Full Circle

INKSCAPE SERIES SPECIAL EDITION

THE INDEPENDENT MAGAZINE FOR THE UBUNTU LINUX COMMUNITY

INKSCAPE SERIES SPECIAL EDITION Vol 8

INKSCAPE

Volume Eight — Parts 50 - 57

First, an apology. Last month I suggested you could get color from a black shadow by using the Fixed Offset column of the Color Matrix filter, and demonstrated using the Source Alpha input. Unfortunately a change was introduced into Inkscape 0.91 which prevents fixed offsets of the color components working on a Source Alpha input (https://bugs.launchpad.net/inkscape/+bug/897236). It does work in version 0.48, as well as in Firefox and other SVG renderers. Apologies to anyone who wasted their time trying to follow my instructions on 0.91, and thanks to Moini in the Inkscape Forum for bringing this issue to my attention. Now, on with the show…

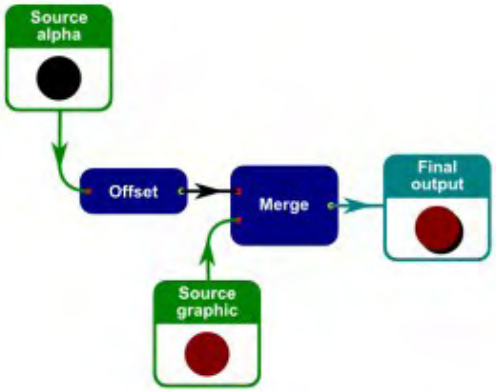
Another type of drop shadow effect that you'll see from time to time is the "stacked shadow". This is created by stacking several hard-edged copies of your original object on top of each other, with each of them having a different fill color.

The easy way to create this effect is just to duplicate your original objects, move them a little, change their fill color, and re-stack them into the correct order. With three objects, the middle one having a white fill and no stroke, it was a matter of moments to produce this:

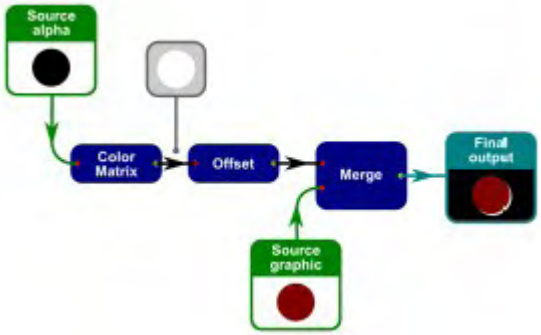
Not a bad start, but what happens when you need to change the text? You would have to alter it for all three objects which triples your chances of making a mistake. Better to use unset fills with clones (see part 30), which can get you to the same result but with only the parent object to edit in order for your changes to propagate through the whole stack.

Even with clones, however, you're still working with three objects. Grouping them lets you move or transform them all as one, but you then have the extra burden of having to enter the group and track down the original object in order to change the text. As you might imagine, filters offer a solution to all these problems.

With the few filter primitives you've learnt in the previous two instalments, you already know enough to create a stacked shadow effect using the fill color at the top, a white copy of the source image below that, and a black copy right at the bottom. I t's really just the same as a simple hard-edged drop shadow (from part 48) with a re-colored drop shadow (part 49) sandwiched in the middle. Let's look at it in graph form first, starting with a basic hard-edged black drop shadow:



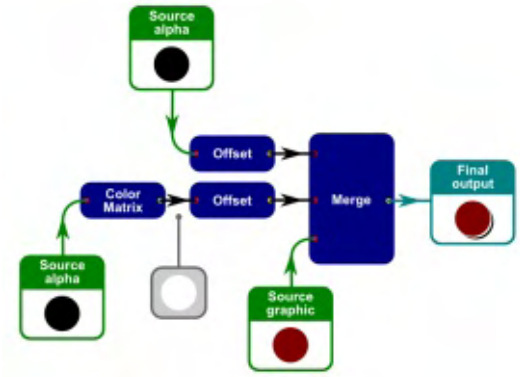
Pretty simple, right? Now let's look at our hard-edged white drop shadow. You'll notice it's essentially the same graph, but with the addition of a Color Matrix primitive to convert the black shadow to a white one (I've used a black background for the final output box, so that the white shadow is visible) :



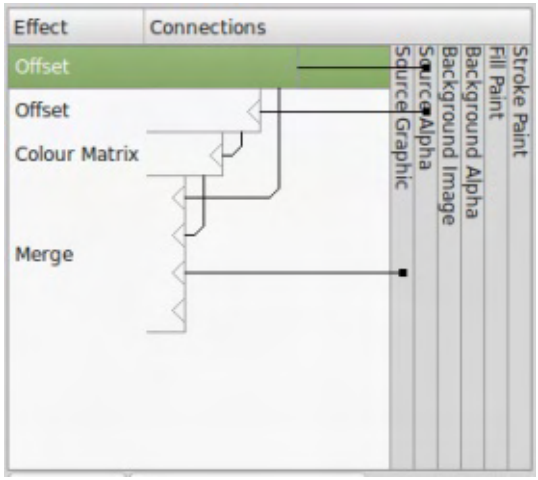
In order to get a white output from the color matrix, each R, G and B row must evaluate to at least 1.0 (which is mapped to 255 in RGB).Our input values are all zero, so no amount of multiplication will get the result we want. Instead we have to put a value of 1.0, or greater, into the fixed offset column for the first three rows:

1.00	0.00	0.00	0.00	1.00
0.00	1.00	0.00	0.00	1.00
0.00	0.00	1.00	0.00	1.00
0.00	0.00	0.00	1.00	0.00

Now that we know how to create the constituent parts of our filter, we just need to combine them into one. In this case, it's a simple matter of merging them in the right order – black shadow first, then white, then the source graphic. The final graph looks like this:



As you can see, our final filter requires four filter primitives – two Offsets, one Color Matrix and a Merge. It also has two connections to the Source Alpha input, and one to the Source Graphic input. Let's take a look at the final filter design in Inkscape:



If you follow each line in the image you'll see that it's the same connected set of objects as in the graph view. Unfortunately, Inkscape's UI manages to make it seem more complex, largely due to the need for lines to cross in order for each "branch" of the filter to come together at the Merge primitive. Now imagine the same filter design, but with even more shadows being stacked: despite each shadow being a separate linear graph feeding into a common Merge, Inkscape's UI quickly becomes filled with a confusing spaghetti of crossing lines. Whenever you're faced with such a complex mess, try sketching out the filter primitives and their connections in graph form to see if it becomes more understandable.

There's a bit of a problem with our stacked shadows filter: it looks distinctly different when placed on a white background compared with a colored background. In the former case, the white shadow vanishes into the background, giving the appearance of a disconnected black shadow, but as soon as you put it on any other background, the white layer stands out.

In some cases you may want the white layer to be visible, but for others you would want that part of the output to be transparent. If you were building the stacked shadows from normal SVG objects, you could use a clipping path to achieve this effect (see part 13), but clipping paths aren't available as filter primitives. Instead there is a primitive called "Composite" which allows you to combine two inputs in myriad ways, including a couple that have a similar effect to a clipping path.

The Composite primitive uses the alpha values of the pixels in the input images to determine what the output pixel should be, using the methods described by Thomas Porter and Tom Duff back in the 1980s, collectively referred to as the Porter-Duff blending modes. These blending modes are selected from the Operator pop-up in the filter editor:

Default – This omits the operator from the filter primitive in the underlying XML file. Per the SVG Filter Effects spec, this causes Inkscape to behave as though a value of "over" had been supplied. For the sake of clarity, I recommend never using this option, and always explicitly

selecting the “Over” option, if that's what you want.

Over – The two images are laid on top of each other, with the top input appearing above the lower input. This is exactly the same as using the Merge primitive with two inputs, except that the order of the inputs is reversed.

In – Only those parts of the top image that are “inside” the lower image will appear in the output. This has a similar effect to a clipping path.

Out – Only those parts of the top image that are “outside” the lower image will appear in the output. This has a similar effect to an “inverse” clipping path.

Atop – The output consists of the lower input image, plus all the parts of the upper input image that are inside the lower image.

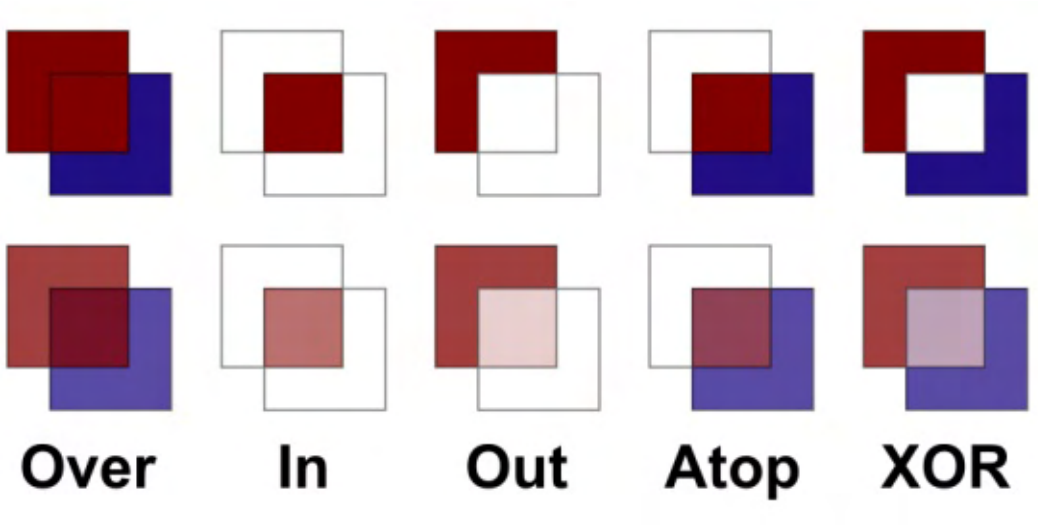
XOR – Performs an eXclusive OR operation between the RGB values of each of the pixels in the two input images. The effect is for the output image to include any non-overlapping parts of the input images.

Arithmetic – This is not one of the Porter-Duff blending modes, but rather is an additional mode that is present in the SVG spec. It will be described in a little more detail later.

Note that the filter UI provides four sliders, but even though these are only used for the Arithmetic operator, they nevertheless remain visible, though disabled, when any of the other operators is used.

The descriptions above are broadly correct, but some subtleties slip in when the input images contain alpha values other than 0 and 255. If you want to use this primitive for clipping, it's therefore advisable to ensure that your input images don't contain intermediate values. The best way to do this is with the Component Transfer primitive, which gained a UI in Inkscape 0.91 and will be described in a future article. Sticking with the filters I've already covered, you can also use the Color Matrix primitive to stretch and offset the range of possible values to achieve a similar result. For example, this matrix will clamp alpha values such that those below 128 are converted to 0, and those above or equal are converted to 255.

```
1.00  0.00  0.00  0.00     0.00
0.00  1.00  0.00  0.00     0.00
0.00  0.00  1.00  0.00     0.00
0.00  0.00  0.00  512.00  -256.00
```

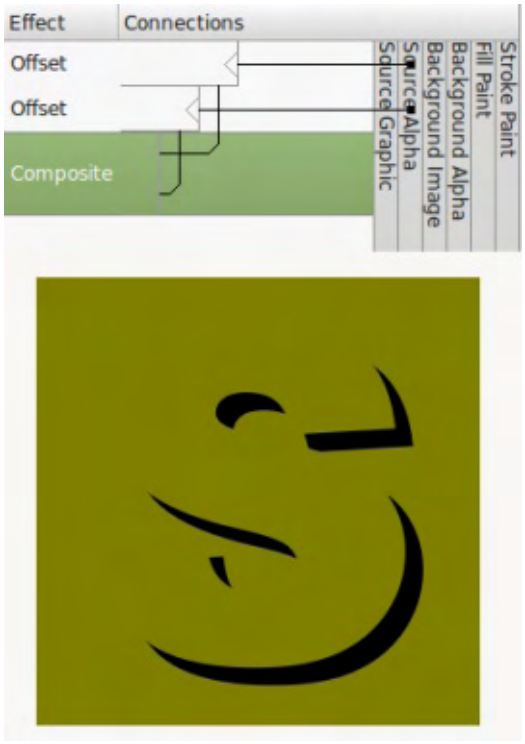


As with so many things in SVG, that's a lot of words to describe something that's better shown as an image. Here are the 5 Porter-Duff blending modes when applied to a couple of squares, first with no transparency, then with the opacity reduced to 75%. Note that the black outlines have been added afterwards to clarify which parts of the images remain – they're not present in the pure filtered output.

Let's get back to our stacked shadow and take a look at how this filter can help to cut away the white layer. Consider just a small part of the output – a single letter. I've removed the Source Graphic so we're just seeing the two offset shadows:

We need to keep the black part that's visible, but remove all the white content, leaving it transparent. In other words, we want to keep the part of the black layer that is outside the white layer. Clearly this is a job for the Composite primitive's “Out” blending mode. Because the Composite filter cares about only the opacity of the input sources, not their color, we can omit the

Color Matrix primitive, giving us this filter chain and result:



All we need to do now is to add a final block to the chain to merge this output with our original source graphic once more, giving us the final result we were looking for – a stacked shadow with a transparent intermediate layer that works on any background (see top middle).

There's one last thing to describe before concluding this month's article: that "Arithmetic" mode of the Composite filter, and

its four sliders (K1 to K4). With this mode, each channel (R, G, B, A) of each pixel of the output image is calculated from the corresponding pixel channel of the input images (i1 and i2), weighted by the K1 to K4 values using the following formula:

```
result = (K1×i1×i2) + (K2×i1)
+ (K3×i2) + K4
```

Breaking this down, you can see that K4 isn't multiplied by anything, so it just represents a fixed offset. K2 and K3 are multiplied by i1 and i2 respectively, so they adjust the amount of each input that goes into the output. K1 is multiplied by both i1 and i2, so acts to stretch the range of the output values.

This mode can be used to combine the output from two other filter primitives, allowing you to control the proportions of each input. The SVG spec suggests it might be useful for overlaying the output from some lighting effect primitives (not yet covered in this series) with texture data from another primitive or image source, but it can be useful whenever you want to mix two images together with some control over the strength of each one.

