Inkscape - Part 122

Right on schedule, Inkscape 1.2 was released last month with a lot of new features, and updates to the existing UI. I will be covering everything in detail in this column, but not just yet. In the meantime, if you want a good overview of the headline features, I strongly recommend taking two minutes out of your day to watch the very professionally produced official release video: https://www.youtube.com/watch?v=1U4hVbvRr_g

For now, these articles are still working through the many new Live Path Effects (LPEs) that were added with Inkscape 1.0 and 1.1. The topic for this instalment is a pair of new LPEs – "Power Clip" and "Power Mask" – which I first mentioned back in part 101. That article detailed the use of the new "Inverse Clip" and "Inverse Mask" menu entries, with little more than a scant reference to the fact that they're actually implemented as LPEs. Now it's time to dig into the effects that back these capabilities in a little more detail.

Power Clip

I won't sugar coat this: I think the name of this LPE is making a very bold claim that is far from matched by reality. As far as I can tell, this effect usefully does one and only one thing, and doesn't really deserve the "Power" prefix at all. The only practical use for this LPE is to create inverse clipping paths.

Adding this effect to a path works a little differently to almost every other LPE. You can't simply select a path and add the Power Clip effect from the usual dialog. If you try, you'll just find that the option is disabled (as is Power Mask, for similar reasons).

A clipping path requires one other element to act on – the thing you're trying to clip. That single element could, of course, be a group consisting of several other items, which allows complex designs to be easily clipped. But the clipping path itself is applied to only one element or group.

With this in mind, you might think that this LPE will become available if you have two items selected. In practice, however, the Path Effects dialog itself will chastise you with an "Only one item can be selected" message, and the "+" button will be disabled to prevent you even opening the Live Path Effects Selector dialog. How, then, do we apply this LPE in the first place?

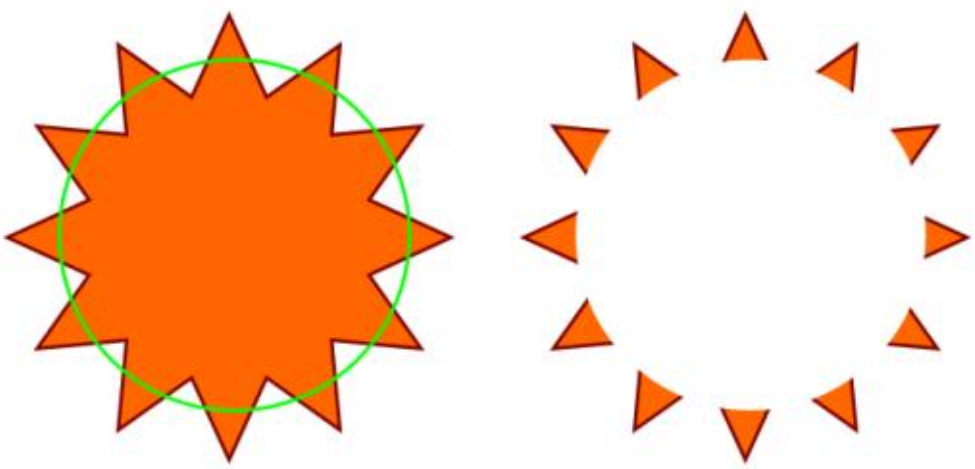
There are actually two methods:

- The first is to apply it directly using the Object > Clip > Set Inverse (LPE) menu entry that I discussed in part 101.
- The other is to apply a normal clip, either via the context menu or using the Object > Clip > Set menu entry, then use the Path Effects dialog to add the LPE as usual.

Yes, if you have an already clipped object selected, only then will you be able to add this LPE in the normal manner.

Whichever approach you take, this LPE really, really wants to create an inverted clip. If you add it directly via the menu entry, then that's what will happen, as you would expect. Adding it via the dialog to an already clipped object, however, will immediately invert the clip that you've already applied – which might be a little more surprising if you're not expecting it.

As a reminder, a normal clip preserves everything inside the path, and hides the content outside. An inverse clip, therefore, hides everything inside the path but preserves anything outside, per the image below.



If you really want a normal clip, then you can uncheck the "Inverse clip" option in the LPE's parameters – though doing so raises the obvious question of why you want to use the LPE at all in that case.

Perhaps the "power" in this LPE actually comes from the other two parameters? Well, the "Hide clip" option simply modifies the clipping path to remove the nodes that actually do the clipping – again, raising a question about why you wouldn't just remove the clip entirely. As for "Flatten clip": this is another entry in my box of LPE mystery parameters that appear to do something, but it's not entirely clear quite what that something is. My own experiments with various combinations of objects and groups, primitives and paths, have failed to present me with any situation in which it does something useful. As usual, if you have any idea of how it's meant to be used, please do get in touch.

One final important note about this LPE is how to go about removing it. If you only want to temporarily stop it being applied, then the usual approach with an LPE is to toggle the visibility icon (the "eye" image) in the Path Effects dialog. This may not, however, work as you expect. It will remove the inverting effect of the LPE, but still leave your object with the non-inverted clip applied. If you really want to see the object with no clip at all, then you're back to using the "Hide" option in the LPE's parameters. This feels a little awkward to me, as it requires users to understand too much about the implementation of clipping masks in SVG to fully appreciate why the normal visibility toggle doesn't work the way they might expect.

Permanently removing this effect is even worse. Should you be foolish enough to delete the effect from the Path Effects dialog, as you might usually do, then you're likely to find that your clipped objects disappear as well – almost certainly not what you wanted! Instead you have to release the clip using the Object > Clip > Release menu entry, or the context menu alternative. Even that comes with a sting in the tail: your original clipping path will also be deleted, so if you wanted to retain both that path and the object being clipped, you're out of luck.

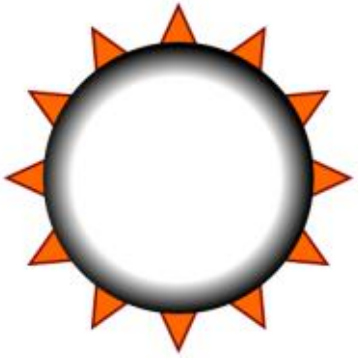
With that exhaustive examination of this "power" LPE, I've come to the conclusion that it serves only one useful purpose: to simplify the task of punching a hole in another object using a clipping path. To be fair, I think this is an exceptionally useful ability. I'm just not sure it warrants the "power" prefix, rather than simply being called "Inverse clip". As useful as it is, do be aware of the problems and limitations that come with it. If you've got enough experience with Inkscape to be able to make your own inverse clipping paths, as I described way back in part 13, perhaps sticking with the old-fashioned approach is still the safer option.

Power Mask

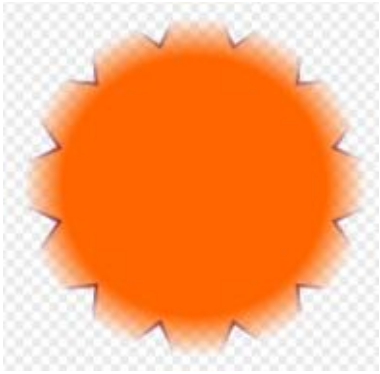
Masks are often thought of as a more "analogue" version of clipping. Whereas clips provide hard edges and a binary visible/hidden state for each part of the clipped object, masks allow for gradients and blurs, with every part of the masked object having its opacity determined over a continuum according to the colors used in the mask. But when it comes to the Power Mask LPE, the behaviour and usage compared with the Power Clip effect is quite different in many ways.

I prefer to keep my masks in

shades of gray to provide clear control over their effects. A typical mask would use white for those areas that are to remain 100% opaque, black for those that are to become completely transparent, and intermediate shades for levels of opacity in-between. Any transparent areas, including those outside the mask shape, are also rendered as transparent once the mask is applied. With that refresher done, here's the shape and mask we'll be experimenting with today.

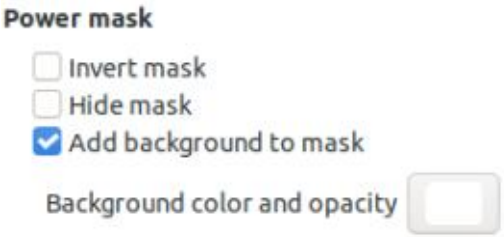


When applied as a normal mask, the white center to the design should ensure that the middle of the star remains solid, while the gradient out to the black edge will fade out the points of the star long before we reach their ends. Selecting both the star and the mask, then using the "Set Mask" context menu entry (or Object > Mask > Set) does, indeed, give the result we would expect (shown against a chequerboard background so you can see which parts are translucent).

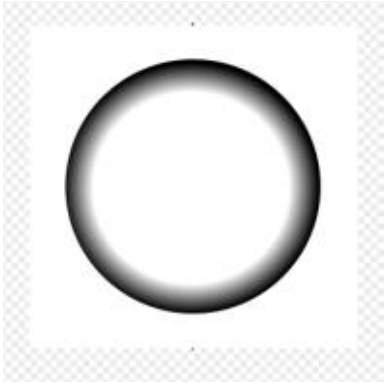


What would you expect the result to be if, instead, we use Object > Mask > Set Inverse (LPE)? Based on the behaviour of the Power Clip effect, you might imagine that the middle will be punched out of the star, leaving the points in place but fading towards a large central hole. That's certainly what I expected the first time I used this effect, but the actual outcome is somewhat different.

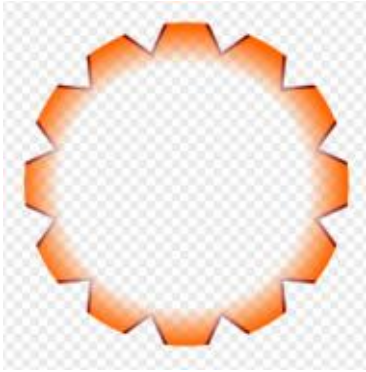
Let's take a closer look at the LPE's parameters to work out exactly what's happening here.



Interestingly, despite having selected "Set Inverse (LPE)", the "Invert mask" option is not checked. Instead all the hard work is being done by the "Add background to mask" checkbox, and the color picker below it. This has the effect of adding a flat colored background to your mask, which covers the bounding box of the element you're masking. In this case, it adds a white background which combines with our original mask to produce an effective mask that looks like this.

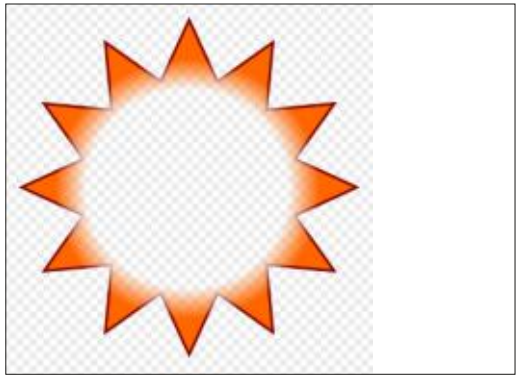


With that in mind, the end result is a little more understandable. But it's not exactly what I would expect from a menu entry that bills itself as an "inverse mask". Perhaps if we enable the "Invert mask" option, we'll get something closer to our expectations…



Hmm… not really. The problem now is that we've inverted every part of our mask – swapping black for white, light gray for dark gray, and so on – but that also includes the color of the background we've added. Now, therefore, the mask looks like this:

Again this explains the output we're seeing, but it's still not really what we had expected from an inverse mask. What we need is the hole in the middle from this version, but with the points of the stars still being visible, as we had in the previous one. In other words, we want a black center fading to a white shape that continues to fill the whole bounding box. The way to achieve that is to change the background color to black, so that the "Invert mask" checkbox has the effect of converting it to white. With that change, we finally get the look I was expecting from the outset.

