Inkscape - Part 108

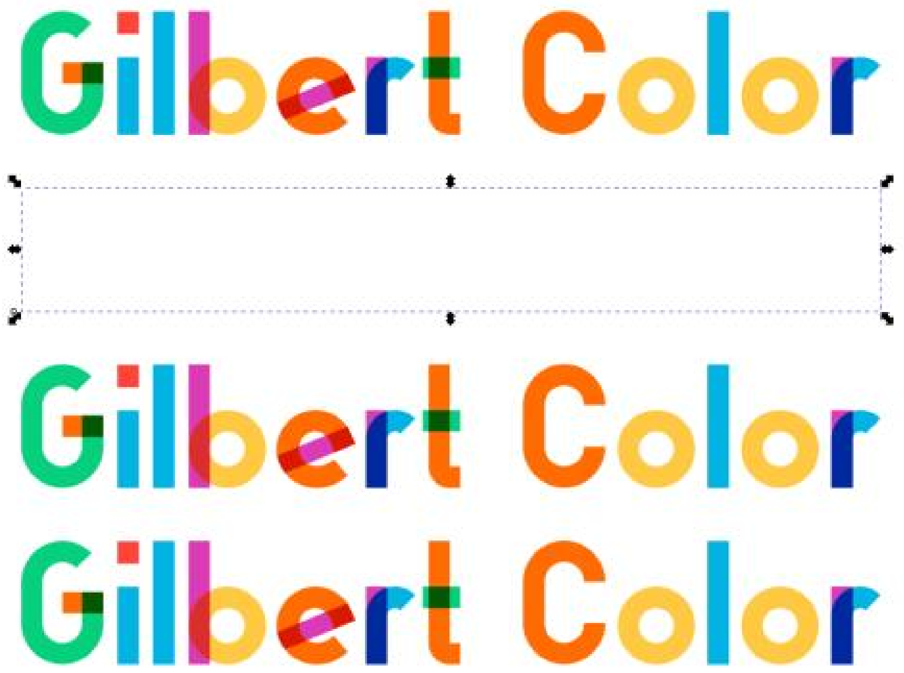
This month, we'll conclude our dive into the new text features in Inkscape version 1.0 by looking at a pair of new font formats that are now supported (and one that possibly isn't). Although I've used the term "font formats", you won't find any new file extensions in use. These formats are implemented inside the standard OpenType Font (*.otf) files that are commonly used on modern computer systems. As such, you can't immediately tell if a font supports these new features from the file extension alone, but we'll look at a solution to that problem later.

Colored Fonts

The first font variation to discuss is termed "SVG in OpenType", and it's pretty much what it sounds like. With these fonts, the shapes of the glyphs are defined using the SVG format, and then bundled into the OTF wrapper in order to benefit from many of the typographical capabilities that come with it.

Basic OTF fonts just define shapes for each glyph, with no concept of color. The program that renders the fonts is responsible for filling or outlining the shapes to make them visible, often with a solid tone, but it could equally be with a gradient or pattern. The key point is that the font provides the path outlines, and the program draws those outlines as coloured objects. With an SVG in OpenType font, this choice is usually taken out of the application's control, with colors defined solely by the font itself. Let's look at an example.

You're probably familiar with the Rainbow Flag, used as an emblem of gay, lesbian and related communities. The man who originally designed it, Gilbert Baker, passed away in 2017. In his memory, a free color font was created which was inspired by the Rainbow Flag, and named Gilbert. Due in part to the fact that there are currently very few color fonts in existence, and that even fewer are free of charge, this one is commonly used in articles about color font support in various browsers and other applications. This piece is no different, so here's a screenshot of Gilbert Color in action within Inkscape version 1.0:



The first thing to note is that it doesn't appear in color within the font selection drop-down, nor in the list in the Text and Font dialog. Neither is there any marker or indication that this is a color font, other than its name. It appears amongst all the other fonts in alphabetical order, rather than being separately grouped. This means that you'll need to keep track of the names of any color fonts you add to your system, as they won't always stand out obviously for selection in the UI.

The second thing to note is that it is, indeed, colorful. As is the case with many color fonts currently, the use of color is brash and obvious, rather than being used for subtle shadows or ornaments. But what happens if the colors used in the font aren't quite what you want to fit in with your own design project? Unfortunately the answer appears to be "tough luck".

In the previous image I have made four copies of the text, as follows:
- With a black fill and no stroke
- With no fill and no stroke
- With a red fill and thick blue stroke
- With no fill and a thick blue stroke

As you can see, removing both the fill and stroke causes the text to be transparent, but if any fill or stroke color is applied, then you get the colored font in exactly the colors that the designer intended, regardless of the color you apply.

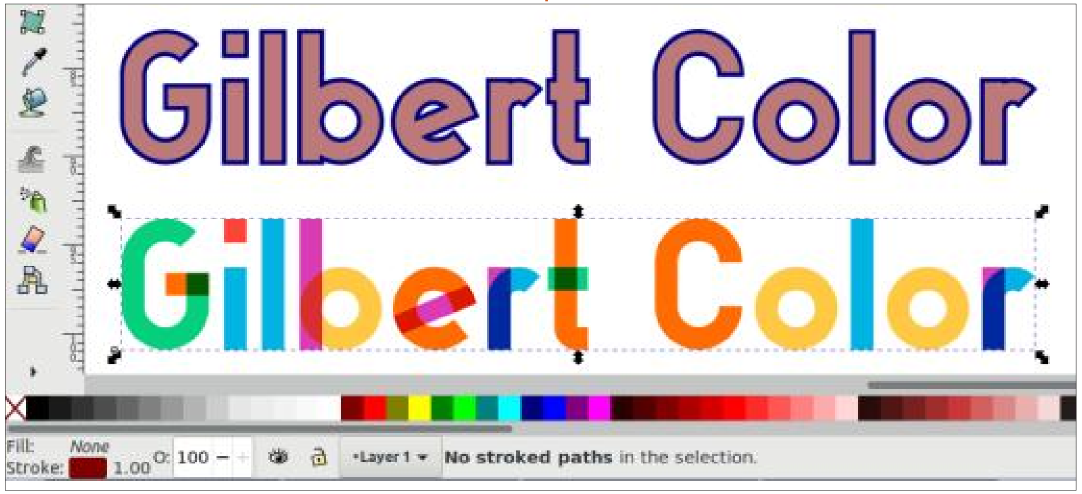
If we can't change the colors in the font using the normal fill and stroke, perhaps we could convert to paths and change the individual parts of each letter that way, right? Wrong.

In the image below, I've converted the top text – with a translucent red fill and blue stroke – to paths, using Path > Object to Path. As you can see, the result is what you would expect from a "normal" non-color font. I specifically used a translucent fill to show that the appearance of overlapping shapes in the original font does not result in overlapping sub-paths after the conversion.

The bottom example has a red stroke set, but no fill, as can be seen at the left-hand side of the status bar. I've used Path > Stroke to Path on this version, but Inkscape has left it untouched, claiming in the status bar message that there are "No stroked paths in the selection".

This inability to preserve the colors – when performing an Object to Path conversion – does somewhat limit the usefulness of colored fonts. Using Object to Path is a common operation to "fix" the style of your text when you can't be sure that the recipient has the font in question – such as when creating an SVG file for use online, or exporting to a PDF. Obviously a solid-fill conversion of the font will not produce the same results at all in these situations.

Instead of performing a conversion, it is possible to load an SVG file containing colored fonts directly into a web browser, either as a local file or by serving it online. However, even if the end user has the font installed on their machine, their experience will vary significantly depending on the browser they use. While Firefox displays Gilbert Color in all its prideful hues, the latest release of Chrome (89 at the time of writing), will show just a solid-filled version.



No stroked paths in the selection.

Similarly, when exporting a PDF from Inkscape, you might be tempted to use the option for embedding the font. My non-exhaustive test of PDF viewers suggests that this may result in a file that can't be opened at all by some applications. Selecting the "Convert text to paths" option in the Save As… dialog results in a readable file, but only because the text is, once again, flat-filled with a single color.

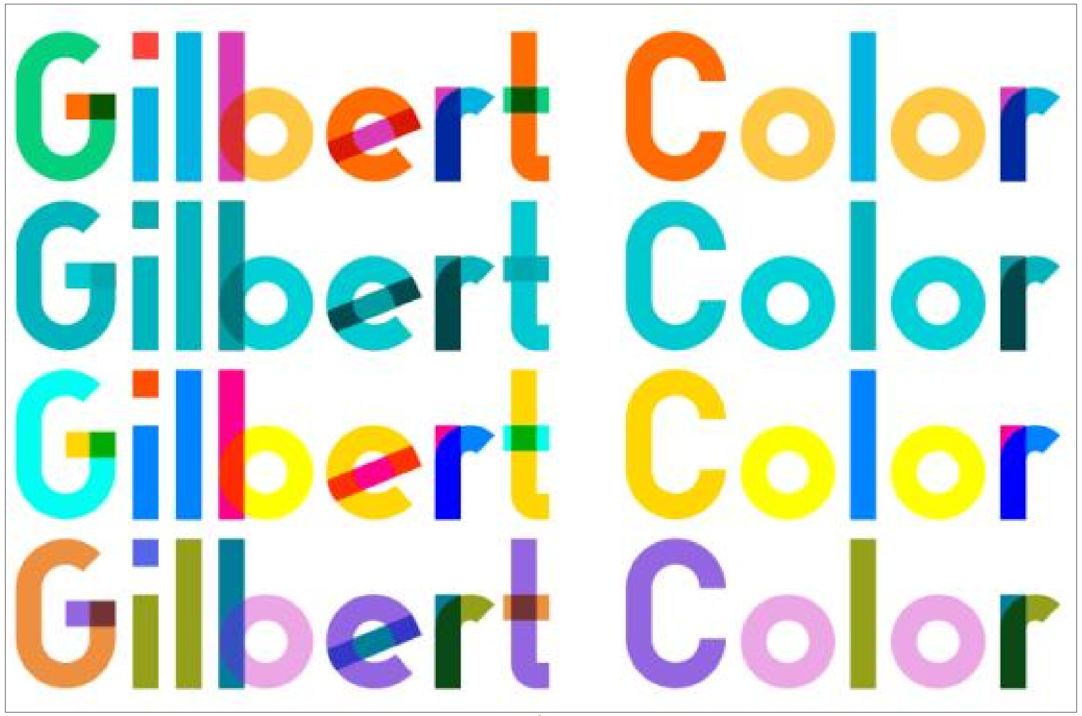
Exporting to a PNG does work correctly. If you wish to create an SVG file for use online, however, the best option may be to use Edit > Make a Bitmap Copy in order to embed a bitmap version of your text into the SVG content, should you wish to ensure its colorful appearance across different browsers.

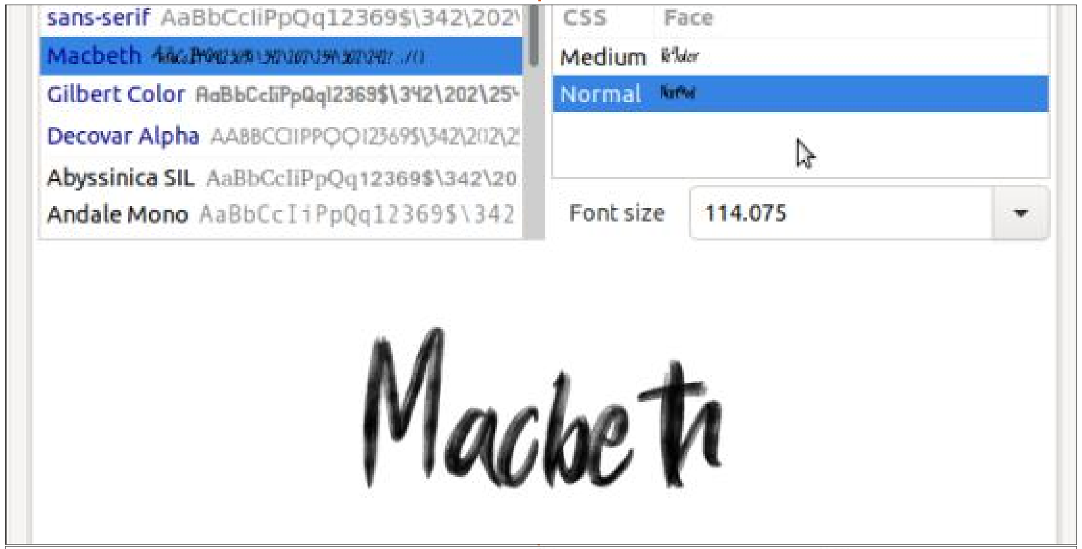
One final tip: if you do want to use different colors to those encoded in the file, you may be able to achieve the result you're looking for by applying a filter. Be aware that this approach gives you only limited control over the choice of colors – unless you're a filter expert who is prepared to spend a long time crafting a complex filter chain. In this final example, the top image is the original text in its natural colors, whereas the other three were the result of randomly using some of the filters from the Filters > Color submenu.

Bitmap Fonts

There's an interesting side-effect of allowing SVG content in OpenType files. Due to the fact that SVG content can include embedded bitmap graphics, this format offers a backdoor through which bitmap fonts can be created. Of course, these are not the bitmap fonts that were prevalent in the 1980s – the OpenType wrapper imbues them with many modern font capabilities, if properly constructed. And unlike the pure bitmaps of the past, these modern files can seamlessly combine vector and bitmap content as required.

Bitmap fonts tend to result in larger files than their purely vector counterparts. For retro-styled fonts that hark back to video game text from decades ago, the size may still be modest. But many bitmap fonts are created to simulate brush strokes, spray paint

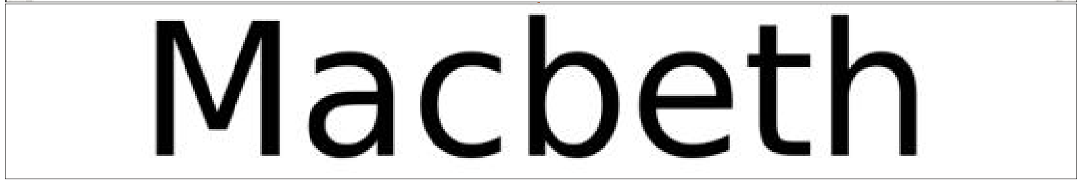




sans-serif AaBbCcIiPpQq12369$\342\202\
Macbeth
Gilbert Color AaBbCcIiPpQq12369$\342\202\25
Decovar Alpha AABBCCIIPPQQ12369$\342\202\2
Abyssinica SIL AaBbCcIiPpQq12369$\342\20
Andale Mono AaBbCcIiPpQq12369$\342

CSS Face
Medium
Normal

Font size 114.075



or photographs of physical objects, at a high resolution, which results in very large file sizes.

