Inkscape - Part 73

Over the past six years, this series has detailed just about every part of Inkscape, from basic shapes through to more complex features such as filters and path effects. With nothing more to cover, it's time for this column to take a sharp turn from its previous tack: I'm going to look at how you can use graphics you've created in Inkscape on the world wide web.

On the surface, this might seem like a simple task – worthy of an instalment or two at most. But SVG is unlike any other graphics format; its XML heritage lends it to the same kind of manipulation with CSS and JavaScript as HTML, allowing for graphical tricks that go way beyond simply displaying a static image on a page. At the same time, you still have the option to treat SVG as a dumb image format, if that's a better fit for your requirements.

Broadly speaking, there are four ways to include an Inkscape-created image in a web page. Each has its own pros and cons, which are outlined in the table shown above.

Method	Scalable	CSS animations	JavaScript	Difficulty
Convert to raster	✘	✘	✘	Low
Link to SVG	✔	✔	✘	Low
Object element	✔	✔	✔	Medium
Inline SVG	✔	✔	✔	High

The first method is to simply avoid the whole SVG question entirely and export your image to a PNG file – possibly even converting it to a JPEG afterwards to reduce the file size, if the image doesn't require transparency. The resultant raster image can be included in a web page via an <img> tag, used on social media sites or blogs, and generally be treated the same way as a photo from a phone or digital camera. Because you're effectively creating a snapshot of the image as it appears in Inkscape, you also don't have to worry about missing fonts, flowed text, or mismatches in browsers' rendering engines. The downside is that the exported raster image is no longer infinitely scalable in the same way as a vector image. You also lose any concept of the individual objects that made up the image, so it's not possible to manipulate them individually using CSS animations or JavaScript.

The second approach relies on the fact that most modern browsers do a pretty decent job of rendering SVG – so why not just use the SVG file directly, rather than converting to a raster format first? This is as simple as referencing an SVG file in your <img> tag, rather than a PNG or JPEG. Because the internal structure of your SVG is preserved, the browser knows about individual objects, so can animate them using CSS. In theory it could also allow JavaScript, but that's prevented when using images like this, for security reasons.

What does security have to do with an SVG image, you may wonder. Consider that many social media platforms, forums, and other websites, will let you upload your own SVG file, then expose it to other people on the site by using an <img> tag. This would effectively give a malicious user a means to run their own JavaScript under the guise of the hosting website. The code could capture user's keypresses, spoof a login dialog to record passwords, or redirect the user to another site entirely. Given that being able to upload an SVG file to a site and

have it rendered as an image is generally a good thing, but that unfettered permission to run JavaScript in such a file could easily be abused, browsers take the sensible approach of allowing CSS animations to run (giving SVG an edge over raster images), but disallowing any JavaScript.

Unfortunately this method does come with one big drawback – though it's a problem that diminishes every year. Because you are relying on your browser's rendering engine to draw each object in your image, you no longer have complete control over the appearance of your image. For simple drawings, this isn't generally a problem. But add in more advanced features, such as filters, and the output of the browsers begins to diverge.

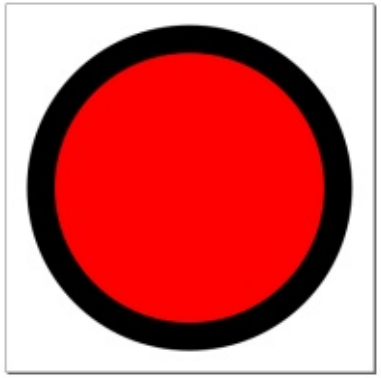
Text rendering is a particularly significant problem. If the person viewing the image doesn't have the same font installed on their machine that you used to create the document in the first place, it will be replaced with a different font. In HTML, this can be annoying, but usually the browser adjusts the page layout a little and no real harm is done. In an image format, however, it can be much more of a problem. Consider something like a comic strip: if the text is rendered at the wrong size it might not fit the speech bubbles; if it's rendered in a different font it could change the feel of the narrative entirely. For this reason, I post all my comic strips as PNG or JPEG images, and link to the SVG files separately where I want to make the source files available for people to download.

The third approach is essentially another way of linking to an SVG file, but this time it uses the <object> element rather than an <img>. Whereas forums and social media sites are happy to expose a user's file via an <img> link, they tend not to do the same via <object> – so, if you want to take this approach, you need to be able to edit the underlying HTML directly. This usually implies that you are creating content for your own domain, rather than just uploading something to a third-party site. What you gain from this extra requirement is an enhanced level of trust from the browser, allowing JavaScript in your image to execute. With sufficient knowledge and experience, it's even possible to create two-way interactions between code on the page and in your image. But, again, you're also at the whim of the browser in terms of the final rendering of your file.

The last approach is a relatively recent option. For years, SVG was neglected by the web developer community, largely because Microsoft refused to support it in earlier versions of Internet Explorer, at a time when it commanded the majority share of the market. But with later releases, Microsoft softened their stance, adding SVG support and paving the way for SVG to be promoted to a first-class citizen in the world of HTML5. Now, you can put a chunk of SVG code directly into your HTML and it will just work. And if you're hand-coding your SVG, you no longer have to concern yourself with the complexities of namespaces in XML – just use the SVG tags in the same way as you would HTML.

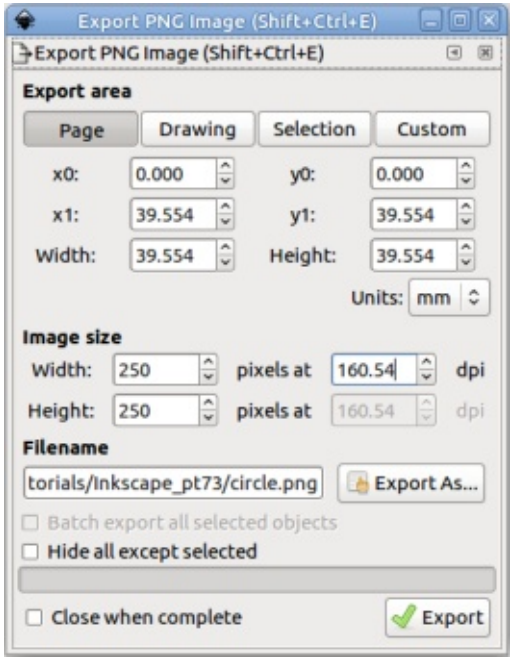
With this last method, the SVG is a direct part of your web page, so writing code to manipulate the image is much simpler than with the <object> approach. But it does mean that your SVG code is right there in the middle of your HTML. If all you're doing is drawing a couple of simple objects, that might not be so bad, but including all the code from a complex drawing can quickly overwhelm the rest of your page, making it harder to work with the surrounding HTML. And, once more, your browser's rendering engine might make poor work of your image.

Enough theory, let's put it into practice. Next month, I'll describe how to use <object> and inline SVG, but, this time, we'll deal with the <img> methods. Begin by loading a simple image into Inkscape, or creating one from scratch if necessary. I'm going to work with this basic circle (the square around it is the Inkscape canvas):



Let's export this to a PNG image via the File > Export PNG Image…

menu item. This opens a dialog in which to set the details for the export:



I covered this dialog way back in part 12 of this series, but, for the purposes of creating a PNG to use online, you can ignore many of the options. You do need to decide what to export using the buttons at the top: usually this will be the entire page, the entirety of your drawing (which could be larger or smaller than the page), or a rectangle that encompasses the currently selected objects. I've chosen to export the whole page, as I want to include a little bit of blank space around my circle.

Now comes the most important decision: how big do you want your image to be? This is set in the Image Size section, by adjusting the image dimensions or dpi (dots per inch). For web use, you can generally ignore the dpi option and go straight for setting the width and height in pixels. If you're creating an image for a profile picture or banner ad, you'll usually be told the dimensions your graphic must be. Note, however, that you can't set these fields independently, so you'll either have to create your design to the right proportions, or export a larger area that you can then crop in a bitmap editor such as The GIMP or Krita.