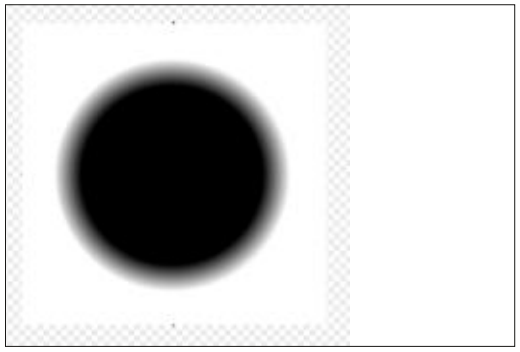


To summarise, in order to create what I would consider to be the inverse of my mask, I had to do the following:

- Draw the mask
- Apply it to the object using Object > Mask > Set Inverse (LPE)
- Open the Live Path Effects dialog to access the parameters
- Set the "Invert mask" option
- Ensure the "Add background" option is checked
- Set the background color to black

That really is a lot of work, and it's clear that the "Set Inverse (LPE)" option didn't really do much to help us out. My advice, therefore, is to skip this LPE entirely, and just construct the mask you need with the right colors from the outset. Here's what I would actually do to achieve the same result:

- Draw a white rectangle that covers the star
- Draw a black circle in the middle
- Blur the circle, or give it a black-to-white gradient to create the soft edges
- Group the circle and rectangle to create the final mask
- Select both the mask and the star, then use Object > Mask > Set



UPDATE:

via @inkscape on Twitter:

There's a bug in Inkscape 1.2 that could destroy gradients, clips & clones in a drawing. You'll only notice it happened after saving file & later opening it again. To work around bug DO NOT use copy-paste while Export dialog is open! The bug will be fixed in Inkscape 1.2.1.

For more details on the bug, head here:
https://gitlab.com/inkscape/inkscape/-/issues/3600

Admittedly it's barely any fewer steps, but I'll wager that this approach is a lot more intuitive to anyone with even a little experience using Inkscape.

And there you have these two new "power" LPEs which, on the surface, promise so much, yet deliver so little in reality. The Power Clip is potentially useful, but may create problems when you want to release the clip later. The Power Mask just doesn't do anything that you can't achieve by hand almost as easily, if not moreso. They're both useful tools to know about, though – if only so that you understand how to achieve the same results without having to get the LPEs involved at all!

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**

Inkscape - Part 123

One interesting aspect of the new 'Live Path Effects' Selector dialog is a small toggle at the right of the toolbar, labelled "Show Experimental". Turn this on, and your collection of LPEs will be augmented by a few others, each with a classic cherry bomb as its icon.

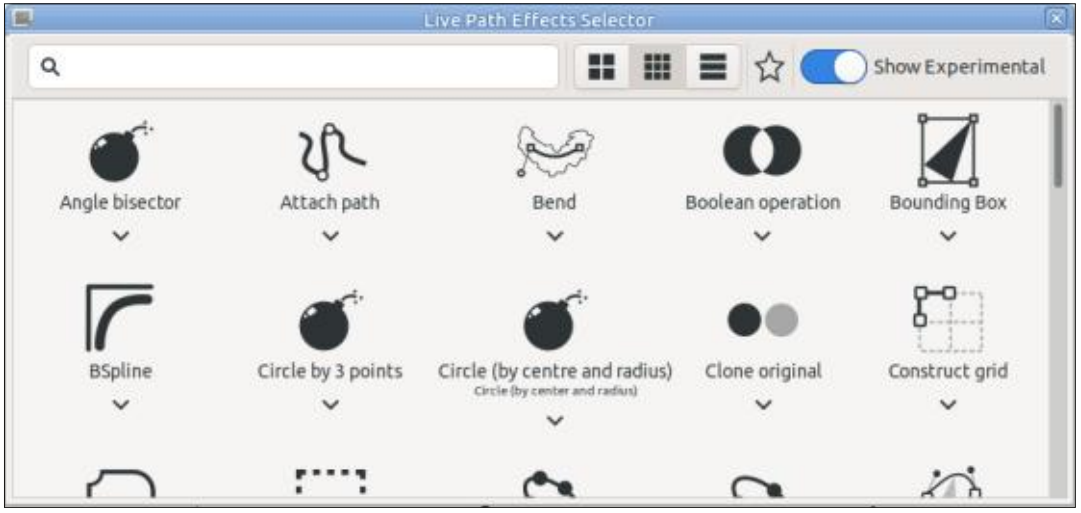
The cherry bomb has a long history in computer interfaces as a means to indicate something has gone wrong – the most famous instances being in the crash dialog of earlier MacOS releases, and to indicate boot problems on the Atari ST (whose GEM interface took rather a lot of "inspiration" from MacOS). In the case of Inkscape, however, it serves as a warning that using these LPEs is risky. Not only are they likely to be buggy, but there's also no guarantee that images you create using them will be supported in future versions. Generally, therefore, I recommend leaving the Show Experimental switch disabled – unless you wish to engage in the selfless act of actually trying the experimental LPEs in order to provide some useful feedback to the developers. Definitely don't use them for real work though.

The good news is that the best LPEs can, in time, be promoted out of the "experimental" stage and into the suite of fully supported effects. Such is the case with the Boolean Operation effect, which was experimental in version 1.0, but received enough polish and stability fixes to make it to the big time in Inkscape 1.1.

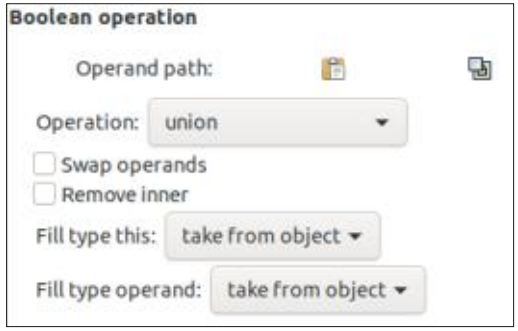
Boolean Operation

I'm going to assume that anyone reading this is already familiar with the existing boolean operations in Inkscape, found under the Path menu. If not, then you may wish to have a read of part 7 of this series, from way back in FCM #67! In short, boolean operations take two paths and manipulate them in different ways – such as joining them together into a single object (the "union" operation), cutting the shape of one out of the other ("difference") and keeping just the parts where they overlap ("intersection"), or where they don't overlap ("exclusion").

Using these operations from the Path menu is as simple as selecting two paths and then choosing the appropriate menu entry. You may also need to alter the stacking order of your paths, depending on the operation you're performing, as some of them produce different results if the roles of first and second operand are reversed. Inkscape uses the z-index of the paths to determine which is which.



With LPEs, things aren't quite so simple when multiple paths are required. You can't just select your two paths and add the effect. Instead, you have to begin by selecting your first operand path and add the Boolean Operation LPE to it in the normal manner. This will bring up the effect's parameters:



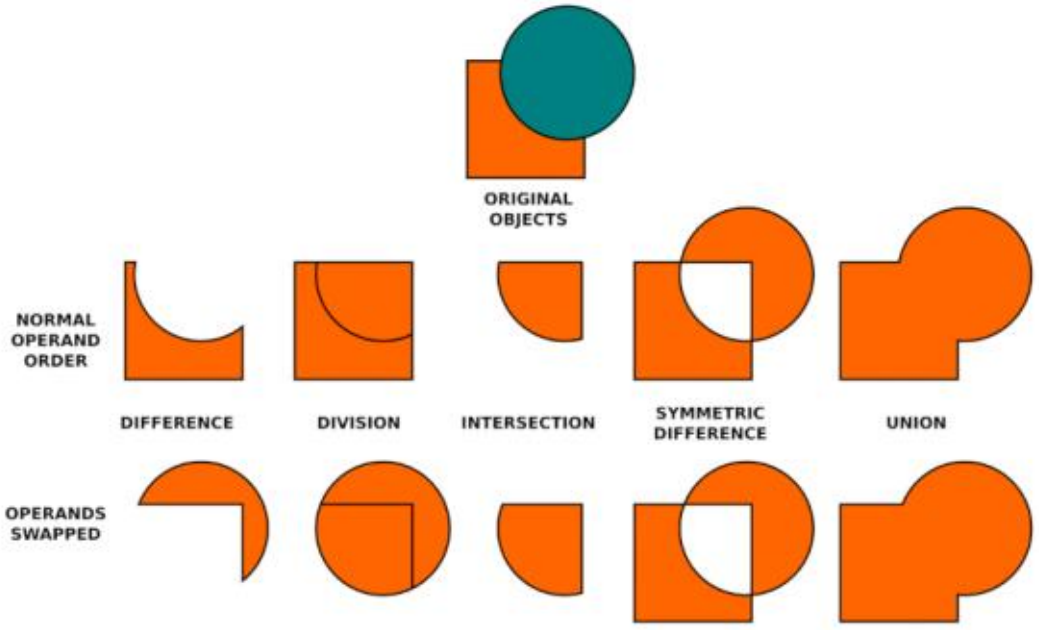
With just one path, a boolean operation won't do anything, so we need to add the second operand. This is done with the usual complex

and frustrating method of copying it to the clipboard and "linking" it into the LPE:

1. Select the second operand path on the canvas. This will de-select the first one, so you'll lose access to the LPE parameters.
2. Copy the selected path to the clipboard (Ctrl-C).
3. Re-select the first operand path in order to re-enable the LPE parameters.
4. Click on the "Link to item" button in the LPE controls (the one that looks like a clipboard).

Note that you're not actually pasting the second path into the LPE, but rather using this convoluted technique in order to add a reference to the path – a "link" in the terminology used in LPEs. That means you are still free to make changes to the second path, and don't need to go through this rigmarole each time: the link is live, so changes propagate automatically.

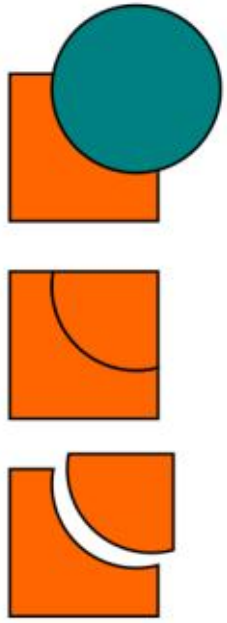
With two paths added, the actual result will largely depend on the value selected in the "Operation" pop-up menu. The "Swap operands" checkbox can be used to change Inkscape's notion of which order the operands should be applied in (equivalent to swapping the z-index order when using the traditional boolean path operations). For some cases this will have no effect on the output. The image below gives you an idea of how these two controls work when applied to a pair of simple shapes.



At the top of the image is the original pair of paths. In each case, the LPE was applied to the orange square, with the teal circle being linked via the clipboard. The columns below show the effect of applying each operation, with the top object being the default output, and the bottom one showing the effect of ticking the "Swap Operands" checkbox.

The results are fairly self-explanatory, especially if you're already familiar with the standard boolean operations in Inkscape. It's worth noting that the operation referred to in the LPE as "Symmetric Difference" is, as far as I can tell, the equivalent of Path > Exclusion. I have no idea why the developers opted for different names, as it just adds unnecessary confusion.

One significant difference between the historical boolean operations and this LPE is with the output from the Division operation. With the old-style path operations the result is a pair of objects which can be dragged apart. You can see this in the image below, which shows the original objects, the immediate result of the Path > Division operation, and the effect of dragging the top-right object away.



With the LPE version, the initial output looks similar, however trying to move the top-right part actually just changes the position of the second operand path. In order to separate the parts (i.e. to actually divide them), you have to take the additional step of "fixing" the LPE by using Path > Object to

Path. Of course, once you've done that, you no longer have the benefit of any "liveness", so you may as well have just used the older approach in the first place.

Moving on with the parameters for the effect, next we have a checkbox labelled "Remove Inner" – but only if you're using version 1.1. This checkbox only had an effect on some operations that were ultimately removed from this LPE for the 1.1 release due to stability issues, so there's no need to consider it further. It's been removed entirely from Inkscape version 1.2.

The last two parameters concern the fill rule which is applied to each path. The first pop-up governs the path that the LPE is applied to, while the second is for the one added via the clipboard. Fill rules are one of the more obscure parts of vector graphics, and essentially set the rules for how complex paths with self-intersections or sub-paths should be treated when applying the fill color. You may have encountered them via the toggle buttons in the Fill & Stroke dialog (see part 95 of this series, in FCM #155).

In this image, you can see what happens when I turn the original objects into complex paths with several sub-paths. The top image shows the objects with no LPE applied (with the "evenodd" fill rule applied via the Fill & Stroke dialog). The second image shows the effect of applying the "Symmetric Difference" (aka Exclusion) effect with both pop-ups set to "non-zero"; the third is the same but with them both set to "even-odd".

In my experience, the "positive" option just makes the objects disappear, but that might simply be an effect of the paths I'm using. The "take from object" option honours the setting in the Fill & Stroke dialog, so, in this case, the output is the same as the bottom image. My advice is to leave these pop-ups set to "take from object", and only fiddle with them if you have complex paths and aren't getting the result you would like in terms of the areas being filled.