If SVGs in OpenType fonts are relatively rare, the bitmap-based subset is rarer still. Finding free fonts that fit this definition is even more of a challenge. So take my Inkscape findings with a pinch of salt, as they're based on a single sample of just one font.

I installed a brush-stroke font called Macbeth on my system, and tried using it in Inkscape 1.0 (and also the recently released 1.1 beta). In the font drop-down, and in the Text and Font dialog, the previews showed exactly what I had hoped for: translucent brush strokes in a dynamic, script style font.

On the canvas, however, it is a completely different matter, as shown in the image below. Either there's a problem with the font, or an issue with Inkscape's ability to deal with bitmap-in-SVG-in-OpenType files. I suspect the latter, but the fact that the previews work gives me hope that this is a temporary issue that will be fixed in a future update.

Despite Inkscape's refusal to render the font correctly, Firefox does display the font as intended within an SVG file. Oddly, despite not supporting color fonts, Chrome does also display the Macbeth font correctly when included in an SVG file, sending somewhat mixed messages about its support for SVG in OpenType.

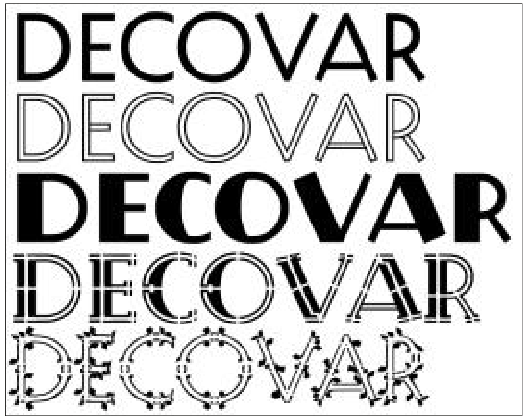
Variable Fonts

Typefaces typically live in "families", consisting of related fonts that vary in weight or style. A single typeface may consist of a large number of separate *.otf or *.ttf files, each holding a separate variation, such as light, bold, black, condensed, expanded, italic or some combination of these. But as non-bitmap fonts are usually made up of the same basic paths which are tweaked for each variant, wouldn't it make more sense to have just a single font file, and expose different parameters for controlling the path shapes? That is the premise behind "variable fonts".

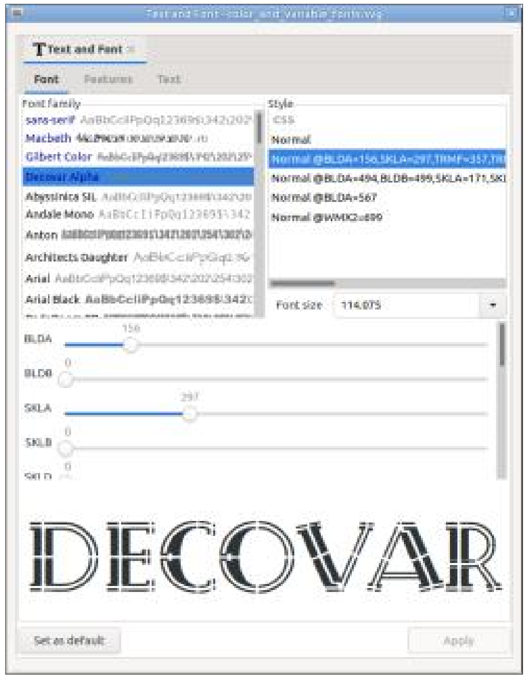
A variable font file – once again in a *.otf wrapper – is typically larger in size than an individual font, but considerably smaller than an entire family. It exposes a number of parameters, which are referred to as "axes" and which can potentially control any aspect of the font's design. The original designer chooses which parameters to expose, with each axis being assigned a cryptic four-character name. Those names are also mapped to more human-readable versions within the font file, but we'll come back to that shortly.

As well as the individual axes, the designer can also define "named instances''. These are specific collections of axis values that are given a name. For example, the designer might include a "Weight" axis that runs from 0 to 1000, but define some named instances for "Light" (300), "Regular" (500), "Bold" (700), and Black ("1000"). If the font also had a "Slant" axis, then the named instances might also include options such as "Bold Italic" and "Light Oblique". Think of named instances as being a shortcut to a pre-defined set of parameters, so you don't have to tweak them all yourself.

In keeping with the tradition forged at the start of this article, I'm going to use the same example font as every other article to demonstrate the use of variable fonts. Decovar is free of charge, and offers a wide selection of axes which can result in combinations of glyph shapes ranging from the conventional through to the bizarre. Here are a few examples rendered in Inkscape and, remember, these are all created from a single font file.



Within Inkscape, the axes are adjusted within the Text and Font dialog. You can see them displayed as a series of sliders just above the font preview in this screenshot. Note that there's a scrollbar on the right to access more sliders: Decovar exposes 15 different axes in total!



Normal
Normal @BLDA=156,SKLA=297,TRMF=357,TR
Normal @BLDA=494,BLDB=499,SKLA=171,SKI
Normal @BLDA=567
Normal @WMX2=899

Font size 114.075

BLDA 156
BLDB 0
SKLA 297
SKLB 0

Unfortunately, Inkscape has a few problems in both the design and the implementation of this feature. First is that the sliders are labelled with the four-letter internal names of the axes, rather than the human-readable names that the font supplies. The second is that the named instances aren't exposed in this dialog, so you have no choice but to set the sliders yourself rather than using the designer's preferred presets.

The biggest issue, however, is that the area that holds the sliders is allowed to grow and shrink according to the available space, but the lower limit is too small. If your font size is large, and your dialog height is small, the sliders can easily collapse down until only one is visible, making it extremely difficult to work with them. My advice is to keep the font size set at a small to moderate value, and the dialog as tall as you can, while you adjust the various axes. Once you're happy with the parameters you can then increase the font size again, should you need to.

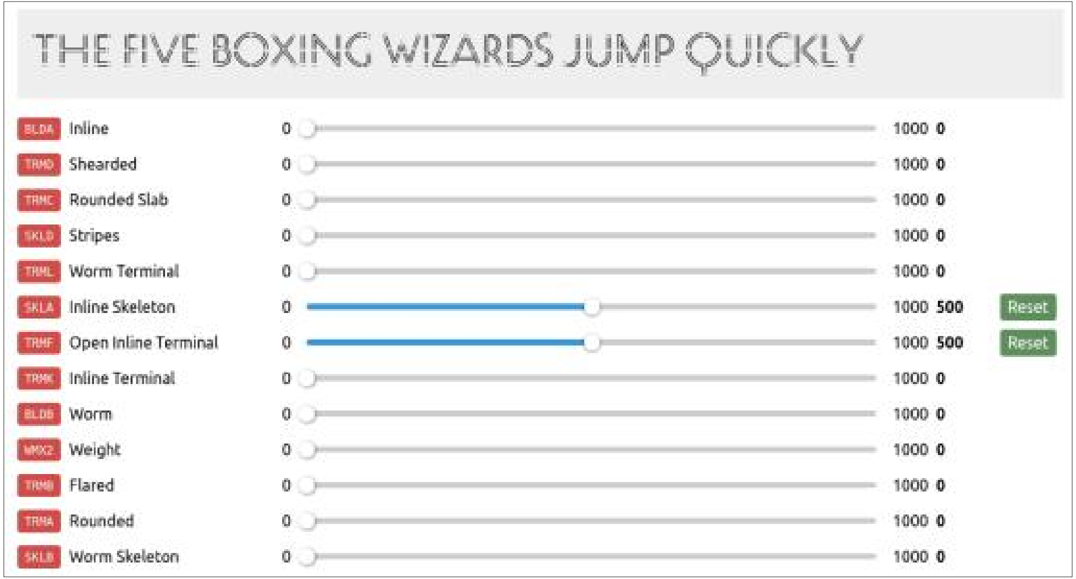
Unlike color fonts, variable fonts do appear to be converted correctly when using Path > Object to Path. Conversely, although variable fonts have broad support across current browsers, SVG files created using Inkscape don't display correctly, falling back to the basic font as though all axes are set to zero. Clearly there's a mismatch between the CSS that Inkscape is creating, and what is expected by browsers. Further investigation is required on this front but, for now, if you wish to use variable fonts in your SVG files you should probably convert them to paths before deploying the files online.

Font Information

Although Inkscape doesn't tell you the human-readable names of the axes, doesn't expose the named instances, and doesn't indicate which fonts contain SVG content, there's a way to explore the information encoded in your fonts which can help to overcome some of these shortcomings.

The bizarrely named "wakamai fondue" website is an invaluable tool. Just drag-and-drop a *.otf or *.ttf file onto the page, and a host of details about the font are extracted from the file. The processing all happens locally, without the font itself being uploaded to their server, so there are no licensing issues to worry about.

Where it's particularly useful is with a variable font. Here's a section of the output for Decovar:


THE FIVE BOXING WIZARDS JUMP QUICKLY

```
BLDA  Inline                 0  1000  0
TRMD  Shearded               0  1000  0
TRMC  Rounded Slab           0  1000  0
SKLD  Stripes                0  1000  0
TRML  Worm Terminal          0  1000  0
SKLA  Inline Skeleton        0  1000  500  Reset
TRMF  Open Inline Terminal   0  1000  500  Reset
TRMK  Inline Terminal        0  1000  0
BLDB  Worm                   0  1000  0
WMX2  Weight                 0  1000  0
TRMB  Flared                 0  1000  0
TRMA  Rounded                0  1000  0
SKLB  Worm Skeleton          0  1000  0
```

Here you can experiment with the various axes (complete with human-readable names), or select from the "Instances" popup. When you're happy with the combination you've found, you can then laboriously apply each slider's value to its equivalent in Inkscape, based on its four-letter name.

This tool offers a whole lot more information as well, and is a great way to check out the hidden capabilities of your fonts. In fact, through using the beta of the next version of the tool I was even able to discover that Gilbert Color actually offers five different variations on its color scheme, via the OpenType "alternatives" feature. Until Inkscape offers a way to access them, however, I guess we'll have to stick to tricks with filters.

Links

Gilbert Color:
https://www.typewithpride.com/
Macbeth:
https://www.dafontfree.co/macbeth-opentype-svg-script-brush-font/
Decovar:
https://github.com/TypeNetwork/Decovar/
Wakamai Fondue:
https://wakamaifondue.com/

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

Written by Mark Crutch

Inkscape - Part 109

There are still a number of changes and new features in Inkscape version 1.0 which I haven't covered over the past few months. In order to get through as many new features as I can, this month I'm going to take a whistle-stop tour of some of the smaller features which don't necessarily require complex examples to demonstrate. Hold on to your hats!

Y-Axis inversion

I've said this before, and no doubt I'll say it again: Inkscape is not a CAD program, despite offering some CAD-like features. Nevertheless, it has historically behaved the same way as most CAD programs – and even traditional paper-and-pen technical drawing – with regard to the position and orientation of its y-axis. Previous versions of Inkscape placed the origin (the 0, 0 point) at the bottom-left of the canvas, with the positive direction of the y-axis going upwards.

If you come from a CAD, drafting or graphing background, this may seem perfectly sensible. But Inkscape is really an SVG editor, and SVG is a product of the web world. Web pages grow from the top downwards. Add some more paragraphs to your HTML page and the browser simply extends the scroll bar to let you reach them. SVG works in the same way, with its origin at the top-left of the drawing area (i.e. the browser window), and the positive direction of the y-axis going downwards.

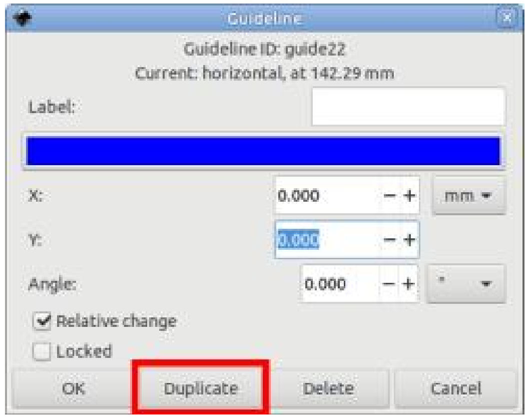
This mismatch between Inkscape's on-screen behaviour and the requirements of the SVG format meant that the coordinates of points viewed in the program would be different to the values that were actually stored in the file. For most users, this wasn't a problem, but when using Inkscape to produce graphics that will ultimately appear on a web page, this mismatch could make life unnecessarily difficult, especially when trying to write JavaScript to dynamically alter the file within a web browser.

With v1.0, Inkscape now defaults to the SVG standard for its origin and y-axis direction. Should this cause you any problems or confusion, you can revert to the previous behaviour by un-checking the "Origin in upper left with y-axis pointing down" setting in the Interface panel of the Edit > Preferences dialog.

Duplicating guides

Version 1.0 introduces a means of duplicating an existing guide line. The behaviour of this differs between v1.0.x and the release candidate for version 1.1 (the full version of which may have even been released by the time this magazine is published). Let's look at v1.0 first, where the change is rather frustrating and unintuitive.

There's a new "Duplicate" button in the Guideline dialog (opened by double-clicking on an existing guide). This duplicates the current guide, in-place, then closes the dialog. No other changes are made to the duplicate, regardless of what other parameters you set in the dialog. This can easily lead to confusion as it's not always obvious that there are now two co-positioned guidelines. Double-clicking on the lines will open the dialog again, where you can make changes that will affect one of them.