Inkscape Forum

The Inkscape Board is forming a committee to organise the creation of an official Inkscape forum. The main existing community forum (inkscapeforum.com) has become a target for spammers, and the owner of the domain has not been responsive to any emails or messages. The chair of the committee will be Brynn, a long standing contributor to the old site, who maintains a separate forum at www.inkscapecommunity.com. The major contributors to the forum are moving to her site, at least as an interim measure. Until a final decision is made about a new forum, it is strongly recommended that support posts or requests are made at Brynn's site, rather than at the old forum.

Mark uses Inkscape to create three webcomics, 'The Greys', 'Monsters, Inked' and 'Elvie', which can all be found at http://www.peppertop.com/

Inkscape - Part 51

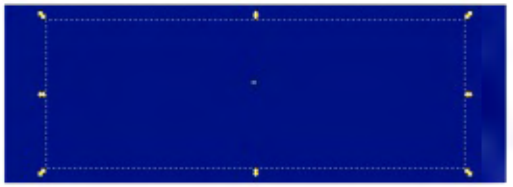
It appears that last month's announcement about forums was premature. Suffice to say that politics and personalities have been at loggerheads in the world of Inkscape support forums, but things have since calmed down.

So I'll stick to pure facts: Both the inkscapeforum.com and inkscapecommunity.com forums continue to operate, each with a different subset of users (and some degree of overlap). Support requests posted to either will generally elicit a response, and there's no need for normal Inkscape users to concern themselves with the behind-the-scenes shenanigans. None of this has any impact on the development of Inkscape itself. If and when there is any more concrete information about an official forum, I'll write about it, but until then I'll be keeping forum politics well out of this column!

Now, where were we...? Oh yes, filters. Let's return to the single-colored drop shadows of Part 49 to show you a simpler way to create the same effect. Previously, I introduced you to the Color Matrix primitive as a means of converting one color to another, but when all you need to do is to introduce a specific fixed color into your filter chain, it's usually easier to use the Flood primitive.

As you might have guessed from the name, the Flood primitive floods an area with color. You may now be thinking along the lines of the bucket tool in Inkscape or other graphics programs, which typically floods an area by working outwards until it hits a differently colored boundary line. But there's no such finesse here; the flood primitive simply fills the whole of the "filter effects region" with a flat color. The filter effects region is the rectangle defined by the Filter General Settings tab (see part 48), and is typically larger than the bounding box of your selected objects.

Starting again with some simple text, create a filter and add the Flood primitive. With the primitive selected, use the controls at the bottom of the Filter Effects dialog to choose a color and opacity, and you should get a result something like this (note that it doesn't matter what the input of the Flood primitive is connected to, as it has no effect on the output):

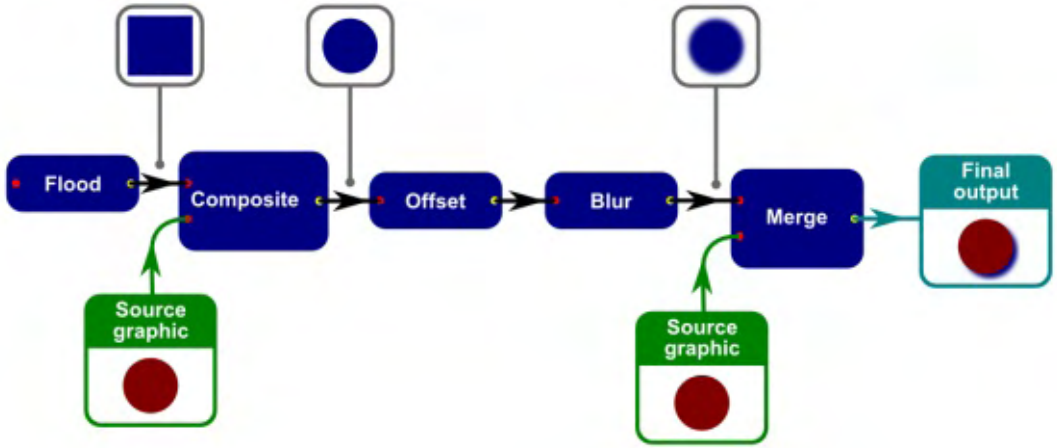


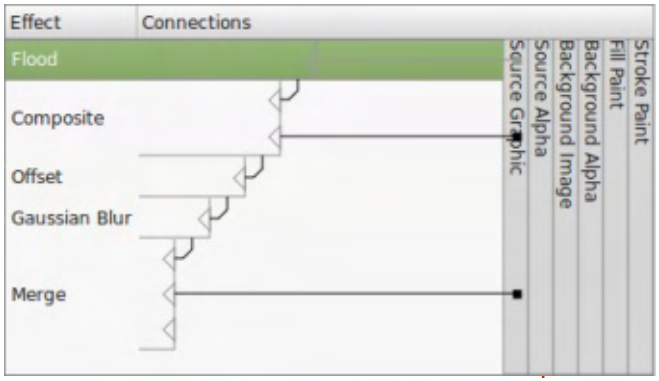
Not terribly inspiring, is it? So the question now is how to turn this big blue rectangle into a softly shaped drop shadow. If you followed last month's tutorial, you'll know that the Composite filter (used in "In" mode) can be used to crop the blue rectangle into the shape of our text.

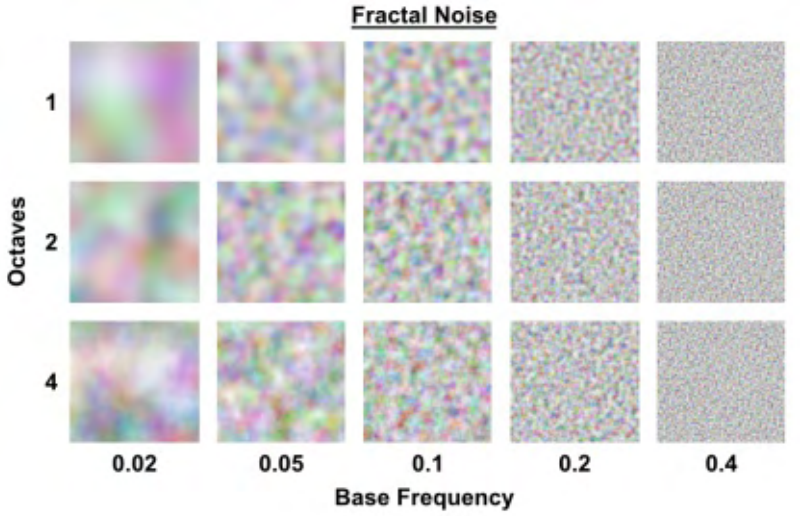
It's not very common that you'll want the output from your filter to be strictly rectangular, rather than following the shapes and curves of your selected objects. So whenever you see a Flood primitive in a filter chain, there's a good chance that there will be a Composite filter following along shortly afterwards to trim it to shape.

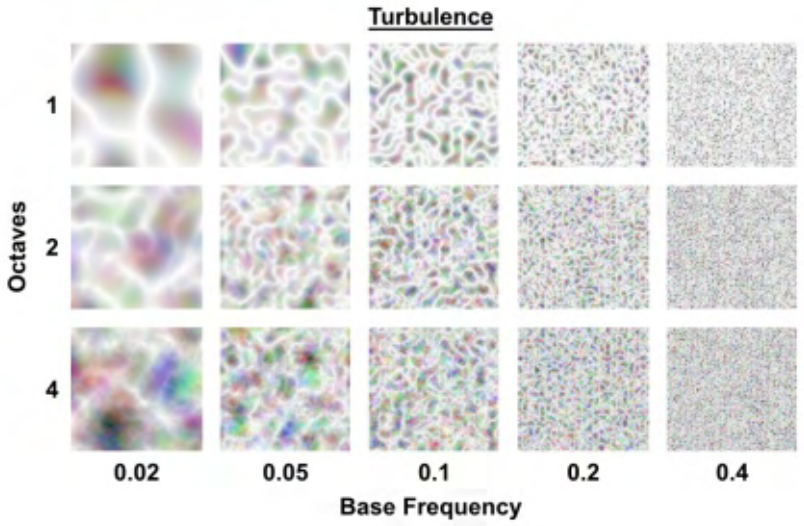
Now that we've got a colored version of the text, it's a straightforward matter to offset and blur it, before merging it with the Source Graphic – you should be adept at those steps by now, so I'll spare you a detailed description and instead present the result, a graph version of the chain, and a screenshot from Inkscape.

Whilst Flood provides you with a rectangle of a single color, the Turbulence primitive gives you a rectangle filled with a chaotic mix of colors. It's not strictly random, in the mathematical sense, as the output is well defined and repeatable (meaning that your









filters should look the same in any renderer – although in reality that may not be the case), but, in the colloquial sense, it's this primitive that you should head for if you want to add a degree of randomness or noise to your image. It has two modes: Fractal Noise and Turbulence. The difference between them is that the latter has more "troughs" in the output, where the background shows through, giving the appearance of joined up lines running throughout the output, whilst the former has more of a cloudy appearance.

Whichever mode you choose, the rest of the controls remain the same. The Base Frequency sliders control how "dense" the noise appears – low values give slow, smooth transitions, whereas higher values result in transitions that change more rapidly, making the output more reminiscent of "snow" on an old un-tuned TV set. The horizontal and vertical frequencies are usually the same, but can be changed independently by toggling the Link button to the right. The Octaves slider controls how detailed or complex the noise appears; taking this much beyond about 4 is rarely worthwhile as the increased detail is too small to see, and it imposes an extra load on the processor. Finally, the Seed value can be used to prime the pseudo-random number generator at the core of the filter to give you a slightly different output pattern without changing the other parameters.

The following images show the effect of varying the Base Frequency and the Octaves sliders, for both the Fractal Noise and Turbulence modes.

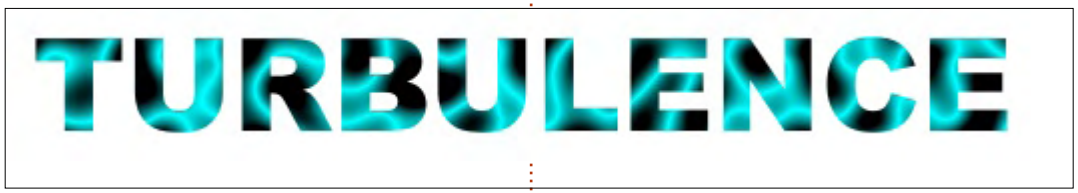
You'll notice that the images are fairly pastel in tone. This is because all four channels (R, G, B, A) are calculated independently – each pixel actually consists of a combination of four pseudo-random numbers. The value of the Alpha channel will override all the others, so even if you happen to have a strong color from the RGB components, a low Alpha can knock it back to a translucent shadow of its former self.

5.00	0.00	0.00	0.00	0.00
0.00	5.00	0.00	0.00	0.00
0.00	0.00	5.00	0.00	0.00
0.00	0.00	0.00	0.00	1.00

You can use a Color Matrix to extract a single channel, or to stretch the output to make it more vibrant. In this example, I've done the latter, as well as wiping out the Alpha channel entirely and replacing it with a fixed value of 1 (fully opaque). The cyan color of the original text doesn't show through at all in this case (any cyan in the result comes purely from the Turbulence filter), but I have used a black background to make the colors stand out even more.

To extract a single channel from the output, zero everything in your Color Matrix, and then populate just one of the first four columns, depending on what you want to get out. For example, setting a value of 1.00 in every field in the third column would take the 0-255 values in your Blue channel and map them to RGBA in the output. A Blue value of 63 on the input would produce (63, 63, 63, 63) – a translucent gray color – as an output. You might want to take the Alpha channel out of the equation by setting a value of 1.00 in the bottom right corner (the Fixed Value column for the Alpha output). In this example, I've used the Green channel to set only the Alpha of the output, and stretched the values a bit by using 3.00 rather than 1.00. This gives an image that runs from opaque black to transparent black, so, by Compositing it (to clip it to shape) and then Merging it with the cyan Source Graphic, it's easy to create an “electric” or “plasma” effect.

0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00
0.00	3.00	0.00	0.00	0.00



You could also bring a Flood primitive into a chain like that to ensure that your result has the right color in the filter, regardless of the color of the object it's applied to.

Don't forget that the Base Frequency controls can be un-linked. By keeping the values close to each other, you can introduce a slight stretching or bias into the patterns, whilst separating them by some distance can result in almost horizontal or vertical lines appearing. Here's the previous filter, but with the horizontal Base Frequency set quite high, and the vertical Base Frequency at zero – the result isn't what you might usually think of as “turbulent”, but can be a useful addition to your filter arsenal nonetheless.

By now you should be starting

to appreciate the power and flexibility of filters. By combining a few primitives in various ways, you can quickly create complex results. Throw in a little pseudo-random chaos and you're well on your way to everything from clouds to marble, whilst Flood primitives can ensure that the important colors in your filter are independent of the objects they're applied to.

One common theme between Flood and Turbulence is that they fill the filter region entirely, usually requiring a Composite operation to trim them to shape. Next month, we'll look at the last of these “fill” primitives – uncovering limitations in Inkscape along the way – then progress on to other ways to change their shape.

Mark uses Inkscape to create three webcomics, 'The Greys', 'Monsters, Inked' and 'Elvie', which can all be found at http://www.peppertop.com/

Inkscape - Part 52

Written by Mark Crutch

Last time, we looked at Flood Fill and Turbulence – a pair of primitives that can be used to fill the filter area with, respectively, a flat color or a pseudo-random cloud of colors. But there's a universe of other fills you might like to use, from stripes to polka dots, flowers to butterflies. To cater for such infinite possibilities, the SVG standard provides a way to pull another image into your filter chain, using the Image primitive. This allows you to not only use bitmap images, but can even reference other parts of your SVG file to let you pull your own creations into the filter chain. There's just one little problem: the Inkscape implementation is well and truly broken.

Let's start with the bit that does work, at least to some degree: importing an external bitmap image into your filter chain. As usual, we'll begin with a bit of text as the object to which we'll apply the filter. You can, of course, use any type of object, but I find that text gives me a quick and easy way to see how a filter will look when applied to a complex shape, rather than using just a simple rectangle or circle.

Create a filter on the test object using one of the methods described in Part 48, and, if necessary, remove any existing filter primitives. Now add a single "Image" primitive to the filter chain, and take a look at its minimal controls at the bottom of the dialog. The "Source of Image" field will be used to hold the path and filename of an external image file, or the XML id of another element in your image. For now, you should choose an external bitmap image by clicking on the "Image File" button and picking one from your hard drive. We'll use our tried and tested Mona Lisa image, giving us the following output (unfiltered text on the left, filtered on the right) when the filter is created in Inkscape 0.48 (bottom left).

