

Top right: *freqx* = 0, *freqy* = 4, *scalex* = 0, *scaley* = 1, *incx* = 0, *incy* = -1, *power* = 1
Bottom right: *freqx* = 6, *freqy* = 4, *scalex* = 0.5, *scaley* = -0.5, *incx* = -0.5, *incy* = 0.5, *power* = 1
Bottom left: *freqx* = 5, *freqy* = 4, *scalex* = 0.5, *scaley* = 0.5, *incx* = -1.5, *incy* = 0.5, *power* = 2

Compare auger, waves2b, wavesn.

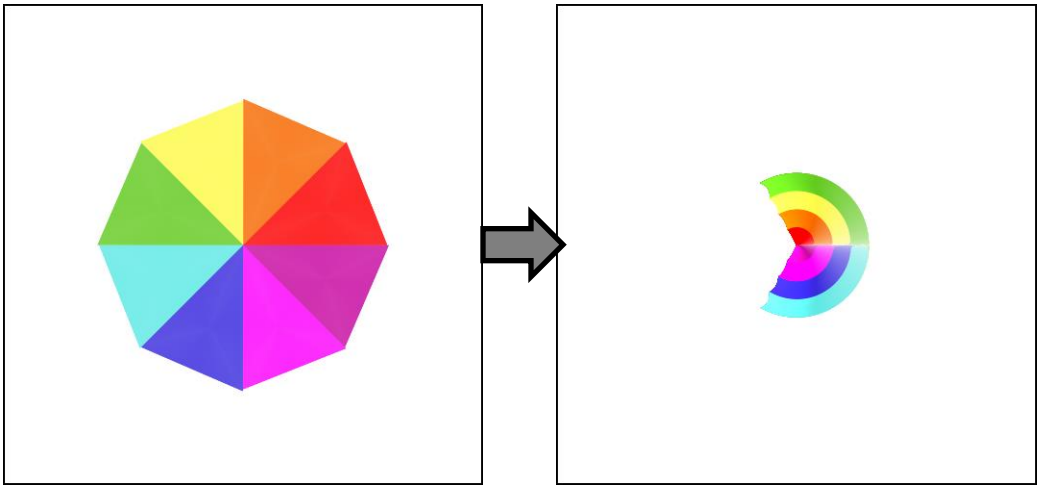


wdisc (2D)

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

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

Compare disc, idisc.



wedge (2D)