Guideline

Guideline ID: guide22
Current: horizontal, at 142.29 mm

Label:

X: 0.000 mm
Y: 0.000
Angle: 0.000 °

[✓] Relative change
[ ] Locked

Suppose, therefore, that you have a horizontal guideline and you wish to create two more parallel guidelines, with 10mm spacing between them. Here are the steps:
- Double-click on the existing line to bring up the Guideline dialog.
- Click the Duplicate button. The dialog closes.
- Double-click on the newly duplicated line (which is on top of the existing line) to open the

contents ^

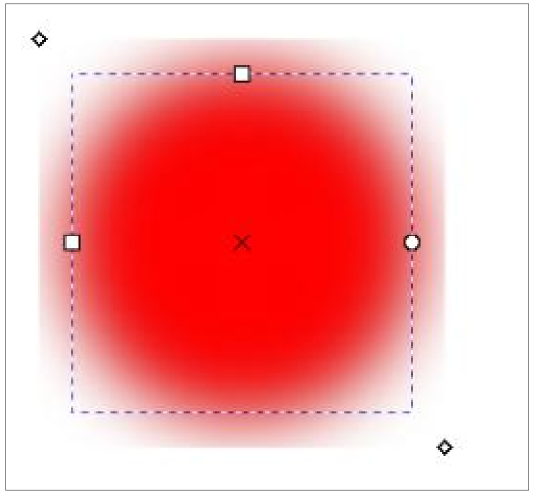
dialog again.
• Enter 10mm into the “Y” field, check the “Relative change” box, and click the OK button. The dialog closes, but you now have two guides on the page, separated by 10mm.
• Repeat all four steps, but starting with your newly duplicated and moved line.

Although the “Relative change” box remains checked between steps, the value in the “Y” field is cleared. This makes it frustrating to create a series of equally spaced guides. If you forget to click the “Duplicate” button before you change the value, you’ll end up moving your original guide by mistake. No amount of hitting Ctrl-Z will undo that mistake: you’ll have to reopen the dialog and move the guide by the opposite amount to get it back to where it started.

Fortunately this feature has been improved in the 1.1 release candidate. In that version you simply open the dialog, put in your relative change, then press Duplicate (rather than OK) to create a duplicate guide with the movement already applied. Much simpler!

Filter Region size

From a frustrating change to a delightful one. In order to reduce the amount of processing required when a filter is applied to an object, SVG includes the ability to set a finite boundary outside of which the filter is no longer calculated. For some filters – particularly those involving large blurs or offsets – the default filter region is too small, resulting in the edges of the filtered content being cut-off with a hard boundary. You can see the effect quite clearly at the sides of this heavily blurred circle.



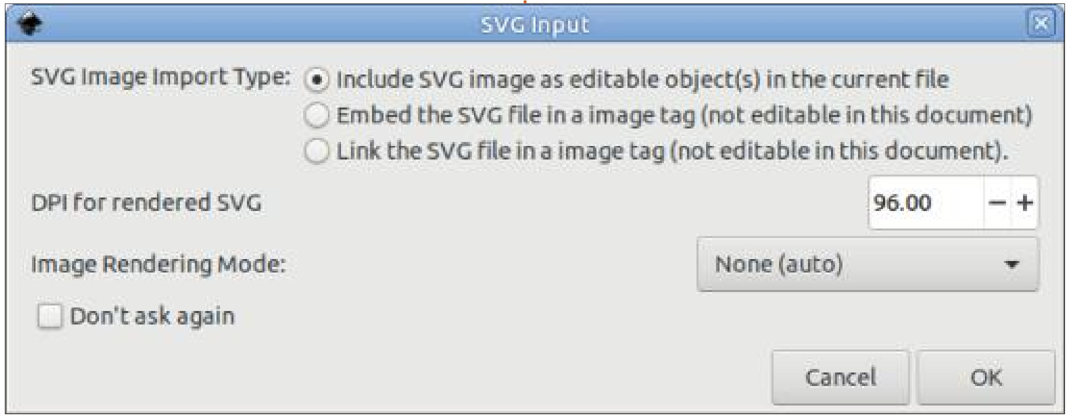
It’s always been possible to adjust the size of the filter region, via the Filter General Settings tab of the Filter Editor dialog. But the four fields you’ll find there are less than obvious to anyone who hasn’t read up on the inner details of SVG filters. You’ll also have to guess – or find out through trial-and-error – what the best values need to be for your particular image. Set the filter area too large and you’ll slow down the rendering of your image. Set it too small, or in the wrong position, and you’ll see your filter being cut-off.

With version 1.0, guessing the optimum values for these fields is a thing of the past. With a filtered object selected, switch to the Node tool (F2) and you’ll see diamond-shaped handles at the top-left and bottom-right of your object which can be dragged to directly adjust the filter region on the canvas.

Importing SVG files

Inkscape v1.0 brings some more options when importing one SVG file into another, whether via File > Import, or by just dragging and dropping an SVG file from your file manager onto the Inkscape canvas. By default, a dialog will be presented offering three different ways to import the image, plus a couple of settings that have an effect on only two of the three choices.

The first option will probably be the one most people use. It’s the same way that previous versions of Inkscape imported SVG images: by adding all the objects from the imported image as first-class, editable objects inside the existing document. Images imported in this way are essentially indistinguishable from any other



SVG Input

SVG Image Import Type: [✓ Include SVG image as editable object(s) in the current file] Embed the SVG file in a image tag (not editable in this document) Link the SVG file in a image tag (not editable in this document).

DPI for rendered SVG: 96.00

Image Rendering Mode: None (auto)

☐ Don't ask again

elements drawn within the Inkscape document.

The second option embeds the SVG content as a base64 encoded string in an <img> tag. If those words are gobbledegook to you, then you're probably not a web developer. In layman's terms, it just means that the SVG content is stored within the file, but as a single image that can be treated much like a bitmap version of the vector image. Indeed, Inkscape actually renders it as a bitmap version, so zooming in, or scaling the image too large, can make it look blocky. More on that shortly.

The third option links to the SVG file. Unlike the prior options, this means that the linked file can be updated outside of Inkscape, and the changes will be reflected in your Inkscape document. But it also means that moving the Inkscape file to another machine will also require the linked image to be sent, and the link potentially updated to reflect the new location of the file on disk. Inkscape also displays linked images as bitmap versions, so the same zoom and scale warning applies as previously.

Although Inkscape displays linked and embedded SVG images as bitmaps, it's important to understand that the linked file, and the base64 encoded embedded content, are both still SVG documents in all their vector glory. For example, even the blockiest looking embedded SVG will look crisp and clean when the containing file is loaded into a web browser.

When linking or embedding an SVG file, you do have some limited control over the bitmap that Inkscape displays as a proxy. The "DPI for rendered SVG" field in the import dialog lets you set the quality of the rasterized content. Higher DPI values will capture finer details from the vector content, which may allow you to scale or zoom with less obvious loss of detail. The "Image Rendering Mode" pop-up lets you select the trade-off Inkscape uses between quality and speed when rasterizing. Most of the time leaving this as "None (auto)" will be good enough.

Because the bitmap representation is just an artefact of the way that Inkscape works, and doesn't affect the underlying vector content, it's even possible to change the DPI and render trade-off after the image has been imported or linked. Right-click on the image and select Object Properties, or use Object > Object Properties. In the dialog that opens, you can modify the DPI setting or change the rendering mode – with even more options available than in the original import dialog. This means that any blockiness that appears as a result of scaling an imported SVG image can be addressed after-the-fact, so there's no need to worry too much about which values you use when importing.

The defaults for the import dialog are set in the Imported Images pane of the Edit > Preferences dialog (where you'll also find the defaults for importing bitmap images). Should you check the "Don't ask again" box in the dialog, then this is also where you need to come to re-enable it once you discover that the defaults don't always suit your needs.

Mesh Gradient polyfill

If you save an SVG image containing a Mesh Gradient (see part 59 of this series), Inkscape will now embed a JavaScript polyfill in the file. As I've remonstrated previously, browsers still don't support mesh gradients in their SVG implementations. This polyfill goes some way towards addressing that shortcoming.

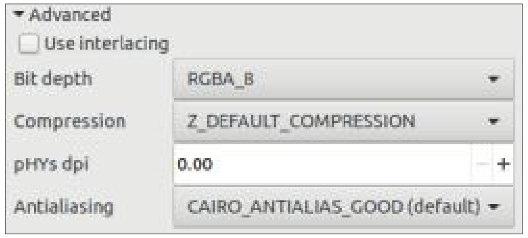
When the SVG file is loaded directly into the browser, or is included within a web page in a way that allows JavaScript to run (i.e. via an <object> tag, or by inlining the SVG content), the polyfill code will execute, rendering a bitmap version of the mesh gradient in order to approximate the original appearance of the file.

The idea is to break the impasse that is preventing mesh gradients gaining browser support. The browser vendors won't put time into their implementation due to a lack of files on the internet that use the feature. But few people put such files online because the browsers don't support them.

If you're technically competent enough to be able to put SVG images online in an <object> tag, then I urge you to create something with mesh gradients and share it with the world. Only by increasing the number of such files out there can we ever hope to get native support in web browsers.

PNG EXPORT

The PNG export dialog has gained an Advanced section which allows you to set a number of parameters for the exported file. Most users will probably never need them, so I won't go into detail about them here, other than to point out that the "pHYs dpi" field is almost certainly what you're looking for before opening a thread on the forum about how Inkscape PNG files don't appear at the "correct" size in some other program. Unless you have a specific need to modify these fields, you can probably leave them as-is. They're hidden in an Advanced section for a reason.


Advanced
Use interlacing
Bit depth: RGBA_8
Compression: Z_DEFAULT_COMPRESSION
pHYs dpi: 0.00
Antialiasing: CAIRO_ANTIALIAS_GOOD (default)

3-DIGIT RGB VALUES

RGB colors are often denoted as 6-digit hexadecimal values. But a common shortcut in CSS is to provide just three hex digits, each of which is doubled to produce the final 6-digit (hex) version. So, a value of #1ca would be expanded to #11ccaa. Inkscape now supports entering three digits into the Fill & Stroke dialog to set colors, though the value that is actually stored in the SVG file will still be the full 6-digit version.

The alpha (opacity) will be set to 100% (a value of 255, or #ff in hex), but if you enter a 4-digit hex value, this will be expanded in a similar way to produce both the 6-digit color and a 2-digit alpha value. So, #1cab would be expanded to #11ccaa and #bb.

SAVE AS A TEMPLATE

It's long been possible to add to Inkscape's set of templates by creating your own SVG file and putting it in the right place on disk. With version 1.0, the process has been made a whole lot simpler with the addition of a "Save Template…" entry in the File menu. Simply create the template you need, setting the page size and units, adding guides, and including some boilerplate SVG content if you wish, then choose the menu entry and enter a few basic details. Your template will appear in the "New from Template" dialog immediately – you don't even need to restart the program.

It's not all perfect, however. Although the creation dialog has a field for Keywords, there's no indication as to how these should be delimited. From looking at the internals of existing templates, I believe that you need to use spaces between the words. Even doing this won't make them searchable in the "New from Template" dialog, though. It appears that there's a bit of a mismatch between the template files that Inkscape creates using this feature, and the code that displays them in the dialog.

Existing templates store the metadata you provide in some XML elements whose names are prefixed with an underscore. Using the "Save Template…" feature stores them in un-prefixed elements. It's possible to modify the XML content in a text editor, adding the underscores to make the template keywords searchable, but that really shouldn't be necessary. I'll be filing a bug report about this one.

Even with this glitch, the ability to more easily create templates is a very welcome addition.

REMOVED FEATURES

A few features have been removed from version 1.0 for various reasons. These are the main ones that you might notice:
File > Import Clip Art: This feature used to allow direct downloading of files from the OpenClipart.org website, however the API that Inkscape used is no longer operational. The website does indicate that V2 of the API is in beta, so perhaps this feature will return in the future.
Save As Cairo PNG: This option has been removed from the Save As dialog as it had limited functionality, and was often confused with the File > Export PNG Image option, which is much more capable.
UniConvertor: Inkscape is no longer built with the UniConvertor library. This means that a number of third party file formats can no longer be opened or saved from Inkscape directly. If you do need to work with any of these file types a

possible workaround would be to install the UniConvertor command-line tool and convert the files to the SVG format prior to loading them into Inkscape, or back the other way after saving.

Selection Sets: Despite being added in only version 0.91, the Selection Sets dialog has been removed. I described this feature in part 62 of this series, and I would rather have seen it polished and improved than dropped entirely. Performing some types of complex selections in Inkscape can still be tricky, and selection sets offer a way to combine several simpler selections to achieve the same result. Nevertheless, I doubt this dialog will be much missed by most users.

When even the list of small changes fills a whole article, it’s clear that Inkscape development is continuing at a pace. The imminent release of version 1.1 will no doubt add to my list of topics to cover, but next month I’ll be continuing to describe some of the more significant changes in 1.0.x.

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

Written by Mark Crutch

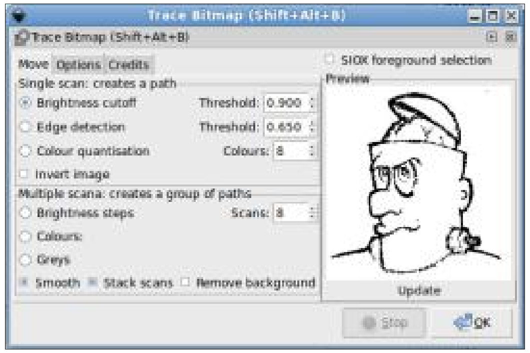
Inkscape - Part 110

I'm starting to think that the Inkscape developers have deliberately synchronised their calendar with the Full Circle Magazine article deadlines to ensure that they announce new releases just a few days after I've submitted this column, leaving the News pages to break the story. As you may have noticed in last month's issue, Inkscape 1.1 has been released, and is now available for download from the Inkscape website. For Linux users there are versions in AppImage and snap format, as well as an official PPA repository, which may be preferable to Ubuntu users in particular.

The 1.1 release does bring a few welcome additions; however, I'm still working my way through the changes introduced with 1.0.x in this column, so the very latest changes will have to wait. This month will be all about the improved UI in the Trace Bitmap dialog.