As we've seen, the Boolean Operation effect will accept only two paths: the first operand (on which the LPE is applied) and the second operand (added via the clipboard). This is unlike some other LPEs which allow you to add multiple linked paths into a list. This is a shame, because some boolean operations could easily be extended to multiple objects, while others could simply treat a list of paths as a sequence of operands, applying them one at a time to the output from the previous operation. Instead, should you wish to apply multiple boolean operations using this LPE, you'll have to add several instances of it to your effects list. In itself, this isn't a problem, but you can quickly end up with a series of entries, all just labelled as "Boolean operation", which can make it hard to keep track of what each operation is for.

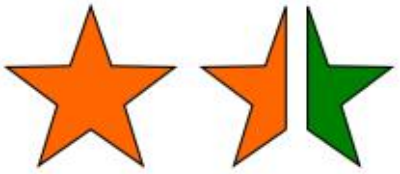
Personally, I'm more likely to stick to the old-fashioned boolean operations. While it's great to have a "live" version of these, allowing for subsequent tweaks and changes, the means of applying them is a little fiddly (as with all multi-path LPEs, to be fair), and their liveness means that they not only require more processing power to display, but are also more likely to expose bugs in the implementation (I suffered several hard crashes while creating the screenshots for this article!). LPEs also aren't recognised by other applications, or web browsers, so although the output should look the same in theory, you get a better guarantee of that with the older operators that make permanent changes to the underlying paths.

Slice

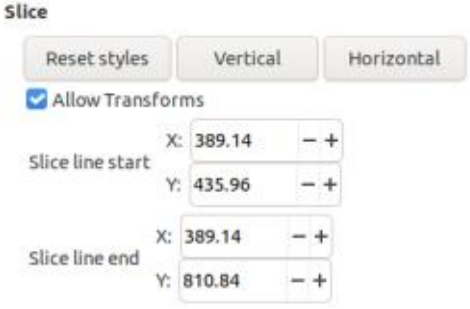
For many use cases, the Slice LPE will likely give a better result than the Boolean Operation effect

in Division mode. As noted above, this mode does not actually produce separate objects until you "fix" the effect. The Slice effect, on the other hand, does split your object into two separate parts that can be moved, and even styled, independently of each other. It is limited to splitting along a straight line, however, so the "square and circle" example I used for Division can't be replicated using Slice.

It's a pretty easy LPE to use. Just select your object and add the effect. The object will be broken into two with a vertical split down the middle, and the two parts can be moved and styled separately, as shown in this simple example of applying the Slice LPE to a star.



The parameters for this effect are pretty simple, too.



The three buttons do exactly what you would expect from their labels:

- Remove any style changes that have been applied to the individual parts
- Use a vertical slicing line
- Use a horizontal slicing line

I've used the term "slicing line" to indicate that this is a straight line, not an SVG-style path that could have corners and curves. But you're not limited to horizontal and vertical slicing: the coordinate boxes below can be used to set any start and end points for the line, allowing angled slicing lines to be used as well. In practice, nobody is likely to fill in the numbers here, though, as you can switch to the Node tool (F2) in order to move and rotate the slicing line on-canvas instead. This approach makes it easy and intuitive to adjust the line to suit your needs.

The "Allow Transforms" checkbox is a vital part of this effect – and one you'll most likely want to leave enabled all the time. If this is unchecked then you won't be able to move the individual parts around (nor skew, scale or rotate them). They'll stay in their original location, relative to the source path, which may be useful if you want to style only parts of the shape differently, but don't want to adjust their positions. For most people, however, slicing a path implies a need to move the pieces around, for which this needs to remain checked.

If you do want your shape cut into more than two parts you can apply the Slice effect more than once. Here's how our star looks with an additional set of angled slicing lines applied to each half, and a bit more variation in the styling.

Of course, the Slice LPE can be combined with other effects. Here are our earlier complex paths, with the Boolean Operation effect applied ("Symmetric Difference" mode, "even-odd" for both operands), followed by a pair of Slice effects to produce four complex paths, each with a different fill color.

After several months, we're now finished with the new LPEs that were added in Inkscape 1.0 and 1.1 (and that's without considering the experimental ones!). But we're not quite done yet: next month, I'll take a look at some important new features that were added to an existing LPE, before moving on to the new extensions that were added in version 1.0.

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**

Update from Inkscape.org

Important Release of Inkscape version 1.2.1 fixes data loss and crash issues

We've just resolved some critical issues in Inkscape 1.2 that were identified by our community. If you're currently using Inkscape 1.2 then this is an important update to install on your device.

It provides a fix for a number of critical bugs in the 1.2 release:
- drawing data about filters, markers, gradients and more, now saves to file after performing a simple sequence of copy-paste-undo with the Export dialog open;
- startup is smoother when many fonts are installed, and
- rasterized (filtered) objects now show up on any page of a multipage document exported to PDF.

This release brings back the built-in ability to open PostScript/ Encapsulated PostScript (.ps / .eps) files with Inkscape on macOS.

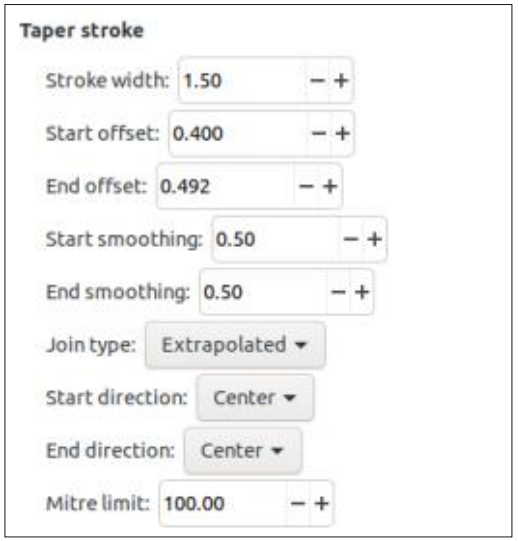
Improvements were also made to user interface translations and documentation translations.

Linux users using the snap packaging format will once again be able to import and export images. Windows users will now see the correct Inkscape version image on install, while users of macOS 10.13-10.15 will be able to open extensions without them crashing.

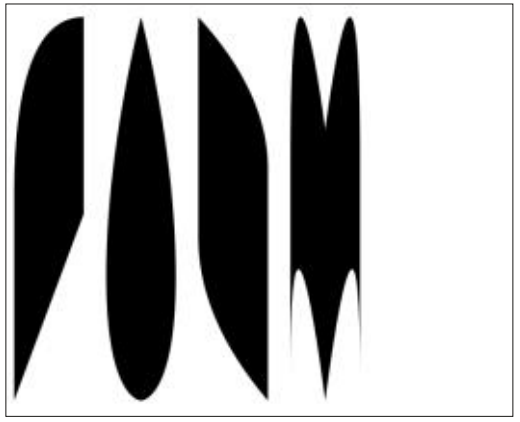
Inkscape - Part 124

I first described the Taper Stroke LPE in part 67 of this series (FCM#127). Back then, it was in the context of manually tracing an image. As an example, I took the 'Frankie' cartoon character, originally drawn by Vincent Mealing for our "Monsters, Inked" comic series, and traced over a scan of the original artwork using a variety of techniques. The Taper Stroke effect was used as a quick-and-easy way to produce more interesting outlines, without going to the full extent of using the Power Stroke LPE. Here's a screenshot that appeared in the article, showing the result of using the effect in the drawing, and the UI for the LPE as it was at that time.

With Inkscape version 1.1, several new additions were made to the Taper Stroke parameters – enough to warrant a quick update in this series. Here's how the UI appears now:



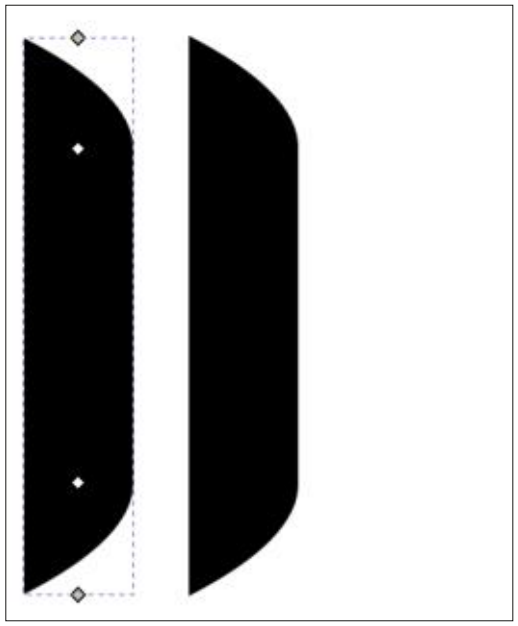
The first change is that the Taper Smoothing control has now been split into two parameters, governing the start and end smoothing respectively. In this context, smoothing refers to the shape of the taper, from a very rounded taper (smoothing: 1.0) to a straight line (smoothing: 0). You can see both of these used on the image below, which shows some of the effects that can be produced with this LPE when used on exaggeratedly thick lines. The Left-hand line displays the sensible limits of the smoothing parameter (1.0 at the top, and 0 at the bottom), while the rightmost line shows the effect of setting these to more extreme values (+/- 10.0 in this case).



This image also shows the other addition to this LPE: the ability to alter the direction of the taper. The second and fourth lines use the default "center" direction, but the first and third have their start and end directions variously set to "left" and "right". These are the only three options available – it's not possible to have a taper that terminates only slightly left of center, for example.

It's worth noting that the "left" and "right" designations are relative to the direction of the path. Imagine walking along the path from start to finish to determine which side of it is which. In the image below, the left-hand line has its directions set to "left", and the right-hand line has its directions set to "right", yet they both appear the same. The difference is that the former was drawn from bottom to top, and the latter from top to bottom. This also means that using Path Reverse on a shape with this LPE applied will flip the visible directions of the tapers, so watch out for that one if

you're forced to reverse the path for other reasons!



I've left the leftmost line selected in order to show the handles that are visible when the Node tool (F2) is active. As has always been the case, you can drag these handles to adjust the length of the start and end tapers. A new addition, however, is that you can now shift-click on them to cycle between the three direction values, making it quick and easy to experiment with this upgraded capability.

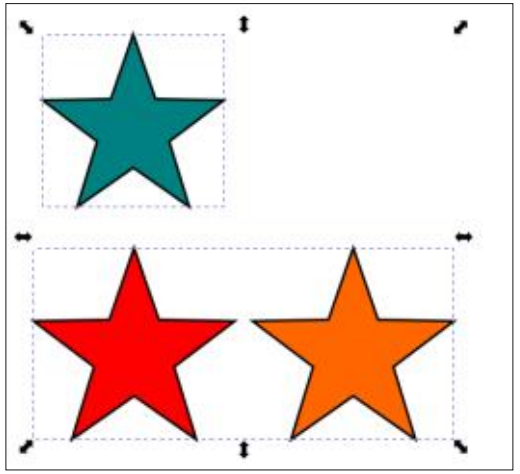
With that, we've reached the end of the new Live Path Effects added in Inkscape 1.0 and 1.1 (I'm skipping the experimental LPEs for the reasons mentioned last time). But these releases also brought with them some new extensions, which I'll describe below, and next month. These can all be found under the Extensions menu, of course, and I've included the relevant submenu at the start of each section title.

Render > Frame

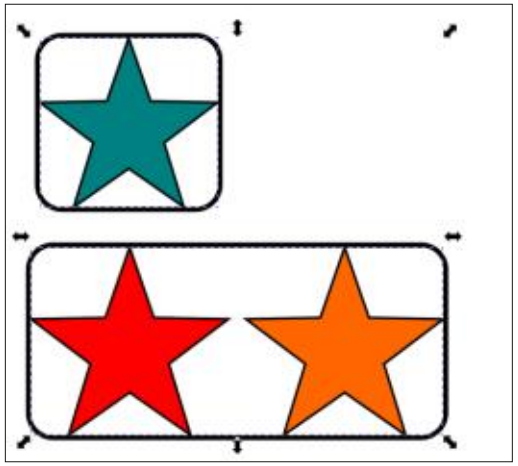
Let's start with a simple extension. This one draws a "frame" around each selected object in your image. I've put the word "frame" in quotes because it suggests something far more impressive than the result this extension produces. You might imagine an ornate picture frame, a Celtic knot style border, or perhaps something sweeping and calligraphic. Instead, what you actually get is what appears to be a rounded rectangle around each selected object. Except it's not even a rounded rectangle – it's a path – so there's no easy way to change the corner radius after the fact.

Drawing a rectangle that exactly hugs the outside of an arbitrary object is a little tricky in Inkscape. But it's not exactly rocket science either. The simplest approach I know of is to duplicate your object, select Object Objects To Guides, then draw a rectangle by snapping to the intersections between the guidelines. This extension reduces those few steps down to one, but doesn't really offer many other advantages as it does so.