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

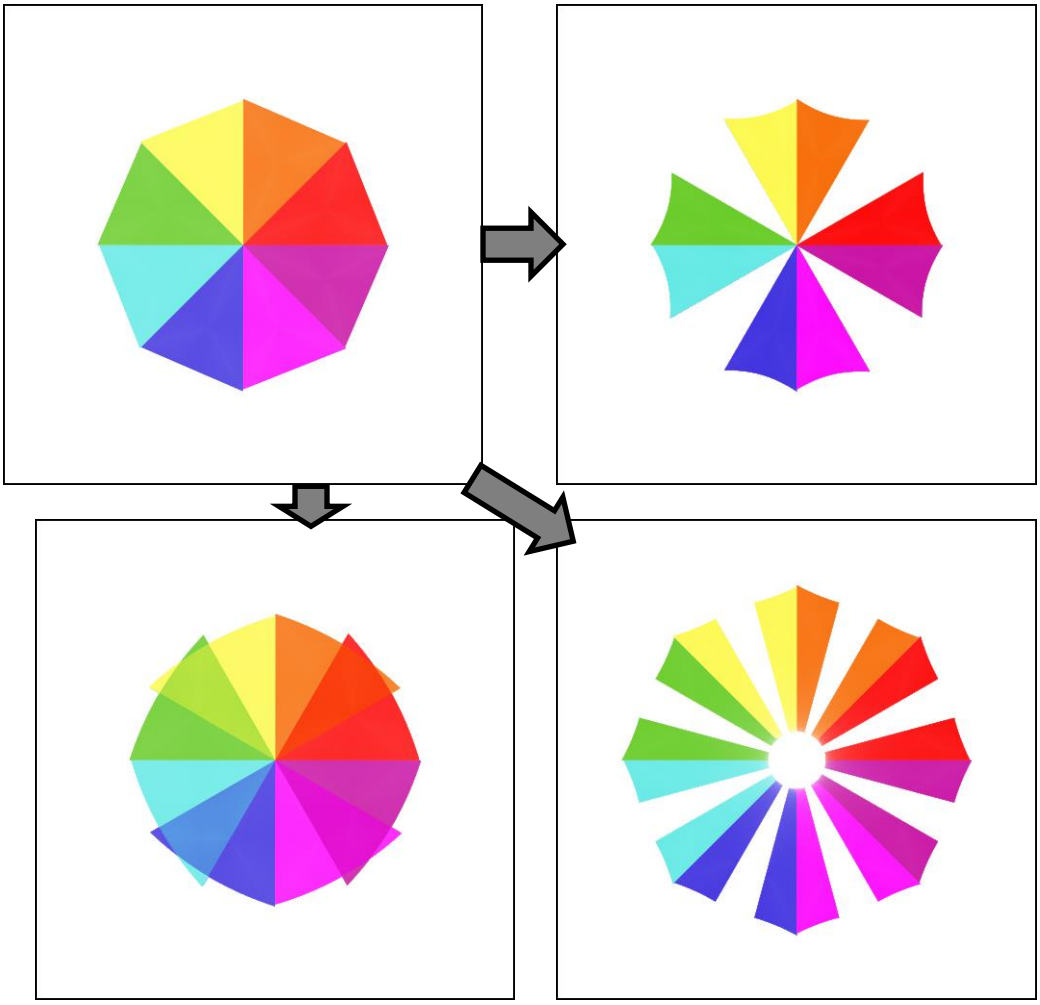
wedge.dll

Divides the plane into *count* wedges and squeezes each by *angle* radians. If *angle* is positive, gaps will be left; if negative, the wedges will be expanded and overlap (bottom left). Then each wedge is moved outward a distance specified by *hole* to leave a hole in the middle (bottom right). Negative values are allowed, which will make the wedges overlap in the center.

If *swirl* is non-zero, an effect similar to swirl will be added (see the swirl variation).

Top right: *angle* = 0.524 (30°), *hole* = 0, *count* = 4, *swirl* = 0
Bottom right: *angle* = 0.262 (15°), *hole* = 0.4, *count* = 8, *swirl* = 0
Bottom left: *angle* = - 0.524 (-30°), *hole* = 0, *count* = 4, *swirl* = 0

Compare shredrad. There is also a plugin **wedge_fl** that allows non-integer values for *count*.



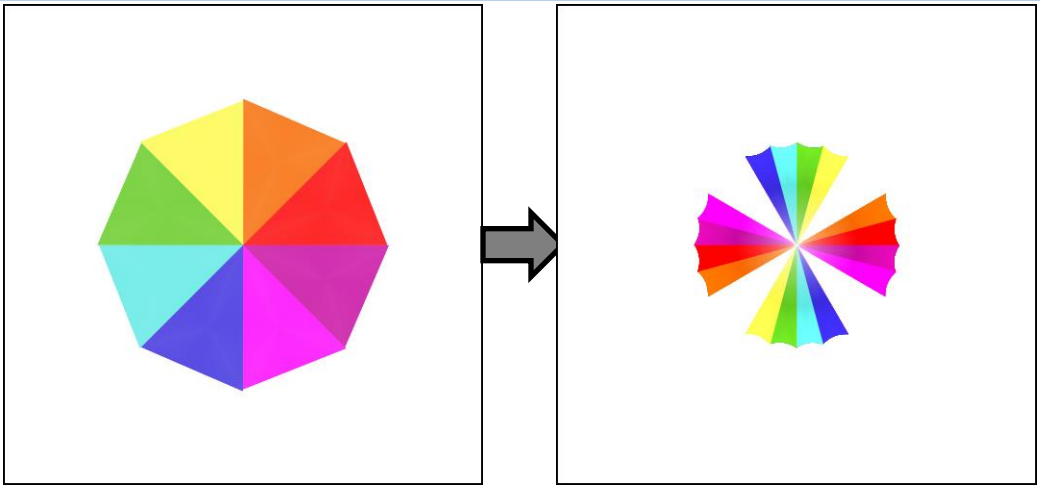
wedge_julia (2D)

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

wedge_julia.dll

A combination of julian and wedge. Example shows *angle* = 0.524 (30°), *count* = 4, *power* = 2, *dist* = 1.

There is also a plugin w**edge_juliaFL** that allows non-integer values for *count*.



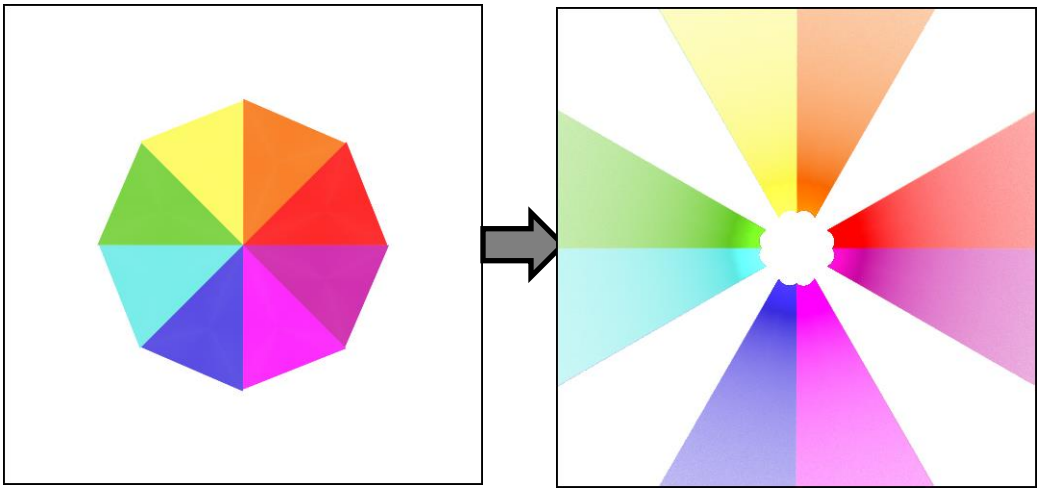
wedge_sph (2D)

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

wedge_sph.dll

A combination of spherical and wedge. Example shows *angle* = 0.524 (30°), *hole* = 0, *count* = 4, *swirl* = 0.