Now you don't need to be an expert in renaissance art to notice that the image has been distorted somewhat. Now the same filter created in 0.91 (bottom right).

Well, we've lost the distorted aspect ratio, but it doesn't exactly fill the filter area – although we can do something about that, as we'll see shortly. This change in behaviour could potentially mean that drawings created in 0.48 may not appear the same in 0.91 if they make use of this filter primitive.

In 0.48, you're stuck with the default position and size of the image – i.e. stretched to fill the bounding box of the object. The official Inkscape manual makes it sound as though you can at least set the position and size of the image within the filter by using the XML editor, but, despite many attempts, I haven't been able to achieve this. To be fair, the manual does state that the implementation in Inkscape "doesn't correctly position images" – though that seems to be something of an understatement based on my own tests.

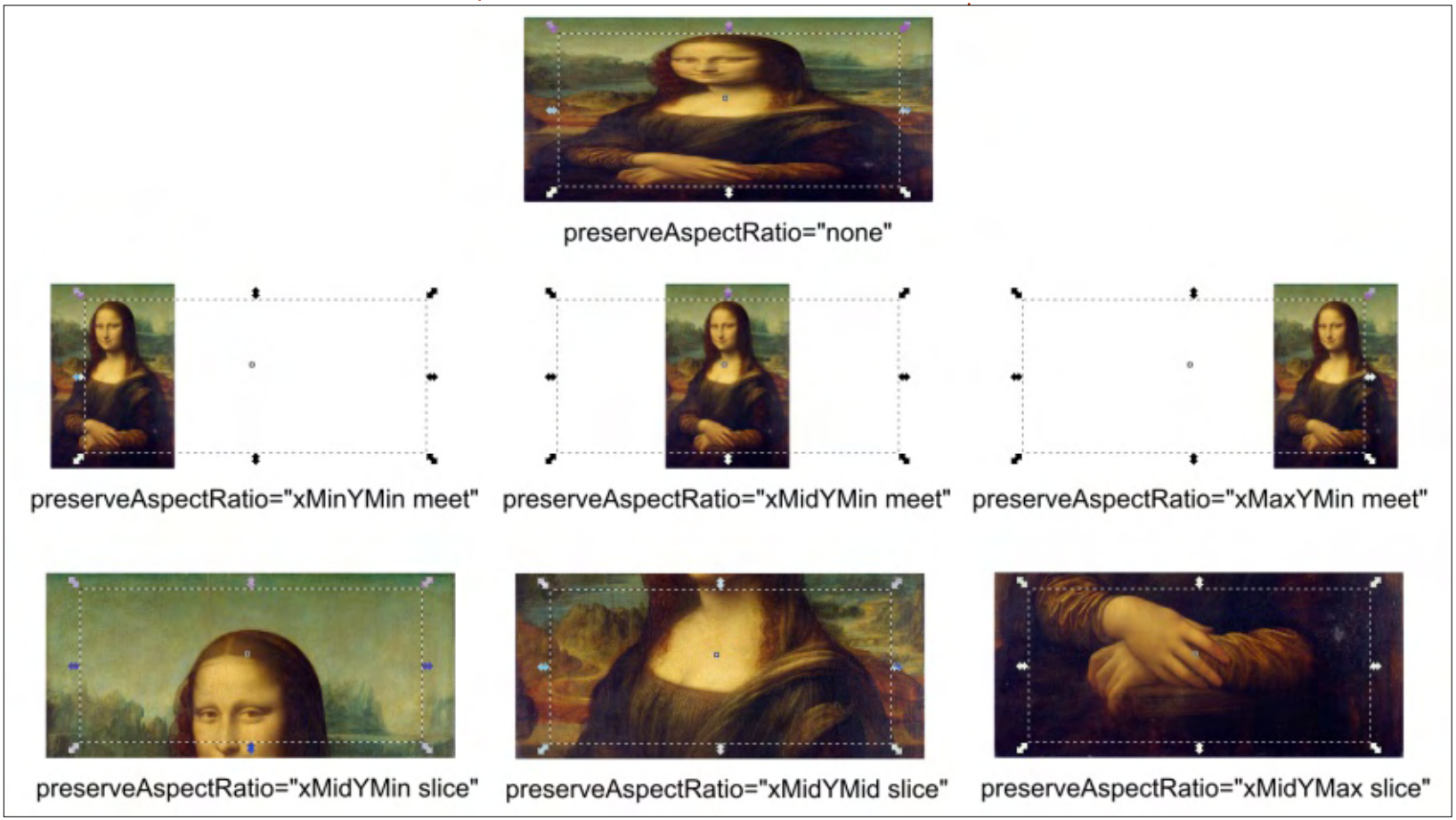
With 0.91, things fare a little better – though you'll still have to make your way to the XML editor to change the parameters, as they're still not reflected in the

GUI. Dig out Part 31 of this series if you need a refresher on how to use the XML editor. The XML element you'll need to modify is the <svg:feImage> which is inside an <svg:filter> in the <svg:defs> section of the file. You can add 'x', 'y', 'width', and 'height' attributes, although, in my tests, none of them took effect until I also added a 'preserveAspectRatio' attribute. In subsequent tests, once such an attribute was already present, I had to change it to a different value and back again for changes in the other attributes to be reflected in Inkscape's canvas. Changing it to an invalid value and back does the trick, so you can just add a single letter to the end of the existing value, click “Set”, remove the letter, then click “Set” again.

So just what is a valid value for preserveAspectRatio? A good starting point is the word “none” - that will cause Inkscape to ignore the aspect ratio of the original image, and stretch it to fill the bounding box of the object, resulting in a similar appearance as with version 0.48. But there are other options – lots of them, all similarly and confusingly named! They all start with a lowercase 'x' and are followed immediately by 'Min', 'Mid' or 'Max' (which, for the x direction, basically means left aligned, centered or right aligned), then followed immediately by an uppercase 'Y' and another 'Min', 'Mid' or 'Max' (top, middle or bottom aligned), followed by a space and an optional keyword of 'meet' (scale the image so that it's all visible) or 'slice' (scale the image to fill the bounding box, whilst preserving the aspect ratio, but hide any parts that extend beyond the bounding box – i.e. just show a slice of the image). Confusing, isn't it? Perhaps some examples (below) would help.

No, I don't know why the SVG working group went for the confusingly similar 'Min' and 'Mid', nor why 'x' is lowercase whilst 'Y' is uppercase, nor why they chose the words 'meet' and 'slice' rather than 'scale' and 'crop'. I do know, however, that my examples are just the tip of the iceberg: there are 19 possible combinations, without considering the aforementioned 'x', 'y', 'width' and 'height', which can have a dramatic effect on what actually appears in your filter.

Until Inkscape gains a UI to make some sense of this madness, I recommend leaving the advanced options of this filter to the experts. But if you do want your image to



be distorted to fit the bounding box, per 0.48, you will have to take a deep breath, roll up your sleeves, and wade into the XML editor to deliver a swift dose of preserveAspectRatio="none".

There's one last thing worth noting about the Image primitive, when used with external images. By default, Inkscape will put the entire path to your image into the filter UI. In order to keep your drawings more portable I strongly recommend keeping any required images in the same folder as your drawing, and then manually editing the entry in the filter settings to remove the path, leaving just the filename. You might consider embedding your image into your document, rather than keeping it in an external file, but read on

The Image primitive should have one more trick up its sleeve. but, yet again, it's broken. It's possible to select an object (or group) in your Inkscape image and then click the "Selected SVG Element" – at which point the Source of Image box will populate with the ID of the element. In this way, it should be possible to pull any other SVG element into your filter chain… except that it doesn't work. It does appear to function in 0.48, in that a rasterised version of the element is pulled in and stretched to fill the bounding box, but in 0.91 even that limited ability seems to have vanished.

So there you have the Image primitive – a filter that promises so much, but delivers so little. The useful parts that work in 0.48 are broken in 0.91, whilst the useful parts from 0.91 require you to wade into the XML editor. Meanwhile, the pitiful UI sits back, laughing at your efforts to attempt something as audacious as setting the position of your image within the bounding box. Let's hope that UI gains a little flesh in a future release, and that the ability to use SVG elements makes a welcome return.

That concludes our look at the "fill" primitives in Inkscape. The SVG spec, though, has one other – "Tile" – which lets you feed in the output from another primitive to be repeated ("tiled") over the whole of the filter region. In order for this to work, the incoming primitive needs to have a filter region that is smaller than the one it's going to be tiled into; but, as Inkscape uses a single filter region definition for the entire filter chain, even if this primitive were to be implemented, it would have no practical effect.

It hardly seems fair to have wasted your reading time with a description of one poorly implemented filter, and another that hasn't been implemented at all, so I'll finish this instalment by adding another useful primitive to your toolbox: Morphology.

Despite its fancy sounding name, this is a very simple filter: all it does is makes things thicker or thinner. And it does so with the minimum of fuss: there's just a drop-down to select between "erode" (make things thinner) and "dilate" (make things thicker), and a pair of optionally linked "radius" sliders to set the amount of erosion or dilation that will take place. Let's see this filter in action – in each case the first text object is unfiltered, and the second is filtered as described, with a radius of 2.5.

These filters are particularly useful when used with the Composite primitive, often in "In" or "Out" modes. In the following example, I've used a Flood filter to create a translucent white fill, then used a Composite "In" to trim it to the size of my eroded text. A little Gaussian Blur and Offset later, and you've got a filter that gives a 3D appearance to your text.

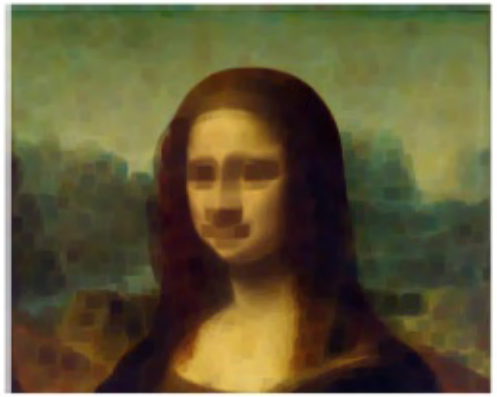
"Out" can work well with dilation, to punch out the center of the dilated image. As a simple case, consider a Morphology primitive that dilates the source, then a Composite Out that leaves only those parts of the image that are outside the original source object. What you're left with is an outline of your object, with a transparent middle.

Now, rather than punching out the source object, what if you punch out another dilated version, such that you're removing a small dilation from the core of a large dilation. Merge the original object back in, and you have an outline that surrounds the original, at a distance set by the smaller dilation with a thickness equal to the difference between the inner and outer dilations (you may need to increase the size of the filter region to avoid the result being cropped).

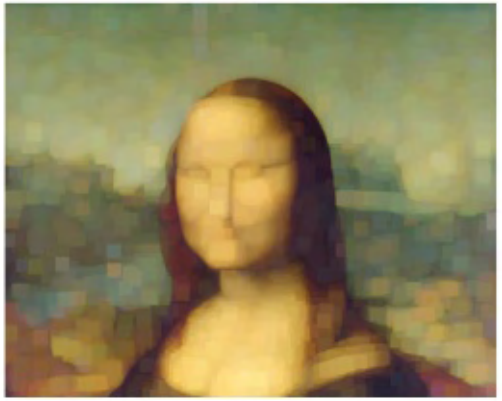
Finally, how about taking the previous idea and stacking it up a little further. You can have several outlines, all at different distances from the original object, then just merge everything together at the end. Things start to get a little complex as you add more outlines, because you're juggling a pair of Morphology primitives and a Composite for each layer of the onion, but, in principle, it's possible to carry on adding as many as you like, so long as you can keep track of them all.

Or how about this version, where I've also used Color Matrix primitives in Hue Rotate mode in order to give each outline its own color:

It's worth remembering that filters are bitmap operations that take place at the rendering stage. Although you can think of the Morphology primitive as thinning or thickening your image, it's not doing so in a vector sense, but rather by just adding or removing pixels in a bitmap version of your object. With that in mind, it also makes sense that you can apply this primitive to bitmap images imported via the Image primitive. This allows you to hide the fine details of an image by eroding them away, or blotting them out through dilation of adjacent areas, without introducing the sort of softness you would expect if you just blurred the images. In either case, Mona ends up looking somewhat worse for the experience!



Erode



Dilate

Image Credits

"La Gioconda" (aka "Mona Lisa") by Leonardo da Vinci
http://en.wikipedia.org/wiki/File:Mona_Lisa,_by_Leonardo_da_Vinci,_from_C2RMF_retouched.jpg

Mark uses Inkscape to create three webcomics, 'The Greys', 'Monsters, Inked' and 'Elvie', which can all be found at http://www.peppertop.com/

Inkscape - Part 53

A major feature of SVG filters is that they're dynamic. The calculations to produce the output aren't simply done once and then stored in the image – as is often the case with filters in bitmap editors. Rather, they're performed time and time again as you zoom, pan, rotate objects or otherwise modify your drawing. This gives you the flexibility to make changes to your filter parameters at any time, but all this recalculation takes its toll on Inkscape's rendering speed. So now that you're (hopefully) starting to create more and more complex filters, I'm going to begin this instalment by looking at a few ways to mitigate this slowdown.

When faced with a program that's slowing down due to too many calculations, there are two approaches that can be used to minimise the problem: reduce the number of calculations, or find some way to speed them up. Remembering that filters are applied on a per-pixel basis, just at the point of rendering the object, one way to reduce the number of calculations is to zoom out. An object viewed at a low zoom, which takes up 10x10 pixels on screen, occupies an area of 100 pixels. Even for the simplest of theoretical filters that means 100 calculations – but in practice it will be many more as, at the very least, there will likely need to be separate calculations for the red, green, blue and alpha channels. Zoom in so that the object fills 20x20 pixels – what we would colloquially consider to be "twice as big" – and the area grows by four times, to 400 pixels and therefore 400 calculations per channel. Zoom right in so that your small object almost fills your HD monitor and there's a lot of calculations to perform!