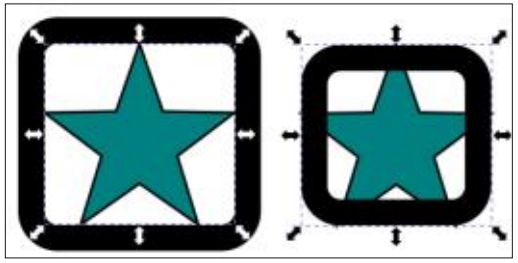
Let's look at it in action. We'll begin by selecting multiple items on the canvas. In this case there are two selections – one single star at the top, and one group of two stars at the bottom.



Now we'll run the extension with some pretty standard values to get started: Position is set to Outside, the checkboxes are left un-ticked, the stroke width is set to 2px and the corner radius to 10. Finally the stroke color is set to black, and the fill (on the second tab) is transparent (alpha = 0). Here's the result:



The Inside/Outside popup can be used to determine whether the frame is drawn inside the bounding box of the selected object, or outside it. The difference is most obvious when the stroke thickness is cranked up a little – to 10px in this example.



The Clip checkbox determines whether or not the selected item should be clipped to the size of the

frame. Often this will make little difference, but if your object has a thick border, for example, this can theoretically hide the portion that extends outside the frame. I say "theoretically" because I have not been able to get this feature to work reliably at all. Most of the time enabling the Clip option just results in the entire object being removed. I have seen it work correctly on one occasion, but I haven't been able to reproduce that behaviour since.

The Group checkbox, on the other hand, does seem to work. But all it does is group the selected object with its corresponding frame, saving you a small manual step if you need to keep the object and its frame together in this way.

The new frame is rendered on top of the selected object. Bear that in mind before setting the Fill tab to use an opaque color, as it will obscure the original element. You can change the stacking order after applying the extension, of course, but it's another step to be aware of.

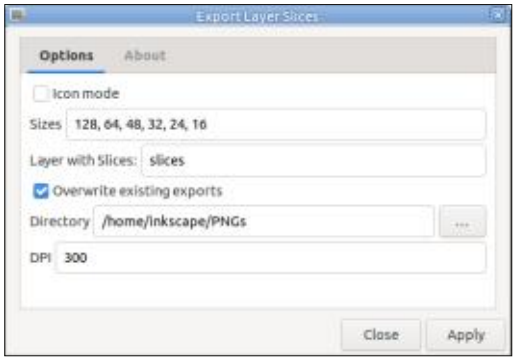
I don't really see the point in this extension. It would make some sense if there was a field in which to add a padding value, allowing for frames that don't hug the content quite so tightly. If the frame could be rendered in the background as well, this would provide an easy way to add a box and background color for multiple items, turning them into buttons or icons, for example. As it stands, I see little benefit of using this extension over learning the small number of steps to perform this task manually, unless you have a very large number of objects that require tightly fitting rectangles around them for some reason. Ironically the next extension would have been a perfect example of a use for this feature, if it wasn't for the fact that the output of the Frame extension is not at all in the right form for Export Layer Slices to use…

Export > Export Layer Slices

Inkscape already has a native mechanism for exporting multiple items in a document as separate PNG files, via the Batch Export feature in the File Export PNG Image… dialog. This assumes that the items you want to export are individual elements, or are already grouped in the right way. You can get around this limitation by simply creating a layer on which to draw some rectangles (or other shapes) that mark out the sections of the page to export, then perform a Batch Export with those rectangles selected. Here's a tip: if you don't want the rectangles themselves to appear as part of the exported files, select them all and then hide the layer before you hit the Export button in the dialog – the Batch Export checkbox should still say "Batch export N selected objects", where N is the number of shapes you selected in your slicing layer, but you're not left with a visible stroke around the edge of each PNG.

The Export Layer Slices extension essentially does the same thing. To use it you have to create a slicing layer, then draw rectangles to mark the areas to be cut out and saved as separate PNGs. The layer has to be at the top level (rather than being a sub-layer), and the rectangles must actually be rectangles (i.e. <rect> elements in the XML), not paths or any other shape. The Export dialog approach doesn't have either of these limitations.

When creating your slicing layer, you need to ensure that the name of the layer is the same as the name used in the extension's UI. By default, this is "slices", so the easiest approach is simply to give your layer the same name.



Set your export directory, and the DPI for the images, and hit the Apply button to create your PNG files. If you want fine control over the filenames, you can first change the ID of each rectangle via the Object Object Properties dialog (don't forget to hit the "Set" button). And you probably also want to tick the "Overwrite existing exports" checkbox, otherwise the extension will refuse to replace any existing files with the same names.

Unfortunately, this extension suffers from a major flaw, as far as I am concerned. Given that you have to create a whole separate layer to define the slicing regions, I would

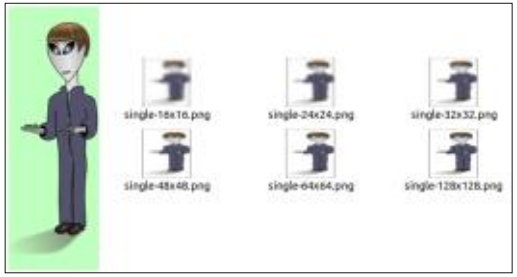
expect that layer to be omitted from the exported images. After all, it xists only to provide some dimensions, it's not actually part of the drawing itself. But no, every exported image includes the rectangle's stroke and fill as part of the PNG. The easy workaround is to hide the layer before running the extension; unlike the equivalent Export dialog workaround, there's no need to select the rectangles first, but it's still a bit of a pain having to repeatedly hide and unhide it, if you're trying to fine-tune the sizes and positions of the slices.

But perhaps you want the rectangle included, to provide a nice border for your exported PNG. You take the time to set the stroke width and color… only to find that your own choice of styles is replaced by the extension after the export. The rectangle's stroke is removed and the fill is replaced with a shade of gray (file already exists and was not overwritten), red (file exists, but was overwritten), or green (file was created for the first time). If you want to try the export again, you'll need to revert all those back to their original styles unless you really want a red overlay added to all your PNGs!

In my opinion, if you want to export multiple slices of your document defined by rectangles, you may as well just use the standard Batch Export approach, with an optionally hidden slicing layer. The benefits of the extension just aren't great enough to make it a compelling new method of performing this task.

This extension does offer one other trick, however: Icon mode. This is enabled via the checkbox of the same name, and while it initially appears useful, it's also got a major flaw that makes it less than practical in a lot of cases. What this mode does is to ignore the DPI setting, and instead create a series of square images for each slicing region, using the pixel sizes defined in the "Sizes" text box. Unfortunately, it does this by stretching each image to fit the square aspect ratio. In short, your slicing "rectangle" in this case absolutely has to be a square, otherwise the content will be stretched out of proportion. A much better approach would have been to scale the content in proportion, and automatically center it in the square. This would work equally well for slices that are already square, and those of a different aspect ratio. But alas, the extension offers no such option.

The image below shows the result of using Icon Mode on a single image from a character sheet. On the left you can see the original Inkscape drawing, complete with the slicing rectangle (in green, due to this screenshot being taken after the initial export). To the right, you can see the icons that were produced. Clearly not the result I would have liked or expected.



So there we have two of the new extensions. I hate to be cynical, but I can't really see much benefit in either of them. The Frame extension could perhaps be useful if it allowed some padding to be specified, and created real rectangles instead of paths. The Export Layer Slices extension could be useful if it didn't also include the slicing rectangles in the output, and if it didn't stretch the images out of proportion in Icon mode.

What's most annoying is that these two extensions are frustratingly close to working well together. Imagine if the Frame extension's default behaviour was to create real rectangles in a new "slices" layer (with optional padding, of course). Instantly, it becomes an easy way to create the initial slicing rectangles for the Layer Slices extension if you have a lot of elements to export. But instead, we get these two extensions, neither of which are great on their own, and which don't work well together. What a shame.

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**

Inkscape - Part 125

Interactive Mockup is the only remaining new extension that arrived with version 1.0 for me to discuss in this series. But what I initially thought was going to be a simple topic to cover has thrown up a few questions and issues which will stretch this topic out to more than one article. To start with, though, let's take a look at this extension, what it does, and how to use it.

Web > Interactive Mockup

This extension is intended to let you use Inkscape to create user interface mock-ups with some limited interactivity. It could also be used to create a simple slideshow, a basic adventure game, or more. It does this by injecting some simple JavaScript into your document, so the final file has to be loaded into a web browser in order to produce the interactivity suggested by its name.

But let's start with a quick definition: the "interactive" mock-ups I'm referring to here are little more than a series of static images containing clickable regions that let you navigate between them. If you were mocking up a series of dialog boxes to form a "wizard" type of interface, you could use this extension to detect a click on the "forward" and "back" buttons in your images. Each click would switch to another image in the sequence, allowing you to demonstrate how your potential user interface might work before you actually start to code it.

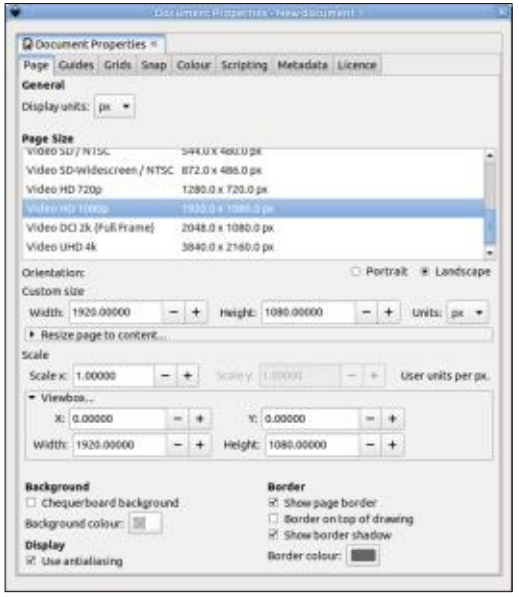
Another example is when mocking up a website. Perhaps you've got a series of images, one for each page in the final site, created using a design tool or from a scanned sketch, but not yet turned into actual HTML code. Using this extension, you could combine those images into a working demonstration of how the page might operate, complete with clickable menus and links for moving between the different pages.

I'm going to use this latter example to show you how this extension operates, by mocking-up a very simple 3-page website. Just make sure you understand that this approach could be used to let you link any series of images together, and isn't restricted to websites. The reason it's found in the Extensions Web submenu is solely because the final result is a file that has to be opened in a web browser in order to work.

Let's start with the main home page for my website. This will be the starting content when the document is loaded into a browser, and will define the physical size for the whole presentation. In this case, I'm going to demo the site using an HD monitor, with the browser set to fullscreen mode (by pressing F11), meaning I'll have every one of those 1920×1080 pixels to play with. To begin with, therefore, I'll open the Document Properties and set my document up as follows:

Display units: px
Page Size: Video HD 1080p (1920 × 1080px)
Scale x: 1.0
Viewbox x, y, width, height: 0, 0, 1920, 1080 (these should be set automatically)

The last two items aren't so important, but ensuring we have a scale value of 1.0 means that we can use specific pixel dimensions in our page design, since there's a 1:1 mapping between each pixel in our drawing, and the corresponding pixel on-screen once the page is rendered by the browser.



Now to add some content to the page, starting with a home page. It

won't win any design awards, but it will do the job for demonstrating this extension (above).

The key part of the design is the section at the top right. That's where I want to have links to navigate between pages. Those are the elements that will become clickable in order to move from page to page in my mock-up. Of course, in order to be able to do this, we actually need some pages to move to! Time to create the second entry in my mock-up: the "About" page.

This is where things start to get a little tricky. We need our new page to have the same dimensions as the first one. If we were creating these as completely separate files, that would be simple enough - just create a new document with the same properties as the first. In practice, I'd probably just use File > Save As… on the first document, then edit the content to suit, but either approach would create multiple files, each with the same basic page properties. But this extension doesn't link between separate documents, so that approach is no good.

Another sensible method would be to create each image on a separate layer. That way all the images share the same document properties, by definition. But this extension doesn't work with layers, either.

What it actually does is inject some JavaScript that will change the viewBox attribute on the <svg> element. This is an approach that has been discussed in this column in the past, with respect to "SVG sprites" (part 79, in FCM #139). More specifically, the extension actually only changes the first two numbers in the viewBox – the ones that define the top-left corner of the view. The other two numbers – those that define the width and height - remain fixed, so it's pretty much essential that all our mock-up pages are the same size.