Trace Bitmap Dialog

This dialog has seen some substantial changes in the layout of its options, partly to avoid some confusion that could easily happen with the previous UI, and partly to integrate some tracing modes, one of which I'll describe later in this article, with the others being held over until next month. Let's deal with the placement of the long-standing features first of all, by taking a look at how the dialog appeared when I first introduced this feature, back in part 19 of this series:


Trace Bitmap (Shift+Alt+B)

Move | Options | Credits

Single scan: creates a path
- [✓ Brightness cutoff] Threshold: 0.900
- Edge detection Threshold: 0.650
- Colour quantisation Colours: 8
- ☐ Invert image

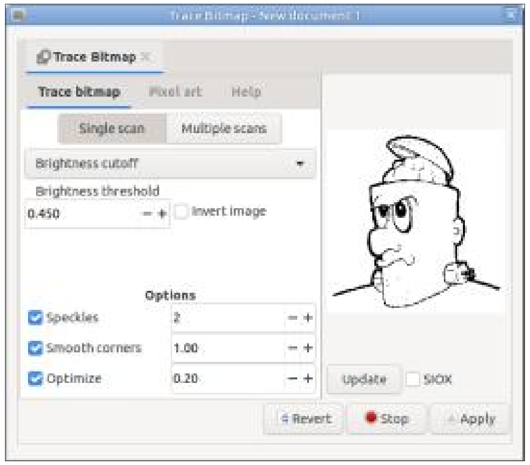
Multiple scana: creates a group of paths
- Brightness steps Scans: 8
- Colours
- Greys
- ▨ Smooth ▨ Stack scans ☐ Remove background

☐ SIOX foreground selection

Here we can see the mis-titled "Move" tab (later versions corrected this to "Mode"), showing the two main operating modes, "Single scan" and "Multiple scana" (which was probably intended to be "Multiple scans").

Each mode then offers different approaches to tracing, as a set of radio buttons that actually encompasses both parts of the dialog. The result is that only a single tracing method can be selected at a time, even if the design of the dialog makes it initially appear as though each mode can have a different method selected.

Within each mode area there are also various other parameters that affect the tracing algorithm. The layout makes it somewhat unclear whether these parameters all apply to every tracing method, or whether some are tied to particular algorithms (answer: it's the latter). Finally there are options that affect all methods and modes, in the "Options" tab, plus a lone "SIOX foreground selection" option that I described in part 20 of this series, but which I summarised with "you would be better off separating out the foreground using The GIMP (which has its own, more interactive, implementation of the SIOX algorithm) and then tracing the result.", here's what I had to say about this dialog in part 19: "This is one dialog in Inkscape that could really do with a little UI love. It's cramped, unintuitive, contains typos, and the spinboxes don't have the nice context pop-ups of most similar controls in Inkscape." Although the spinboxes still don't have any context-sensitive pop-ups, I'm pleased to say that all my other concerns have finally been addressed, with this much cleaner dialog in 1.0.


Trace Bitmap - New document 1

Trace Bitmap

Trace bitmap | Pixel art | Help

[✓ Single scan] | Multiple scans

Brightness cutoff

Brightness threshold: 0.450 ☐ Invert image

Options
- ✓ Speckles: 2
- ✓ Smooth corners: 1.00
- ✓ Optimize: 0.20

I will start with a small complaint. It's a general Inkscape issue, but is particularly obvious in this dialog. As I prefer floating dialogs in windows over docked dialogs at the side of the canvas, I often have several windows, each of which contains only a single dialog inside it – as in this image. In that situation, I think it would be nicer for Inkscape to collapse the tab bar down to a smaller "grab handle" that could be used to re-dock the dialog, rather than showing a lone tab in a bar that stretches across the whole dialog, wasting quite a bit of space. The window title already tells me this is the Trace Bitmap dialog; I don't think I need that information on a tab as well. And with this particular dialog, the first tab within the dialog also bears the same name, making the dialog tab seem doubly redundant, and potentially confusing.

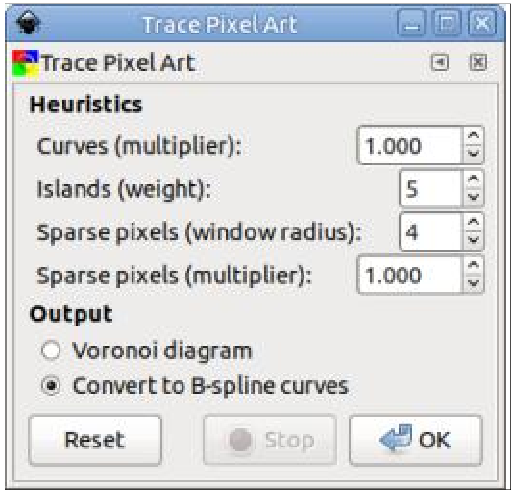
Moving on to the real content of the dialog, you can see that the tabs along the top have been completely replaced. The old "Mode" tab is now "Trace bitmap", and within that section are two large buttons for switching between the Single scan and Multiple scans modes. Gone are the radio buttons for selecting the tracing method, replaced instead with a single pop-up which greatly reduces the visual clutter. The parameters below change according to the selection in this pop-up, avoiding any confusion about which fields apply to which methods.

Finally, the global parameters that formerly lived in the "Options" tab have been moved to a common section at the bottom of this dialog, making them far more obvious. This move also makes it clear that these options apply to the "Trace bitmap" modes, but not to the "Pixel art" mode, which has its own tab entirely.

Aside from the UI changes, the operation of this part of the dialog remains unchanged from previous versions – apart from the addition of some new tracing modes that I will look at next month. For details on how to use the older modes, therefore, parts 19 and 20 of this series still apply.

Trace Pixel Art

The "Pixel art" mode isn't actually a new thing – it first appeared in version 0.91 – but I haven't covered it in this series before. Back in the 0.9x days, it had its own dialog, opened via an entry in the Path menu.



Heuristics

Curves (multiplier): 1.000
Islands (weight): 5
Sparse pixels (window radius): 4
Sparse pixels (multiplier): 1.000

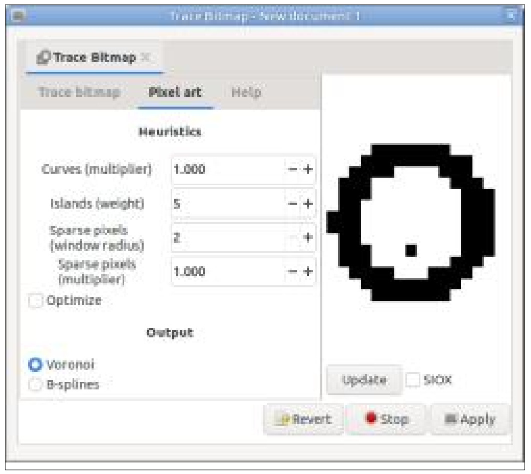
Output

○ Voronoi diagram
● Convert to B-spline curves

In 1.0, the UI is essentially unchanged, save for an additional checkbox labelled "Optimise". But it's now accessed via the "Pixel art" tab of the Trace Bitmap dialog, rather than being a wholly separate dialog of its own. This means it also shares the image preview pane of the Trace Bitmap dialog, although it's largely useless here, showing only a black-and-white picture which bears little resemblance to the traced image you'll probably end up with.

As the name suggests, this mode is intended for tracing pixel-based graphics, such as icons, emojis, or other small bitmap images. When scaling up a small image, you will often find that the interpolation algorithms that are commonly used can result in a blurry appearance, as intermediate colors are used for the new pixels that need to be created, losing any crisp transitions that are an essential part of the design. The Pixel Art tracer attempts to produce vector elements that will reproduce the original image, but which can be scaled without introducing a blurred or softened result. How effective it is at this will depend largely on the source image, and the result you're looking for.

As an example, I decided to trace the 16px × 16px image used as a "favicon" on the Full Circle Magazine website. It's a tiny representation of the orange circle with white swirls that forms the logo of this magazine. Let's see what the preview pane in the dialog makes of it.



Heuristics

Curves (multiplier) 1.000
Islands (weight) 5
Sparse pixels (window radius) 2
Sparse pixels (multiplier) 1.000
☐ Optimize

Output

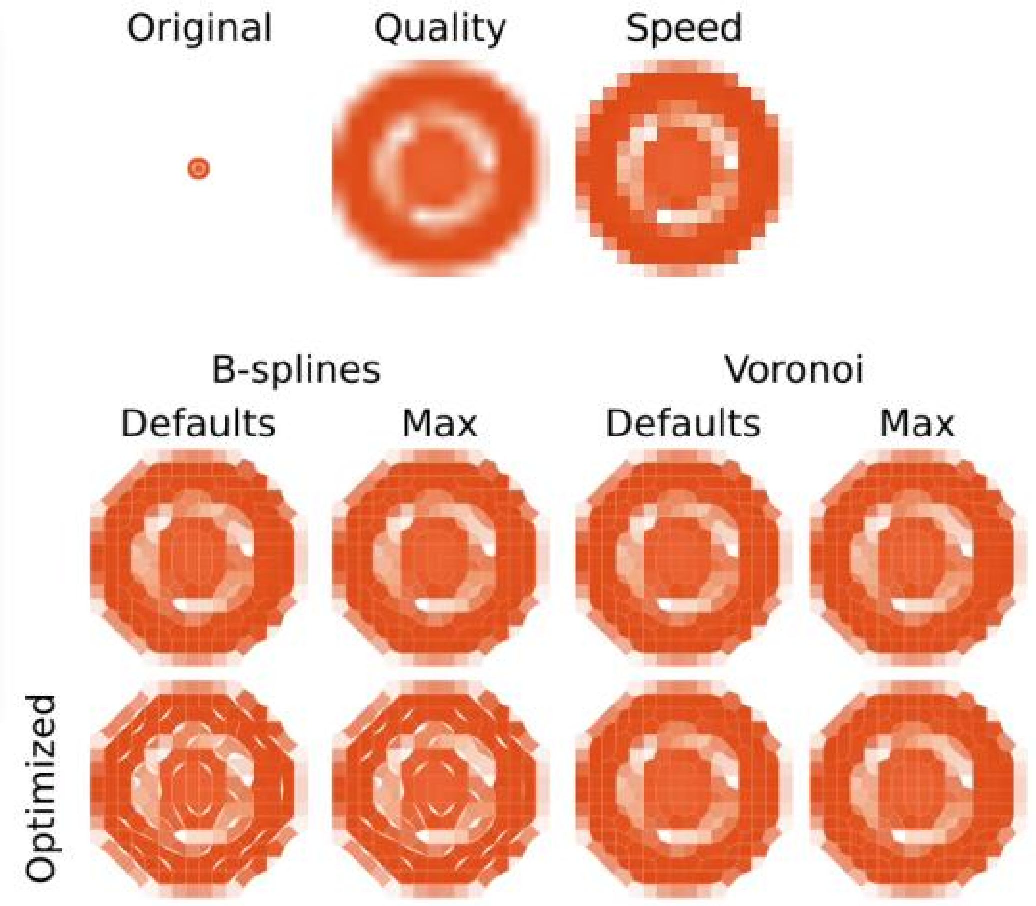
● Voronoi
○ B-splines

The various spinboxes in the Heuristics section are a mystery to me. They each have tooltips… which do nothing to clarify their purpose. They refer to parameters in the tracing algorithm, but the original research paper that is linked from the Inkscape 0.91 release notes is no longer available, nor have I been able to find it elsewhere. When faced with this situation, I resort to a tried-and-trusted technique to try to work out what effect they have: I first traced images using the defaults, then I cranked the values up to their maximum levels (10, 20, 8 and 10), and traced the same images. The results, in every case that I tried, were so close as to be irrelevant.

The "optimize" checkbox at least has an understandable tooltip. It claims that enabling it will try to optimize paths by joining adjacent Bézier curve segments. With my test images, however, it served to only cut off the corners of some paths, resulting in holes within the design.

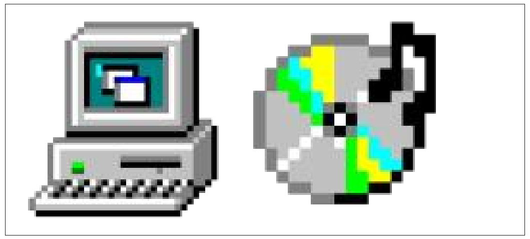
In the Output section, you can decide between producing shapes that have entirely straight edges (Voronoi), or which may include curves (B-splines). When tracing very small images, this is a matter of personal taste. With larger images – even those of only a few hundred pixels – the Voronoi output can take a lot longer, and may not produce the result you expect.

Let's look at the result of playing with some of these options. In this screenshot, the original 16px square favicon is at the top-left. To the right of it you can see how it looks when scaled directly in Inkscape – first using the "optimize for quality" mode, and secondly with the "optimize for speed" mode (remember, the selected mode can be changed in the Object Properties dialog). In both cases, however, the end result is still a bitmap image, so its individual parts can't be manipulated as vectors.

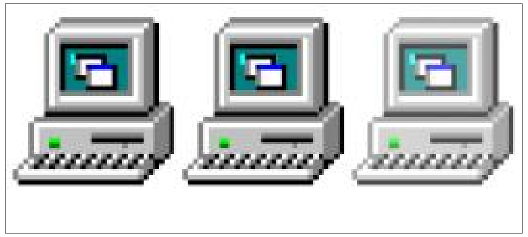


The lower part of the image shows the result of the tracing algorithm, with the B-spline versions on the left, and the Voronoi versions on the right. In each case, there's a column showing the results when the parameters are left at their defaults, and one showing the results when all the parameters are set to their maximum values. The bottom row shows the result of enabling the Optimize checkbox. In most cases I suspect that creating an un-optimized B-spline using the default values will be good enough.