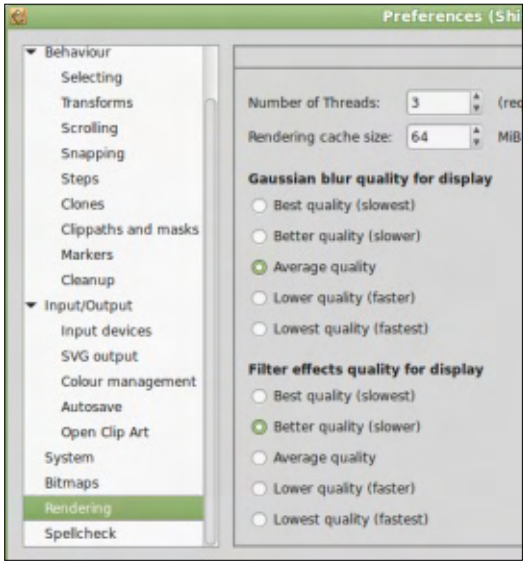
As well as avoiding large zooms, you can reduce the number of pixels to recalculate by simply resizing your Inkscape window. Does it really need to be full-sized to stretch to the whole width of your widescreen monitor? Try reducing the canvas size to something with a squarer aspect-ratio in the middle of your screen, with dialogs dragged out to floating windows at the sides.

Sometimes you don't really need to see the filtered version of an object if you just want to zoom in to tweak its shape. For those occasions, there's the View > Display Mode > No Filters option. There's also an option for viewing the Outline of objects only, which can be useful for finding elements you've lost through one of the myriad ways of making things invisible, but which doesn't really offer anything more in terms of dealing with slow filters. I mention it simply because there's also a Toggle option which cycles through all three modes – if you only do one thing today, learn the keyboard shortcut for it (CTRL-5 by default, where "5" is the key on the numeric keypad). The great thing about this is that you don't have to change modes before zooming – if you zoom in and the redraw is too slow, just press CTRL-5 to switch modes, abandoning the current redraw.

How about when you've finished tweaking a filtered object, at least for the time being? If you don't need to refer to it when working on other parts of the drawing, it's worth putting it into its own layer or sub-layer. Turn the layer visibility off, and there's nothing for Inkscape to re-calculate. If you do still want to see it, you can make a bitmap copy of the filtered object before you move the original to another layer. Select your object and use Edit > Make a Bitmap Copy (or press ALT-B): Inkscape will render a bitmap of your object, complete with filters applied, meaning that (once the original is hidden) it doesn't need to re-calculate the filters as you work on your document. When you've finished your drawing you can delete the bitmap version and re-display the hidden layer with your original content. The resolution of the bitmap copy is set in the Inkscape preferences – lower values will be created faster, but won't be as accurate when you zoom in closely. Usually this is fine, though, as the bitmap is generally there as a position or color reference, rather than needing to

be a high-resolution representation of your object.

These methods reduce the amount of calculations that need to be performed, but there are also ways to speed up filter performance even when you need to have the original filtered objects visible. Within the Inkscape Preferences (File > Inkscape Preferences… on 0.48, Edit > Preferences on 0.91) there is a panel for adjusting the rendering of filters, labelled as “Filters” on 0.48 and “Rendering” on 0.91.



Within this panel you can set the number of threads that Inkscape uses for rendering Gaussian Blur filters (0.48), or filters in general (0.91). If you have a multi-core or hyper-threading processor in your machine, increasing this value to suit can speed up rendering. The usual recommendation is to set it to the number of cores minus 1. That, in theory, allows a single core to be used for the main Inkscape process, whilst using your remaining cores to render the filters. In practice there's a whole operating system between Inkscape and your cores, so although it's a useful guideline there's no guarantee that your OS will distribute the threads so neatly.

On 0.91 you can also set aside some memory in which to cache the results of your filter calculations. This should have an effect on things like panning – where an already calculated filter result is moved in and out of view – but it will likely have less effect if you zoom in and out, as the filters will need to be recalculated for each zoom level anyway. Nevertheless, if you have plenty of spare RAM it might be worth assigning a bit more to this option to help speed things up where possible.

Finally there are a couple of sets of radio buttons governing the trade-off between display quality and speed. Filters can be approximated by rendering at a lower resolution, giving a faster redraw but with less accuracy. The buttons here let you adjust that balance for filters in general, but also for Gaussian Blur in particular (since that tends to be the most commonly used filter primitive). Note that these radio buttons only affect the display of your image on screen – exporting to a PNG file always uses the highest possible quality.

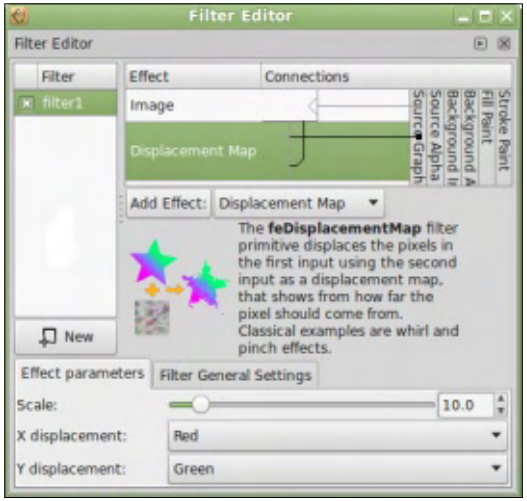
Moving on from performance, and back to filters themselves, a small correction of the previous article: it seems that the Image primitive in 0.91 does let you use an SVG element from your drawing as its input, after all. The problem is that the element is included relative to the top left of the page – so if you try to include something that's located away from that corner, there's a good chance you'll only see empty space pulled into your filter (that's what led me to think it wasn't working at all). There are two possible solutions to this: draw your included SVG element at the top left of the page (you can put it onto a hidden layer if you don't want it to be visible there in the final image), or increase the size of the filter region until your included element is visible, then use an Offset primitive to move it to the right place. Neither of these are great options, in my opinion, but, of the two, I tend to prefer placing the included element (or a clone of it) at the top left corner, on a hidden layer, as the latter results in a larger filter area to calculate – and hence slows down rendering.

Another problem with this feature in 0.91 occurs if you try to use the same object both as a target of the filter chain, and as an input to the Image primitive. This is fairly easy to do by mistake, as the clumsiness of Inkscape's filter UI makes it likely that you'll lose track of what is selected and why, but the result is an instant crash of Inkscape, with no warning and no backup file saved. If you plan to use SVG objects as inputs to the Image primitive in 0.91 it's probably best to save your file just before you add the link.

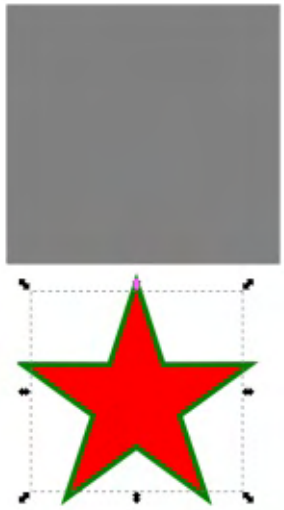
A good use for the Image primitive is in conjunction with the Displacement Map filter. This

replaces each individual output pixel with one taken from elsewhere in your image, so can be used to create various whorls, waves and distortions. It takes two inputs: the first is the image you want to distort, whilst the second is another image that acts as a "map" to tell the filter where to find each output pixel. The process is really quite simple when considered on a pixel-by-pixel basis, but soon becomes rather complex when you try to create a displacement map to perform a specific distortion.

To begin to understand this primitive, let's start with a most basic of chains:



As you can see, the first input to the displacement map is our Source Graphic, whilst the second comes from an Image primitive. In practice the Image is just a 50% gray rectangle pulled in as an SVG element (and positioned at the top left of the page so that it works in 0.91). There are also two stars in the image: the filter is applied to the red one, whereas the green one is simply there as a reference so that you can see the effect more clearly. The effect parameters are set to a Scale of 10, with the Red and Green channels being used as the source of the X and Y displacements respectively – those settings will become clear shortly.



The result of the filter is... absolutely nothing! To understand why, let's consider a single pixel in our output image. That pixel comes from somewhere in the source image, with the exact nature of "somewhere" being defined by the displacement map (the second input image). Each pixel in the displacement map is made up of a combination of four values (Red, Green, Blue and Alpha), and the settings in the filter dialog let you choose which of those values should be used for the X offset, and which for the Y offset. From there, Inkscape goes through the following steps to find out what color the output pixel should be:

1) Find the color of the equivalent pixel in the displacement map.
2) Extract the X and Y offsets from the color components that have been set in the filter.
3) Divide the offsets by 255 to normalize them into a range of 0 to 1.
4) Subtract 0.5 from the offsets to shift them into a range of -0.5 to 0.5
5) Multiply the offsets by the Scale value set in the filter.
6) Add the offset values to the X and Y coordinates of the pixel to get a new pair of coordinates.
7) The output pixel should be set to the color of the pixel from the input image that is located at the new coordinates, or an interpolated color based on the surrounding pixels if the coordinates don't point to a single pixel.

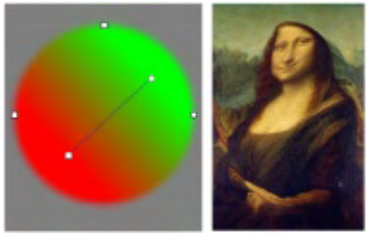
Bear in mind that our map consists only of 50% gray, with RGB values of 127, 127, 127. If you follow the steps above you'll find that gives an offset of about -0.02 pixels for both X and Y – close enough to zero to effectively mean that the output pixel is taken from the same position as the input pixel. Extend that over every pixel in the filter, and it's clear why our output looks exactly the same as the input.

Changing the rectangle to a black fill (0, 0, 0) alters the calculation somewhat. Now the offset becomes -5, -5 so our output pixel is the color of the pixel located a little up and to the left in the original image. That gives the appearance of the whole image having moved down and to the right.

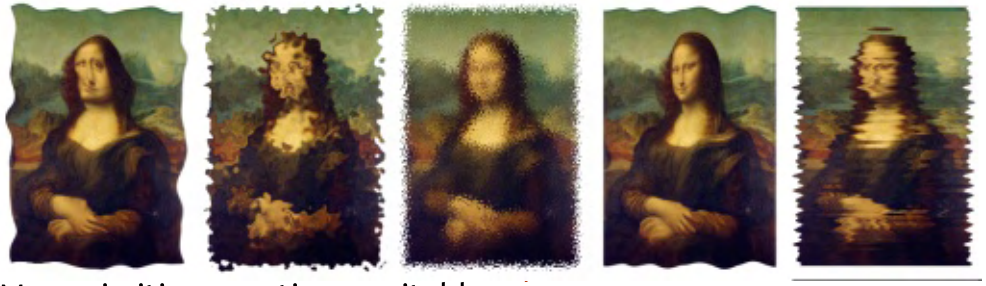
Changing the rectangle to white (255, 255, 255) has the opposite effect – the image appears to move up and to the left. Because we've specified Red and Green for the X and Y displacement, filling it with pure red (255, 0, 0) produces different displacement values for the two coordinates, effectively moving the image down and to the left; pure green (0, 255, 0) moves it up and to the right. In all cases, the value of the Blue component (or, indeed, the Alpha component) doesn't make any difference. Pure cyan (0, 255, 255) has exactly the same effect as pure green, since we've configured the filter to consider just the Red and Green components.

Used with a flat color like this, Displacement Map is just a very poor replacement for the Offset primitive. Where it comes into its own is when your displacement map contains various colors in order to use different offsets for each pixel. We know that a black fill pulls its pixels from up/left, and a white fill from down/right – what happens when we use an image with both black and white in it? Let's give it a try with a group, containing a black spiral on a white background – and we'll apply it to something a bit more complex than a red star.

By adding a little Gaussian Blur between the Image primitive and the Displacement Map you can soften the edges to give a nice ripple effect – with its intensity adjusted by changing the Scale parameter. Or how about a red-to-green gradient to give a fish-eye type of effect?



It's a bit of a cheat, because using just red and green only "stretches" your image in two directions. Overlaying a circle with perpendicular gradient that runs from white to transparent to black gives a more accurate result, but does start to hint at the biggest problem with the Displacement Map primitive: creating a suitable map image for the effect you want to achieve isn't always easy or obvious. But there is one way of creating a map that's quite simple, and extremely useful: the Turbulence primitive.



If you need a refresher on this primitive, take a look at Part 51 of this series. In short, it's a fast way to create areas filled with pseudo-random colors which, when used as a distortion map, will pull your image this way and that as you tweak the parameters. Use a low frequency Fractal Noise setting to add grotesque distortions to your image. Crank up the values a little to produce the sort of modesty-providing distortions you might find in a bathroom window. Further still and you've got a pointillistic masterpiece of shattered pixels. Unlink the horizontal and vertical frequencies and you can have a fluttering flag, or horizontal ripples.

But make sure you take the time to look at the edges. And what edges they are! From slight undulations, through spattered ink, to fuzzy vignettes. Imagine how these filters might look on shapes with even more edges, such as squares, stars and text. Better still, don't imagine; roll your sleeves up, dive into Inkscape's editor, and create your own filters.

Image Credits

"La Gioconda" (aka "Mona Lisa") by Leonardo da Vinci
http://en.wikipedia.org/wiki/File:Mona_Lisa,_by_Leonardo_da_Vinci,_from_C2RMF_retouched.jpg

Mark uses Inkscape to create three webcomics, 'The Greys', 'Monsters, Inked' and 'Elvie', which can all be found at http://www.peppertop.com/

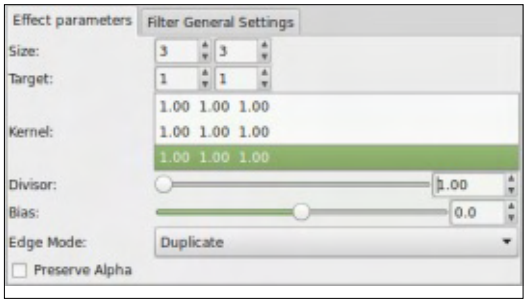
Inkscape - Part 54

This month, we're going to look at the Convolve Matrix filter primitive. Convolution is a mathematical term for a process of repeatedly applying one function to the varying output of another function. In the computing world, it's commonly used with discrete values, rather than continuous ones, as you might get when dealing with sampled audio or, indeed, with individual pixels in an image. So, in digital signal processing, convolution generally means using a function to map a series of values to a new series. In SVG filter terms, that means mapping one set of pixels to a new set. The "function" is defined using a matrix of numbers, hence the filter name is "Convolve Matrix" - although "map pixel values using a matrix" would have perhaps made it a little more understandable to the layman.