There is also a plugin **wedge_sphFL** that allows non-integer values for *count*.



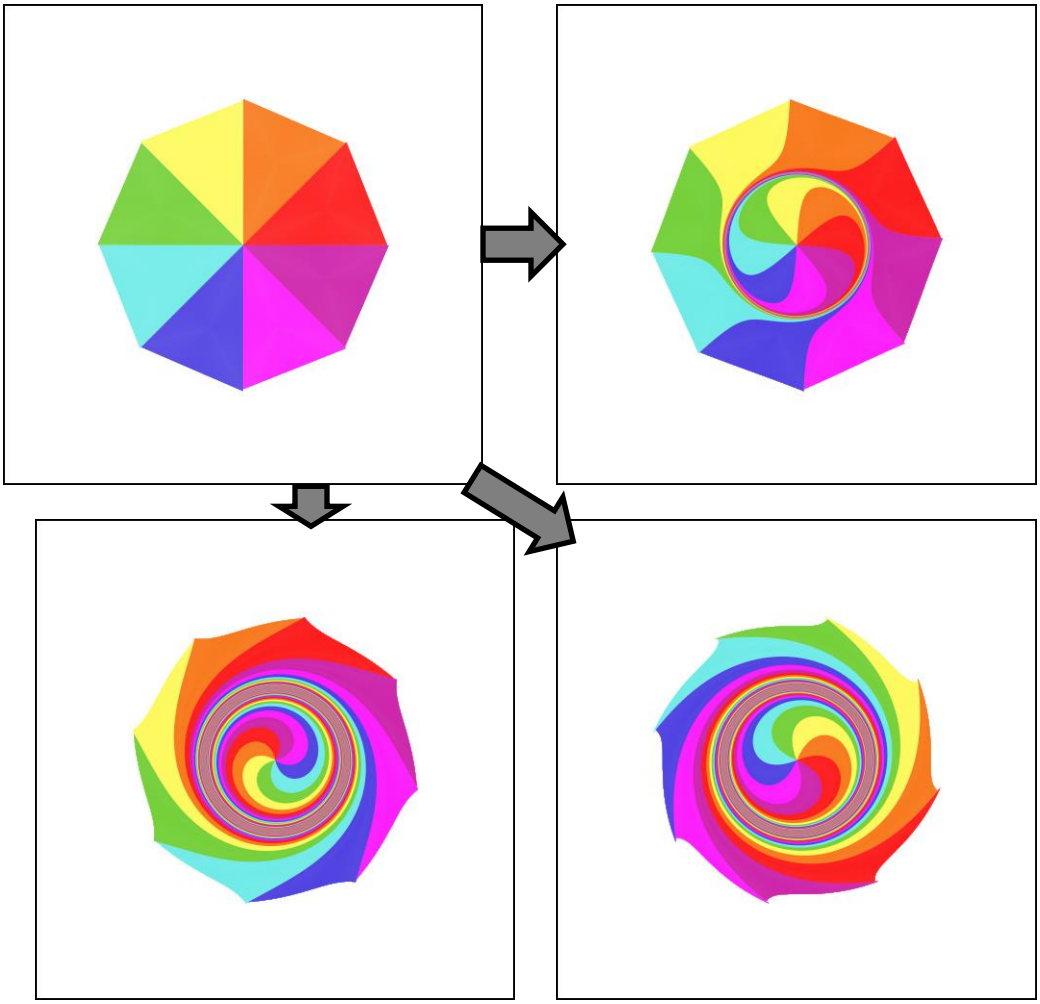
whorl (2D)

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

whorl.dll

Divides the plane into two parts with a circle the size of the variation value, and rotates the inside and outside independently based on the variables *inside* and *outside*. The rotation is weakest at the center and outside, and gets stronger near the boundary. The top right example has small values for both variables to demonstrate this effect.