The last setting required is the filename to export to. Click the Export As… button and navigate to your destination folder in the file selector. Enter a filename in the selector, making sure it ends in ".png", and then use the file selector's Save button to return your selected location and name to the export dialog. Despite the name of the button, this won't have saved a PNG file anywhere; it's just saved the export location. In order to actually save the file, you have to click the Export button.

You should now have a PNG file on disk that can be uploaded to just about any site that accepts user images. If you want to use it in your own HTML page, that's easy too. If you're already familiar with writing HTML then you almost certainly know how to link to your image. But if you've never written a line of HTML in your life, why not give it a try now – it's not as tricky as you might think. In a text editor type the code shown below.

Modify the "circle.png" string to suit the filename of your own PNG, and save the document into the same directory as the exported image, making sure to give the filename an extension of ".html". Open the web browser of your choice, press CTRL-O to open the file selector, and load your HTML file. You should see a text heading with your PNG image displayed below it.

That wasn't too tricky, was it? Linking directly to an SVG file is just as simple. If your original Inkscape file isn't in the same directory as the HTML file, use File > Save a Copy… to put a copy of it there. When you look in the directory you should now see your HTML file, your exported PNG, and an SVG file. To get the latter to appear in your web page, just add the following lines below the existing image link:

```
<h1>IMG tag (SVG)</h1>

<img src="circle.svg"></img>
```

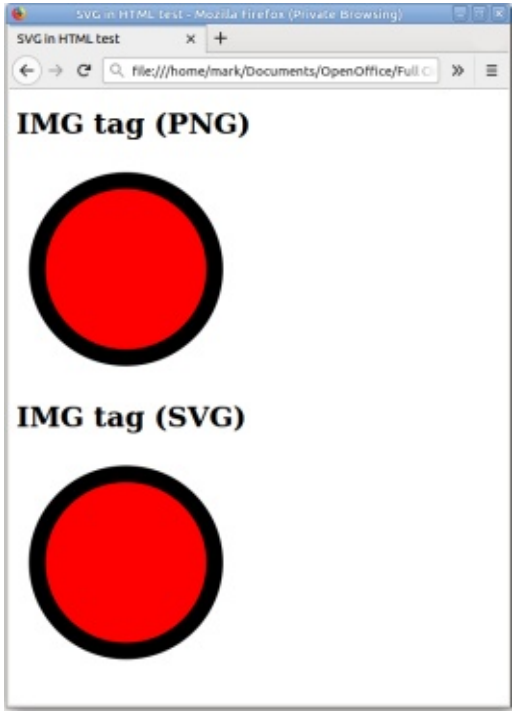
Save the file, switch to your

```
<!DOCTYPE html>

<html>

<head>
  <title>SVG in HTML</title>
</head>

<body>
  <h1>IMG tag (PNG)</h1>
  <img src="circle.png"></img>
</body>

</html>
```

browser, and press F5 to reload the page. You should see both your PNG image and the SVG version. There's a good chance that the images aren't the same size, but SVG files are scalable so we can fix that easily. Given that my PNG was exported at 250px in width, I can modify the <img> tag for the SVG file to set it to the same:

```
<img src="circle.svg"
width="250"></img>
```

Another press of F5 in the



browser and you should find that both images are the same size. But what if we wanted them both to be larger? Try setting a width attribute on both of them, with a value of 1000 or more. Notice how blocky the PNG becomes, whilst the SVG is still rendered as smoothly as possible. That's the main advantage of using an SVG file in a web page instead of a PNG.

Linking to an image via an <img> tag is a common way to display user-submitted images on the web. But, when coding your own pages, images are often used in other ways as well, typically via CSS. You can use an SVG file wherever you would usually use a PNG in these cases. For example, let's fill the background of our web page with copies of the SVG file using this chunk of CSS, inserted between the </head> and <body> sections of the HTML code:

```
<style>

body {

  background:
url("circle.svg");

  background-size: 50px;

}

</style>
```

Reload the page to see the result. As a rule, modern browsers will let you use an SVG image wherever a raster is allowed. But although the browser might be happy to do that, many websites still only let you upload pure raster formats such as PNG, GIF and JPEG. If you write your own HTML you have full control over what formats to use, but if you're trying to upload an image to a third party site you may find your options limited. If you can use an SVG, though, I encourage you to do so. Only by ever more SVG content making its way online will browser vendors have an incentive to provide better support for the format, and the SVG working group have a chance to introduce more of the new features that would make this useful format even better.

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

Inkscape - Part 74

Last time I showed you the basics of including an Inkscape image in a web page by treating it as an image object in an HTML <img> tag or via CSS, either by exporting as a PNG, or by linking directly to the original SVG image. This time I'm going to consider the other two methods of getting an Inkscape image into a web page: via the <object> tag, or through the use of inline SVG.

At first glance the HTML <object> tag isn't too dissimilar to the <img> approach. The "src" attribute is replaced with "data" and, because <object> tags can include more than just images, we have to include a "type" attribute that contains the SVG MIME type so that the tag knows what format the data will be in.

```
<!DOCTYPE html>

<html>

<head>
  <title>SVG in HTML</title>
</head>

<body>
  <h1>OBJECT tag</h1>
  <object
type="image/svg+xml"
data="circle.svg"></object>
</body>

</html>
```

SVG as an <img> is sandboxed by the browser such that any JavaScript in the file won't run and no external resources can be loaded (i.e. no web fonts). Even code in the hosting page won't be able to dig into the contents of the SVG to dynamically change the image. In fact the SVG image behaves largely the same as a rasterised version of the same picture.

With the change from <img> to <object> comes vastly more power. The SVG file is treated as a complete, self-contained, document which is allowed to execute JavaScript, load external resources, and even communicate with the host page. In future instalments of this series we'll explore some of those possibilities in more detail.

The fourth and final approach to including SVG in your web page is so-called "inline SVG". Inline, in this case, simply refers to directly intermingling the XML code of your SVG file with the HTML code of your web page. For example, here's the code for an HTML page with an inline SVG image of a red circle with a thick black stroke:

```
<!DOCTYPE html>
<html>
<head>
  <title>Inline SVG</title>
</head>

<body>
      <svg>
            <circle
            fill="red"
            stroke="black"
            stroke-width="11"
            cx="75"cy="80"
            r="60">
            </circle>
      </svg>
</body>

</html>
```

At a glance the code looks pretty straightforward. Most of it is just the sort of content you would expect of a typical web page, except that the contents in the <body> tag consist of an <svg> block, with some SVG code inside it. The SVG itself is just a single <circle> element with a few attributes to create the desired result. It's all very neat and tidy. Perhaps too neat and tidy, in fact.

This code was hand crafted to ensure that only the bare minimum appears in the SVG block. A typical SVG file from Inkscape, on the other hand, is filled with all manner of extra content, most of which isn't necessary when using it in this way. Here's a screenshot from an HTML page that contains both the hand-crafted code above, and a copy of the XML created by Inkscape for the same basic shape:

Inline code (hand-crafted)

Inline code (Inkscape)

Can you spot the difference? No, me neither. But the top circle required SVG code amounting to 107 characters, whereas the second used 2118 – nearly twenty times as much code for the same result! Surely Inkscape can do better than that?

There are two reasons for the bloated file size. The first is simply that a hand-coded SVG image always has the potential to be smaller than a computer-created equivalent. Inkscape is a general purpose tool used for all manner of tasks, so it creates general purpose files which are good enough for most circumstances, but which lack the targeted optimisations that would bring the file size down. Unless you want to shave every last byte of flab from your files it's not usually worth worrying about this first problem. But the second issue is definitely worth looking at, as it's responsible for far more of your file's expanded girth: XML namespaces.

XML stands for eXtensible Markup Language. Yes, I know. I can only guess it's because ".eml" was already taken as a file extension for email files, and "X" is way cooler than "E". But that X or, more to the point, extensibility, is at the heart of the file bloat we can see with Inkscape. One of the more sensible design decisions in XML was to allow different XML-based languages to coexist, even inside the same document. You can see the remnants of this decision in the code above, where we switch from HTML (a not-quite-but-close-to XML language) to SVG, without the browser kicking up a fuss. But in this specific case HTML has been extended to allow the inclusion of SVG elements; XML allows such inclusions in a more generic manner, without the need to extend the host language in any way.