Let's look at how a convolution matrix actually works by picturing its effect on a simple image made up of a small set of pixels. For this demonstration, we'll use black and white pixels with values of 0 and 255 (with numbers in-between being shades of gray). In a real filter there are three color channels, so our single-channel black-and-white image here is merely a model to represent the calculation process. The shape we'll use is just a 9-pixel square inside a larger 25-pixel square.

255	255	255	255	255
255	0	0	0	255
255	0	0	0	255
255	0	0	0	255
255	255	255	255	255

Our first matrix will be a 3×3 array, with each cell containing the number 1.00, and the "target" specified as the center cell of the matrix. Here's how that looks in the Inkscape GUI:



The convolution process itself consists of taking our matrix and positioning it so that the target cell in the matrix is positioned over each pixel in the input image, in turn. We're going to look at the calculation that takes place for the first black pixel in our input image – the one with the red outline. The 9 pixels covered by the matrix are all multiplied by the corresponding value in the matrix cell, then added together. The result is clamped so that it doesn't exceed 255 or drop below 0, and is then used as the value for the output pixel. This image might clarify things a little – the green area represents the 3×3 matrix, with each pixel's contribution to the output shown.

255 × 1.00	255 × 1.00	255 × 1.00	255	255
255 × 1.00	0 × 1.00	0 × 1.00	0	255
255 × 1.00	0 × 1.00	0 × 1.00	0	255
255	0	0	0	255
255	255	255	255	255

The value of the output pixel is therefore:

```
(255×1.00) + (255×1.00) +
(255×1.00) +
(255×1.00) + (0×1.00) +
(0×1.00) +
(255×1.00) + (0×1.00) +
(0×1.00)
```

It doesn't take much mathematical skill to realise that the five white pixels each contribute a value of 255 to the output, whilst the black pixels contribute nothing. So the value used as the output pixel is just 255×5 = 1,275. Except that the output values are clamped, so the actual output value is just 255 – this matrix has turned the black pixel into a white one.

255	255 × 1.00	255 × 1.00	255 × 1.00	255
255	0 × 1.00	0 × 1.00	0 × 1.00	255
255	0 × 1.00	0 × 1.00	0 × 1.00	255
255	0	0	0	255
255	255	255	255	255

Moving on to the next pixel, we get a similar result. This time there are only three white pixels that contribute to the output, but that's still a value of 765, which gets clamped, so the output is again white.

Considering the remaining black pixels in our test image, it's pretty obvious that all the outside ones will turn white. In fact it's only the very center pixel that remains black. So the output from this particular convolution matrix is just a single black pixel at the center of a white square.

Some of you may have noticed that I've conveniently started with a pixel that is not on the very edge of the filter area. How does Inkscape calculate the value for the top-left pixel, for example, given that five of the points covered by the matrix simply don't exist? The answer lies in the Edge Mode popup in the filter settings: “Duplicate” copies the pixels from the outer edge to fill any missing values; “Wrap” uses the pixels from the opposite side of the image to fill the gaps as though it were working on a tiled version of the input; “None” just sets the channel values for the missing pixels to zero.

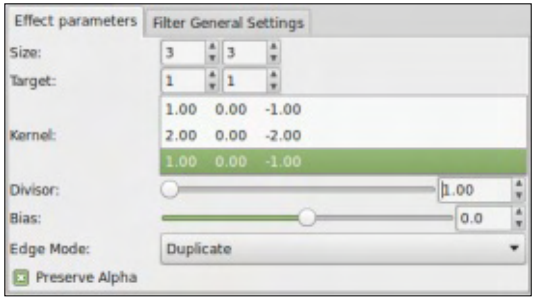
Or at least that's how it's supposed to work. According to the official Inkscape manual, this parameter is completely ignored by Inkscape, despite being present in the UI. It doesn't specify what method is used to calculate the missing pixels and, as the official manual has not been updated for version 0.91, I'm not sure if this situation has changed with the more recent release. So we'll just ignore the question, and assume that Inkscape does something to populate the missing pixels or omit them from the calculation, so that we don't have to worry too much about it.

Because the values we've chosen tend to result in calculations that get clamped, our filter, as it stands, pretty much just creates black and white pixels in the output. Before clamping, we were getting results of 1,275 and 765, but we then proceed to throw away any difference between the values because they're both greater than 255. By using the Divisor control in the filter settings, we're able to scale the output of the calculations prior to any clamping, allowing us to rein in the values to preserve those differences. A good rule of thumb is to set the divisor to the same value as the total of the individual numbers in your matrix. By setting it to 9 in our example image, the outputs of 1,275 and 765 are reduced to 142 (1,275 ÷ 9) and 85 (765 ÷ 9), giving us this result:

255	255	255	255	255
255	142	85	142	255
255	85	0	85	255
255	142	85	142	255
255	255	255	255	255

Now each output pixel is the average of nine pixels from the input image. Although it might not be clear from this small example, the outcome is a simple blurring of the input image. In reality, it would be better to use a Gaussian Blur primitive if you just want to soften your image a bit, but this was, of course, just a demonstration of the mathematics that takes place behind the scenes with the Convolve Matrix.

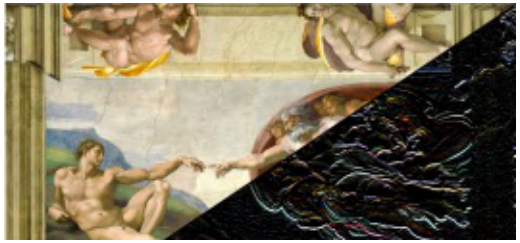
Now let's move on to some more interesting matrices. I'm going to stop with the pixel-by-pixel approach, and the mathematical explanations – it's all just an extension of the examples I've shown so far, but with larger images and multiple color channels. We'll use a different classical image to demonstrate these because Mona, quite honestly, isn't that interesting when you apply a convolution matrix. So we'll switch to Michelangelo's “Creation of Adam”, with each image showing the unfiltered version at the top left, and the filtered one at the bottom right. We'll start with a “Sobel” matrix:



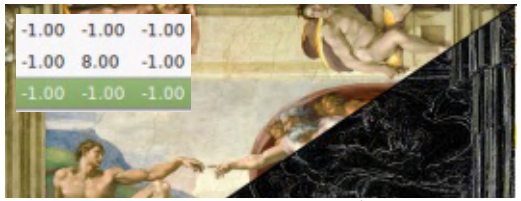
A Sobel operator emphasises the differences between adjacent pixels in one direction or another. The result is essentially a map with bright colors where there is a sharp

transition between pixels, and dark colors where there is little or no difference between adjacent pixels. In practical terms, therefore, it acts as an edge detection filter, in this case highlighting vertical edges (note, particularly, the coving at the right of the image).

Rotating the values of the matrix through 90° (so that the top row contains 1, 2, 1 and the bottom row is -1, -2, -1) turns it into an edge detection filter for horizontal edges. In this case the coving vanishes, but any near-horizontal shapes are accentuated:



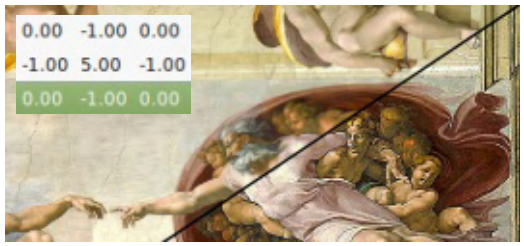
A more general form of edge detection, which highlights both vertical and horizontal lines resulting in an "outline" version of the original image, can be achieved with the following matrix:



It's easy to imagine this, followed by a Color Matrix primitive, forming the basis of a "pencil sketch" filter chain, but you can achieve a similar result by using the Bias parameter in the filter preferences. This lets you add a fixed offset to the result of each calculation, and acts to brighten or darken the output image. Setting this parameter to 1.0 with the previous filter gives this result:

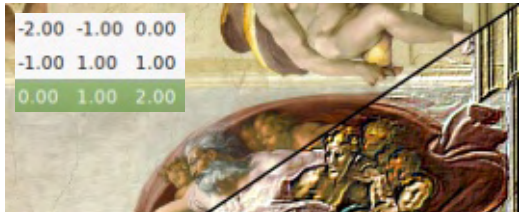
A variation on edge detection is edge enhancement. This matrix will emphasise edges but still allow the original colors to show through, resulting in a sharpened appearance to the image:

Here's another matrix that



darkens some edges whilst lightening others, giving rise to an embossed appearance.

As you can see, there are a



variety of effects that can be produced with this primitive, although it's far from intuitive to work out what values you need to enter into the matrix to get a particular output. Although the matrix approach allows for a vast number of possibilities, there are really only a few well-known matrices that are commonly used. A search online will provide you with some more examples, but they all fall into the same basic themes I've covered here.

Before concluding, there are a few more controls in the filter UI that need to be explained. The Size parameter, as you might expect, sets the size of the matrix. I've only used 3×3 matrices in this article, but 5×5 is also a common size, and you could go higher still, to consider a wider area around each source pixel. Just remember that the size of the matrix defines how many pixels need to be read and calculated for each output pixel, so increasing this parameter can quickly impose a much larger processing burden on Inkscape.

In the examples here, I've assumed that the center of the matrix is positioned over the target pixel for each calculation. It's possible to change that using the Target fields in the UI, where 0,0 would set the top-left cell of the matrix as the target. All this does is shift the output a little, so there's little reason to worry about it too much.

Finally, the Preserve Alpha checkbox determines whether the alpha of the original pixel is transferred to the output unchanged (checked), or if the alpha channel is also subject to the convolution process (unchecked). I tend to leave this checked, as it's one less channel of calculations for Inkscape to perform, and I haven't yet found myself needing to convolve the alpha channel.

Image Credits

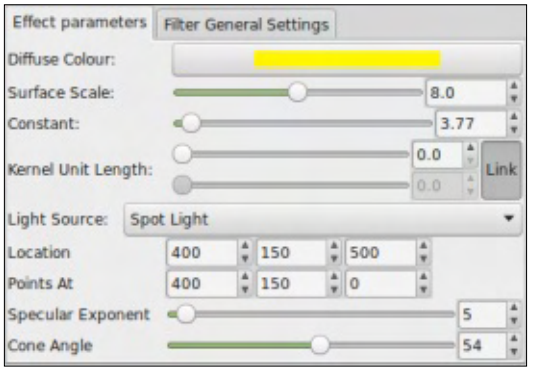
"The Creation of Adam" by Michelangelo
https://en.wikipedia.org

Inkscape - Part 55

This month we'll be looking at the last of the filter primitives available in Inkscape 0.48, Diffuse Lighting and Specular Lighting. These are used to simulate the effect of lights shining on your objects, and constitute two thirds of the Phong reflection model. The third part, Ambient Lighting, refers to light that's present everywhere in an image rather than coming from a specific light source. There's no need for a specific filter for this part as it is formed by the fill and stroke colors of the objects in your image.

Diffuse Light refers to the general light and shadow on an object that doesn't change significantly as you move your viewpoint. Specular Light, on the other hand, refers to the bright spots or reflections that shift and change as you move. Look at a shiny object near you and move your head around to see the difference – notice the specular highlights on edges and corners that move with you, and the diffuse shadows and glows of the main body of the object that remain largely unchanged.

To begin, create an object or group to apply the filter to, and then add a Diffuse Lighting primitive in the usual way. There are a few parameters to modify, but mostly it's a case of moving sliders by trial-and-error in order to achieve the result you want.



The first parameter to choose is the color of your light. This has a huge effect on the output of the filter, as the lighting effect completely replaces the original color of your objects, rather than mixing with the underlying hues. In the example that follows, all the text objects are teal (a blue-green color), but the color used in the filters is yellow. Notice that no teal appears in the output images.

In practice, it's only the alpha channel of the input image that's used by this primitive – so it doesn't matter whether you connect it to the full Source Graphic or just the Source Alpha, the result will be the same. The alpha channel is used as a "bump map" to determine each pixel's position along the z-axis – more opaque areas protrude further from the background. The Surface Scale and Constant sliders can be used to scale and offset the alpha values in order to alter the apparent depth of the object.

The Kernel Unit Length parameter can be largely ignored. It's not used by Inkscape, but may have an effect on other SVG viewers, where it's used to define the size of the pixel grid used for the filter calculations. I usually just leave it at zero.

Finally, it's time to choose the type of light source: Distant, Point, or Spot. The first indicates a light source that is an infinite distance from your object, such that all the rays of light that arrive are parallel to one another. The Azimuth parameter sets the location of the light source as an angle – 0° places it to the right of your object, with increasing values moving it clockwise around the image until 360° puts it back at the right again. Drag this slider to see the effect in real-time. The Elevation parameter sets the angle to the drawing plane: imagine a light sitting flush with your computer screen at 0° (casting low, dark shadows); as you move the slider towards 90°, the light swings out of the monitor, towards you, until it's directly over your objects; continue towards 180° and it carries on following the same arc until it's flush with the monitor on the opposite side of your image; any further values continue moving the light in a semicircle behind the monitor, and tend to not be particularly useful.

Specifying two polar values like this defines a bearing in three-dimensional space. If you ever watch an episode of Star Trek where a crew member states their

course as “249 mark 38,” this is what they're doing – just stating an azimuth and elevation to describe the direction the ship should head in. It always amazes me that they're able to judge those values to the nearest degree, but then I haven't had the benefit of a Starfleet Academy stellar cartography course!

With two polar values able to define a bearing, it only takes a third parameter, distance, to specify a particular point in space. When selecting the Point Light option, you might expect to see the same two sliders, joined by a third. But the SVG working group decided that defining a specific point in 3D should be done using Cartesian coordinates, so instead you have three anonymous fields with a single “Location” label, representing the location of the point light using x, y and z coordinates. There's no means to graphically pick an x and y location on the canvas, and the values are in terms of the coordinate system of the object being lit (which is not necessarily the same as a the coordinate system of the main drawing). So, yet again, it's down to some trial and error.

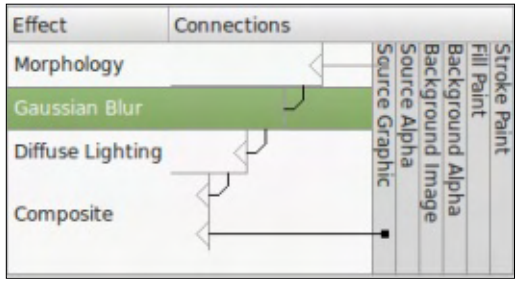
Whereas the Distant Light, at its infinite distance from the scene, projects an even illumination, the Point Light is far more nuanced. It illuminates areas near to the light source more than those at a distance, leading to gradients in the final color.