I find that the easiest way to do this is to enable the snap-to-page option and then draw a rectangle from the top-left corner of the page to the bottom-right. Alternatively, you could simply draw a rectangle anywhere and then set its width and height to the correct dimensions via the tool control bar. In either case, you want to end up with a rectangle that is the same size as your main document page, but you should then drag it outside of the page boundary. Repeat this for each frame you will require in your mock-up, so that the new 'pages' are distributed around the real Inkscape page, and don't overlap. It doesn't matter if they're arranged in a row, column, grid, or randomly

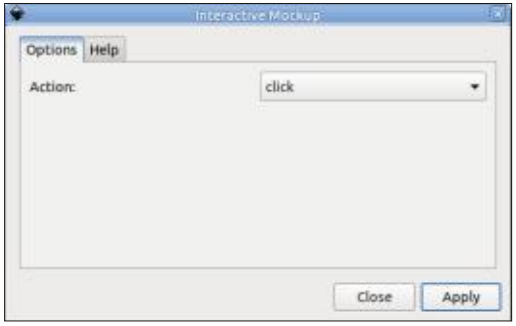
placed, so don't get unnecessarily caught up in the positioning of each page.

Now you can draw the content of each image within the confines of its rectangle. As you are likely to need some sort of background color in your mock-up (even if that color is only white), I suggest giving each rectangle a solid fill and then locking it, or the layer it lives on, to avoid it being accidentally moved when drawing on top of it.

Following this approach, the selection of pages in my mock-up website ends up looking like the image on the previous page, bottom right.

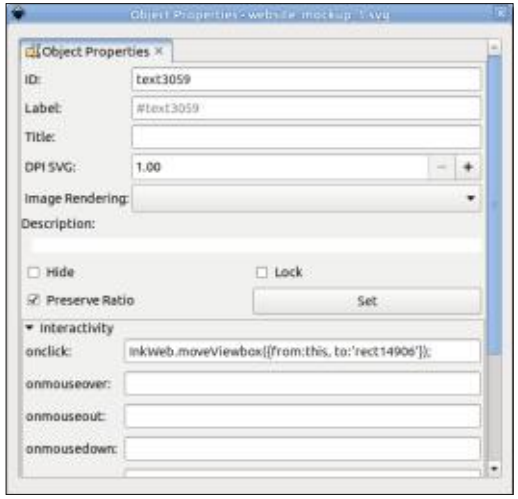
Now that we have some pages in place, we can start adding a little interactivity using the extension. The first thing I'm going to do is to ensure that the "About" link on the Home page connects to the About page. The extension requires you to have two elements selected: in this case I'll select the "About" text element on my Home page, then hold Shift as I click on the background rectangle that encloses the About page. With those two items selected, I can click on Extensions > Web > Interactive Mockup, to be presented with this rather underwhelming dialog.

As you can see, there's nothing



but a Help tab and an "Action" pop-up. The latter offers various ways in which interacting with the first element in your selection will switch the viewBox to point to the second element in your selection. Most of the time you will probably want to leave this set as "click". In almost all cases, therefore, there's nothing for you to do here but to click on the "Apply" button and close the dialog.

Nothing will have changed visibly in your page, but the extension has added some JavaScript to the first element in your selection. Open the Object > Object Properties dialog and expand the "Interactivity" section; you'll notice a small piece of JavaScript has been added to one of the fields (you may also notice that the list of fields available in



this dialog maps suspiciously closely to the options in the pop-up within the extension dialog).

I'll look at the JavaScript itself in future, but for now it's sufficient to know that this is what is ultimately responsible for changing the viewBox when the first selected element is interacted with (i.e. clicked on, in this case). At this point you can save the file as a normal Inkscape SVG document and load it directly into a web browser. Browsers don't actually make this capability terribly obvious these days, but you can either drag the file directly into the browser window, or hit Ctrl-O to bring up an Open File dialog, just as you might in any other program.

When the page loads in the browser you'll probably notice that some of it is cut-off.

This might seem a little strange, given that we're working with a scalable image, but it boils down to the fact that we set a specific width and height for the file, in pixels, in the Document Properties dialog. You may recall that the intention was for it to fit the dimensions of an HD screen when the browser is switched to fullscreen mode. Press F11, therefore, and our mock web page fits perfectly!

Now we can test the interactivity. Click on the "About" link and, as expected, the view switches to the relevant mock page in our document. At this point, however, no other links have been set up, so we can't click the "Home" button to return to the previous view. Neither can you use the Back button (or keyboard shortcut) within the browser, since the URL hasn't actually changed – the JavaScript has just dynamically updated the content of our page in order to set the new viewBox.

Let's return to Inkscape and fix some of these issues by adding more links. First we need to add a connection going from the "About"

link on the Contacts page to the rectangle of the About page itself, in just the same way that we did with the previous link from the Home page. With that in place it's now possible to jump to the About page from either of the others. As this shows, it's okay for the same element to be the target of several different links.

We also want both the Home and About pages to link to the Contacts page. We could connect each link item individually, as we've been doing so far, but the extension offers a small shortcut that can help us. If you call the extension with more than two objects selected, the last one is used as the target, and all the others are linked to it. This means that we just have to select the first "Contact" element (on the Home page), then hold Shift while we click on the second one (on the About page), and continue to hold it while we click on the target background rectangle for the Contacts page.

With all three items selected we'll now use another little shortcut, offered by Inkscape itself. We know we don't have to change the single parameter within the extension dialog, so we can just use Extensions > Previous Extension to run the extension directly, without having to interact with the dialog. The JavaScript is added to both our menu links as one step. This approach makes it quick and easy to finish all the remaining links between our three pages.

With that done, all that remains is to save the file, reload it in the browser, switch to full-screen (F11) if needed, and click the fake links to switch between all three pages of our mocked-up website. Pretty cool stuff, right?

Of course this is all well and good if you've created your mock-ups within Inkscape. You can easily select the individual objects that are to act as triggers. But what if your "page" is actually a bitmap image, exported from a paint program or scanned from a sketch on a napkin? Or it might be a more complex Inkscape image, with your objects nested deeply inside a range of groups, making it tricky to select both the trigger and the target element at the same time. In those cases, there's a simple little trick that can help you out: just draw fresh elements over the top of your page which will act as your triggers and/or destinations.

Start with a simple shape with a colored fill that covers your trigger item completely. It doesn't even need to be a rectangle, if your design calls for something more complex. Give it a little opacity so you can see what you're doing as you start to add more of them to cater for every possible link in your mock-up. Use the extension to add the interactivity then, just before saving, set the opacity way down on these new shapes. You can set it to zero to ensure they're not visible on the page at all, but that does make it trickier to re-select them for editing later. A useful trick is to set their opacity to 1: this is usually so transparent that there's no hint of them on the final page, but they're much easier to select in Inkscape itself if they need to be modified in future.

With this approach, it's fairly trivial to turn even a series of pencil sketches into something that's interactive enough to work as a proof-of-concept demo. Want to improve the UI of your favourite Open Source application? Before wading in with a code editor, you could take some screenshots, edit them in The GIMP, and use this extension to test out your ideas on other users first.

I'm sure you can see how this simple extension makes it very quick and easy to create interactive mock-ups. For a lot of situations, this may be all that you need, but next time I'll take a look at some of the problems presented by such a simple system, and how they can be addressed with just a little extra effort.

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**

Inkscape - Part 126

Last month, I looked at the Web > Interactive Mockup extension using a simple website mock-up. I demonstrated how you can use this to create a demo that works well on a specific screen size. But even though SVG files are scalable by their very nature, this extension doesn't do a great job of producing demos that will work across a variety of devices. This is a limitation we'll try to address this month, following on with the same file I created last time, so make sure you've read last month's column before pressing on with this one.

First, I need to manage your expectations. Although we'll be trying to make our presentation work better on different screen sizes, there is a limit to what can be done with this simple extension or the small amount of code we'll be writing. All we're aiming to do is to make the output scalable – we're not going to be creating a fully responsive mock-up where parts of the design move around or change appearance in order to suit mobile, tablet, and PC displays from a single file. Think of it instead as creating a file that will work on both a laptop and desktop PC, even if the screens are slightly different sizes.

Let's begin by exposing the problem we have. Last month we created a mock-up that was specifically sized at 1920px × 1080px – the dimensions of a high-definition screen. To display at that size we loaded the file into a web browser and pressed the F11 key in order to remove the browser UI and render the content full-screen. But let's see what the same file looks like before the F11 button is pressed, when the available resolution for the SVG content is smaller.

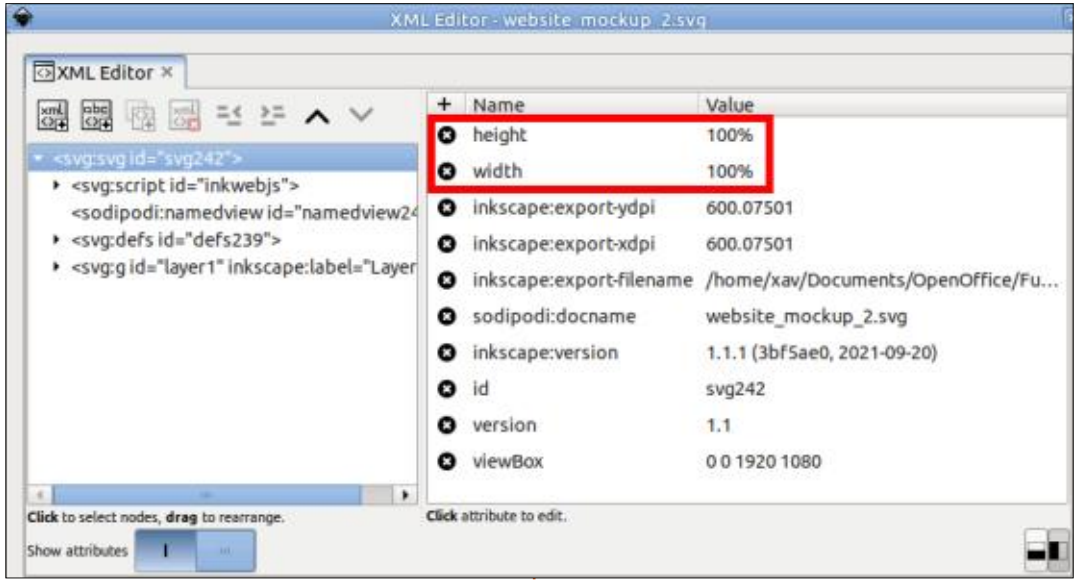
Vertically we can no longer see the bottom of the page. The available space to draw the image has been reduced by the browser's UI, by the window's title bar, and by the top and bottom panels of my Mate desktop environment.

Horizontally all the content fits, because the browser isn't showing any scrollbars and the window theme has thin borders, so almost the whole width of the monitor is available when drawing the content. If I were to resize the window, however, or open the developer tools to one side, the right-hand side of the image would be cut off.

To make it easier to experiment with different screen sizes, we're going to use the browser's developer tools to simulate the page's behaviour across a range of screen sizes. These instructions are for Firefox, but all modern browsers have similar tools available, and even the keyboard shortcuts tend to be common between them.

- Open the Developer Tools via the menu, or by pressing F12.
- Switch to Responsive Design Mode using the button at the top of the developer tools (it looks like a couple of rectangles, representing a phone and a tablet), or by pressing Ctrl-Shift-M.
- The screen content should switch to some sort of mobile view.

You can use the controls at the top to select specific devices to emulate, and to switch between



landscape and portrait mode. For our purposes, however, just drag the handle at the bottom-right corner of the screen around to quickly try the page in various sizes and aspect ratios. On Chrome and Chromium, the handle may not be visible: in that case you first need to select "Responsive" from the pop-up menu at the top of the page area.

Now it should be pretty clear that the content of the file is not scaling to suit the window, and is being cut off along the right or the bottom (or both) depending on the size of the content area.

The reason for this is pretty straightforward. Inkscape requires us to specify a page size when we create a document, and in this case we set it to our target size of 1920px × 1080px. When loading the image into a web browser, however, it's trying to honour those dimensions still, meaning that if the available document size isn't large enough, the excess will just get cut off. What we need is to replace the width and height in the file with something more dynamic – something like "100%" or "auto". Or even to remove those fixed values altogether and just let the browser do the right thing. Unfortunately none of these approaches work within Inkscape's Document Properties dialog.