Top right: *inside* = 0.1, *outside* = 0.1
Bottom right: *inside* = 0.4, *outside* = -1
Bottom left: *inside* = -1, *outside* = 0.6



x (2D)

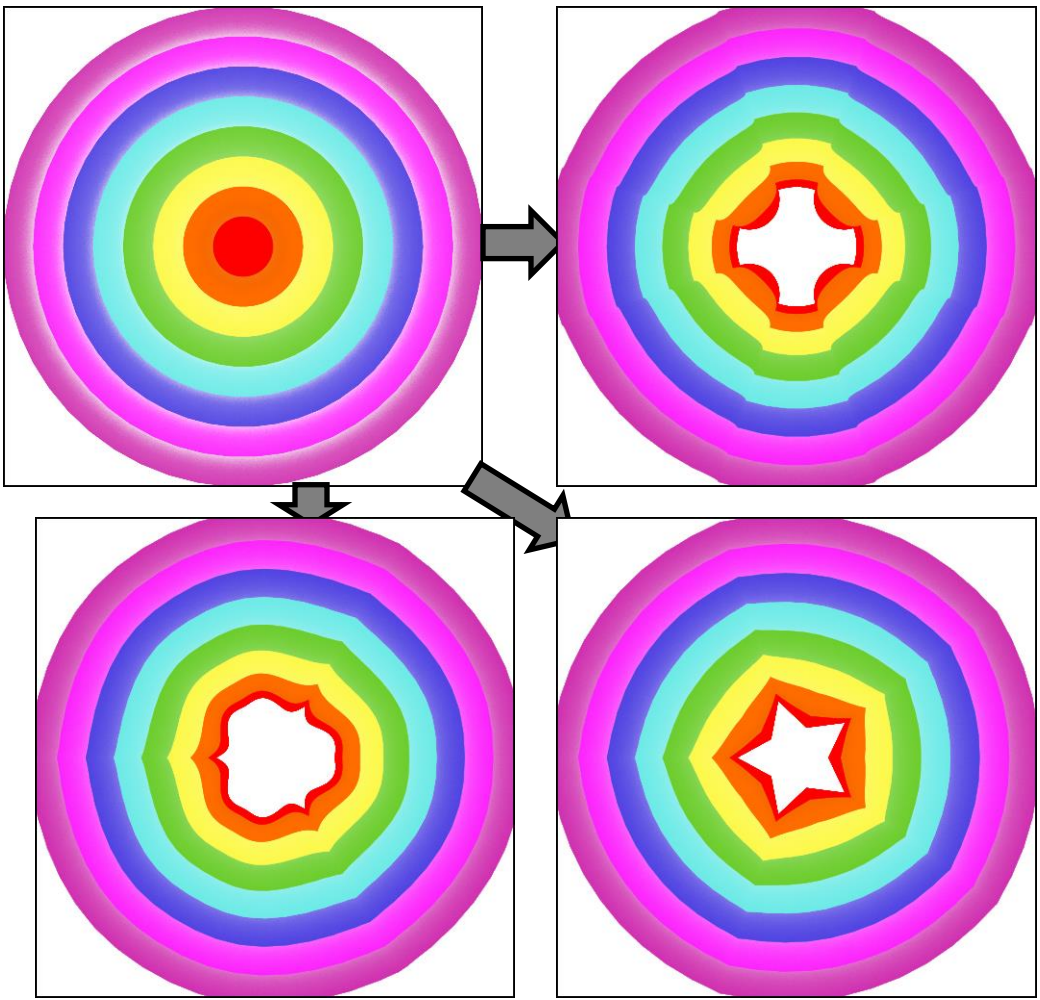
2.09	7X15B	7X16	jwf	ch
dll	dll	dll	no	dll

x.dll

Contorts the plane to various shapes (hypergon, star, and supershape shown; lituus and combos not shown). The effect is strongest at the center and diminishes outwards (contrast with variation z, which does not diminish).

Top right: *hypergon* = 0.5, *hypergon_n* = 4, *hypergon_r* = 0.5
Bottom right: *star* = 0.5, *star_n* = 5, *star_slope* = 0.5
Bottom left: *super* = 0.5, *super_m* = 6, *super_n1* = 0.75, *super_n2* = 1, *super_n3* = 3

See also w, y, and z.