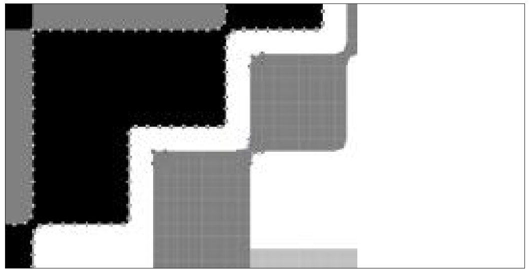
The close similarity between all the results may simply be because my choice of test images doesn't really do justice to the algorithm, but it was actually rather difficult to find genuinely small pixel images. Many examples of "pixel art" that you might find online have the appearance of being pixellated, while actually being made up of a much larger collection of pixels. For example, here's one I tried from Wikimedia Commons:



A count of the obvious “pixels” in the computer image on the left suggests that part is only 31px wide. Except that it’s actually 248 pixels, with each apparent pixel in the image actually consisting of an 8×8 square of pixels. Nevertheless, we should be able to trace that with Inkscape, right? Here’s the original image on the left, followed by B-spline and Voronoi versions after tracing.



In both cases, the tracing took a lot longer than with the favicon, with the Voronoi version taking the longest by a clear margin. And why does it look so faint? A close-up view of the result, with some paths selected, shows us a bit more of what’s happening.



The top-left shows shapes from the B-spline tracing. The black pixels that form a diagonal edge in the original image have been turned into a single path, with far more nodes than necessary, as a result of the 8×8 pixel blocks. The resultant trace consists of a group of 69 individual objects, each of them an overly defined path.

The Voronoi mode fares even worse. Here the individual pixels of the 8×8 blocks have been converted into separate paths, resulting in a group of 63,000 objects! That’s quite a result from what was clearly designed as a 32px × 32px icon (1,024 pixels in total). It also explains why it looks faint: the large number of closely spaced objects plays havoc with Inkscape’s anti-aliasing algorithm, as it tries to resolve the individual edges of the shapes.

Trying to convert something as small as an icon into a vector graphic will always be a trade-off that will work better for some images than others. The lesson from this is that the pixel art mode is not really designed for “pixel-style art”, but specifically for very low resolution pixel images. If your source material is more than about 48px in either dimension, you probably need to consider scaling it down in a bitmap editor before throwing it at this algorithm.

As great as it is to have advanced algorithms in Inkscape, what I would really like to see in terms of tracing pixel art is a mode that simply takes each individual pixel and converts it to an Inkscape square, with an option to merge adjacent squares of the same color into a path. You can kind-of achieve this with the Tiled Clones dialog, if you know what you’re doing, but adding that as an option to the Trace Bitmap dialog would likely cover most people’s pixel tracing requirements, without the need for opaque parameters and complex results.

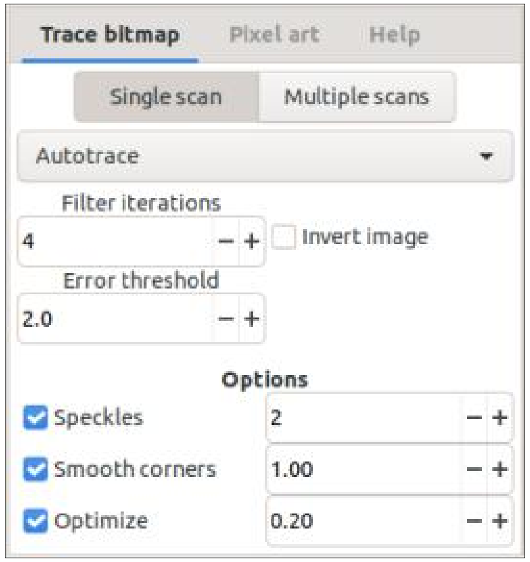
Mark uses Inkscape to create three webcomics, 'The Greys', 'Monsters, Inked' and 'Elvie', which can all be found at http://www.peppertop.com/

Inkscape - Part 111

This month, I'll be concluding my look at the Trace Bitmap dialog by examining the new tracing modes that have been introduced with Inkscape v1.0, including one that has been much requested on the forum over the years.

Autotrace

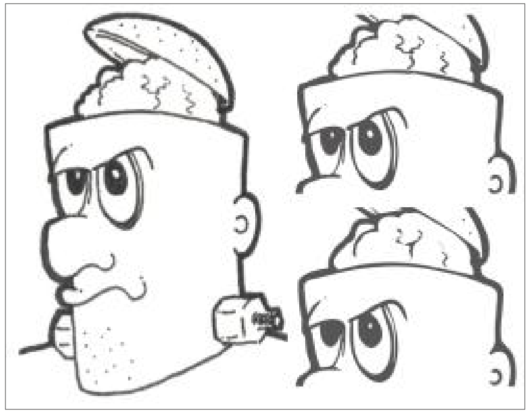
Inkscape has long offered tracing of bitmaps using the Potrace library. While this does a fine job of vectorising raster graphics, it's not the only open source tool that performs this trick. Autotrace is a command-line tool of similar vintage, which has now been integrated into the Trace Bitmap dialog. It's available in the mode selection pop-up, in both single-scan and multiple-scan varieties. The latter is labelled as "Autotrace (slower)", suggesting that it probably shouldn't be your first choice for a multiple-scan conversion – a suggestion that I'll come back to later. For now, however, we'll start by looking at the single-scan variant.



Autotrace
Filter iterations: 4
Invert image
Error threshold: 2.0
Options
Speckles: 2
Smooth corners: 1.00
Optimize: 0.20

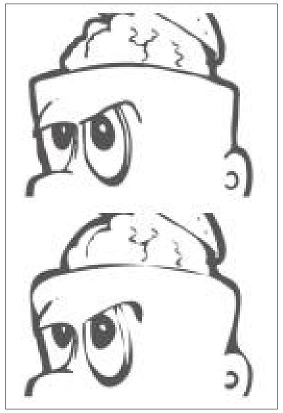
Once selected, it offers two spinboxes and an "Invert image" checkbox. The latter, as the name suggests, inverts the image colors before tracing, making it easier to trace a light-on-dark design without having to manually process it first. The effect of the two spinboxes is somewhat harder to guess at.

The scant documentation for Autotrace – basically an online copy of the man page for the command-line tool – describes the "Filter iterations" option thus: Smooth the curve the specified number of times prior to fitting. Without knowing what is meant by "fitting" in this algorithm, it's hard to infer what this actually does. Trial-and-error suggests that it reduces the fidelity of the trace somewhat, smoothing out the generated vectors and tending to result in a trace with fewer nodes. Here's what the most extreme values look like when applied to dear old Frankie. The original bitmap is on the left, with clipped versions of the traced head to the right. For the top trace, the filter iterations was set to 1; for the bottom it was set to 20.



Note that the higher iterations have reduced or wiped out the whites of the eyes, while the veins on the brain are a mere shadow of the original. The shape of the ear and of the brain's outline are also significantly smoother. The total node count for the traced head is 485 for the top image and 410 for the bottom one.

The second spinbox, for setting the "Error threshold", is described like this in the man page: Subdivide fitted curves that are offset by a number of pixels exceeding the specified real number. This one I can at least have a guess at. The process of tracing a bitmap consists of generating paths that approximate the shape of the original pixels. The paths will most likely be a close match in some areas, but not as accurate in others. This parameter allows you to set the threshold before which a path segment will be considered too far away, and will be subdivided into two paths to make it easier to adjust them to fit. Setting a small value here allows only slight deviation from the pixel positions, at the expense of a lot

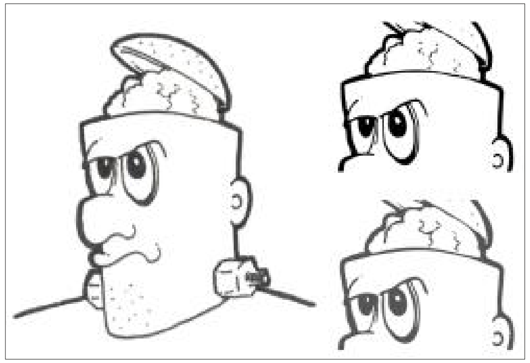


more subdivisions and, therefore, more nodes in the result. Let's take a look at how Frankie fares with values of 1.0 (top) and 10.0 (bottom).

It's pretty clear that the higher value results in a trace that is so smoothed out as to lose many of the original shapes completely. The top image, where the paths were much more heavily subdivided, consists of 587 nodes; the bottom one has only 327.

As is often my advice in this column, I suggest most users should at least start with the default values for both spinboxes, and start tweaking them only if you need to improve the fidelity of the trace, or want to take the counter-approach of reducing the number of nodes. Even in the latter case, I would probably be more inclined to trace with the defaults and then use Path > Simplify afterwards.

Perhaps the biggest question is how the Autotrace results compare with the Potrace equivalents. Here's another pair of traced Frankies created using the default settings: The Potrace-based "Brightness cutoff" at the top; the Autotrace version at the bottom. Again, the full head on the left is the original raster image.

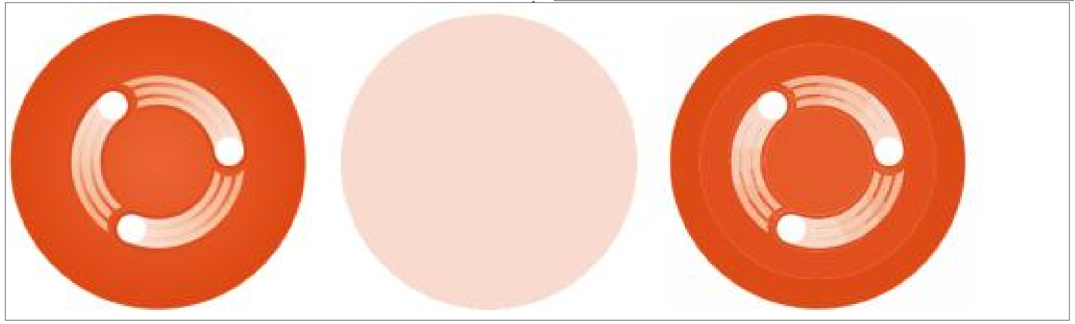


The first thing to note is that the Autotrace version has maintained the grey color of the original image – though that's such a trivial thing to change that it shouldn't be used as a reason to select one over the other. The Potrace result is a lot crisper, with the paths more accurately maintaining the sharp corners of the head, and thinner lines of the eyebrows. This accuracy is reflected in the node count: 1090 for Potrace but only 440 for Autotrace.

But it's not that clearcut. The extremely thin lines on the brain are actually better preserved by the Autotrace algorithm. On the whole, I think the old Potrace code works best, at least in this case. But I also wouldn't rule out creating a hybrid result by using node editing or Boolean operations to paste together the best parts from each result.

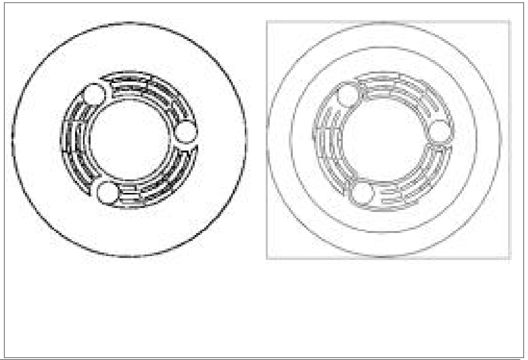
Autotrace (multiple scans)

What about using the "Autotrace (slower)" mode for scanning color images? My advice is to avoid it completely and stick to the Potrace-based modes. I tried scanning the same images that I used for part 19 of this series: the Full Circle Magazine logo, and a Wikimedia Commons copy of "La Giaconda" (The Mona Lisa). In both cases I used the default settings. The logo, which takes less than a second to trace with Potrace, took several minutes to complete. With such an amount of effort involved you might expect something impressive, but this is what the result looks like (original bitmap on the left, Autotrace in the middle, Potrace on the right):



To you and I it may appear as though Autotrace spent several minutes producing a salmon-colored circle. But no: what you're actually looking at is a group of 4180 objects! For comparison the Potrace version contains 8 objects – one for each color set via the "Scans" spinbox.

Switching to the outline view does suggest that the shapes have been traced, and are hidden somewhere in the salmon fillet before us, but the thicker outlines definitely hint at complex paths compared with the simplicity of the Potrace version.



And what of La Giaconda? After many minutes of processing my memory, swap and CPU were all maxed out, then Inkscape disappeared off my screen entirely. There was no appearance of the usual crash dialog I see when it dies, leading me to suspect that its demise was perhaps the fault of the Linux kernel killing it due to lack of available resources.

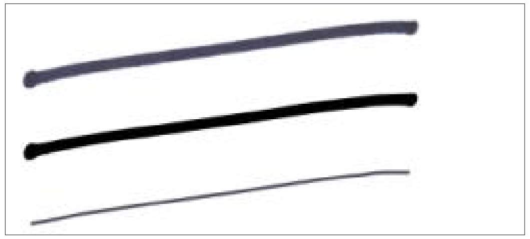
Even with a “successful” trace, the sheer number of objects created is practically unmanageable. There may, perhaps, be some image types for which this mode offers an advantage, but I would try it only if the Potrace methods aren’t yielding acceptable results – and make sure to save your file first!

Center Line Trace

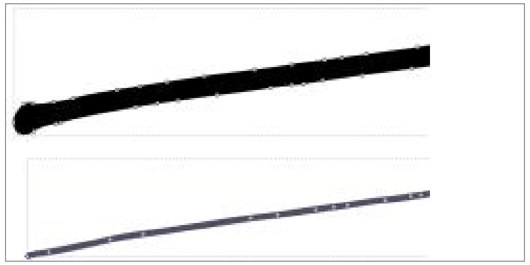
If Autotrace offers little or no improvement over Potrace, and in some cases is far too resource hungry, why bother adding it to Inkscape at all? The reason is that it offers one type of frequently requested tracing mode that Potrace does not – center line tracing. In fact the menu entries described previously are there only as a side-effect of including this mode. After all, if you’re adding the library anyway, why not also expose the standard tracing mode as well, to give your users more options.

Center line tracing is really applicable to only line art in which the shapes are made up of individual pen or pencil strokes. Using other tracing modes, each stroke is converted into a closed, filled path that reflects the thickness and shape of the original artwork. With this new mode, however, the tracing algorithm attempts to determine a single path that traces out a line following the middle of the original stroke.