Let's imagine that we wish to combine two XML languages in a single document. The first is our old friend SVG. The second is a new language I've just made up for garden designers – let's call it GardenML. Before you cry foul, XML is explicitly intended to allow you to create your own domain-specific languages, so although my example is a little contrived, it's not without precedent in the real world. So, back to the plot: we've got an SVG file, filled with <path> elements that describe lines in an image. But we want to include our GardenML content, which is filled with <path> elements that describe… well, paths in a garden. The sort you walk on, made up of bricks or stone slabs, and certainly not the same thing as a path in SVG.

How does an XML consuming program reconcile these different uses of the same element. How does it know when you're using an SVG <path>, and when it's a GardenML <path>? The answer is a thing called a "namespace". All XML elements have a namespace, but it's usually set as a default for the whole document, then just implicitly used for each element. Here's an SVG document which does exactly that:

```
<svg
xmlns="http://www.w3.org/2000
/svg">

      <path d="M 130,70 A
      60,60 0 1,1 130,69.9 Z"

      fill="red"
      stroke="black" stroke-
      width="11" />

</svg>
```

Notice the "xmlns" (XML NameSpace) attribute in the <svg> tag? That defines the default namespace for the document. The value is a URI – a Uniform Resource Identifier. It may look like a web address, also known as a URL (Uniform Resource Location), but it's subtly different. Whereas a URL points to the location of a resource such as a web page, a URI is just a unique identifier which doesn't have to have a corresponding resource at the specified address (though they often do). In other words, a URL is a path to a real document, whilst a URI is a unique string used to make sure that each namespace is distinct from all the others. A URI may also be a URL, but doesn't have to be.

When a browser sees that the document's default namespace is "http://www.w3.org/2000/svg" it knows that any <path> elements are SVG paths, not GardenML paths. But suppose we want to create a file that will both display in the browser and can be interpreted by something that understands GardenML. In that case we need to define more than one namespace in the <svg> element, give all but the default one a handy shortcut name, then prefix the elements and attributes with the shortcut as appropriate.

```
<svg

xmlns="http://www.w3.org/2000
/svg"

xmlns:garden="http://fullcirc
lemagazine.org/GardenML">

      <path d="M 130,70 A
      60,60 0 1,1 130,69.9 Z"

            fill="red"
            stroke="black"
            stroke-width="11"
            />

      <garden:path
      start="back door"
      end="shed"
      type="gravel" />

</svg>
```

This time the default namespace is still SVG, so the <path> element still renders in a web browser. The second <path> element, however, has a prefix of "garden", identifying it as being from the GardenML namespace that's defined in the <svg> tag. The browser doesn't try to render the second path because it doesn't know what to do with elements in that namespace – but equally it doesn't complain about them either.

By default attributes are in the same namespace as their element – so in the previous example the

```
<svg
   xmlns:dc="http://purl.org/dc/elements/1.1/"
   xmlns:cc="http://creativecommons.org/ns#"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:svg="http://www.w3.org/2000/svg"
   xmlns="http://www.w3.org/2000/svg"
   xmlns:sodipodi="http://sodipodi.sourceforge.net/DTD/sodipodi-0.dtd"
   xmlns:inkscape="http://www.inkscape.org/namespaces/inkscape"
   width="40mm"
   height="40mm"
   viewBox="0 0 40 40"
   version="1.1"
   id="svg8"
   inkscape:version="0.92.2 (unknown)"
   sodipodi:docname="circle.svg">
```

"d" attribute is in the SVG namespace, whereas the "start" attribute is in the GardenML namespace. But you can prefix individual attributes as well, should you need to. In this example we've got an SVG path to which I've added some custom attributes of my own:

```
<path d="M 130,70 A 60,60 0
1,1 130,69.9 Z"

      fill="red"
      stroke="black" stroke-
      width="11"
      garden:start="gate"
      garden:end="front door"
      garden:type="paved"
/>
```

Now all of this might seem a little esoteric, but there are two reasons for explaining it. The first is that namespaces are integral to XML documents, so when we get round to manipulating Inkscape files using JavaScript a little later in this series you'll be glad of a good grounding in the topic. The second is that it explains why our Inkscape generated files are so much larger than a hand-crafted version. Here's the opening <svg> element of a typical Inkscape file:

Notice all the different namespaces being defined, and a couple of them being used on the last two attributes. The "svg" namespace you now know about, but what of the others? The "inkscape" namespace is used to store extra attributes that hold Inkscape-specific data. Without these, Inkscape would be limited to the features defined in the SVG spec, and wouldn't be able to provide extra capabilities such as Live Path Effects. The "sodipodi" namespace serves a similar purpose – Inkscape was forked from the Sodipodi program many years ago, but its history lives on in attributes that date from before the split.

The remaining namespaces are used for the metadata about your document that you can enter into the Document Properties dialog. There are several of them because they each refer to a different XML language. Inkscape could have just used it's own namespace for all of them, but by referring to other well-known languages it improves the ability of the metadata to be automatically parsed and understood by indexing programs or other XML tools. It does bloat

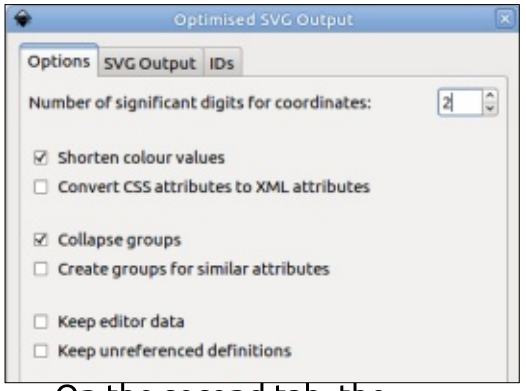
the size of the file quite considerably, though.

An obvious way to reduce the file size, therefore, is to remove the additional namespaces, elements and attributes. Doing this obviously compromises the SVG file in various ways, from removing the editability of Inkscape-specific features, to loss of metadata. Inkscape does, however, provide a couple of options for doing this for you. They're both alternative formats in the File > Save As… dialog, but in practice I recommend still saving a normal Inkscape SVG file, then creating your slimline version using File > Save A Copy…. This approach avoids the problem of forgetting to save in the full-fat format when you've made an edit, and losing data in the process.

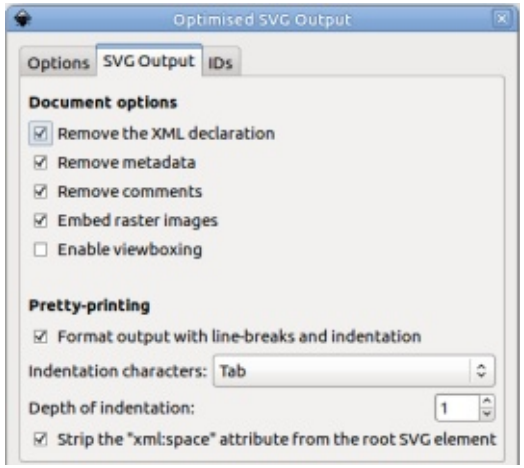
Your first choice is to save as "Plain SVG". This strips out the proprietary Inkscape and Sodipodi namespaces, and their associated elements and attributes. It still leaves the other namespaces intact, so the file will still contain any metadata you entered into the Document Properties dialog. This option is ideal for use when linking to an SVG file via the <img> or <object> tabs, as it removes the data that the browser doesn't understand, but leaves any copyright or license information that you may have added. With my simple example file, saving as Plain SVG reduced the file size from 2.1kB to 1.3kB.

The other choice is to save as "Optimised SVG". This presents an additional dialog that lets you choose from a wide range of optimisations that can potentially reduce the file size. Be warned that this option can lead to quite extensive changes in the structure and content of your SVG file, so always test the resultant file to make sure you haven't optimised away something important. Be particularly wary of reducing the number of significant digits too far, as this will affect the fidelity of your image.

When it comes to the additional namespace-related data, there are a couple of key fields to pay attention to. On the first tab, the "Keep editor data" checkbox determines whether the Inkscape and Sodipodi namespaces are preserved.