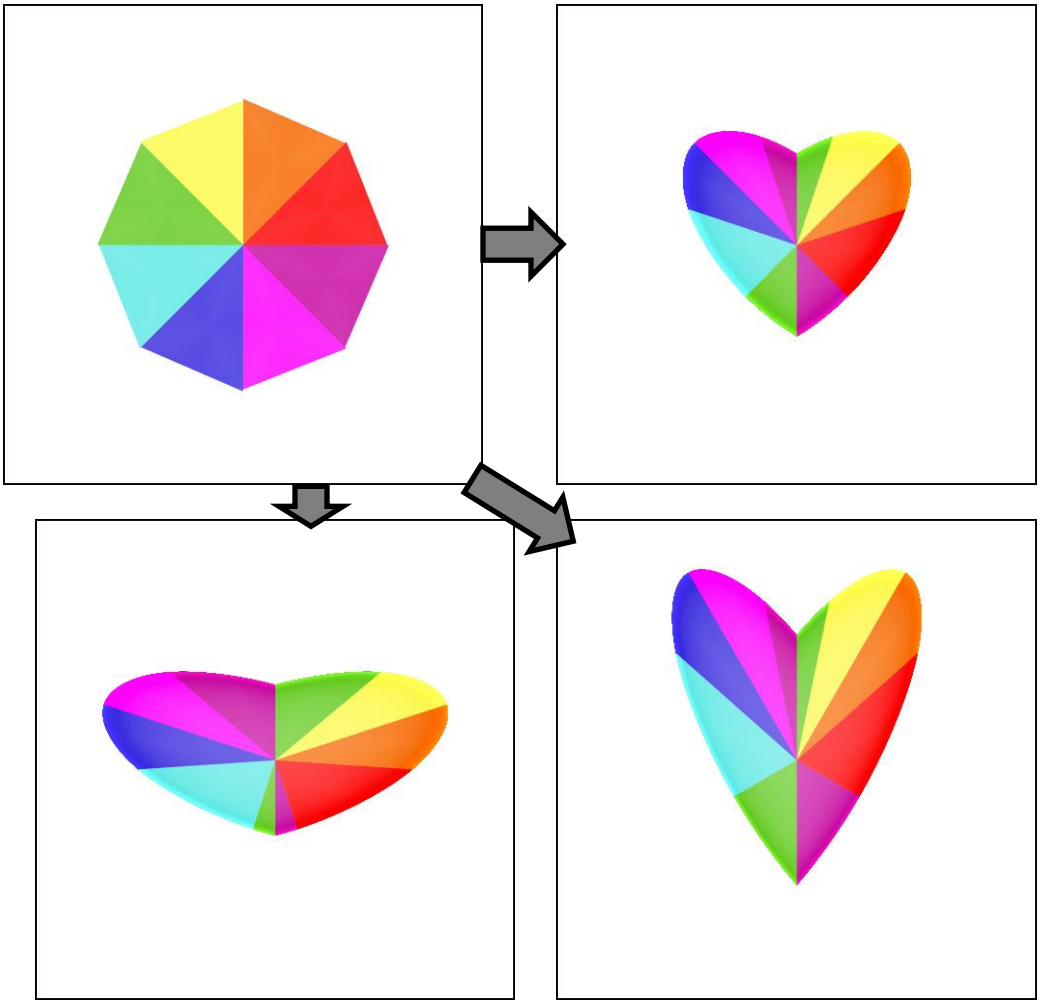
xheart (2D)

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

xheart.dll

Maps the plane to an ellipse with the variation value as the width and (*ratio*+3)/2 as the height, and rotates it according to *angle* (-2 for a vertical ellipse (no rotation) to 2 for a horizontal one). It then flips the left half vertically, resulting in a heart shape if the ellipse is tilted.

Top right: *angle* = 0 (45° rotation), *ratio* = 1 (height twice the width)
Bottom right: *angle* = -0.667 (30° rotation), *ratio* = 3 (height three times width)
Bottom left: *angle* = 1.2 (72°), *ratio* = 2 (height 2.5 times width)



xtrb (2D)

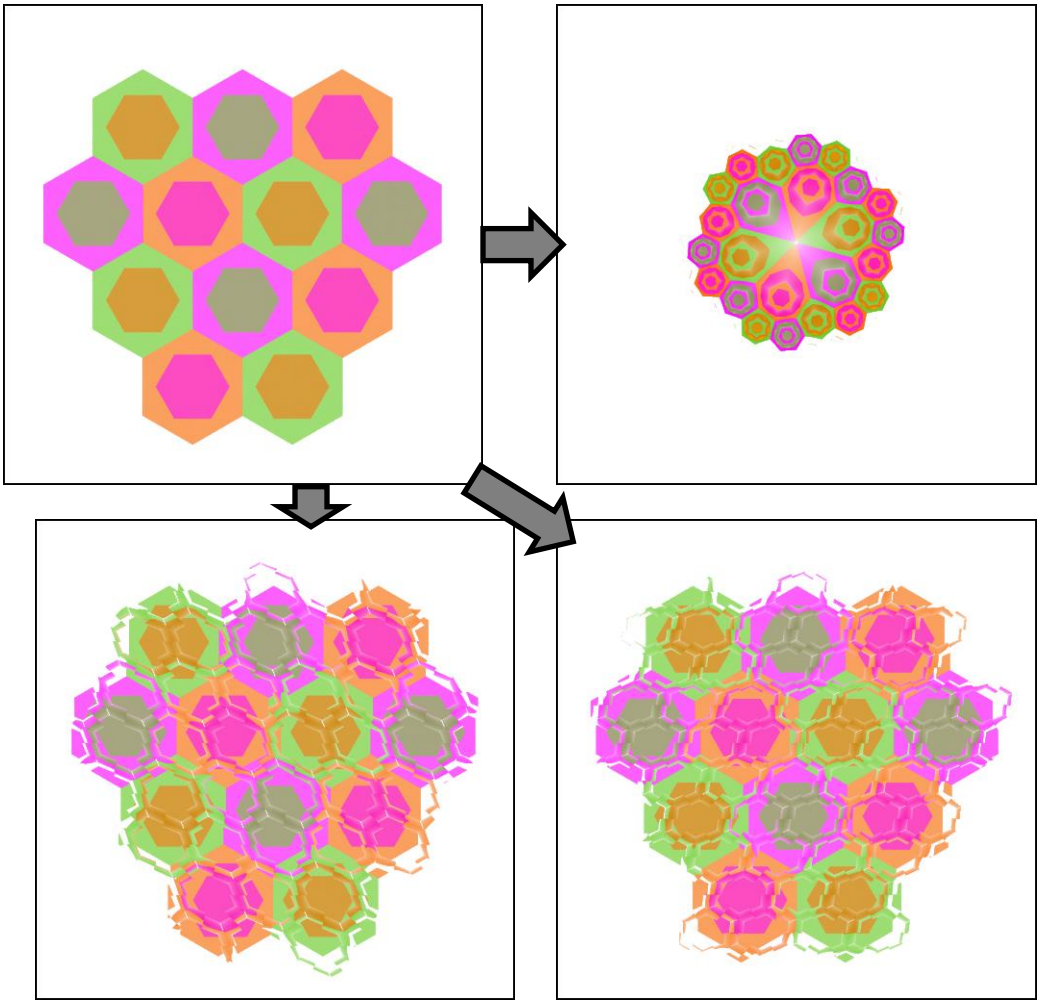
2.09	7X15B	7X16	jwf	ch
dll	dll	dll	yes	dll