The Spot Light option is even more precise in its effect. This requires two sets of coordinates – one to specify the location of the light, and the other to define the direction it's pointing in (which is actually achieved by specifying the point in space it's aiming at). The light is projected in a cone from the source to the target, with an additional two sliders to set the characteristics of that cone: the Specular Exponent sets how focused the light is, whilst the Cone Angle defines the shape of the cone. The cone has a hard edge to it; any points outside it will not be illuminated at all, so you will need additional filter steps if you want a softer edge.

This example shows the three types of light in use on some text objects, all of which are actually blue as their base color!

You'll notice how “flat” all of these are. Because the bump map is created from the alpha channel of the input image, and our input image has alpha values of only 0 and 255, there's no scope for gentle transitions in height. If you want a softer edge to your lighting you'll need to introduce some variety in the alpha channel. The easiest way to do this is by using a Gaussian Blur primitive to the input image.

Just adding a blur will tend to spread the edge of your text outwards as well as inwards (second image in the next example). For a more pronounced effect, it's often worth using a Morphology filter to erode the input image before you blur it. By thinning your objects first, the full extent of the blur can be kept inside the boundaries of the original shapes (third image). If you then add a Composite filter, set to “In”, to the output of your lighting primitive, you can clip the result to give you something more like the rounded text you were probably looking for (fourth image).



Still we're left with that yellow color from the lighting filter. This is where the “Arithmetic” option of the Composite filter comes in (re-read part 50 if you need a refresher on this primitive). The output from the Diffuse Light filter

is intended to be multiplied with the source image in order to overlay the lighting effect onto the underlying objects, but rather than providing a nice, obvious shortcut to this operation, the Inkscape UI just exposes the parameters of the underlying SVG model. For each channel of each pixel, the Arithmetic operator performs the following calculation:

```
result = (K1×i1×i2) + (K2×i1)
+ (K3×i2) + K4
```

Where K1-4 are constants set in the UI, and i1 and i2 represent the values from a pair of input images. By setting K1 to 1.0 and all the other constants to 0, this equation simplifies down to:

```
result = i1×i2
```

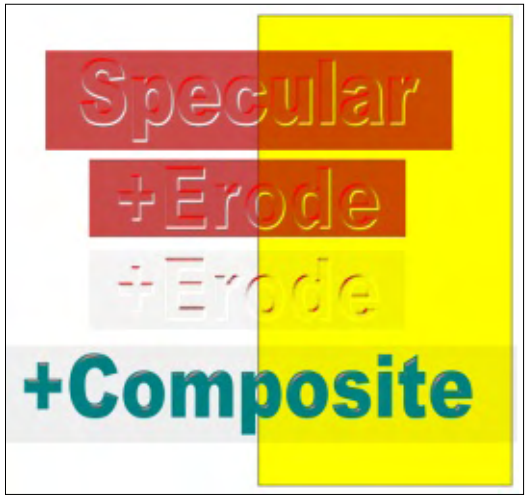
In other words, a simple multiplication of input values, which is exactly what we want. Changing the "In" operator in the previous filter chain to "Arithmetic", and setting the constants to 1, 0, 0, 0 results in a

green-looking output – the result of shining yellow light on a teal object.

Now we have an illuminated object whose base color has an effect on the output. This is obviously much more flexible than the simple "In" operator, which would have us changing the lighting color in the filter itself every time we want to alter the result. If you're worried about losing the "clipping" effect of the "In" operator, don't be: the multiplication operator also applies to the alpha channel, so all those areas in the source image with alpha=0 will result in transparent pixels in the output as well.

Moving on to the Specular Light filter, things look pretty similar in the filter UI. There's one additional parameter, but otherwise it's all the same as for the Diffuse Light primitive. That extra parameter is "Exponent" which, according to the SVG spec and the Inkscape tooltip, is used to make the specular lighting more "shiny".

Unlike Diffuse Light, this filter results in an image with mixed alpha values. Watch out for this, as seemingly bright reflections might actually just be a white background showing through! In the following image you can see that effect quite clearly on the first and second examples, where bright white "reflections" to the left of the filtered text are exposed as holes in the alpha channel once the yellow background is added behind them. Note that I've used a red Point Light in these examples, but still with teal text as the original object.



The four images above show the effect of the Specular Light filter on the plain text, then on an eroded version of the same. I didn't add a Gaussian Blur this time, as I wanted the specular reflections to be sharp and clear. Cranking up the Exponent value in the third image gets close to an output that just shows the highlights, which can then be added back to the original source image again using another Composite Primitive (fourth image).

This time the "Arithmetic" mode is used again, but with values of 0, 1, 1, 0 – which has the effect of reducing the equation down to:

```
result = i1+i2
```

This primitive therefore adds the reflections to the original image, which is the recommended approach from the SVG specification. Note, however, that a little background opacity has also sneaked through, so you might want to apply another Composite Filter, set to "In", to ensure that the result is clipped to the shape of your original objects.

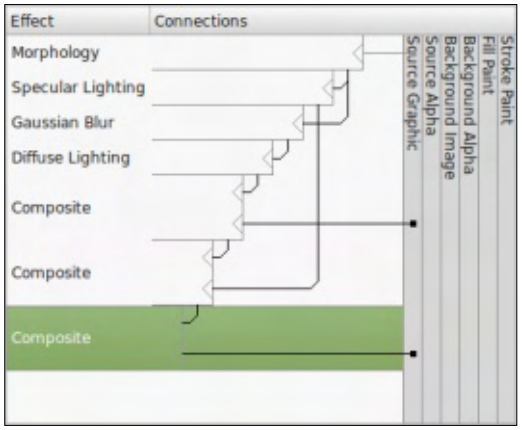
Finally it's time to combine both lighting filters to produce a fully lit object, with both diffuse and specular light. Once again, the original text is teal, so the yellow diffuse light gives it a green

appearance – but you can also see the glinting highlights from the red light source of the specular filter making an appearance.

The full filter chain for this effect isn't too complex if you take it one step at a time. First the Morphology primitive erodes the text of the Source Graphic a little, with the output from that going straight into the Specular Lighting primitive, to give those sharp, red highlights. The Morphology output also goes to a Gaussian Blur to soften the image before it's used in the Diffuse Lighting primitive.

From there, it's just a matter of combining everything together: the first Composite filter ("Arithmetic" mode: 1, 0, 0, 0) multiplies the Source Graphic with the output from the Diffuse Lighting. The second Composite ("Arithmetic" mode: 0, 1, 1, 0) adds in the Specular Lighting highlights. Although the result is almost perfect, there was a slightly visible background, albeit with a low alpha value. A third Composite filter ("In" mode) simply tidies everything up a bit.

Although they're no match for real raytracing or 3D modelling, the lighting effects in SVG can be useful for adding a little pseudo-depth to your images. This needn't be anything as obvious as the 3D text presented here: just a little highlighting can turn an otherwise bland texture into something far more interesting, or make your objects stand out from the background. As usual, the best way to find out what can be done with them is simply to experiment.

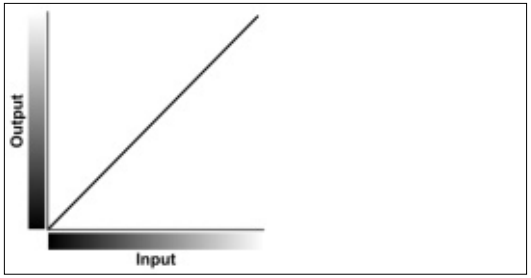


Mark uses Inkscape to create three webcomics, 'The Greys', 'Monsters, Inked' and 'Elvie', which can all be found at http://www.peppertop.com/

Inkscape - Part 56

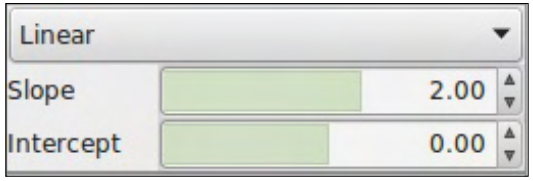
There's one last filter primitive to visit in this series, which I've kept until last simply because it's a new addition in 0.91, so isn't available to users who are still using version 0.48. The filter is called Component Transfer, and its purpose is to use a function (called a "transfer function") to adjust the distribution of values within each color channel (or "component"). It allows you to adjust brightness or contrast, or to set hard thresholds for posterization effects. As usual, I'll begin by considering the filter's operation on a single color channel, then you can extrapolate from there to how it behaves with three channels plus alpha.

A single color channel of a single pixel is represented by a number from 0 (no color) to 255 (completely saturated). The distribution of the values is linear – ramping up along a straight line – and the default settings for the Component Transfer primitive leave this line untouched. A value of 0 into the filter results in 0 out. 136 in gives 136 out. And so on. This can be represented as a graph, where the value of the channel coming into the filter is shown on the x-axis, and the value that comes out of the filter is shown on the y-axis.

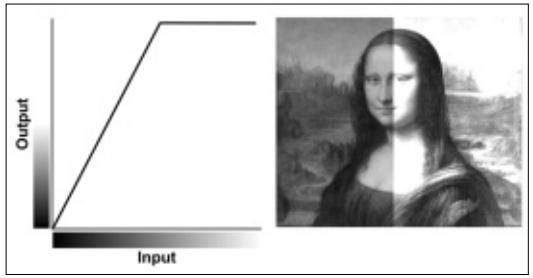


In practice, this primitive maps the input values to a range from 0 to 1 rather than 0 to 255, but the result is the same: with the default settings in the filter ("Identity"), every input channel is mapped to the output without being affected. The purpose of the Component Transfer filter is to play around with that simple 45° graph to let you change the way that input values are mapped to output values.

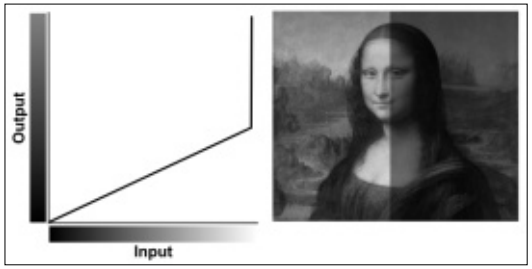
Basic mathematics tells us that a straight line graph like this can be defined by the slope of the line and the point at which it intercepts with the y-axis. One way to modify the mapping, therefore, is to alter the slope and the intercept point – a pair of values provided by the "Linear" option in the filter. The identity line has a slope of 1 – that is, for every increase of 1 along the x axis, the y value also increases by 1. By setting it to a value of 2 we can make the slope steeper, causing the output to appear brighter. Here's how it looks for one channel in the filter dialog:



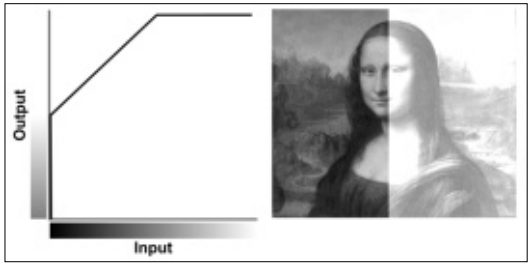
As well as showing the effect on the slope, I've also included a grayscale version of Mona, with the right-hand side showing the result of applying this change to all the color channels:



Changing the slope to a smaller value, 0.5 in this case, reduces the brightness of the image:

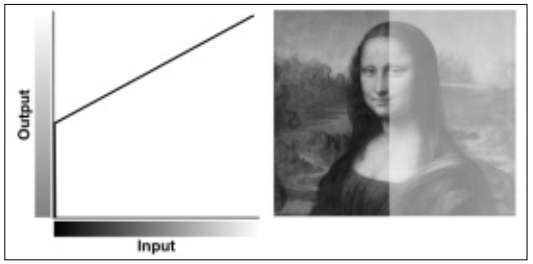


By changing the intercept you can alter the contrast of the image; you may also want to tweak the slope to ensure you don't also change the brightness at the same time (unless that's your intention). For example, setting an intercept of 0.5 with a slope of 1 would give you this result:

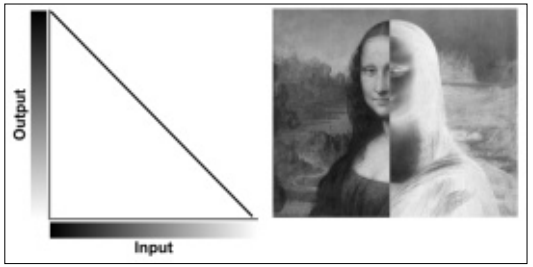


Bear in mind that color channels can't go below 0 or above 127, so the graph changes shape when you hit these limits. As you

can see, it becomes horizontal halfway along the x-axis, washing out any values above 127 by turning them completely white. Compensating for this by changing the slope to 0.5 preserves the detail a lot more, because all 255 input values are mapped, rather than just clamping half of them.

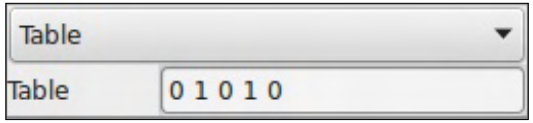


The intercept value can also be negative, to give a darker output, again with reduced contrast. It's worth noting that the slope can also be negative, which inverts the mapping so that larger input values are converted to small output values, and vice versa. With a slope of -1 and an intercept of 1, the output from the channel is completely inverted:

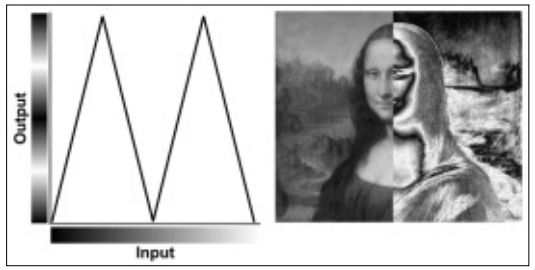


The linear mode of this filter primitive assumes that you want a simple mapping from input to output, to adjust the brightness or contrast by altering the slope and position of a single line. But there are times when a single straight line (even one that flattens out at the limits of the color range) just doesn't cut it. What happens if you want the output to ramp up, then down again, such that values at the extreme ends of the range are mapped to low numbers, whilst those in the middle are mapped to high numbers? For that we have the "Table" mode.