Although "%" appears as a unit in many of Inkscape's unit pickers, the size pop-up in the Document Properties dialog isn't one of them. Don't get fooled by the "pc" unit – that's "picas" (1/8th of an inch). Trying to clear the Width or Height fields, or typing "auto" into them, results in them being populated with a value of 0.00001 instead. Typing "100%", with the units appended to the number, just gets converted to a value of 100, using whatever units are selected in the pop-up.

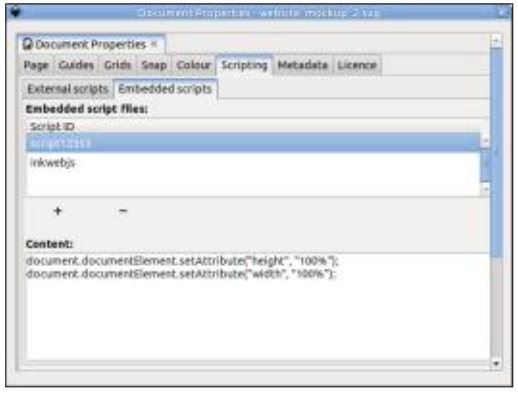
Although we can't set the values we want via the Document Properties dialog, there are other options open to us. If you're familiar with editing XML or HTML then you could load the SVG file into a text editor and remove or change the "width" and "height" attributes on the <svg> element directly. Alternatively you can achieve the same via Edit XML Editor, by selecting the top entry in the left-hand pane (the <svg> element), and then editing the attributes in the other pane.

Both these approaches work, but they also have a drawback. If you have reason to go into the Document Properties dialog and edit other fields in there, you may find that these attributes return with their original values, leaving you playing a constant game of cat-and-mouse, having to remember to check or edit them just before saving each time you edit the file.

Instead I propose a more elegant solution. We know Inkscape wants some 'real' units to define its page size. But we also know that we want to change those units to

something more abstract when the page is viewed in a web browser. My solution, therefore, is to dynamically make that change when the image is loaded into the browser, by including a small snippet of JavaScript in the Inkscape file. Don't worry, it's not as tricky as it sounds…

- Open File Document Properties.
- Switch to the "Scripting" tab.
- Within that, switch to the "Embedded scripts" tab.
- There will probably already be an entry for "inkwebjs" which you can ignore.
- Click on the "+" button to create a new embedded script. It will appear in the list with a random ID.
- Select the new entry in the list.
- Put the cursor into the "Content" field, then enter the lines of code below.



The code itself consists of two nearly identical lines. We need to access the <svg> element first: since this is the main container for the whole file, it can be accessed as "document.documentElement" in JS (make sure you type in the right case). We then need to call the setAttribute() method, telling it the name of the attribute we wish to set or update ("width" or "height"), together with the new value it should have ("100%"). The complete code, therefore, looks like this:

```
document.documentElement.setAttribute("height", "100%");

document.documentElement.setAttribute("width", "100%");
```

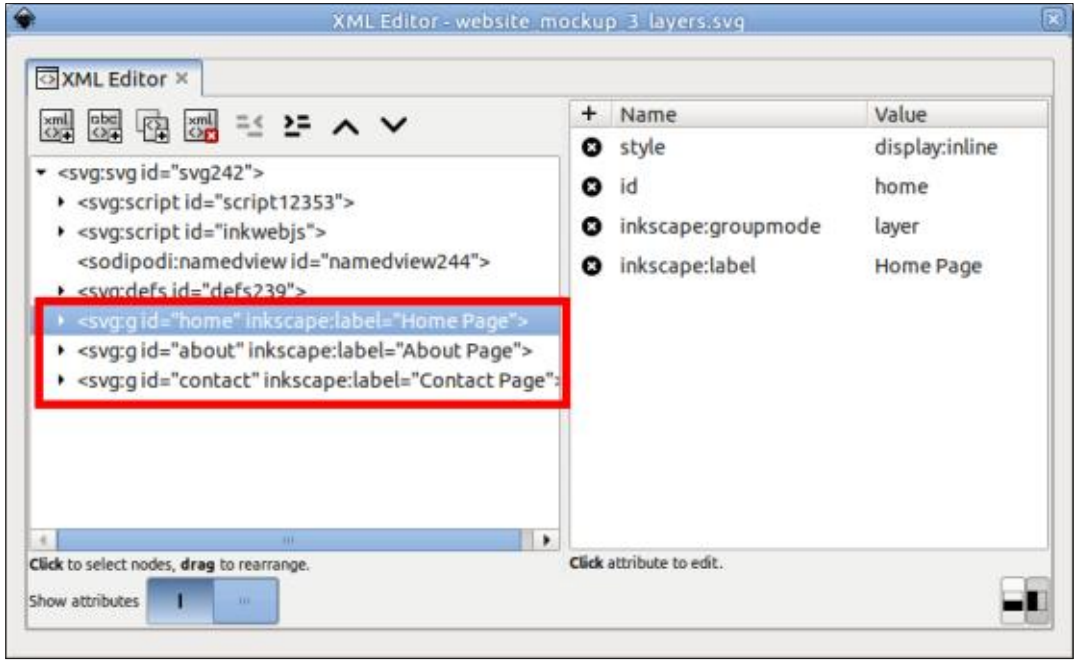
Because that code isn't inside a function, it will automatically run when the page loads in the browser. Job done, and we don't need to keep remembering to adjust any values each time we save an edit to the file.

That solves one problem: our content now scales to suit the width and height of the browser, while still maintaining its original aspect ratio. But it also exposes another issue with the way this extension works. You may have already seen the problem when playing with the resize handle in the responsive view: if the window size is tall enough, then we can also see some corners of the other pages in our mock-up (outlined in red in this image).

The reason for this is that our new width and height values tell the browser how to size the main content (the bit inside the viewBox), but the browser will happily render anything outside that area, if there's space on the screen to do so. We're simply seeing the parts of our document that fall outside the current viewBox region. There's an obvious and easy fix for this: just move the pages further apart in the original document. If they're way outside the bounds of the viewBox then they're far less likely to appear on screen when they shouldn't.

This is, of course, something of a band-aid. Although it works in most practical cases, there will always be some extreme aspect ratio which is sufficient for the other pages to just creep in at the edge. For most situations it's probably good enough, but it would be better if we could have a solution that caters for all cases.

What we require is some more JavaScript that hides all the pages in our mock-up except the one we're currently viewing. This requires us to have some simple way to define what counts as a "page" – and it strikes me that simply putting each page onto a separate layer is the easiest way to do that. Yes, I know Inkscape 1.2 has multi-page support, but using layers will work for older releases as well. In the case of our demo file, we need three top-level layers, one for each page. These will all be direct children of the <svg> element. To make our code more readable, we'll change the IDs for the layers to "home", "about" and "contact" using the XML editor.



To simplify the code we have to write, we'll also move all the layers back onto the canvas, stacking them on top of each other. That means you'll have to turn them on and off in order to edit the content, but it also means our JS code only has to deal with the visibility of the layers, not with also moving the viewBox around.

The code itself is a little more complex than the two lines we created earlier, though not by much. We'll want a function that accepts a layer ID, and which turns off all of the layers before turning on the one we've provided. We'll also need to call that function when the page loads to ensure the first layer is visible. Finally we'll call that function from each of the "interactive" elements in our mock-up, by attaching the function call to an onclick event or one of the other interactivity events we've seen before.

Let's begin by adding the function we need. Once again open the Document Properties dialog, select the "Scripting" tab, the "Embedded scripts" tab, and then the script we created earlier. Append a blank line or two, then add the following code (shown below - again, remember it's case-sensitive)

The first line in the body of this function uses the querySelectorAll() method to search the document for any elements that match the supplied CSS selector. The gobbledegook in quotes actually means "find any <g> elements that have an attribute called 'groupmode' (in any namespace), with a value of 'layer', but only if they're an immediate child of an <svg> element". This is good enough to pull out only our top-level layers, without accidentally catching any sub-layers or other <g> elements. The second line then iterates over the collection we've just created, hiding each of them by setting their CSS "display" property to "none".

The last two lines are similar, but only deal with a single element. This time we use querySelector() (without the "All") to just find an element with an id that matches the one we've provided. In CSS terms an ID starts with a "#" character, so we'll add that to construct the selector, meaning we can just supply a page name, such as "home", rather than having to pass "#home". The last line sets the CSS display property for this one element back to "inline" in order to make it visible.

Now we need to add another line that will call this function to make one layer visible by default when we load the file into the browser. This needs to happen after a short delay (we're using 100ms) to give the page a chance to load and settle down before we start poking with it. Append a line like this at the end, after the closing curly brace, replacing "home" with the name of your own initial layer.

```
function showLayer(id) {
  const layers = document.querySelectorAll("svg > g[*|groupmode=layer]");
  layers.forEach(layer => layer.style.display = "none");


  const layerToShow = document.querySelector("#" + id);
  layerToShow.style.display = "inline";
}
```

```
setTimeout(() =>
showLayer("home"), 100);
```

To avoid any visible glitching, it's also worth ensuring that your default page is the topmost layer within Inkscape, and is not hidden.

Now all that remains is to add a call to this function to each of our interactive elements. We'll have to do this one-by-one, without the convenient shortcuts we had when using the extension for adding multiple items at once. That makes it a little more fiddly, but I think it's worth it for the better end result.

- Select an interactive element.
- Open the Object Object Properties dialog.
- Expand the "Interactivity" section, if necessary.
- Choose the field for the type of interactivity you want (usually "onclick").
- If there's already code in there from the Interactive Mockup extension, remove it (this will be a call to the InkWeb.moveViewbox() function). We don't actually need anything from this extension any more!
- Type in a call to the showLayer() function, using the ID of the layer that should be displayed when the element is clicked – e.g. showLayer("home")
- Repeat for each interactive element on each page. Hint: you don't need to close and reopen the dialog, it'll update as you select each element.

Save the file and load it into your browser. If you've done everything correctly you should now have an interactive mock-up that scales correctly for any screen size, and doesn't suffer from other pages peeking into view. All it took was a few lines of JavaScript, and a single function call added to each interactive element. And to clarify, this is a replacement for the Interactive Mockup extension, not an enhancement of it: you can add this JS to a new Inkscape file to create interactive mock-ups without ever going near the extension.

Using the extension is definitely simpler, especially if you're not comfortable with JavaScript. If you know you only have to target one specific screen size, then that's probably the approach for you. But if you need the flexibility of scaling to suit any screen size, or prefer to keep all your pages stacked in layers rather than distributed across the canvas, these few lines of JS may be just what you need to create an interactive demo that suits your needs.

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**

Inkscape - Part 127

Over the past two articles, I've introduced the Web > Interactive Mockup extension, and gone on to show how it's possible to create the same effect – and with fewer problems – with just a minimal amount of JavaScript. This time, I'll be finishing this project by adding a little more code that will demonstrate some additional mock-up capabilities that simply aren't possible using the extension.

As a reminder, so far I've created a mock-up design consisting of three layers, each representing a different page in a website (which could equally well have been a design for an app, tutorial or presentation). By stacking the layers on top of each other, the JS code simply has to hide all the layers, then re-show the right one when the mock UI is clicked. This code is stored in the Inkscape document, and accessed via File > Document Properties, then the Scripting tab, the Embedded Scripts tab, and finally clicking on the randomly-generated Script ID in the list. Your code will appear in the Content pane at the bottom of the dialog – which is unfortunately not resizeable (you may wish to copy/paste between Inkscape and a text editor to make it easier to modify the code). After last month's additions, the code looks like that shown above.

```
document.documentElement.setAttribute("height", "100%");
document.documentElement.setAttribute("width", "100%");

function showLayer(id) {
  const layers = document.querySelectorAll("svg > g[*|groupmode=layer]");
  layers.forEach(layer => layer.style.display = "none");

  const layerToShow = document.querySelector("#" + id);
  layerToShow.style.display = "inline";
}

setTimeout(() => showLayer("home"), 100);
```

Within each interactive element, a single line of JS triggers the change to a different 'page' of the demo. These can be found by right-clicking on one of the elements, selecting Object Properties, then expanding the Interactivity section at the bottom of the dialog. For my examples, I'm just triggering changes on mouse clicks, so the 'onclick' field contains something like this:

```
showLayer("about")
```

That's all we needed to do in order to create an interactive mock-up that scales with the size of the web browser, and doesn't allow any non-active pages to be visible. Now let's push things a little further with the addition of some new features.