xtrb.dll

Extends tri_boarders2, adding a julian function (variables *power* and *dist*; see top right sample), and variables *a* and *b* to control the angles of the hexagon shapes (set both to 1 for normal hexagons).

Top right: *power* = 1, *radius* = 0.5, *width* = 0.5, *dist* = 1, *a* = 1, *b* = 1
Bottom right: *power* = 1, *radius* = 0.25, *width* = 0.75, *dist* = 1, *a* = 1.25, *b* = 1
Bottom left: *power* = 1, *radius* = 0.25, *width* = 0.75, *dist* = 1, *a* = 1, *b* = 0.75

See tri_boarders2



y (2D)

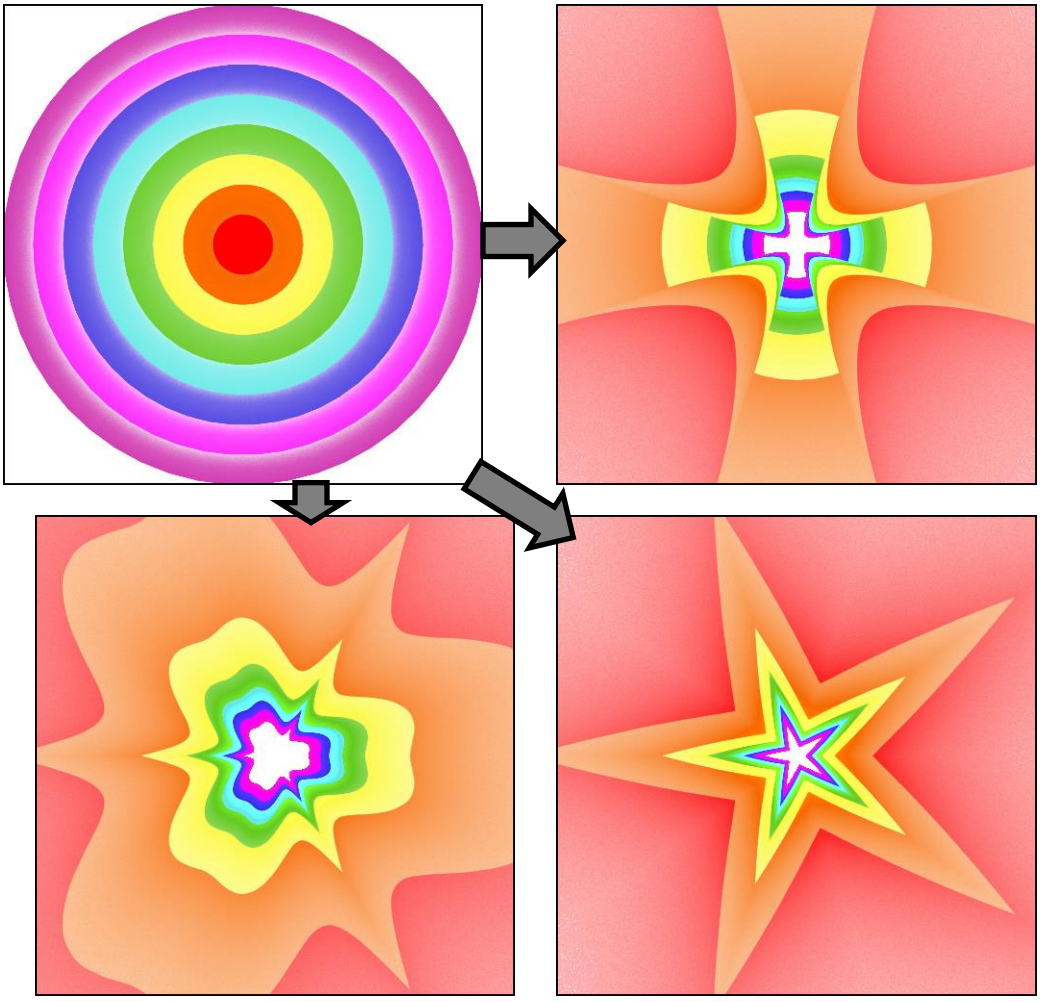
2.09	7X15B	7X16	jwf	ch
dll	dll	dll	no	dll

y.dll

Contorts the plane to various shapes (hypergon, star, and supershape shown; lituus and combos not shown), then turns it inside-out as spherical does.