On the second tab, the "Remove metadata" option will lose all the other namespaces, together with any metadata you may have added to the Document Properties.



Also on the second tab, pay attention to the Pretty-printing options. Turning this off can reduce the size further, but if you're trying to create a file to put inline in an HTML page, a little formatting can make it a lot easier to work with.

So how well does Optimised SVG stack up against hand-written code? It doesn't reach the 107 characters of my carefully crafted version, but at 277 characters of fairly readable SVG it doesn't do too bad a job. For most cases where you want to inline your image into an HTML page it will be good enough, with far less scope for errors than trying to write everything by hand.

Phew! We've covered quite a lot this month, from <object> tags to XML namespaces. If all you want to do is to include a static image in a web page, stick with the methods described last time. But as we move on to more advanced topics, such as incorporating code into our files, the details of this article will become more relevant.

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

Inkscape - Part 75

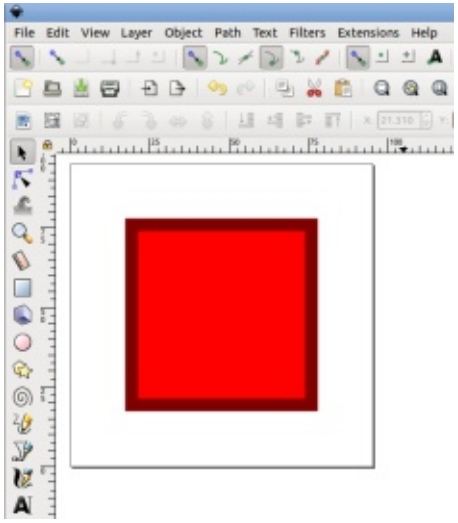
Having looked at the four main ways of including an SVG file in a web page, we're going to start examining what you can actually do with SVG files that you can't achieve with simple raster graphics.

Speaking of simple raster graphics, that's the first of our four ways of including an image, and it's one that I'm going to dismiss immediately. Once you convert your SVG to a raster graphic, be it a png, jpeg, or gif, it becomes no different in capability to a photo from a digital camera. You can display it in your web page, but that's about it. Sure, with modern CSS and JavaScript you can make it respond to mouse events, and can use some SVG-style techniques such as masking, clipping and (basic) filtering – but none of those abilities come from the image itself. Your raster image is essentially a rectangle that can only be modified as a single entity, losing all notion of the individual objects in your original SVG file.

So let's move on to the second way of including an image: by linking directly to the SVG file. For this example, we're going to use the following simple HTML page, which just contains a link to our SVG file in an <img> tag:

```
<!DOCTYPE html>

<html>
<head>
  <title>SVG in HTML</title>
</head>

<body>
  <img
src="square.svg"></img>
</body>
</html>
```

As for the SVG file, it's just a simple square, drawn in Inkscape, and centered in a square page.



I'm going to wade in and add some handwritten CSS to this file, so, for the sake of clarity (not to mention space), I've saved the image as an Optimized SVG. Everything I'm going to do would also work on an Inkscape SVG, but if you're not familiar with editing XML files, it's probably worth using optimized files, at least at first, so that you don't have a load of extra elements and namespaces getting in the way. My optimized file looks like the codew shown below (with a few line breaks added for clarity – they won't affect the image).

A hand-coded version of this image could be even smaller – there would be no need for a <g> with a transform when the x and y coordinates of the <rect> could be adjusted directly. But this is, after all, an Inkscape column, so I'll work with the output it gives me.

With my SVG file created and saved into the same directory as the HTML document, loading the latter into a web browser gives exactly the result you would expect: a web page with a square in it. So far, we haven't really gained a lot over using a bitmap. Yes, technically it retains a better quality when scaled, and SVG files are often (though not always) smaller than their raster equivalents – but in many cases those are modest benefits at best.

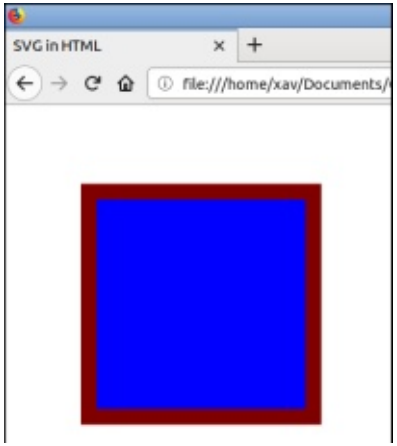
```
<svg width="360" height="360"
  version="1.1"
  viewBox="0 0 95 95"
  xmlns="http://www.w3.org/2000/svg">

  <g transform="translate(-7.6 -5.2)">
    <rect x="29" y="27"
      width="53" height="53"
      style="color:#000000;fill:#f00;
        paint-order:stroke fill markers;
        stroke-width:8;stroke:#800000"/>
  </g>
</svg>
```

But, even when used in an <img>, there are some things we can do with an SVG file that can't be done with a raster image. Unlike a raster image, an SVG file can include its own CSS code. Let's begin by making our red square blue (note, I've abbreviated the <svg> element for space – in practice the file still contains the full element from the previous example). Code is shown top right.

If you're not familiar with CSS, then here's what we've done: first there's a pair of opening and closing <style> elements. The CSS code will be held within them. The CSS code itself consists of a selector that identifies what element(s) in a page the rules should apply to. In this case we've used a simple element selector: essentially if you just use the name of an element (in this case "rect"), the rules will apply to every instance of that element. As we've only got one <rect> in our file, we know that the rule won't accidentally affect anything else.

After the selector come the rules, in a block between a pair of braces (the "{" and "}" characters). We've got only one rule at the moment: to make the fill blue (using hexadecimal RGB syntax for the color). So, let's reload the page and take a look at our…red square.

As you may already know, CSS stands for "Cascading Style Sheets", and it's the "cascading" part that has interfered with our plans. Without going into all the gory details, it's possible to have more than one source of CSS rules affecting an element, and the details of the cascade dictate which rule will take precedence over the others. In almost all cases, if there's a rule directly placed on an element using a "style" attribute, that will win. So, in our case, if we just remove the "fill:#f00;" from within the style attribute of the <rect>, the rule we've added to the <style> block will win instead. This time if we reload the page… our square is blue.




```
<svg ...>
  <style>
  rect { fill: #00f; }
  </style>


  <g transform="translate(-7.6 -5.2)">
    <rect x="29" y="27"
      width="53" height="53"
      style="color:#000000;fill:#f00;
        paint-order:stroke fill markers;
        stroke-width:8;stroke:#800000"/>
  </g>
</svg>
```

That's all well and good, but so far we haven't really achieved anything that couldn't have been done just by drawing a blue square in the first place. But what we have done is to move the rule that governs the color of the square from the <rect> element (where we can't easily override it) to a separate block of CSS (where we can). Our options for how to override it, however, are rather limited. This is just an image in our web page, so mouse events aren't propagated into the file itself. That means no color changes on clicks, or on hovering the mouse over the image. In fact there's really only one thing we can do to modify the image in this situation: animate. And that requires more CSS.

To begin with, we have to expand our existing set of rules to also include some information about the animation we want to use. At a minimum we have to create a name for the animation (so that you can have more than one in a file), and tell it how long the animation should last. That means our rect selector now looks like this:

```
rect {
  fill: #00f;
  animation-name:
myAnimation;
  animation-duration: 3s;
}
```

The choice of animation name is up to you, but it has to match whatever name you use in the next section. This is where we define the keyframes – specific points in the animation when we set the values the CSS should take. Here's

a simple example for animating the fill from red to blue:

```
@keyframes myAnimation {
  0% { fill: #f00; }
  100% { fill: #00f; }
}
```

The syntax is pretty straightforward. First we start with the string "@keyframes" to indicate that this is a set of keyframes, followed immediately by the name we defined earlier. Then, inside a block of code between braces, we have individual entries for each keyframe.