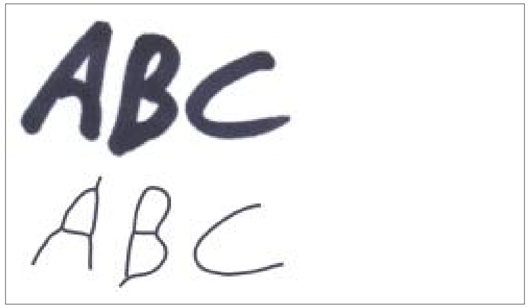
For the most simple real-world example, consider a single pen stroke on paper, scanned and imported into Inkscape. The top line in this image is the original scanned raster graphic. The second shows the result of a normal trace – note that the bulbous ends of the line are reproduced in this mode. The third line is the result of a center line trace – no thickening of the line at the ends of this version.



The real difference becomes clear when we take a closer look at the nodes used to make up the two traced paths. The first is a closed, filled path, so you can see that the nodes make up the outer shape of the stroke. The center line trace, on the other hand, results in an open path made up of a simple line of nodes: any suggestion of line thickness is purely down to the value set for the stroke width.

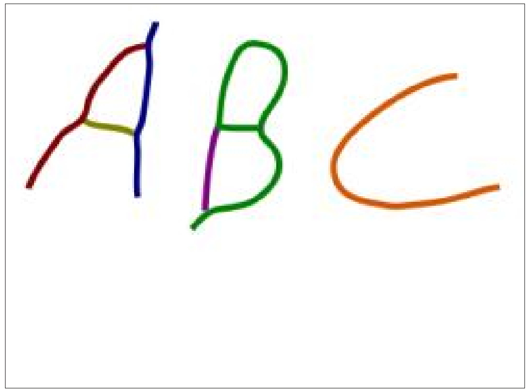


What happens when you try this mode with a more complex example? How about a few handwritten letters?



As you can see, the trace doesn’t really reflect the shapes and writing style of the original scanned image. The fairly straight left leg of the A becomes kinked in the middle, as the algorithm struggles to work out where the center line actually is. The sharply angled line where the top and bottom bowls of the B meet is lost entirely, and replaced with a horizontal crossbar.

You may also have realised that such shapes can’t be made up of a single path segment. In this instance we’ve ended up with a single complex path consisting of all the different segments that make up the letters combined into one object. Path > Break Apart allows us to reduce the complex path down to its constituent parts, which we can then give different colors to demonstrate the paths that the algorithm settled on.



The A, not unreasonably, is made up of three separate paths. But the complexity of the B is captured in only two paths: one complex curving line that encompasses most of the shape of the letter, and a small straight segment to fill in the remaining gap. Wrangling such shapes into something more befitting the original outlines could quickly become tedious on larger projects.

You may think I'm being unfair on the algorithm here. My scanned text was from a thick Sharpie, rather than the thin strokes of a pencil or ballpoint pen. But based on my testing, you'll likely face similar issues, even when starting with thinner lines in the source material.

That's not to say that the new mode is useless or unwelcome. For many images it will prove to be far more effective than the existing tracing methods, especially if you're interested in only the core shapes of the elements rather than the exact details of the stroke outlines. Just remember that it's working only with pixels, and has no concept of the order in which lines were laid down, or the difference between two lines that meet at an angle compared with a single line that has a sharp corner in it.

As is so often the case with the Trace Bitmap dialog, I can only recommend that you give it a try on your image, but don't expect miracles. Even if the results aren't perfect, it may save you some manual tracing time on part of your design, or at least give you a starting framework to build upon.

Next month, we'll take a look at the new "Selectors and CSS" dialog, which promises to make Inkscape a little more useful as a web development tool.

Links

Potrace: http://potrace.sourceforge.net

Autotrace: http://autotrace.sourceforge.net/ https://github.com/autotrace/autotrace

Autotrace man page: https://linux.die.net/man/1/autotrace

"Frankie" and other images: http://www.peppertop.com/fc/

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 112

I'm going to start this month with yet another complaint about the state of font support in web browsers. Once again, a single browser vendor has effectively scuppered a format that is supported by their peers, leaving us users and developers without a practical alternative.

I've written in the past about how Firefox's lack of support for SVG fonts effectively killed the format. This was a real blow: the ability for a web developer to dynamically modify fonts on-the-fly using JavaScript could have been truly revolutionary. More recently, it looked like SVG in OpenType would provide some new capabilities in terms of color fonts and bitmap support, but Google has closed the relevant issue on the Chromium bug tracker (#306078) with a status of WONTFIX, indicating that they won't be adding support for this format to the browser. Given Chrome's dominant market share, this effectively neuters some of the new font support in Inkscape 1.0 that I covered in part 108 of this series.

Variable fonts, as described in that article, already work and will continue to do so. It's the SVG in OpenType format that is now effectively dead, despite being supported in Firefox, Safari and even older versions of Edge. My own experiments suggested some level of support for the fonts I was using (e.g. the Gilbert Color font was rendered in Chrome, but only as a monochrome font). I now suspect that was due to the fonts themselves providing non-SVG fallback content, rather than Chrome providing any support at all for this format.

What's most disheartening about this development is that Google's primary reason for not supporting SVG in OpenType appears to be that they are developing their own format for color fonts (again, embedded into OpenType). If you were in any doubt that Google might use their market dominance to force the direction of web standards, perhaps this will make you realise that a single vendor having so much control is not really in the best interest of the users. The need for multiple, independent rendering engines is the reason I continue to use Firefox as my daily driver. The battle against IE may have been won, but the war for a truly open and independent web still wages.

Time will tell whether or not Google's new format will win out, or font foundries will just embed both the Google and the SVG versions of color glyphs into ever-more-bloated font files. For now, however, despite Inkscape's support for color fonts – and its partial support for bitmap fonts – you might want to think twice about using these in your projects.

CSS in SVG

The main topic for this month was supposed to be the Selectors and CSS dialog that was added experimentally in Inkscape v1.0, and then promoted to non-experimental in v1.0.1. But in order to understand what this dialog does, it's essential to first have a decent grounding in how CSS works in a stand-alone SVG file. This month's article will put in that ground work – if you're already a CSS aficionado, then you may want to skip this one and come back next month for the details of the dialog.

Let's begin with some history. The SVG format has always been a little confused about its identity. It was created during the great XML uprising of the late 1990s, when the World Wide Web Consortium (W3C) were pushing for a world in which XML formed a common basis for many file formats, allowing for tools and workflows that could easily combine and convert different types of data. SVG was a stand-alone vector format, not yet directly integrated into any web browsers, but the intention was clear that it should conform to, and work with, existing web standards. This left it with something of a dichotomy: on the one hand, it needed a way to directly define fonts, colors and dimensions in order to be used as a generic vector format with no dependency on a browser engine; on the other hand, it had to work well with the growing power of CSS to define styles for those users who did want to work in a more web-focused way.

And so we have ended up with a profusion, and confusion, of ways to style SVG content. First there are the classic "presentation attributes" from the SVG format. These are attributes that can be applied directly to SVG elements in order to style them individually. For example, a red rectangle with a thick black border might be defined like this:

```
<rect x="20" y="20"
  width="100" height="100"
  fill="red"
  stroke="black"
  stroke-width="10"
/>
```

This approach requires only the consuming program to understand XML and SVG, and doesn't impose the need for a complete CSS parser (though things like color values and most units are still taken from the CSS spec).

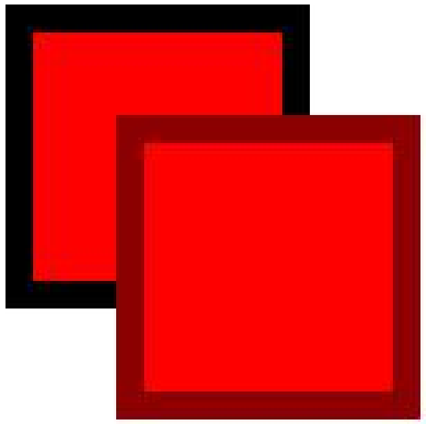
In the early days of HTML, presentation attributes were common there, too. Veteran web developers might remember the "border", "color" and "bgcolor" attributes, amongst others, but those capabilities were rapidly subsumed by the growing scope of CSS. In order to override CSS styles on a per-element basis, the relevant CSS rules had to be combined into a single "style" attribute. This method also works with SVG, meaning that our thick-stroked red rectangle could also be written like this:

```
<rect x="20" y="20"
  width="100" height="100"
  style="fill: red; stroke:
black; stroke-width: 10;"
/>
```

If CSS only offered a way to put all the style information into a single attribute, it wouldn't be terribly useful. But as well as setting styles on each element individually, CSS also offers a mechanism for applying styles across an entire document (or even across multiple documents, but that doesn't work with Inkscape). It makes it trivial to give all your rectangles the same fill color; or to divide your objects into common classes that share similar styles; or to add override rules that make, for example, the third rectangle in a group have a different stroke width to the others.

These tricks are achieved by moving the style information out of the style attributes, and into a common stylesheet in a <style> element. The style attribute still applies, and will override the stylesheet, as shown in this abridged example where both the <rect> elements share the same fill color, stroke width, and their width and height, but have a different stroke color:

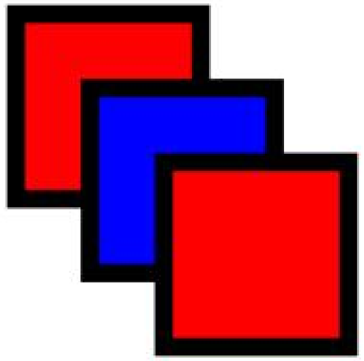
```
<style>
  rect {
    fill: red;
    stroke-width: 10;
    width: 100;
    height: 100;
  }
</style>

<rect x="20" y="20"
  style="stroke: black;"
/>
<rect x="60" y="60"
  style="stroke: darkred;"
/>
```



Notice that the syntax of the stylesheet is quite different to that of the SVG content. I won't go into details about how to write CSS here, but suffice to say that it's made up of multiple rules (delimited by curly braces: { and }), with each rule starting with a "selector" that determines what elements the rule will apply to. In this case, the selector is just the string "rect" which will make this rule apply to all the <rect> elements in the document.

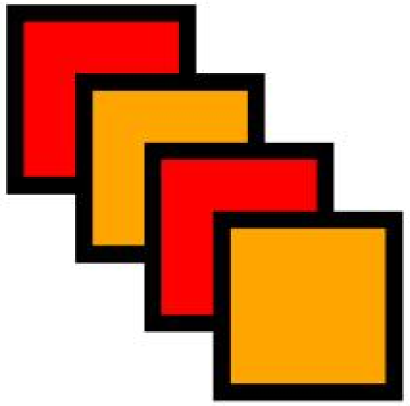
We can target a specific element by ensuring it has an "id" attribute in the SVG content (which Inkscape does by default), then using that ID, prefixed with a hash character (#), as our selector. In this next example, we have three rectangles which all share a common base style, but one of them has its fill color overridden in this way.

```
<style>
  rect {
    fill: red;
    stroke: black;
    stroke-width: 10;
    width: 100;
    height: 100;
  }
  #r2 {
    fill: blue;
  }
</style>

<rect
  id="r1"
  x="20" y="20"
/>
<rect
  id="r2"
  x="60" y="60"
/>
<rect
  id="r3"
  x="100" y="100"
/>
```



We can also target multiple items by giving them the same value in a “class” attribute and using it preceded by a dot (.) as the CSS selector. This example has four rectangles, split into two different classes. Note that an element can have more than one class, separated by spaces, which allows for a huge amount of flexibility in combining objects into overlapping sets.

```
<style>
  rect {
    stroke: black;
    stroke-width: 10;
    width: 100;
    height: 100;
  }

  .important {
    fill: red;
  }

  .warning {
    fill: orange;
  }
</style>

<rect
  class="important"
  x="20" y="20"
/>
<rect
  class="warning"
  x="60" y="60"
/>
<rect
  class="important high"
  x="100" y="100"
/>
<rect
  class="p3 warning low"
  x="140" y="140"
/>
```



CSS in web browsers also provides attribute selectors. These give the ability to select elements based on the presence, and optionally the value, of specific attributes on the elements. This would be particularly useful when dealing with some of Inkscape’s native objects, some of which are implemented as <path> elements, with a custom attribute to tell Inkscape which sort of native objects they are. Similarly, groups and layers are both implemented as SVG <g> elements, with an Inkscape-specific attribute indicating which ones are layers, so this would offer a means to distinguish between them in CSS selectors. Unfortunately it appears that the Selectors and CSS dialog doesn’t recognise the syntax for attribute selectors, so I shan’t consider it further here.

These basic selectors can be combined to further refine the elements that will match. Using “rect.important”, for example, would only match <rect> elements with a class of “important”. It would not match <rect class="warning"> or <path class="important">. In a similar manner, “#r1.high.important” would only match the element with an id of “r1” if it also had both the “important” and “high” classes assigned.

The browser provides some classes for free, in the form of “pseudo-classes” which allow you to create selectors that target things that the browser calculates at runtime. These may change dynamically, especially if your page uses JavaScript to manipulate the document. Most of them aren’t applicable to the non-dynamic world of Inkscape, but some relate to the order in which elements appear in the document model, and do work in a limited manner in the Selectors and CSS dialog. For example, a selector of “rect:first-child” would select any <rect> element that is the first child of its parent (e.g. the first element inside a group). Inkscape doesn’t quite allow this syntax, though, but there is a workaround. I’ll take a deeper dive into pseudo-elements and how to work with them in Inkscape as we progress with the Selectors and CSS dialog.