Top right: *hypergon* = 0.75, *hypergon_n* = 4, *hypergon_r* = 0.5
Bottom right: *star* = 0.75, *star_n* = 5, *star_slope* = 0.5
Bottom left: *super* = 0.75, *super_m* = 6, *super_n1* = 0.75, *super_n2* = 1, *super_n3* = 3

See also w, x, and z.



z (2D)

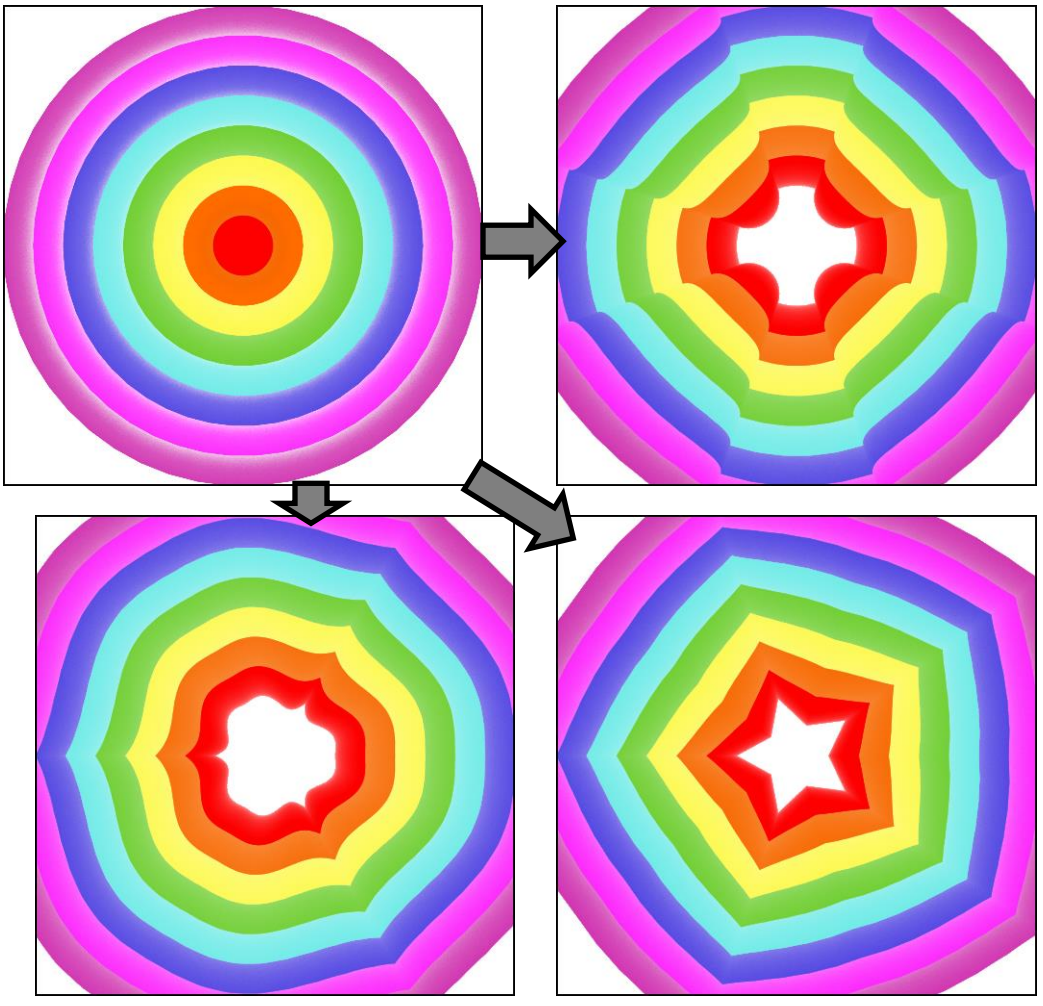
2.09	7X15B	7X16	jwf	ch
dll	dll	dll	no	dll

z.dll

Contorts the plane to various shapes (hypergon, star, and supershape shown; lituus and combos not shown). The effect is equal from the center outwards (contrast with variation x, which diminishes further from the center).