A keyframe entry is just the same as any normal block of CSS, except that the selector is replaced with a percentage, representing how far through the animation this block should apply. You pretty much always need at least a block for the start of the animation (0%) and one for the end (100%), but you can add more in-between if you want. Because animations with just two keyframes are so common, you can also use the words "from" and "to" instead of 0% and 100%, which arguably makes the code a little more readable.

So, taking both of the blocks above together, we've essentially said "create a set of keyframes called 'myAnimation' and run them on the <rect> over a period of 3s. At the start the fill color should be #f00 (red), at the end it should be #00f (blue)."

There's one little caveat to catch: notice that there's still a fill value on the rect CSS block. At the end of the animation, the "normal" CSS for the element will kick in, so if we don't also make it blue in there, the fill will change to black. Sometimes that behaviour is desirable; you want to change the style back to the defaults after the animation has run. But, more commonly, you want the style at the end of the animation to persist.

As an alternative to duplicating the final rules for the element, we can make the last style stick by adding the "animation-fill-mode" rule to our rect block instead. The behaviour we're seeing is equivalent to setting this to "none", but you can also use "forwards" to make the element continue using the style from the last keyframe. There's also "backwards" which does the same with the first keyframe for animations played in reverse, or "both" which works in either direction, and is usually the simplest option to choose.

Animations played in reverse? Yes, there's another property that will let you do that: "animation-direction". The default value is "normal" (i.e. your animation plays forwards from 0% to 100%), but you can also set it to "reverse" (100% to 0%), "alternate" (0% to 100% then back to 0%) or "alternate-reverse" (100% to 0% then back to 100%).

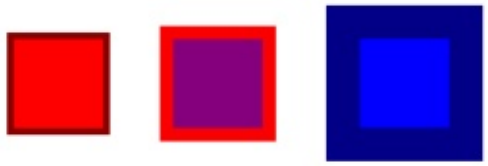
Once you've decided what direction(s) your animation should run in, you might want to also determine how many times it will be performed. For that we have "animation-iteration-count" which can take the value "infinite" to make the animation run forever, or a number – to specify a particular number of cycles. Note that a single cycle goes from 0% to 100% (or vice versa), so if you set "animation-direction" to "alternate" or "alternate-reverse", you'll need to set the count to 2 rather than 1 to display both halves of the animation. With these values, any odd number will finish at the end of the animation, whilst an even number will return you to the start. This is also why it's usually worth setting "animation-fill-mode" to "both" – you can then change the animation count without having to worry about a sudden change in style, regardless of whether there are an odd or even number of cycles.

Let's put all this together, and add another keyframe for good measure. I've also thrown in an "animation-delay" so that there's a short pause between the file loading and the animation starting. Our <style> block now looks something like the text shown on the next page, top right.

So far, we've animated only one property at a time. But because each keyframe carries a block of CSS, it's possible to animate more than one property on an object. Here I've also animated the stroke color and width to substantially change my square over the course of the animation. The code is shown on the next page, bottom right.

The result, captured at 2s (the length of the delay, just as the animation starts), 3.5s (halfway

through the animation), and 5s (the end of the animation), looks something like this – though, obviously, the real thing isn't a series of three images, but a single image that transitions through these three states:



In practice, the iteration count of 3 and direction of "alternate" means that, after a 2s delay, the square progresses through each of these states (iteration 1), then back again in the opposite direction (iteration 2), then finally through them in order again (iteration 3). At the end of the animation the square remains blue with a thick stroke, due to the presence of animation-fill-mode: both.

As you can see, animating an SVG image allows for possibilities that aren't really available with raster images – at least not in a high quality way that works on most browsers. Next time, we'll expand on these capabilities to push our SVG animations even further.

```
<style>
  rect {
    animation-name: myAnimation;
    animation-duration: 3s;
    animation-fill-mode: both;
    animation-direction: alternate;
    animation-iteration-count: 3;
    animation-delay: 2s;
  }

  @keyframes myAnimation {
    0% { fill: #f00; }
    50% {fill: purple; }
    100% { fill: #00f; }
  }

  </style>
```

```
@keyframes myAnimation {
    0% {
      fill: #f00;
      stroke: #800;
      stroke-width: 8;
    }

    50% {
      fill: purple;
      stroke: #f00;
      stroke-width: 16;
    }

    100% {
      fill: #00f;
      stroke: #008;
      stroke-width: 40;
    }
  }
```

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

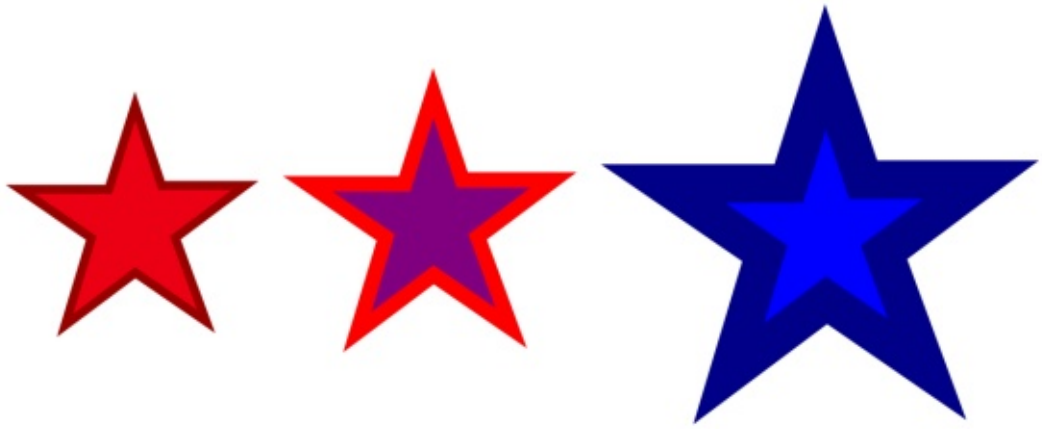
Inkscape - Part 76

Last time, we looked at how it's possible to include animations in an SVG file that is loaded as an image. We introduced the general idea by animating the fill and stroke colors, and the stroke width, of a square we'd drawn in Inkscape. If we're just working with a square – or even a rounded rectangle – we could equally achieve the same effect in HTML by animating the "background" and "border" CSS properties, with no need to go near SVG. But the important point to note is that we actually used CSS to animate properties that only make sense for SVG. In other words, although CSS is most commonly used with HTML, we can still use it to modify many SVG-only values. Furthermore, we can animate those properties not only on squares and rectangles, but also on arbitrary paths.

By reusing the same animation that we had in the previous article, we can produce the same set of three snapshots, but this time of a more complex shape. For this example I created a star in Inkscape and converted it to a path. The first CSS rule, which previously had a selector of "rect", also needed to be updated to change the selector to "path".

But what if you want to animate more than just the style of your elements? Perhaps you want to make them move around, to spin, or to change size. You might even want the shape of the path itself to change. All this and more is possible with SVG… but, as will become clear, I don't necessarily recommend doing it.

As I've previously discussed in this column, there was a time when the W3C went full tilt in favor of XML. They defined and specified a wide range of XML languages – including SVG and XHTML (a pure XML version of HTML) – as well as working on various supporting technologies that could work with any XML language. One of these technologies was the "Synchronized Multimedia Integration Language", or SMIL (pronounced "smile" apparently). SMIL is itself an XML language that describes how an XML document should change over time, or in response to certain interactions such as mouse movements and clicks. In the case of SVG, therefore, SMIL can be used to describe the way in which arbitrary attributes should change over time, allowing any part of the image to be animated.



If the W3C's plan for XML domination had panned out, SMIL would probably have become a universally implemented technology for animation and multimedia. In practice, however, browsers veered away from the XML-centric approach in favor of the more lax requirements of HTML, and Microsoft never implemented SMIL in their browsers. Faced with an animation standard that wasn't supported by Internet Explorer, it's no wonder that so much early animation on the web was outsourced to Flash. The result was that SVG languished for a long time and SMIL never really took off. In the meantime, CSS gained more and more abilities that were once the remit of SVG, and has begun to encroach on the domain of SMIL with CSS animations and transitions.