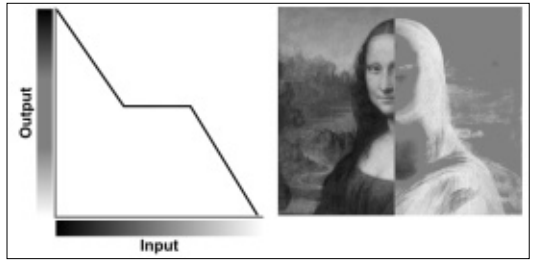
"Table" may be a little misleading, as the table you have to supply is one-dimensional. "List" might have been a better name, but table is what the SVG Working Group decided to go with, and what Inkscape exposes. The numbers in the list represent the start and end values for a series of straight line segments; the number of values in the list determines how many segments there are. For example, the table below has five values (you can use spaces and/or commas to separate them):



These five values give rise to four separate segments in the graph, causing the output values to ramp up and down rapidly as the input varies:



A table consisting of just (0, 1) would be the same as the identity mapping, whereas (1, 0) would invert the image. To flatten a section of the line, use the same value twice in succession: (1, 0.5, 0.5, 0) gives an inverted image where the details in the low and high values are preserved, but the middle third of numbers are all mapped to 127:



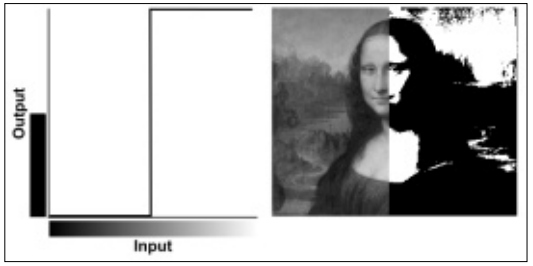
As you can see, the input range is divided evenly based on the number of values in your table, and the line ramps smoothly between them. Sometimes, however, a smooth transition is the last thing you want. Suppose that you have to reduce the number of colors in an image ("posterizing"), or even reduce it down to a stark black-and-white version. For these cases there is the "Discrete" mode.

With discrete mode you still provide a "table" of values, but rather than defining start and end points that will be interpolated between, you provide a list of the only output values that are allowed, and Inkscape will map them to sections of the input range. Provide only two numbers and any input value of 127 or less will be mapped to the first value, 128 or greater will be mapped to the second value. Instant monochrome! Provide four numbers and values from 0-63 will be mapped to the first, 64-127 to the second, and so on.

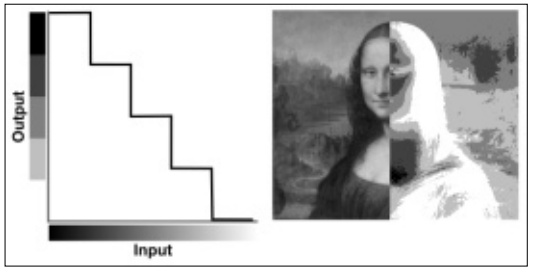
Except there's a bug in Inkscape that prevents it working correctly. In discrete mode the last value in your list is skipped – so if you provide two values expecting to get a monochrome output you'll find that every input value is mapped to the first number, and

the second is never used. The workaround is obviously to provide three numbers (typically just duplicating the last one), but then the filter will not work correctly in other SVG programs or web browsers. The issue is tracked on Launchpad as bug #1046093, and a fix has been committed for the forthcoming 0.92 release of Inkscape, which is good – but it does also mean that if you provide an extra value to get the filter to work in 0.91, your image will look wrong when you upgrade to 0.92.

For the examples below I've pretended that Inkscape works the way it should – just bear in mind that when I say (0, 1) you should actually use (0, 1, 1) to get it to work on the current release. Speaking of which, here is that monochrome output, using a discrete table containing (0, 1):



This one uses values of (1, 0.75, 0.5, 0.25, 0) to posterize Mona down to five shades of gray, whilst inverting the output at the same time:



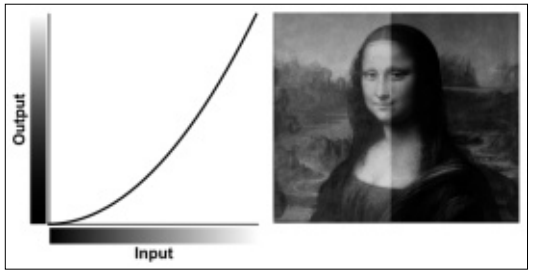
One thing you've undoubtedly noticed about all of the modes so far is that the graphs consist entirely of straight lines – either horizontal ones in the case of Discrete, or angled in the case of Table, Linear and Identity. The last option adds a bit of curvature to the graph, but don't get too excited; it doesn't allow you to draw an arbitrary Bézier curve, but rather just supply three parameters for a gamma correction curve.

In case you're not familiar with gamma correction, it's a non-linear mapping of input to output values, which is used to adjust the brightness and contrast of an image to compensate for differences in perceived brightness at the ends of the range. Think of it as a more sophisticated option than just changing the slope and intercept using the Linear mode, because it allows lower values to change at a different rate than higher values.

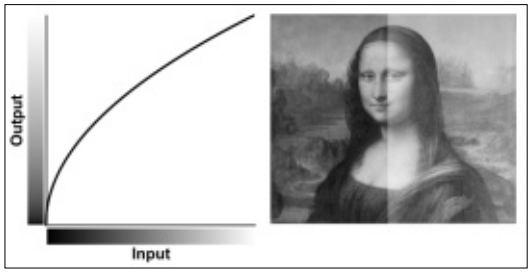
The Gamma mode takes three parameters: Amplitude, Exponent and Offset. The output value from the transfer function is calculated using the following formula:

$$output = Amplitude \times input^{Exponent} + Offset$$

That is, the input value (which is in the range 0 to 1) is raised to the power of the Exponent value, multiplied by the Amplitude and added to the Offset. Often the Amplitude is left as 1, and the Offset as 0, so the output is simply the input raised to the power of the Exponent. For an Exponent of 2, therefore, the result looks something like this:



To lighten an image simply use an exponent value of less than 1 – such as in this example with a value of 0.5.



Notice the similarity to the Linear mode with slope values of 0.5 (to darken) and 2 (to lighten). Gamma mode often gives a more detailed result, particularly where there are subtle changes in the darker areas of the input range.

Although I've used a grayscale image to illustrate this filter, in practice you can use a different transfer function for each color component, and also for the alpha channel – useful for leaving the alpha channel untouched in Identity mode whilst you alter the color channels, or alternatively for only affecting the alpha channel whilst the colors remain untouched.

To finish, therefore, here's a final image of Mona in all her colorful glory, with four different component transfers applied. The top left quarter has a Table (1, 0) applied to just the green channel, with the others left as Identity; the top right uses Table (0, 1, 0, 1, 0) on

all the color channels; the bottom right uses Discrete (0, 0.25, 0.5, 0.75, 1) on the color channels to posterize the image, and the bottom left uses Table (1, 0) on all the channels to produce a "photographic negative" effect.

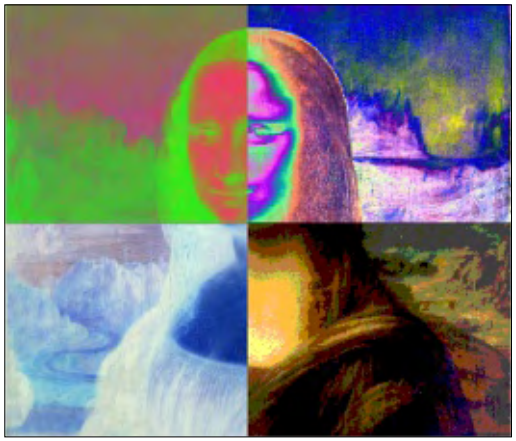
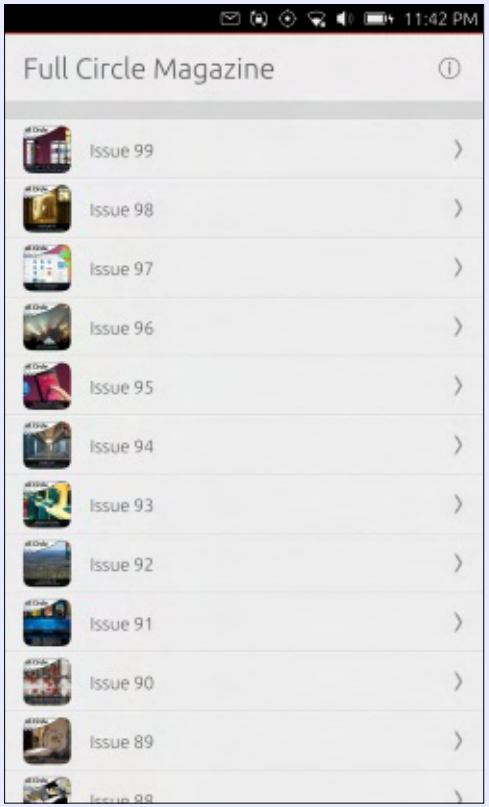


Image Credits
"La Gioconda" (aka "Mona Lisa") by Leonardo da Vinci
http://en.wikipedia.org/wiki/File:Mona_Lisa,_by_Leonardo_da_Vinci,_from_C2RMF_retouched.jpg

Mark uses Inkscape to create three webcomics, 'The Greys', 'Monsters, Inked' and 'Elvie', which can all be found at http://www.peppertop.com/

The Official Full Circle App for Ubuntu Touch

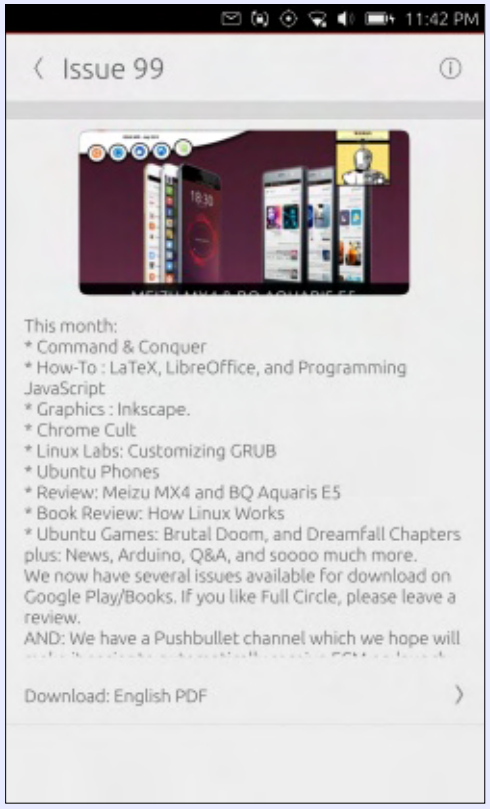


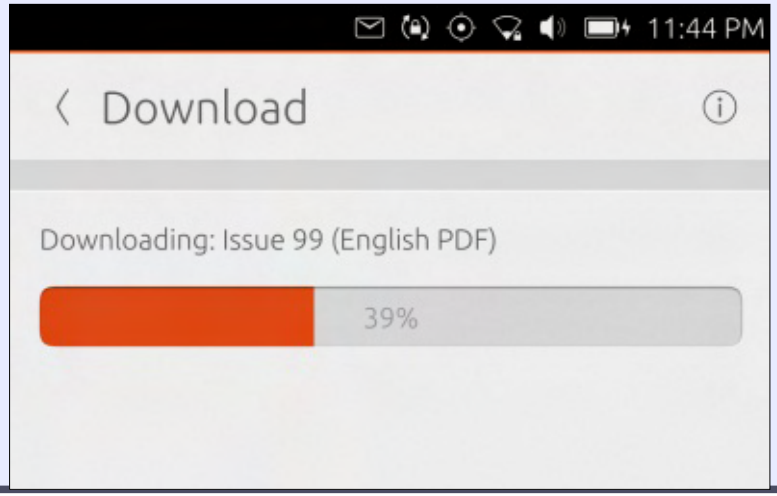
Brian Douglass has created a fantastic app for Ubuntu Touch devices that will allow you to view current issues, and back issues, and to download and view them on your Ubuntu Touch phone/tablet.

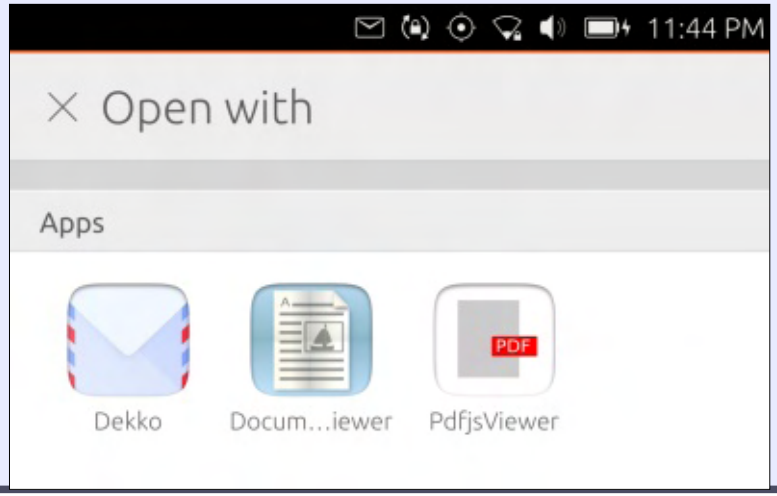
Install

Either search for 'full circle' in the Ubuntu Touch store and click install, or view the URL below on your device and click install to be taken to the store page.

https://uappexplorer.com/app/fullcircle.bhdouglass

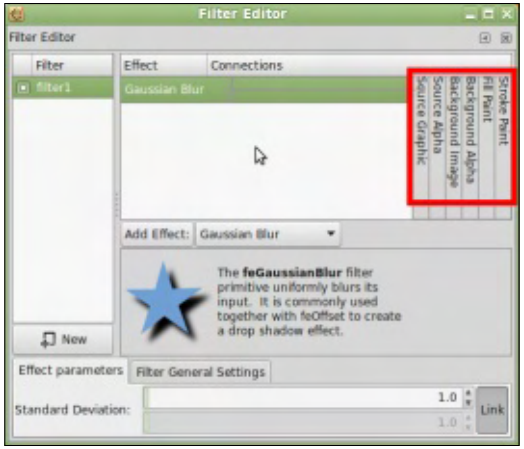






Inkscape - Part 57

Filters are an important topic for making the most out of Inkscape – at least for artistic endeavours. There's always a danger with vector graphics that they can end up looking too precise and sterile for some uses, and filters offer a way to add back in some of the subtle (and not so subtle) variations in texture and color that are often a hallmark of bitmap graphics. At least that's my justification for having spent the previous nine instalments of this series discussing filters but, having described each primitive in some detail and shown a few filter chains along the way, this article is the last on the topic, and I'll move on to something else next month.