Top right: *hypergon* = 0.5, *hypergon_n* = 4, *hypergon_r* = 0.5
Bottom right: *star* = 0.5, *star_n* = 5, *star_slope* = 0.5
Bottom left: *super* = 0.5, *super_m* = 6, *super_n1* = 0.75, *super_n2* = 1, *super_n3* = 3

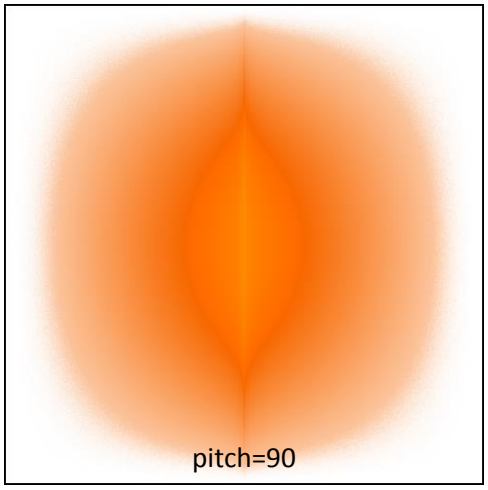
See also w, x, and y.



zblur (3D blur)

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

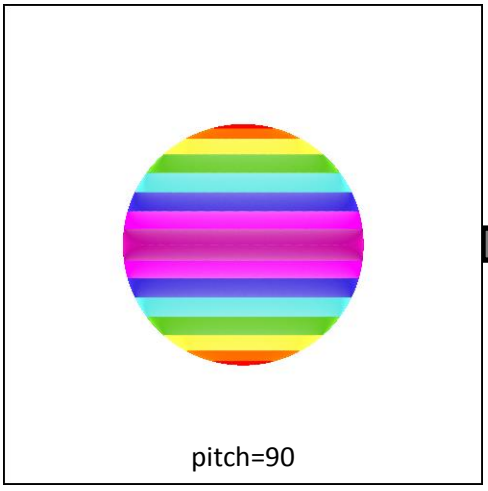
A gaussian blur for the z axis only; no effect on x or y. The sample includes gaussian_blur to provide x-y input and prevent it from being a simple vertical line. The view is from the side; the z axis is vertical. (The sample was rendered by JWildfire; the various Apophysis versions render it differently, probably due to random number generator variations.)

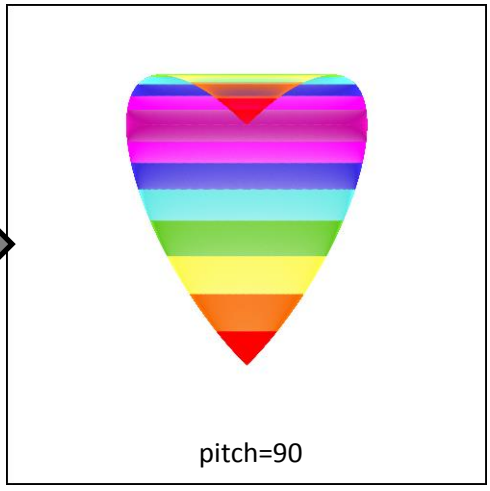


zcone (3D, transforms z)

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

Adds the x-y distance of each point to z, thus transforming a plane into a cone. Shown here is the effect on a sphere, viewed from the side.

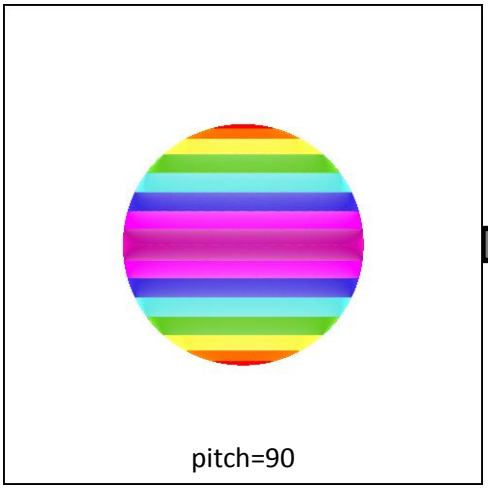


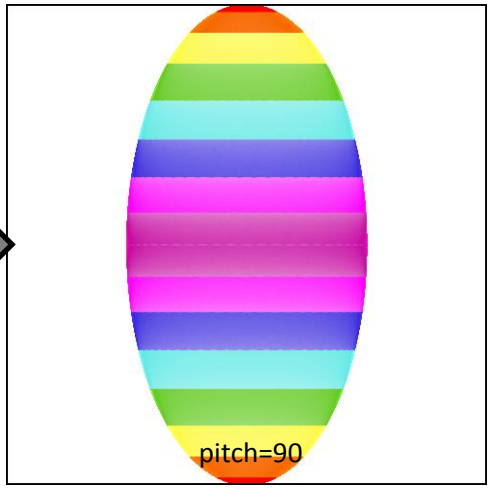


zscale, pre_zscale (3D, transforms z)

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

Multiplies the z value of each point by the variation value, so using zscale with value 1 will simply pass z if the other variations don't already. In the example, zscale is 1, but linear3D with value 1 is also used, which adds an additional zscale of 1, making a total zscale of 2, so doubling the height (or depth) of each point.





ztranslate, pre_ztranslate (3D, transforms z)

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

Adds the variation value to the z value of each point. In the example, the value of ztranslate is 1, so the shape is moved up by 1 unit.