The penultimate nail in the SMIL coffin came with the release of version 45 of Google's Chrome browser, which officially listed the technology as deprecated. It would still work for the time being, but that was notification that it's

eventually going away. After a backlash from the community they revoked this announcement, so SMIL is no longer deprecated in Chrome – but it's only a matter of time before they decide that CSS animations are capable enough for them to try again. So that's why I don't really recommend using SMIL: with no support in Internet Explorer or Edge, and the prospect of being deprecated then removed in Chrome at some indeterminate point, it's not a technology that can be relied on as a viable solution for use on the web at large.

Given this situation I'm not going to discuss SMIL in any great detail. But as it is (currently) usable in (most) web browsers, I'll spend a couple of articles giving you a brief introduction in case you feel it is a technology that you can use and wish to investigate further – and as an insight into the sort of web we might have now, had Microsoft played ball, and XML gained stronger support from the browser vendors.

There are four types of animation that can be performed using SMIL with SVG, in each case by adding the relevant animation tag (shown in brackets) inside the element you want to animate:

• Animate the attributes of an SVG element (<animate>).

• Animate the transformation that can be applied to an SVG element (<animateTransform>).

• Animate the color of the fill and stroke (<animate> or <animateColor>).

• Animate the position and, optionally, rotation of an object by making it follow another path (<animateMotion>).

Let's look at a simple example, by trying to replicate the CSS animation of the same red star I used earlier. In this case we will need to animate the fill and stroke color, and the stroke width, using the <animate> tag. So rather than having those values stored as CSS properties, they need to be moved out to independent attributes. This results in our SVG file looking something like that shown top right.

Note that, for simplicity, I've rounded all the coordinates down to whole numbers. There's also a transform attribute that translates the star 20 units to the right, and 20 units down: I could have done the calculations to adjust the coordinates for the path, removing the need for this entirely. As Inkscape seems rather keen to put transforms onto its content, however, I decided to leave it in to better represent the sort of (minimised) output you might see from the program.

```
<?xml version="1.0" encoding="UTF-8"?>
<svg
   xmlns="http://www.w3.org/2000/svg"
   width="120mm"
   height="120mm"
   viewBox="0 0 120 120">
  <path
     fill="#f00"
     stroke="#800"
     stroke-width="4"
     d="m 40,15 6,18 19,0 L 49,45 56,64
        40,53 24,65 30,45 14,34 34,34 Z"
     transform="translate(20, 20)" />
</svg>
```

As with the CSS animations from last time, I'm going to start by just animating the fill color. This involves putting the <animate> tag as a child element of the <path>. In case you're not very familiar with XML, in the previous code the path is written as a self-closing tag: <path … />. This can legitimately be rewritten as a non-self-closing tag: <path …></path>. It's this latter approach we'll need in order to add a child element (note that the rest of the SVG file and many of the path attributes have been omitted for clarity - shown below).

```
<path
     fill="#f00"
     ...
     transform="translate(20, 20)" >
  <animate attributeName="fill"
    from="#f00" to="#00f"
    dur="5s" fill="freeze" />
</path>
```

The new element is pretty self-explanatory: attributeName defines which attribute in the parent element we're animating, from and to are the start and end values, dur is the duration of the animation, and fill="freeze" is used to make sure the attribute retains the last value, rather than flipping back to its original value.

That's easy enough, but our original animation wasn't a simple transition from one value to another, it also had a specific intermediate value, giving us three keyframes in total. That can be achieved with SMIL as well, by replacing our "from" and "to" attributes with "values" and "keyTimes" attributes which hold the three values we want to hit, and the proportion of the way through the animation that each is applied. The values in each list are separated by a semicolon. Code is shown top right.

I'm using a generic <animate> element for animating this fill color. There is also a specific <animateColor> element that could be used instead, but since <animate> can do everything that <animateColor> can achieve, plus more, there's no point using the latter these days. Even the SMIL specification now recommends using <animate> rather than <animateColor>, so who am I to argue.

Our original animation not only changed the fill color, but also the stroke color and width. This obviously entails animating three attributes – which we do just by using three <animate> elements (shown right).

Notice that I've had to adjust the stroke-width values somewhat to get a similar result to the CSS animated version. I'm not sure why there's a difference – perhaps the CSS values are interpreted as pixels, whilst the SMIL values are treated as SVG user units, based on the viewBox declared at the start of the file. Whatever the underlying reason, a little tweaking was required but I was nevertheless able to produce the same result.

Animating attributes works well where the attribute can hold only a single, simple value, such as a length or color. The "transform" attribute is a more complex case, as it can hold a combination of translate(), rotate, scale(), and skew() functions. If your element already has a transformation applied – as many will do if they're created in Inkscape – you probably want your animation to be added to the current transformation rather than replacing it entirely. To achieve this requires more than a simple <animate> element; it needs something that understands the syntax and structure of the transform attribute. It needs <animateTransform> (next page, top right).

Here I've used a couple of <animateTransform> elements to zoom and rotate the star during the course of its animation. The first element scales the star

```
<animate attributeName="fill"
  values="#f00; purple; #00f"
  keyTimes="0; 0.5; 1"
  dur="5s" fill="freeze" />
```

```
<path

    fill="#f00"
    stroke="#800"
    stroke-width="2"
    ...
    >

  <animate attributeName="fill"
    values="#f00; purple; #00f"
    keyTimes="0; 0.5; 1"
    dur="5s" fill="freeze" />

  <animate attributeName="stroke"
    values="#f00; purple; #00f"
    keyTimes="0; 0.5; 1"
    dur="5s" fill="freeze" />

  <animate attributeName="stroke-width"
    values="4; 8; 20"
    keyTimes="2; 3; 6"
    dur="5s" fill="freeze" />

</path>
```

(type="scale"), first making it bigger (1 > 1.5) and then smaller (1.5 > 1 > 0.75), covering four keyframes in the five second duration. You might think that attributeName="transform" is redundant when the element itself is called <animateTransform> – and I would agree with you – but the animation doesn't work without it.

The second element rotates the star. In this case the values each consist of three space- or comma-separated numbers. These represent the amount of rotation (in degrees), and the x and y coordinates to be used as the center of rotation. I've selected 40 for each, to put the center roughly in the middle of the star, so it doesn't spin off-screen entirely.

Thanks to the scale animation, however, it does still move around quite a bit. As the star is scaled in size, so its center point moves and the values in the rotation animation are no longer correct. This could be accommodated, to some extent, by tweaking the x and y coordinates in the rotation, or by adding another animation to set a translate() transform to compensate for the moving center point. A better approach would be to re-draw the star centered on the 0,0 coordinates, then position it on the page by tweaking the transform attribute on the element itself. That way, there's no need to specify a center point for the rotation, and the values for the "rotate" animation could be reduced to a simple list of angles.

You'll have noticed that both the <animateTransform> elements have an additive="sum" attribute. SMIL animations can be configured so that the animated values completely replace any previous value on the element, or so that the effects of the animation are cumulative. Without this extra parameter, the scale animation replaces the transform="translate(20, 20)" attribute that Inkscape created; then the rotate animation replaces the scale animation. The result is that the star only rotates, and doesn't scale – and it does so in the wrong location. The additive="sum" attribute tells the browser to keep any existing transforms around, and add the new animated value to them to give a cumulative effect. In other words, the star both zooms and rotates, and does so (roughly) in the location specified by Inkscape's translate() value.

```
<path
    ...
    transform="translate(20, 20)">
  <animate attributeName="fill" ... />
  <animate attributeName="stroke" ... />
  <animate attributeName="stroke-width" ... />

<animateTransform attributeName="transform"
    type="scale" values="1; 1.5; 1; 0.75;"
    keyTimes="0; 0.33; 0.66; 1" dur="5s" fill="freeze"
    additive="sum" />

  <animateTransform attributeName="transform"
    type="rotate" values="0 40 40; 120 40 40; 360 40 40"
    keyTimes="0; 0.5; 1" dur="5s" fill="freeze"
    additive="sum" />

</path>
```

Over the years, CSS has gained ever more of SVG's capabilities, with the result that the SMIL animations I've shown so far can all be implemented as CSS animations. The fill and stroke we covered last time, but even <animateTransform> has a counterpart in CSS, now that the latter supports a "transform" property that allows elements to be translated, rotated, scaled, and skewed. Unlike SVG, it also offers a few options for transforming your element in three dimensions.