Way back in part 48, I briefly mentioned the source input columns at the right of the filter dialog (outlined in red). We've spent a little time with “Source Alpha”, and a lot more with “Source Graphic”, but that still leaves four other options that have, so far, been completely ignored. There's a good reason for that: quoting from part 48, I wrote “of the six inputs shown in the UI, two of them require special treatment… and another two don’t work at all!”

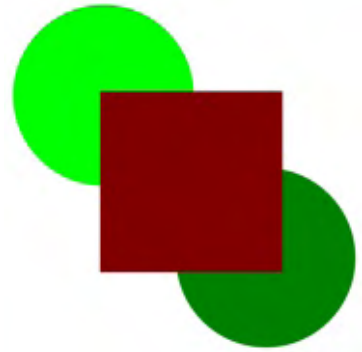
Let's first of all rule out the two that don't work. “Fill Paint” and “Stroke Paint”, according to the SVG specification, should do exactly what their names suggest. They should act in a similar way to the Flood primitive, by filling the filter area with a color, but rather than specifying the value within the filter primitive itself, it is taken from the selected object's Fill or Stroke color. This sounds like a great way of pulling a couple of colors into your filter chain, and allowing you to create filters that can adapt to the colors of the objects they're applied to. Except it doesn't work at all in Inkscape.

There's one obvious technical issue with these input sources: a fill or stroke in SVG can be more than a simple flat color. This doesn't really affect their use in a filter chain – a pattern can be repeated to fill the filter region, as can a gradient if the definition allows it – but it does significantly complicate the rendering process for Inkscape, and has not (yet) been tackled by the developers. Nevertheless, even just being able to use solid colored fills and strokes would be a useful addition. If there are no plans to add even that much, it's long past the time when these couple of columns should be removed from the UI to avoid further confusion.

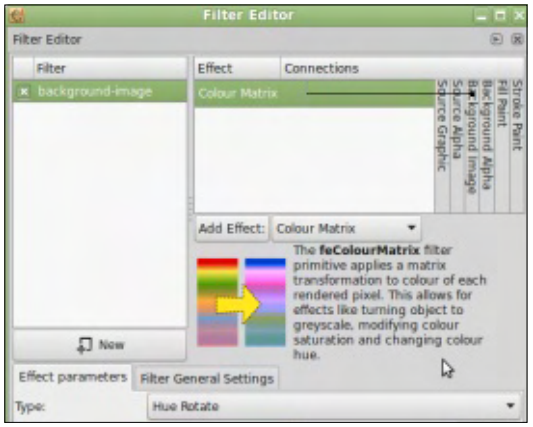
The remaining source inputs, “Background Image” and “Background Alpha” can be used within Inkscape, but only after a little preparation. These inputs represent an “image snapshot of the canvas under the filter region at the time that the 'filter' element is invoked” (according to the SVG spec). In other words, they pull in a flattened bitmap version of the drawing behind the filter region (or just the alpha channel of the same area), and make it available inside the filter chain, much like a bitmap pulled in via the Image primitive. The spec also points out, however, that holding a copy of the background image in memory “can take up significant system resources”, so the SVG content must “explicitly indicate” to the application that the document needs access to the background before these two input sources will have any effect. It then goes on to define how a document should specify that it needs access to the background by putting an attribute called “enable-background” onto an ancestor container element, giving it a value of “new”. You can fiddle around with the XML editor, or even modify your file's source code in a text editor, to achieve this, but there is a much easier way.

Before explaining the simpler method, I'll use a very basic test

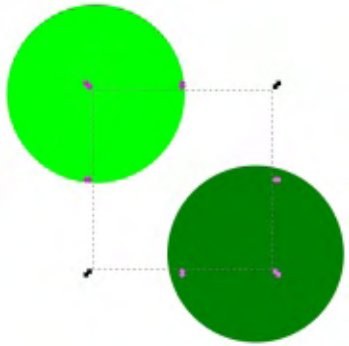
file to clarify exactly what I'm talking about. Here I have a pair of green circles as my background objects. The background consists of any content below the filtered object in the z-order, so could just as easily have been a single shape or an entire drawing. In front of the circles is a red square, the object I'll be applying the filter to.



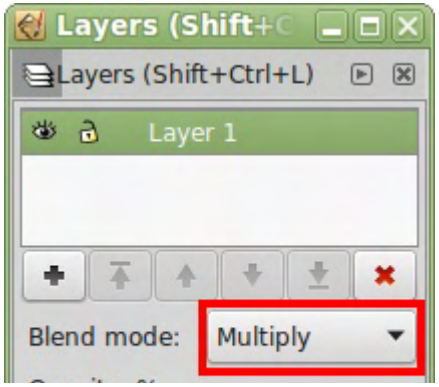
The filter itself is quite simple – just a Color Matrix primitive set to Hue Rotate mode, using Background Image as the source.



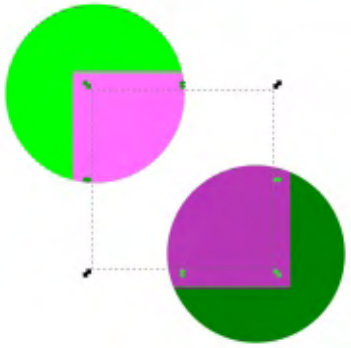
The result, at this point, is rather disappointing. The square simply becomes transparent, with no effect on the background circles at all.



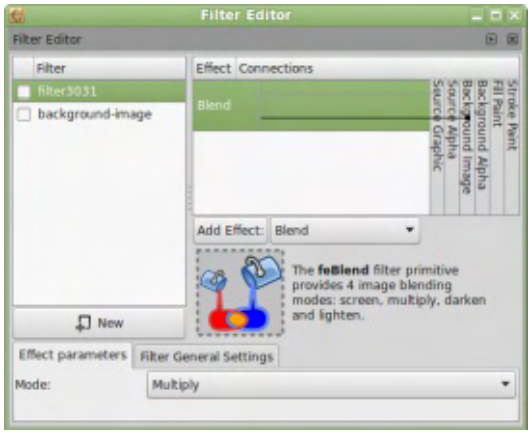
Now to add the “enable-background” attribute. Just open the Layers dialog and change the blend mode for one of the layers to something other than “Normal” (see Part 9 of this series if necessary). Don't panic if it has an unexpected effect on your image, as you can immediately change it back to “Normal” once again. The magic will already have been done.



My test image now looks like this, with the background colors rotated within the area covered by the square's filter region. By default the filter region extends beyond the selected object, which is why the color shift is present outside the dotted outline of the selection box. The square itself has disappeared, because there's nothing in the filter chain that pulls in the “Source Graphic” input.

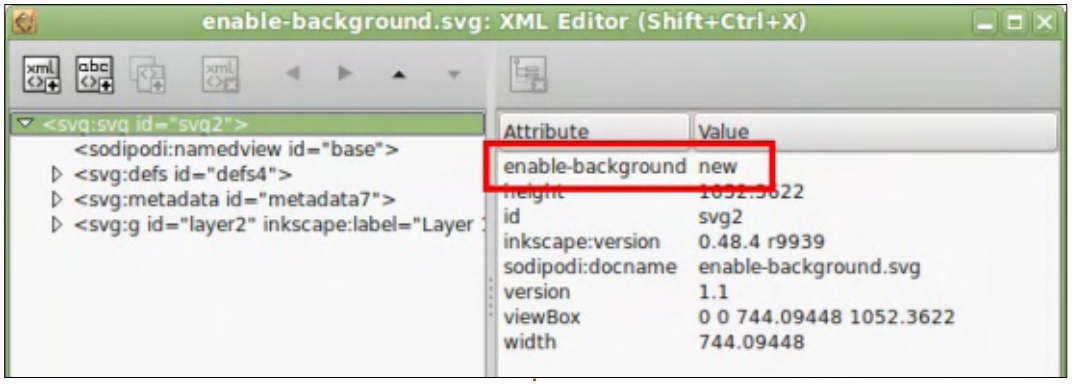


So what happened? What was the black arts and voodoo that made the filter work just by changing a blend mode, then immediately changing it back again? It's actually quite simple: the blend mode popup in the layers dialog is just a shorthand way to add a filter with a Blend primitive to the document. You can even see it appear in the filter editor.



Although it appears in the dialog, the filter isn't attached to any objects that you can select on the canvas. Rather it is linked to the layer itself. Remember that layers are just a group with some Inkscape-specific attributes added, so it's not really any different to having a filter applied to a group of objects. When the filter is created, Inkscape automatically connects the inputs of the Blend primitive to the Source Graphic (i.e. the layer that's actually a group), and to the Background Image. At the same time, it adds the “enable-background” attribute to the root node of the SVG document, visible here in Inkscape's XML editor.

The key thing is that switching the Blend Mode back to “Normal” leaves this attribute intact, although it does remove the filter.



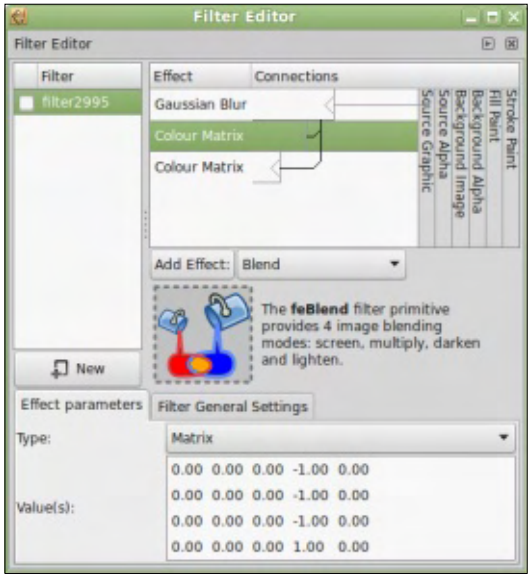
From that point on, however, you are free to use the Background Image and Background Alpha inputs in your own filter chains.

That concludes our detailed examination of the mysterious art of creating your own filter chain. But, if you've been experimenting, you've doubtlessly noticed that Inkscape already supplies an extensive list of ready-made filters, grouped by type, that make up the bulk of the Filters menu. Whilst there are those gallant masochists who dare to brave the shortcomings of Inkscape's UI to create their own complex filters from scratch, many more users simply work with the default set provided. But with the knowledge you've gained over the past few months you can do better than that: you can start with a standard filter, but then dive into its guts to edit and tweak it to suit your needs.

I shan't spend any time going exhaustively through the list of default filters, but instead encourage you to explore them on your own. Try creating a test sheet with some different objects and groups to work on: some of the filters work best on small objects, others on large ones; some require colorful content, others work just as well with a monochrome shape; some are wasted on intricate outlines, whilst others fail just as easily on featureless blocks of color.

A useful ability of Inkscape is that, when you copy and paste an object from one document to another any attached filters are copied with it. Why not start a "filter library" – a document into which you copy any particularly useful or impressive filters? Each time you create or find a great filter, just apply it to a suitable object, then copy and paste it into your library file. Similarly, when you want to use a filter from the library, just copy the object from the library file and paste it into your current creation. The filter will appear in the filters dialog, and, once you've applied it to something else in your image, you can safely delete the object you pasted in. Other users have already posted their own filter collections online – search for "Inkscape filter pack", for example – so you might find that someone else has already created just the filter you need, and it's only a copy and paste away from being used in your drawings.

When constructing your own filter chains from scratch, there's never really a question about what happens when you combine two primitives. You want a blurring and desaturating filter? No problem, just chain a Gaussian Blur primitive with a Color Matrix primitive (in Saturate mode). But what happens when you want to do the same with the default filters? There's an ABCs > Simple Blur (which consists of just a Gaussian Blur primitive) and also a Color > Desaturate filter in the menu (which provides a single Color Matrix primitive). What happens when you add both of them to an object? If you try it, you'll see that you get a blurred, desaturated result, so it is possible to combine the default filters in this way. But there's something odd going on in the filter chain. We don't have just the two primitives we might anticipate, but also a third one: an additional Color Matrix between the two primitives we expected.



If you look closely you'll see that it's not even connected to the last primitive, so plays no active role in this chain. You can delete it entirely and it won't have any effect. So why is it there?

It turns out that this is actually a rather nice addition on the part of the Inkscape developers. Let's suppose you want to add another filter to this chain, but it's one that would normally use Source Alpha as an input. To prevent any unexpected results, you need it to use the alpha from the previous filter output, which will usually not be the same as the Source Alpha at all. These extra Color Matrix primitives act as intermediate alpha outputs within the chain. So with the addition of these, you can not only link any new primitives into the image output of each filter, but also to its alpha output as well.

It's impressive that you can combine filters in this way and have them work as expected, but it can quickly lead to long, complex and hard to manage filter chains. Often a better approach is to apply one filter, then group your object before applying the next filter to the group. You can repeat this as often as necessary, creating ever deeper nesting of groups, each with its own filter applied. This certainly makes it easier to manage them in the filters dialog, as there's far less confusion about which filter you're modifying, especially if you name them well.

One final thing to note is that, in 0.91 (and the just released 0.92!), many of the default filters now have an ellipsis (three dots, "…") after their name. Choosing one of these opens a dialog which lets you enter parameters for the filter, and even see a live preview. Of course this is just a shortcut to setting parameters in the individual filter primitives, but is a welcome addition that can expose the most important parameters from across a number of primitives, whilst hiding all the other options and settings that aren't relevant in most cases. Unfortunately, there's no way to get this simplified UI back once you've dismissed it, so any subsequent tweaks will mean diving into all the gory details of the individual primitives again. Some of the separate filters from 0.48 have also been dropped, since the new parameterized filters can achieve the same effects and more. If you can't find an old favourite filter in the newer releases, look for a similar name with an ellipsis, and start tweaking the parameters!

Mark uses Inkscape to create three webcomics, 'The Greys', 'Monsters, Inked' and 'Elvie', which can all be found at http://www.peppertop.com/

FCM POLL

I've set up a poll which I hope you'll fill in. It's located at: https://goo.gl/Q8Jm4S.

We're interested in what you like/dislike about FCM. What I can change/add, and anything else you want to add.

We'll publish the results in a future issue.

LINK: https://goo.gl/Q8Jm4S