When looking at the pages of our mock website, it's clear that they have some common elements – in this case the whole header section. Wouldn't it be nice if we could keep those on a separate layer, so that any changes to those elements can be made in a single place, rather than having to apply them to each separate layer in our file? This is the sort of thing for which many applications use a 'Master' layer. Although not as politically charged as the use of 'master-slave' relationships in the computing world, it's nonetheless a term that can offend people, and which is tending to be phased out. So rather than propagate a troublesome word for no real gain, I'll be using the term 'Main layer'.

Our first step, therefore, is to split the file into a single Main layer, plus one additional layer for each page. The Main layer will contain all the common elements, and the others will contain just the page-specific parts. We therefore want our Main layer to be at the bottom of the z-stack, and to remain visible at all times. Here's how our existing three layers are split into the four we now need:

On the left we have the previous three pages. On the right we now have our Main layer at the bottom, with the three content layers above. I've added a green border around each of the content layers to indicate their extents: they each now have a transparent background, so without that it wouldn't be so clear exactly how they relate to the positions in the old pages. These green borders are a temporary addition while developing the mock-up, and are removed before the layers are actually used. Additionally, although I've spread the pages out for this image, in practice they're all stacked on top of each other within the document's viewBox, as before.

By showing the Main layer, plus one of the others at a time, we can therefore reproduce the same appearance as the three layers in the old version. All we need to do now is to modify our code to do the same thing on our behalf. To make the new code a little more readable, we'll first use the XML editor to change the ID of the new layer to 'main', in the same way that we changed the layer IDs previously. When viewed in the XML editor, the top level of our document now looks something like that shown above.

Looking back at our JavaScript file from earlier, we still want our function to perform the same basic task: hide all the layers, then show a specific one. Except now we also want it to show a second layer at the same time. It's these two lines that are responsible for re-showing the specified layer in the existing code:

```
const layerToShow =
document.querySelector("#" +
id);

layerToShow.style.display =
"inline";
```

We could simply add a similar pair of lines, hard-coding the ID in the querySelector() call as "#main". That would definitely do the job, but it's not very flexible. What if we want to show two 'main' layers later, perhaps to separate the text from the graphic elements? To give us this extra flexibility let's create an array of layers that we want to show, then loop over them to turn them all on. If you're not a programmer you may not be familiar with arrays: for our purposes you can think of them as a special type of variable that can hold a list of things. For this simple mock-up, our list will always contain 'main' and the id that was passed into the function, but you should be able to guess how you might extend it to include 'main-text' and 'main-graphics':

```
const layersToDisplay =
["main", id];
```

Now we need to step through the array, pulling out one item at a time to work with. As we pull each of them out (using a forEach() loop), we get to assign the value to a variable. By naming this variable 'id', we are able to reuse our existing code for finding and showing the layer. The end result is

```
function showLayer(id) {
  const layers = document.querySelectorAll("svg > g[*|groupmode=layer]");
  layers.forEach(layer => layer.style.display = "none");

  const layersToDisplay = ["main", id];
  layersToDisplay.forEach(id => {
    const layerToShow = document.querySelector("#" + id);
    layerToShow.style.display = "inline";
  });
}
```

something very similar to the code that was previously at the end of the showLayer() function, just with a little more wrapped around it (shown above).

The last thing we need to do is to make sure that all the clickable elements still call the showLayer() function, passing the correct ID, after the re-working of layers that we did earlier. It's particularly important to double-check any items that you've moved to the Main layer. Once you're happy, load the page into a web browser and ensure each of the elements functions as you expect it to when you click on it – if any don't, then double-check the code associated with them.

So far, so good. But on trying out your interactive mock-up, you may have noticed that the mouse pointer doesn't change to indicate that elements are clickable. It's a minor visual thing, but we can definitely improve it. There are various ways to tackle this, but they all end up with us needing a line of CSS that tells the browser what cursor type to use. We want this to apply to all the elements with an 'onclick' handler. In our SVG, these are all implemented using 'onclick' attributes directly in the XML content – which means we should be able to add a style rule using an '[onclick]' selector (matches any element with an 'onclick' attribute). That sounds like a perfect use for Inkscape's 'Selectors and CSS dialog', right?

Wrong. As I mentioned in part 112 of this series, the dialog doesn't recognise the attribute selector syntax. An alternative is to create a suitable <style> block directly in the XML, either using Inkscape's built-in XML editor, or by editing the SVG file in a text editor. Both of these approaches are a little awkward, especially if you're not already an XML aficionado. Instead, let's take a similar approach to the one we used for setting the height and width attributes: we'll write a short bit of JavaScript that will manipulate the document directly when it's loaded into the browser. We want this code to run once, on page load, so the following snippet should be added to the JS outside the showLayer() function. Just appending it to the bottom of the existing code is probably the easiest option.

```
let css =
document.createElementNS("htt
p://www.w3.org/2000/svg",
"style");

css.textContent = "[onclick]
{ cursor: pointer; }";

document.documentElement.appe
ndChild(css);
```

The first line of this code creates a new <style> block (in the SVG namespace) and assigns it to the 'css' variable. The second line just inserts a single CSS rule into the block: if any element has an 'onclick' attribute, the mouse cursor should be set to 'pointer' mode when it moves over the element. Finally, the third line inserts our new style block as a child of the <svg> element, after all the other content, where the browser will pick it up and automatically apply the rules.

There's one last thing I'd like to do to really make this mock-up work effectively. You may have noticed that each page includes a 'hamburger menu' at the top-right. Let's see if we can make that work, at least to some extent.

One approach would be to create six pages instead of three: a second version of each page would simply duplicate the original, but with the addition of the open menu. If you were solely using the Interactive Mockup extension, then that is pretty much your only option. But we're already up to our elbows in real JavaScript, so we have more subtle tools at our disposal.

We've already discovered that we can have more than one layer visible at a time, and rely on transparency to ensure that all the right parts are displayed at once. This is, after all, what we did when we added the Main layer. So why not do the same with the menu? In this scenario each 'page' consists of the Main layer, the relevant page layer, and an optional menu layer that sits on top of them all. Let's begin by designing the menu as a new layer at the top of the z-stack (shown above).

Each of the first three entries in the menu carries the same onclick handler as the equivalent on the main layer (in fact I copied and pasted the objects from there). We'll deal with the 'Sign Out' option later. Now the question is how to make the menu pop-up when we click on the hamburger button – but that's really not so tricky. If we use the XML editor to give the 'Menu' layer an ID of 'menu', then you can probably guess what this function (shown bottom right) will do.

All we need to do now is to call the showMenu() function from the onclick handler of the hamburger menu that lives on the Main layer. We're not calling the existing showLayer() function, so none of the existing layers is hidden. All that happens is that the Menu layer is displayed in addition to the others that were already visible – exactly what we wanted.

```
function showMenu() {
  const layerToShow = document.querySelector("#menu");
  layerToShow.style.display = "inline";
}
```

As it stands, the mock-up is good enough for demo purposes, but perhaps a little clunky in parts. When the menu is 'opened', for example, there's no way to 'close' it other than to navigate to one of the pages. One possible enhancement might be to add an almost-transparent rectangle to that layer, behind the main content. A suitable closeMenu() function, and an onclick handler added to the rectangle, would allow you to click outside the menu to close it. I'll leave that one as an exercise for the reader.

And what of that 'Sign Out' option? You could create another layer containing a mocked-up sign-out dialog, but do you really need to? Once you're comfortable with

showing and hiding things in JavaScript, it's always tempting to go a little too far, and turn your 'interactive mock-up' into something approaching a full UI demonstration. Sometimes that might be appropriate, but often it's better to do the bare minimum you can to help people to visualise how the final website or application might work. Too much detail or functionality can actually be a distraction, and can even inhibit further discussions or ideas. In this case, therefore, the 'Sign Out' option will simply get an onclick handler containing this:

```
alert("You are now signed out");
```

With that addition our simple mock-up is complete. The key thing to take away is that the code for doing something like this probably isn't as complex as you thought. While the Interactive Mockup extension can definitely be useful, you can easily get more functionality, and certainly a lot more flexibility, by just learning enough JavaScript to be able to target some elements in the page and selectively change their 'style.display' properties.

```
document.documentElement.setAttribute("height", "100%");
document.documentElement.setAttribute("width", "100%");

function showLayer(id) {
  const layers = document.querySelectorAll("svg > g[*|groupmode=layer]");
  layers.forEach(layer => layer.style.display = "none");

  const layersToDisplay = ["main", id];
  layersToDisplay.forEach(id => {
    const layerToShow = document.querySelector("#" + id);
    layerToShow.style.display = "inline";
  });
}

function showMenu() {
  const layerToShow = document.querySelector("#menu");
  layerToShow.style.display = "inline";
}

setTimeout(() => showLayer("home"), 100);

let css = document.createElementNS("http://www.w3.org/2000/svg", "style");
css.textContent = "[onclick] { cursor: pointer; }";
document.documentElement.appendChild(css);
```

If you do want to reproduce something like my mock-up, perhaps as a bit of practice to get a feel for the JS side of things, here's the complete code we ended up with in the Document Properties dialog for your convenience (see above).

In addition to that, each clickable element on the page has a single function call in the 'onclick' field of the 'Interactivity' section at the bottom of the Object > Object Properties dialog. In most cases, this was just a call to the showLayer() function, passing in the name of the page to display (e.g. showLayer("contact")). In the case of the hamburger menu, it was a call to the showMenu() function. And our final addition was a call to the browser's built-in alert() function for the 'Sign Out' option.

When you take a step back and look at it, that's really quite a lot of functionality in this interactive mock-up, for not a huge amount of code. But we're done with this now – and with 'Interactive Mockup' being the last of the new extensions, we're done with the features that were added to Inkscape 1.x. Next month, I'll start what is sure to be a long series on the new features and additions in Inkscape 1.2.x.

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**

Inkscape - Part 128

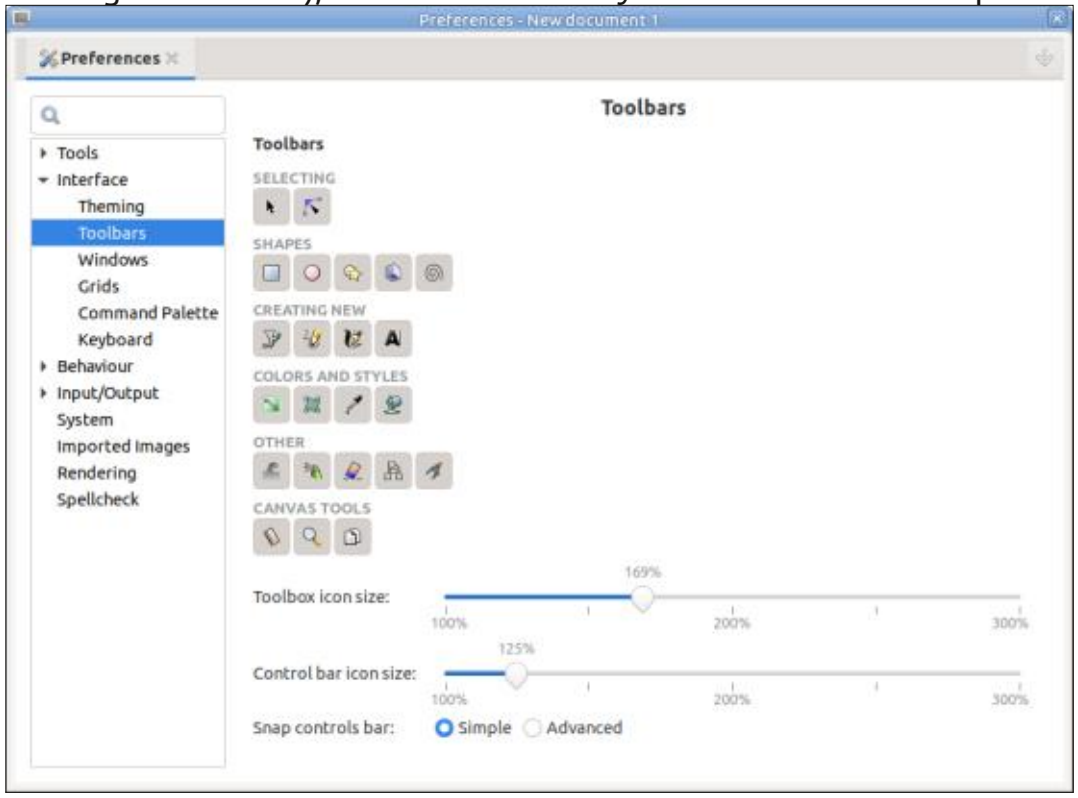
After months of looking at the features added and changed in Inkscape 1.1.x, it's finally time to move on to the 1.2.x series. As I write, the project has just announced the second point release in this series, which features a number of bug-fixes. Inkscape 1.2.2 can be downloaded from the project's website (https://inkscape.org/) as usual, with Linux versions available as an AppImage, Snap, or via a PPA. I like to keep multiple versions installed – which is a must if you write a monthly column about the program – so for the past few releases, I've tended to use the AppImage. In theory, however, it shouldn't make much difference which you use.