But there are still some tricks up the SMIL sleeve which CSS animations can't compete with (yet). One of these is the ability to animate a path between one shape and another. Much like animating a color, this is just a special case of animating an attribute – in this case the "d" attribute that holds the data representing the shape of a path. As a reminder, here's what the path data for our star looks like:

```
d="m 40,15 6,18 19,0 L 49,45
56,64

   40,53 24,65 30,45 14,34
34,34 Z"
```

I'm going to animate the star turning into a large arrow, pointing to the top right of the screen. To do this, I just use another

<animate> element, transforming the “d” attribute from its default value to a new path definition. This will do the trick:

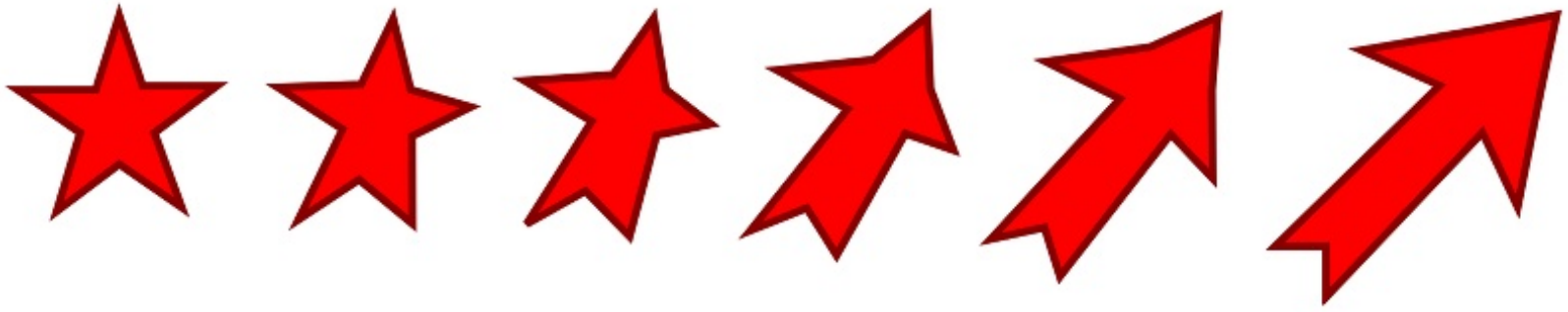
```
<animate attributeName="d"
  to="M 92,26 88,45 82,75 73,56 33,98
      33,85 20,85 62,44 42,35 71,30 Z"
  dur="5s" fill="freeze" />
```

By commenting out the other <animate> and <animateTransform> elements (using the XML/HTML method of wrapping them in <!-- and --> tags), we can see what this path animation looks like in isolation. Here (above) are six frames from the transition.

Removing the comment tags to reinstate the other animations results in a star that smoothly changes to an arrow whilst it’s rotating, changing size, and it transitions from a red fill with a thin dark red outline to a blue fill with a thick dark blue outline (via a purple fill along the way). Stacking up multiple individual transitions to create one larger animation is something that SMIL excels at.

It’s important to note that the animated “d” values must all contain the same number of parameters for a smooth transition. Obviously the browser wouldn’t know how to animate from a 5-pointed star to a 12-pointed star; which lines should the new nodes appear on? But the same goes for the other parameters in a “d” attribute – if you want curved lines in your animation, you need to start out with them curved (even if the curvature is visually non-existent) to ensure that each part of the animation contains the same line types and number of coordinates for all the nodes and Bézier control points. If your paths contain different line segments the animation will still take place but, rather than a smooth transition from one shape to another, you’ll get sudden, discontinuous jumps from shape to shape.

To my eye, SMIL animation is easier to follow than its CSS counterpart. Admittedly, it’s rather verbose which can result in having to edit a lot of elements to make a simple timing change. But that verbosity also has its advantages, as we’ll see next time – when we’ll also send our star on a trip along a path.



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

Inkscape - Part 77

Last time, we looked at how to animate an SVG element using SMIL rather than CSS animations. We'll continue with this topic for one more article, but my warnings from last time need to be repeated: although SMIL is (currently) more capable and flexible than the CSS option, Microsoft have chosen not to implement it in any of their browsers, leading to a slow but inexorable demise in support across all the browser vendors. For the time being, it still works in most browsers but, unfortunately, this is not a technology to bet on in the long term.

I much prefer SMIL to CSS animations. I find the code easier to understand, and the fact that the animation data tend to live inside the elements they're animating avoids any confusion as to which rules apply to which objects as your document becomes more complex.

With a complex document it becomes ever more likely that your animations won't all have to run at the same time, but instead might run sequentially – or a mixture of both. Consider trying to animate something as sophisticated as a cartoon: being able to finely adjust the timing of each character's movements is essential. CSS animations offer little to help you in this case, other than the brute-force option of adding a delay to your animations with this the sort of approach (shown right).

Here we're changing the fill color of a pair of rectangles (with IDs of "rect1" and "rect2"). The first changes from red to blue over 3 seconds. After a 3 second delay, the second rectangle changes from red to white over a 5 second period. Due to that 3 second delay, the animations occur sequentially.

Now what happens if you want to change the length of the first animation? You must also keep the delay for the second animation synchronised to the same value, otherwise it will either overlap the first, or occur some time after the first has finished. CSS does now support variables, so you can set

```
#rect1 {
  fill: #f00;
  animation-name: myAnimation;
  animation-duration: 3s;
}

#rect2 {
  fill: #f00;
  animation-name: myDelayedAnimation;
  animation-duration: 5s;
  animation-delay: 3s;
}

@keyframes myAnimation {
  0% { fill: #f00; }
  100% { fill: #00f; }
}

@keyframes myDelayedAnimation {
  0% { fill: #f00; }
  100% { fill: #fff; }
}
```

```
:root {
  --anim1_length: 3s;
}

#rect1 {
  fill: #f00;
  animation-name: myAnimation;
  animation-duration: var(--anim1_length);
}

#rect2 {
  fill: #f00;
  animation-name: myDelayedAnimation;
  animation-duration: 5s;
  animation-delay: var(--anim1_length);
}
...
```

the animation length once and reuse it as necessary, but the syntax is ugly and unwieldy (previous page, bottom right).

Now imagine what your CSS would look like when you want to chain five animations, or ten – or a hundred! How much better it would be if you could just tell the second animation that it should start when the first one finishes. With SMIL, that's exactly what you can do. The SMIL animation elements have an optional "begin" attribute which allows various ways of defining when the animation should start. At its simplest, you can just enter a delay, giving the same effect as the CSS animation above (shown top right).

But you can also define the beginning of an animation to be triggered by the end of another by using the ID of the other animation, followed by ".end" (bottom right).

What happens if we want to change the length of the first animation now? No problem! Just modify the "dur" attribute and the second animation will still dutifully follow after the end of the first one. As well as the ".end" syntax you can also use ".begin" to link animations together so that they always start at the same time. You can optionally add an offset, such as "anim1.begin+2s" to make the animation begin 2 seconds after "anim1" starts, or even "anim1.end-0.5s" if you want your animation to begin half a second before the end of "anim1".

SMIL allows animations to be repeated by adding a "repeatCount" or "repeatDur" attribute. For example, repeatCount="5" would cause the animation to repeat five times, whilst repeatDur="01:00" would cause it to repeat for one minute. When a repeating animation is used as the trigger for a second animation, this also allows an additional syntactic form to be used: begin="anim1.repeat(2)" would cause the second animation to begin immediately after the second repeat of anim1 completes. Again, changes to the duration or start time of anim1 are automatically handled for you.