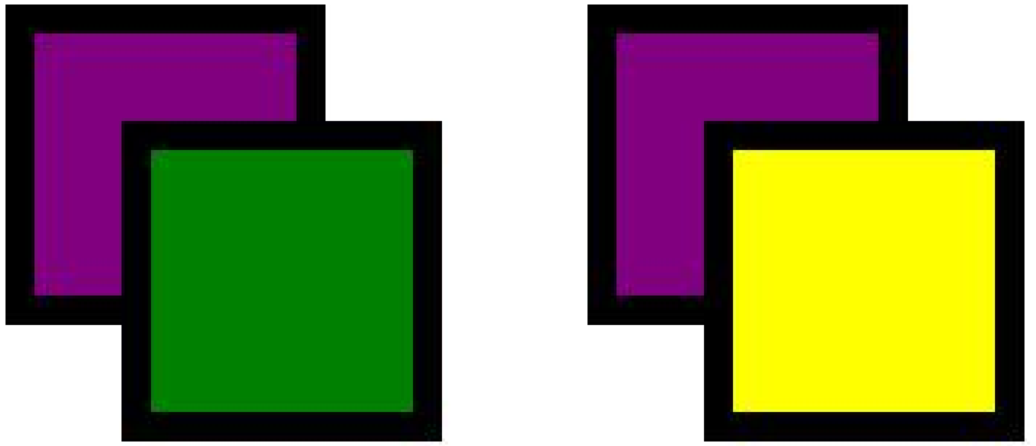
Selectors can be combined in different ways to match more elements, or to further refine the match based on the hierarchical structure of the SVG document. “#id1, #id2, .important” would match the two elements with the specified IDs, but also any element with the “important” class. “text + path”, meanwhile, would match any <path> element that is an immediate sibling of a <text> element. That one can be useful for styling any underlines you’ve drawn with your text elements, for example.

A simple space character creates a rule that matches if the second element is a descendent (possibly even a deeply nested descendent) of the first. E.g. “g.primary rect” matches a <rect> that is somewhere inside a group with a class of “primary”. On the other hand, “g.primary > rect” would only match the <rect> if it’s an immediate child of the primary group, not if it’s nested further inside other groups.

With the information from the last few paragraphs, see if you can make sense of this example file:

```
<style>
  rect {
    stroke: black;
    stroke-width: 10;
    width: 100;
    height: 100;
  }

  g.primary rect {
    fill: yellow;
  }

  g.primary > rect {
    fill: green;
  }

  g > rect:first-child {
    fill: purple;
  }
</style>

<g class="primary">
  <rect x="20" y="20" />
  <rect x="60" y="60" />

  <g
transform="translate(200)">
    <rect x="20" y="20" />
    <rect x="60" y="60" />
  </g>
</g>
```

There’s even more to CSS than the rules I’ve laid out here, but these are the ones that are most useful or relevant when working with the Selectors and CSS dialog. Even so, it’s a lot to take on board if you’re not already familiar with CSS. Next month, it will hopefully start to make a bit more sense as we begin our look at the new dialog.

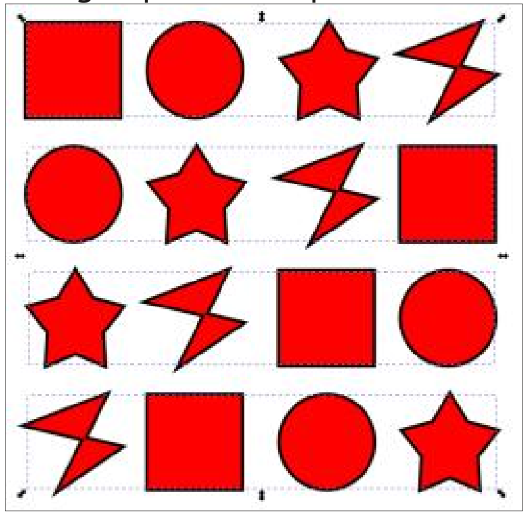


Mark uses Inkscape to create three webcomics, 'The Greys', 'Monsters, Inked' and 'Elvie', which can all be found at http://www.peppertop.com/

Inkscape - Part 112

Last time, I indulged in a whistle-stop tour of CSS support in SVG files, in preparation for this month's topic: the Selectors and CSS dialog. As I noted previously, this dialog first appeared as a hidden option in Inkscape version 1.0, and was then officially exposed in version 1.0.1. It can be opened using the Ctrl-Shift-Q keyboard shortcut (even on v1.0), with later releases also exposing it via the Object menu. In my experience, it's still a little buggy, even in version 1.1, and can lead to full application crashes, so please make sure to save regularly if you decide to use it.

To describe the operation of this dialog requires a sample document



on which to work – preferably one with a mixture of elements to target via CSS selectors. For the purposes of this article, therefore, I've created a collection of sixteen elements (four squares, four circles, four stars, and four Bézier paths), arranged as four rows, with the content of each row being grouped.

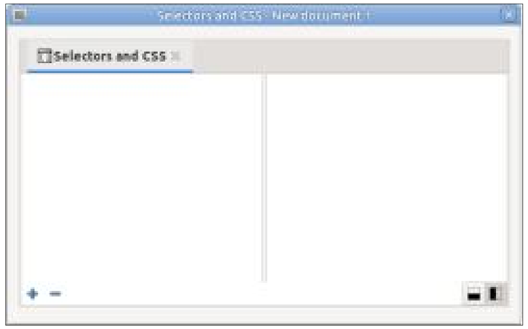
The style of each object is the same, and that information is stored in the element's "style" property, as Inkscape does by default. Looking at the first row, therefore, the structure of the SVG content boils down to something like that shown top right.

As usual, I've omitted a lot of the SVG content for clarity, including additional style properties and the namespaces of most of the elements and attributes. Note the first path, however, where I have specifically included the sodipodi:type attribute which tells Inkscape that this is a star primitive. The Bézier path, on the other hand, does not have such an attribute, as that is just a standard SVG path element

```
<svg>
  <g>
    <rect id="rect31"
      style="fill:#ff0000;stroke:#000000;…" />
    <circle id="circle1145"
      style="fill:#ff0000;stroke:#000000;…" />
    <path id="path1243" sodipodi:type="star"
      style="fill:#ff0000;stroke:#000000;…" />
    <path id="path1253"
      style="fill:#ff0000;stroke:#000000;…" />
  </g>
  …
```

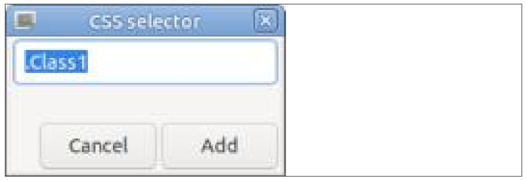
and doesn't require additional hints for Inkscape to understand how to work with it. The IDs are just the arbitrary values assigned by Inkscape – in a real file you might want to edit them to be more descriptive or user-friendly.

Now it's time to open the Selectors and CSS dialog (hereafter referred to as the Selectors dialog, for brevity), via the Object > Selectors and CSS… menu entry.



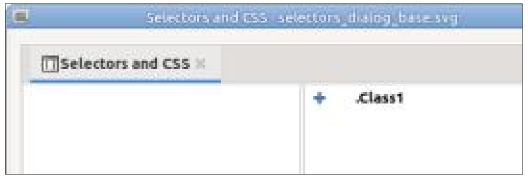
With nothing selected on the page, be prepared for something of a disappointment, as it hardly screams functionality at this point.

The buttons in the bottom-right corner can be used to switch between horizontal and vertical orientation – though in both cases, in my opinion, the panels are presented in the wrong order. The only other interactive elements at this stage are the plus and minus buttons in the bottom-right corner.



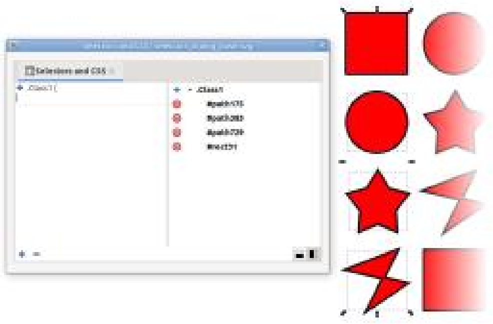
The latter does nothing right now (and should probably be disabled), leaving us with little to do but press the plus (+) button. This will open a small “CSS Selector” dialog, pre-filled with a class name (including the preceding dot, as required in CSS syntax).

Let’s just run with this for now, accepting the default selector value by clicking the Add button. Brace yourself for the overwhelming complexity of the change it makes to the main dialog.



When you’ve fully recovered from the excitement, you might want to take a moment to really appreciate what has happened: there’s now an entry in the right-hand panel labelled “.Class1”, and with another plus button next to it! You can select the new row, but double-clicking won’t let you change the value – there’s no way to edit a selector once it’s been added, so if you make a mistake your only recourse is to delete it and start again. Speaking of deleting it, our minus button now has a purpose: with the new entry selected, clicking the button will delete the entire entry. And what of our new entry-specific plus button? Clicking on it at this stage does absolutely nothing. There’s not even a tooltip to suggest what it is meant to do.

The purpose of this button is to add the class name to any selected SVG elements, causing them to match the CSS selector and appear in the dialog. Let’s look at a practical example by selecting the first element from each row. Remember that each row is a separate group, so we have to hold Ctrl+Shift to select an object within the group (Ctrl) and add to the existing selection (Shift). With those keys held, it’s a simple task to click on the first object in each row, giving us a selection of four items. Clicking on the plus button now will add the “Class1” class to each of the elements, and the Selectors dialog updates to show us the IDs of the elements that match the selector.

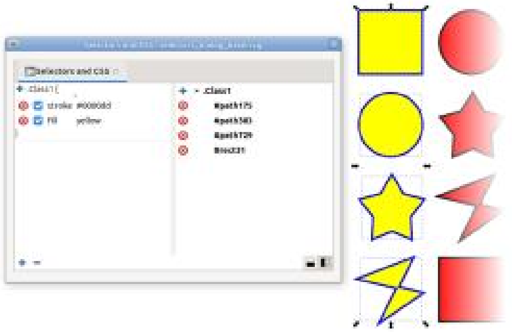


In the XML code those elements now look something like this:

```
<rect class="Class1" … />
```

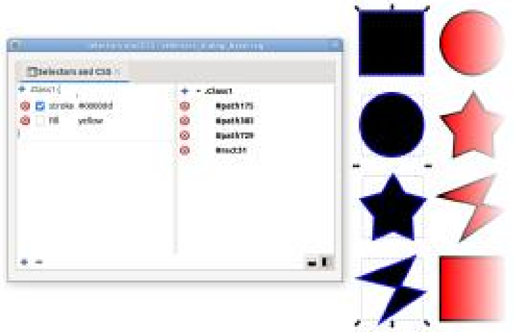
Note that Inkscape gives <circle> elements an ID that begins with “path”, so the three path and one rect IDs listed actually represent two paths (the star and the Bézier path), a circle and a square. It’s a little confusing at first, but just remember that an ID is just a unique label, and doesn’t have to relate to the type of object. Clicking on a single ID in the dialog will select the associated object on the canvas, so it’s pretty easy to keep track of which elements have which IDs. For now, we’ll assume the CSS selector itself (“.Class1”) is selected, per the previous image. You’ll notice that the left-hand pane has now gained a little content – and a plus button of its own.

The left pane shows the CSS properties that apply to the currently selected elements, in this case reflecting the stylesheet that Inkscape has created in the head of the document. The plus button lets us add more properties. The UI is a little clunky, but when you click the button, a new blank entry will be created with the property name field focused, ready for you to type into. Put in the property name, then press Enter or Tab to move to the value field, where you can type a suitable value for your property. Here, for example, is what happens to our selection when I set a couple of CSS colors, one for the stroke and another for the fill.



Each property has a button, which can be used to remove it completely, and a checkbox that can be used to toggle it on and off. But these don’t necessarily work the way you might expect. In this example, you might think that turning off (or deleting) the fill property would revert the objects back to their previous red fill, but that’s not the case at all. Instead the fill changes to black.

What’s going on here? Remember from last month that CSS properties in a “style” attribute override those from the stylesheet.



```
Selectors and CSS
.Class1 {
  stroke #0000dd
  fill   yellow
}
```

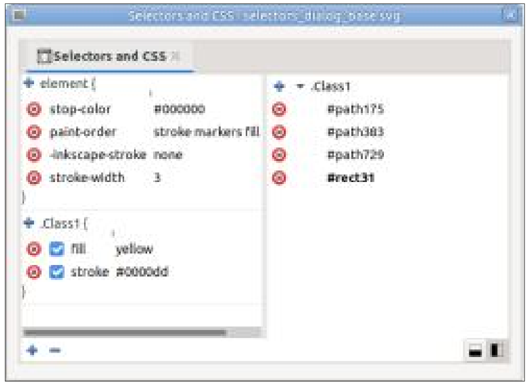
```
.Class1
  #path175
  #path383
  #path729
  #rect31
```

But our elements all had their fill and stroke set via style attributes when we created them, so how did the yellow fill (which comes from the stylesheet) ever work in the first place? The answer is that Inkscape automatically deletes conflicting properties from the style attribute when you start adding them via the Selectors dialog.

In this case, it means that the red fill from the style attribute was removed as soon as the yellow fill was added to the stylesheet. Turn that entry off, however, and Inkscape no longer has any idea what fill color to use. In that case the SVG spec is quite clear: the initial value for fill – the one that Inkscape falls back to – is black. The spec also defines the initial value for stroke to be "none" - so disabling or deleting the stroke property actually results in the whole stroke being removed, rather than a thick black stroke remaining.

Side effects like this mean that working with the Selectors dialog can be a little tricky if you're not careful, or don't have a thorough understanding of the SVG and CSS rules. It's easy to assume that you can add a fill color, then just delete it to revert your change if you want, but, as this example demonstrates, the act of adding a rule can make a change to your XML content in a way that won't be reverted when you remove the rule.