This month, I'm going to take a look at a couple of the more significant UI changes that have taken place with version 1.2. First, however, I'm going to mention a part of the UI that, unfortunately, hasn't changed at all: a problem with the icon sets.

When icon themes were added to Inkscape, it seemed like a great opportunity to provide a symbolic icon set for those people who prefer stark simplicity, while keeping the older, more colourful images for users who find it easier to distinguish between different icons if they're not all similarly coloured. I fall into the latter category, so was disappointed when 1.1 presented me with a mixture of flat, symbolic icons alongside the colorful ones – replacing some of the old standards in the process. Unfortunately. version 1.2 continues in the same manner (top left)

As you can see, the system-level operations – loading, saving, printing, and clipboard – all appear as flat symbolic icons, while other tools and buttons get the more colourful icons of the past. This screenshot was taken with the 'Hicolor' theme selected (Edit > Preferences > Interface > Theming > Change icon theme), but a similar problem occurs with the 'Tango' theme. If you find yourself in a similar situation, then try switching to the 'Use system icons' option: on my Ubuntu Mate 20.04 box, this has allowed me to return to a consistently colorful set of icons in the toolbars (bottom left)

On the subject of icons, following the Theming section in the Inkscape preferences is a new entry labelled 'Toolbars'. This panel



provides a number of extremely welcome controls for changing the size and content of the main toolbars in Inkscape.

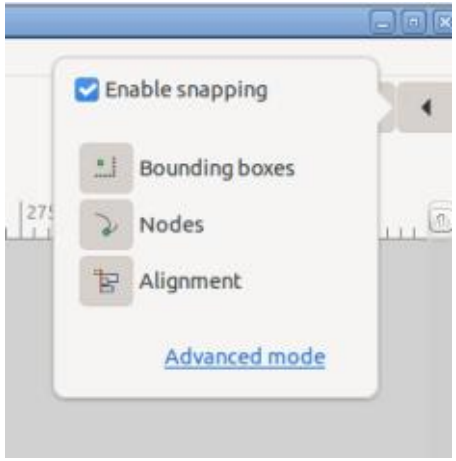
First of all are a group of toggle buttons that allow you to turn each of the individual drawing tool icons on and off in the main toolbox. This is great news for anyone on a small screen who wishes to slim down the number of icons to something that better fits the available space. It's also useful as a means to hide infrequently used tools more generally – at last I can prevent that pesky 3D Box tool from taking up space, and I can hide the Pen and Calligraphy tools to stop me inadvertently clicking them when I meant to choose the Bézier tool. When hiding icons, make sure you know which tool is which, as they all share the same generic tooltip in this dialog. If you do choose to turn off some icons, those tools aren't lost to you: you can still access them via their keyboard shortcuts. It's a shame, however, that there's not a shortcut to temporarily display all the icons, which would let you pare the toolbox down to the most commonly used features while still giving you an easy way to get to the less frequently used ones.

Below these toggles are separate sliders for adjusting the size of the Toolbox (the one on the left, with the tools in), and the Control bar (the two-row toolbar along the top of the window). Being able to adjust these independently is a nice touch.

Finally, there's a control to switch between 'Simple' and 'Advanced' modes for the Snap controls bar – but this really isn't the place you want to visit if you want to toggle that option… at least not until version 1.3 comes out. But to understand that statement, we first need to look at what's happened with the Snap controls in Inkscape 1.2.

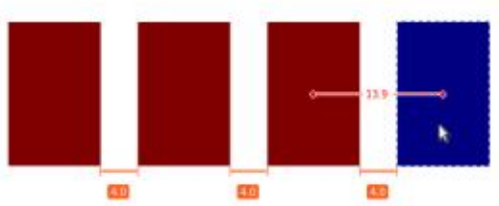
The snap toolbar has been a part of Inkscape since the beginning of the project. By default, it appeared on the right-hand side of the window in the past few releases, but selecting View > Custom would place it at the top. This option has been removed from version 1.2 because the toolbar itself no longer exists. Instead, the snap controls are now hidden in a pop-up menu at the top-right of the Inkscape window.

The controls themselves are so unremarkable in appearance that I've had to include the window border and the end of the horizontal ruler in that screenshot, to help you orient yourself as to where these UI elements appear. The button on the left is a simple toggle to turn snapping on and off as a whole – the same effect as using the '%' keyboard shortcut. To the right of that is a pop-up menu for choosing which options should apply when snapping is enabled, and it's here that things get a little more interesting. Let's look at this menu in its default, 'Simple' mode:



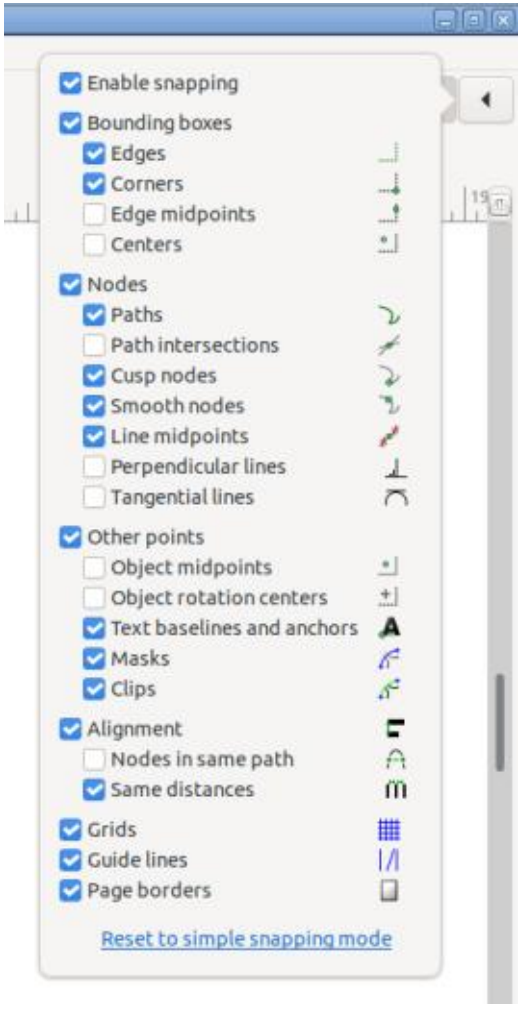
First, we have a checkbox that duplicates the functionality of the snapping toggle, or the '%' shortcut. Also having this option in the menu itself means one less mouse-click if you need to enable snapping and adjust the options at the same time. But the options you can modify are clearly somewhat limited: you can just turn various classes of snapping on and off – for snapping to bounding boxes, nodes, and 'alignment' – but you're not given any details about which snap targets are actually enabled for each of those three entries.

The 'alignment' option is a new addition, which makes it easier to position objects relative to others by displaying temporary alignment guides and optional dimensions as you drag objects around the canvas.



Note that the dimensions aren't enabled by default – usually you just get the orange and red guides shown in the screenshot. For users

who just want things to align neatly but aren't worried about the specific dimensions, that's fine. But if you do want to see the numbers as well, you need to enable the 'Show snap distance in case of alignment or distribution snap' option in Edit > Preferences > Behaviour > Snapping. Excuse the pun, but couldn't the developers have come up with a more 'snappy' title for that option?



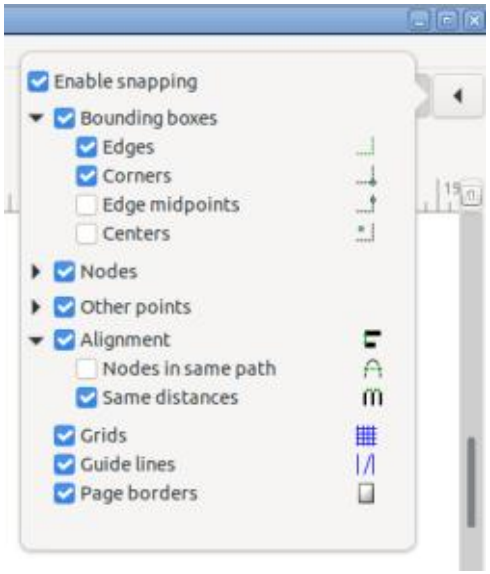
You've undoubtedly noticed the 'Advanced Mode' link at the bottom of the pop-up. Clicking that reveals something more like the snap controls bar from versions past (shown left), with many, many snapping options to choose from.

I won't describe all these options – most of them have been covered previously in this series. If you want complete control over what parts of your drawing should act as snap targets, then this is the place to come. And if you just want to return to the more straightforward (and limited) set of three toggle buttons, the 'Reset to simple snapping mode' link at the bottom of the pop-up will do that.

But that's where the first of my issues with this UI arises. Clicking that link doesn't just switch you back to the simpler view, it also resets your snap settings. You may have spent some time setting up just the right combination of snap targets for the way you work, but switch back to simple mode and all your selections are thrown away. You might assume that toggling, say, the Nodes button in simple mode would do the same thing as toggling the Nodes checkbox in advanced mode – i.e. just turn your entire combination of options on or off. But that's not the case: as soon as you switch back to Simple mode your individual combination of choices is lost entirely and replaced with whatever defaults the developers have chosen.

Simple mode also fails to offer options for snapping to 'Grids', 'Guide lines' and 'Page borders' – all fairly common snap targets that beginners might find useful, but which have to be accessed via the advanced mode. This means that, for many users, a trip to advanced mode will sometimes be a necessity, without it being obvious that returning to simple mode for day-to-day use will lose their settings.

I quite like the idea of a pop-up for snapping options. It's certainly a lot clearer to have a title alongside each option, rather than the slow hover-for-a-tooltip that was often required over almost every option in the toolbar from earlier versions. But I feel a better UI would have been to make the sections in the advanced mode collapsible. That would allow users to reduce the list down to the main toggles for a simpler interface, but let them expand each section out for full control when needed. Here's a quick mock-up of how it might appear in practice, with a couple of the sections expanded.



With what's currently in Inkscape 1.2, however, I really can't recommend the use of Simple mode at all. If you need to use snapping then it's almost inevitable that you'll need at least one option from the Advanced menu at some point, so my advice at the moment is to only use the Advanced mode, and never switch back to Simple mode at all. It may seem more complex initially, but it still beats having your selections thrown away without warning.

So, back to the conundrum of the 'Snap controls bar' option in Edit > Preferences > Interface > Toolbars. I'd pretty much ignore it, use the link at the bottom of the pop-up snap menu to enable advanced mode, and then leave things alone. But if you don't like the new pop-up menu at all, it's worth knowing that the option to switch back to an old-fashioned snap toolbar is probably coming in version 1.3 – at that time the setting in the preferences dialog will gain a third option to let you use the toolbar rather than the pop-up.

While talking about snapping, it's worth pointing out a few more snap-related changes that have taken place in 1.2. One relatively minor difference is that the snap settings are no longer stored as part of the document itself, but are stored globally for Inkscape as a whole. To be honest I doubt this change will really affect anyone in any practical way, and it makes sense to take that Inkscape-specific metadata out of the SVG file.

One less well-known feature that has been removed is the ability to set snap tolerances for some individual snap types. It used to live in the 'Snap' tab of the Document Preferences dialog, but given that snapping is no longer a per-document feature, that entire tab has been removed. In theory it was nice to be able to prioritise certain snap types, by giving them a larger target radius so that they would kick in as your pointer wandered towards the target without requiring too much accuracy in your aim. In practice, however, it was likely a rarely used feature that won't really be missed.

On the plus side, the removal of that tab has also meant that a couple of rather hidden snap target types have now been promoted into the pop-up menu instead. Previously the options for snapping perpendicularly and tangentially were in the Document Preferences, but now live as part of the 'Nodes' section in the Advanced pop-up. But that adds yet another reason to only use the Advanced mode, and skip the Simple option entirely.

Next month, I'll continue to look at – and offer an opinion on – some of the other UI changes that have taken place in the program with this release.

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**