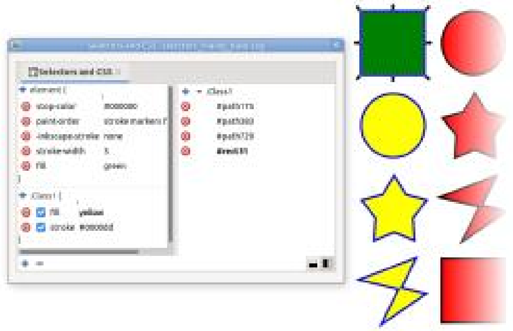
With both rules re-enabled, let's take a look at the left-hand pane when only a single ID entry is selected in the right pane.



```
Selectors and CSS - selectors_dialog_base.svg
Selectors and CSS
element {
  stop-color         #000000
  paint-order        stroke markers fill
  -inkscape-stroke   none
  stroke-width       3
}
.Class1 {
  fill    yellow
  stroke  #0000dd
}
```

```
.Class1
  #path175
  #path383
  #path729
  #rect31
```

The left pane now shows not only the rules that are applied due to the new class that was added, but also those that are defined in the style attribute on the object itself in a section labelled "element". You can edit existing values, or add new ones via this part of the dialog. If we set the fill color on the element itself, you'll also note that the value supplied by the class – the one from the stylesheet – acquires a strike-through to indicate that the property has been overridden by another one.



```
Selectors and CSS
element {
  stop-color         #000000
  paint-order        stroke markers f
  -inkscape-stroke   none
  stroke-width       3
  fill               green
}
.Class1 {
  fill    yellow
  stroke  #0000dd
}
```

```
.Class1
  #path175
  #path383
  #path729
  #rect31
```

Hopefully you can now see why I think these panes are in the wrong order: you first have to select an item in the right-hand pane in order to populate the content in the left one, whereas conventional left-to-right UI design (for an LTR based language) would suggest the pane in which you perform the action should be to the left of the one which reacts to that action.

As well as buttons to delete individual properties, you've no-doubt noticed that there are delete buttons next to the IDs in the right hand pane. Clicking one of these will remove the class name from that element's "class" property, removing it from the dialog, and removing any class rules that apply to it. Once again, however, any conflicting rules that were previously in the "style" attribute are long since gone, so you're likely to end up with a black fill and no stroke once more.

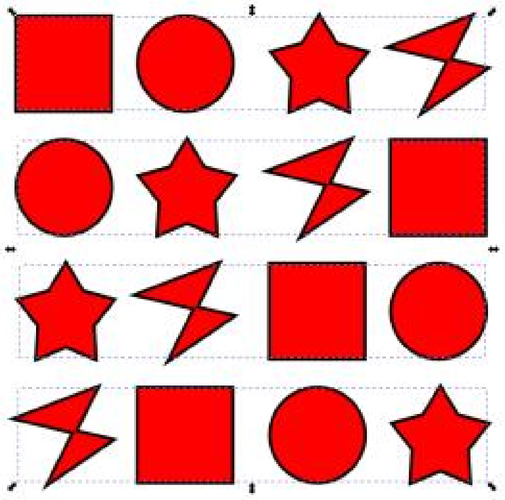
Next time, we'll add a few more items to our selection, look at how best to use this dialog as a replacement for the old "Selection Sets" feature, and begin to explore some of the more varied CSS selectors that can be used in this dialog – albeit with some pretty big caveats!

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

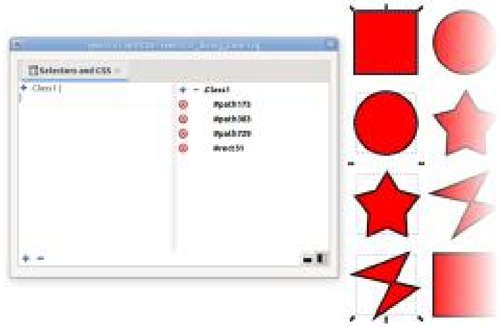
Written by Mark Crutch

Inkscape - Part 114

This month, we'll continue looking at the Selectors and CSS dialog (or just "Selectors dialog" for short). Now would be a good time to review the previous couple of instalments if you need to, before we dive in deeper. As with last month, we'll be working with this collection of shapes arranged as four groups of objects, one for each row.



Starting with a blank Selectors dialog, last month I showed you how to select the first item in each row (Ctrl-Shift-Click each one), and add a new class to them all, so that they appear in the dialog. Let's do the same thing this time:

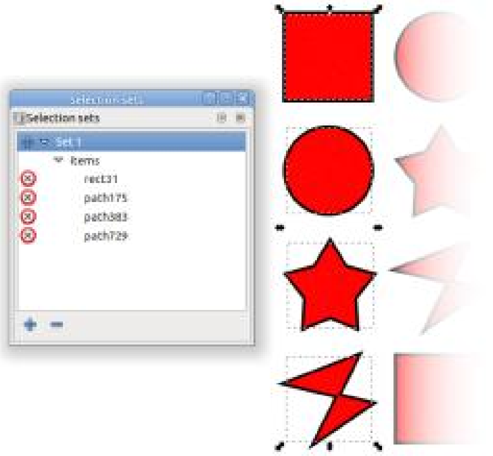


Selectors and CSS

Class1 {
}

.Class1
#path175
#path383
#path729
#rect31

Previously, we used this capability to add some override styling for those objects. But what if we don't do that? What if we just leave the styles alone? In that case, what we've created is a convenient way to re-select all those objects any time we need to: clicking on the ".Class1" entry in the right-hand pane is all that is required.



Selection sets

Selection sets

Set 1
Items
rect31
path175
path383
path729

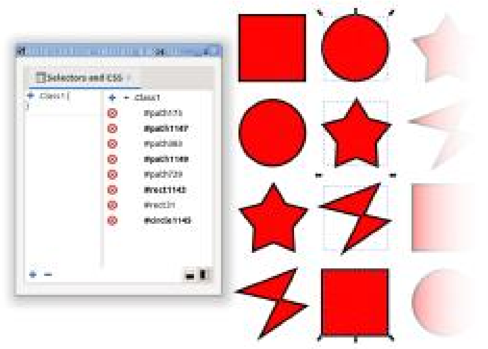
This, essentially, returns the capability of the Selection Sets dialog, which was introduced with version 0.92, but removed with version 1.0.

The Selection Sets dialog allowed you to name (and re-name) your sets, change the label for each object, and add or remove items from sets. The new Selectors dialog is slightly more limited. Double-clicking on the CSS selector won't let you edit it – so you're stuck with the class name you initially created. The listed items are now shown as IDs rather than labels (they include the "#" prefix that CSS uses for IDs, making this difference a little clearer). Double-clicking on an ID just selects it, it doesn't allow you to edit the label or the ID of the object in question.

The Selectors dialog updates in real-time to changes in the SVG content, however, so if you really want to change your class name, you can edit it in the Edit > XML Editor dialog. Just be aware, however, that editing the selector name in the stylesheet won't automatically apply that change to the class attributes in the individual elements – you will need to manually change all of those to suit, as well. In most cases, therefore, it's easier to delete the class entry from the Selectors dialog and re-create it.

You can still, however, add and remove things from the "set" – though in reality what you're actually doing is adding and removing the class from the object. We looked at removing items last time – you just need to click on the delete button to the left of each thing you want to remove. Doing this will also remove any style overrides, most likely returning you to a black fill with no stroke, but if you're using this dialog just for managing selection sets and haven't modified any styles, then this won't be a concern.

Adding new items to the set is also pretty straightforward. You simply have to select them on the canvas, then click the "+" button next to the CSS selector. In this example, I've selected the items from the second column and clicked the button, and you can clearly now see that the set contains eight items.

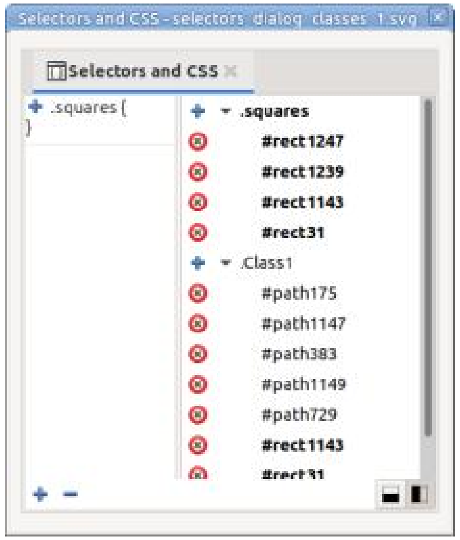


Clicking CSS selector (the ".Class1" entry on the right) will select all eight items on the canvas.

Using this technique, you can create new classes (i.e. selection sets) for any combination of items you want. Let's go through the steps to create a set that contains all the squares:

- De-select everything – by using Edit > Deselect, by pressing the Escape key, or simply by clicking on the canvas background.
- Click the "+" button at the bottom of the Selectors dialog. Unfortunately, you'll be presented with the same default class name (".Class1") every time you do this. It doesn't auto-increment, so you need to make sure you change it by hand to something unique. In this case, I'm going to create a class called "squares" by just typing the name into the prompt, preceded by a dot. You don't need to supply the leading dot to indicate a class – Inkscape will add that for you – but I recommend always including it, for reasons I'll go into later in this article.
- Select all the squares, by Ctrl-Shift-Clicking each of them.
- Click the "+" next to the ".squares" CSS selector line in order to add the "squares" class to each item, thus making them appear in the set.

If done correctly, the dialog should now be showing both the ".squares" class (with four items) and the ".Class1" class (with the eight items we added previously).



Selectors and CSS - selectors_dialog_classes_1.svg

Selectors and CSS

```
+ .squares {
}
```

```
+ ▾ .squares
      #rect1247
      #rect1239
      #rect1143
      #rect31
+ ▾ .Class1
      #path175
      #path1147
      #path383
      #path1149
      #path729
      #rect1143
      #rect31
```

The sections can be collapsed and expanded using the toggle triangles next to the class name, and selecting either of them will select the corresponding set of items on the page.

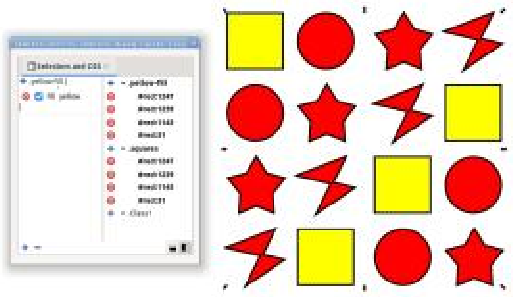
At any time you can select more items on the page and add them to the existing class using the plus button next to the relevant CSS class selector in the second pane. Or you can create another class entirely, using the plus button at the bottom of the dialog.

As mentioned, clicking on a class selector in the second pane will not only show you the relevant CSS properties on the left, but it will also select all the objects with that class on the page – a replacement for the Selection Sets dialog. You should use this capability carefully, however. I recommend treating "Selection Set" classes and "Styling" classes as two different things.

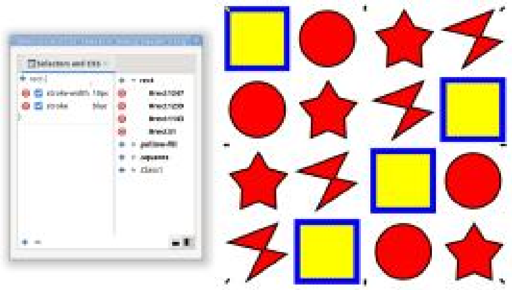
What do I mean by that? Last time, I demonstrated how you could use classes to change the style of the selected elements. But if you do that, it's very easy to end up with elements that have no styling applied at all, if you subsequently remove the class name from them.

So if you want to use this dialog to manage classes as though they were selection sets, it's best not to also apply any styles as part of those classes. I recommend using one class (or set of classes) for selections, with no style changes in the left-hand pane, and different classes for styling changes, even if it means you have two classes that actually select the same objects.

Here's that method in practice: the ".squares" class is as it was before, it just selects all the squares, but doesn't change their style. The ".yellow-fill" class contains exactly the same objects, but changes their fill color. Now if I want to also change the stars to have a yellow fill, I can add them to the latter class, without disrupting the "selection set" formed by the ".squares" class.



Up to now, I've looked at creating just class-based selectors in this dialog, but you can also use it to create other types of rules in your stylesheet. So far, we've added the "squares" and "yellow-fill" classes to all of the squares in the image, but because these are all the same type of SVG primitive, there's an easier way to select or style them all: the element selector. The squares are all <rect> elements in the SVG, which means they can be targeted with a selector that literally just contains the word "rect". Just create a new entry using the "+" button at the bottom of the dialog, but enter "rect" into the prompt, with no preceding dot. In this way, I've changed the stroke to a thick blue line on all of the squares, while still leaving them with the yellow fill from the previous class-based rule.



Inkscape did not automatically prefix the word "rect" with a dot, as it would have done when creating a class selector. This implies that the program must maintain a whitelist of element names which are allowed to pass through untouched. To avoid accidentally creating an element selector when you meant to create a class selector, therefore, I recommend always including the leading dot when entering a new class name. In this case, for example, typing ".rect" into the prompt does create a class selector as expected, whereas omitting the dot creates an element selector.

One advantage of element selectors is that they are dynamic in nature. Add another rectangle to the page and it will automatically be included in your "rect" selector. But be aware that inclusion in the selector may not be enough for it to behave the way you expect. In the example above, the "rect" selector defines a thick blue stroke, yet drawing a new rectangle doesn't produce that result.

The reason for this is that the new elements you draw will have their style properties set via their "style" attribute, as usual. As we found out previously, these will override those in the stylesheet, preventing the element rules from working. The solution is to remove the override rules from the "style" attribute – which can be done by selecting the element and then deleting the rules from the "element" section of the left pane in the Selectors dialog. Nevertheless, it's an additional few steps that may get overlooked, and can become tedious when dealing with many elements. Unless you've got a good reason to do otherwise, it's probably best to stick with class-based selectors for styling – at least you can then select all your new elements and add them to the class with a single click in the dialog.

Element selectors also exhibit some odd behaviour within the dialog. Each matching element has its ID shown below the selector, as you would expect, but they still have delete buttons next to them. Clicking one of those buttons does nothing: it doesn't delete the element itself (which would be the only practical way to remove them from the selector), so they appear to be completely redundant. Worse than that, clicking the selector itself ,then the "-" button at the bottom of the dialog, does not delete the selector. Removing an accidentally created element selector, or one that is no longer needed, requires a trip to the XML editor.

Element and class-based aren't the only types of CSS selectors supported by this dialog, however. The other types also come with a few potential inconsistencies and problems, but that's the subject of next month's instalment…

Mark uses Inkscape to create comics for the web (www.peppertop.com/) as well as for print. You can follow him on Twitter for more comic and Inkscape content: **@PeppertopComics**