In theory, these values can also be used for the "end" attribute, rather than for "begin". That should allow you to specify that a second animation should finish three seconds after the end of the first animation, with the browser calculating when the animation should start in order to produce that result. Similarly, you should be able to specify values for "begin" and "end" with no duration set. In practice browsers fail to honor anything but a simple time-based "end" value.

```
<rect id="rect1" fill="#f00" ...>
  <animate attributeName="fill"
    from="#f00" to="#00f"
    dur="3s" fill="freeze"
    begin="0s" />
</rect>

<rect id="rect2" fill="#f00" ...>
  <animate attributeName="fill"
    from="#f00" to="#fff"
    dur="5s" fill="freeze"
    begin="3s" />
</rect>
```

```
<rect id="rect1" fill="#f00" ...>
  <animate id="anim1"
    attributeName="fill"
    from="#f00" to="#00f"
    dur="3s" fill="freeze"
    begin="0s" />
</rect>

<rect id="rect2" fill="#f00" ...>
  <animate attributeName="fill"
    from="#f00" to="#fff"
    dur="5s" fill="freeze"
    begin="anim1.end" />
</rect>
```

Speaking of features that don't work in the browsers, the "begin" (and "end") attributes can, theoretically, take several other forms. You should be able to use an ISO8601 format time or datetime value to trigger your animation at a particular absolute time, but I was unable to get that to work in a browser. Another option is an "accessKey" - i.e. a single key on the keyboard that,

when pressed, would trigger the start of the animation. The presence of a demo for this on the Mozilla developer site suggests it used to work, but neither Firefox nor Chrome behaved as expected in my own tests.

There's one final option that does sort-of work, however: events. The following syntax, for example, will (in principle) trigger an animation when the rectangle is clicked on:

```
<rect id="rect1" fill="#f00"
...>
  <animate id="anim1"
    attributeName="fill"
    from="#f00" to="#00f"
    dur="3s" fill="freeze"
    begin="rect1.click" />
</rect>
```

There are various events available, covering not only clicks but also mouse movements, scrolling and even changes to the structure of the document. Although the example above uses the parent element to trigger the animation, in practice you could use the ID of another element in the image – allowing a click on one element (styled as a Start button, perhaps) to trigger an animation on another.

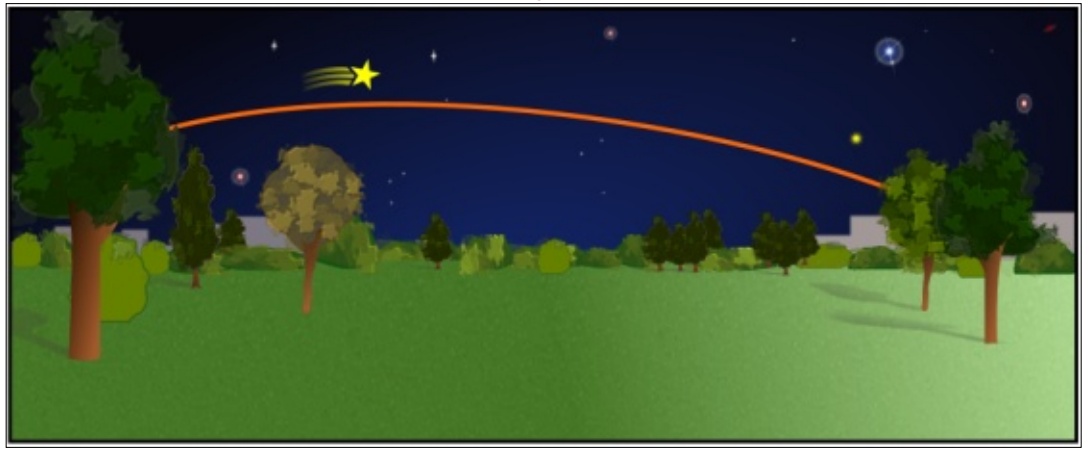
In practice this option does work, but only in situations when JavaScript would also be executed: when the SVG image is loaded directly, via an <object> tag, or is included inline with the HTML. It doesn't work when SVG is loaded via an <img> element, which is a real shame as it could theoretically offer a safe way to produce interactive animations without the security risk of allowing JavaScript code to run.

And that pretty much sums up the failed promise of SMIL. If fully implemented it would have allowed the creation of complex animations triggered by mouse events or keypresses, with each component synchronised to other parts, all with a fairly simple declarative syntax that makes it safe to use via an <img> tag. Imagine a complex interactive animation, of the sort that you might see in a museum, but with the ability to be shared on forums or social media as easily as any other image.

Before bidding farewell to SMIL entirely, I'm going to briefly discuss the last of the animation elements that are supported by SVG: <animateMotion>. This allows you to animate the position of an element along a path, either defined within the element itself or by reference to another path in the file. Consider this delightful evening scene (below).

```
<path

    fill="#f00"
    stroke="#800"
    stroke-width="2"
    ...
    >

  <animate attributeName="fill"
    values="#f00; purple; #00f"
    keyTimes="0; 0.5; 1"
    dur="5s" fill="freeze" />

  <animate attributeName="stroke"
    values="#f00; purple; #00f"
    keyTimes="0; 0.5; 1"
    dur="5s" fill="freeze" />

  <animate attributeName="stroke-width"
    values="4; 8; 20"
    keyTimes="2; 3; 6"
    dur="5s" fill="freeze" />

</path>
```

Note the orange path across



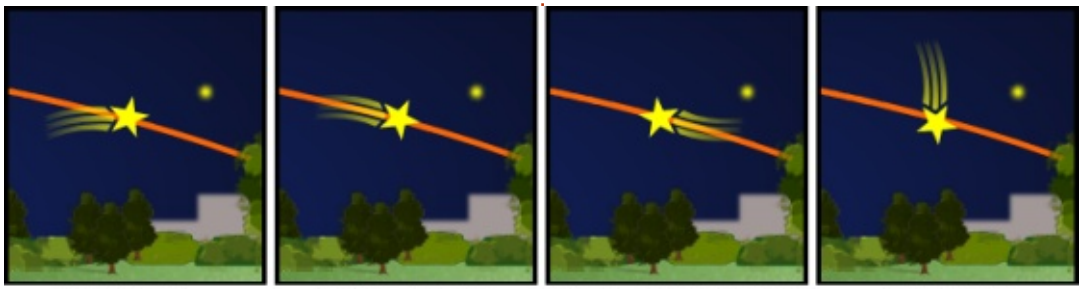
the night sky, which I've given an ID of "animPath". The yellow shooting star is made up of a group of objects, drawn so that the center of the star is at the top left of the document area (0,0 in SVG coordinates) – though I've moved it into the middle of the scene for this screenshot so that you can see it. By adding an <animateMotion> section inside the group, the shooting star will follow the orange path across the night sky (top right).

The <animateMotion> element gets the usual animation attributes of "dur" and "fill" (and could have had "begin" and "end" if required), but has two attributes that are specific to this type of animation. The first is a "d" attribute which can contain path data of the same form that you would find in a <path> element. If present, this is used as the path along which the parent element will be animated.

An alternative to directly including the path data in the <animateMotion> is to reference a separate path that is present elsewhere in the document. This is the approach I've taken here, by including an <mpath> ("motion path") child element that refers to the ID of our animation path via the "href" attribute. Although modern browsers understand "href" as a native attribute in SVG, using the "xlink" namespace provides better compatibility with older software, so that's what I've done here. The big advantage of using a linked path like this is that the path is an element that can be modified in Inkscape, whereas an embedded "d" attribute isn't.

The second attribute that is specific to <animateMotion> is "rotate", which is an SVG addition which is not present in the base SMIL specification. This can take a number, in which case the object is rotated by that number of degrees, although a fixed rotation is probably better achieved using a "transform" attribute. More usefully, this attribute can take a value of "auto", in which case the rotation of the element follows the shape of the path (there's also an "auto-reverse" option which does the same, but rotates the animated element through 180° first). Here's the effect of each option as the star descends on its path; notice particularly how rotate="auto" has turned it to suit the descending curve of the line.

```
<path id="animPath"
  d="…"
  style="…" />


<g id="shooting_star">
  <path id="star" … />
  <path id="tail1" … />
  <path id="tail2" … />
  <path id="tail3" … />

  <animateMotion
    dur="5s"
    fill="freeze"
    rotate="auto">
      <mpath xlink:href="#animPath" />
  </animateMotion>
</g>>
```

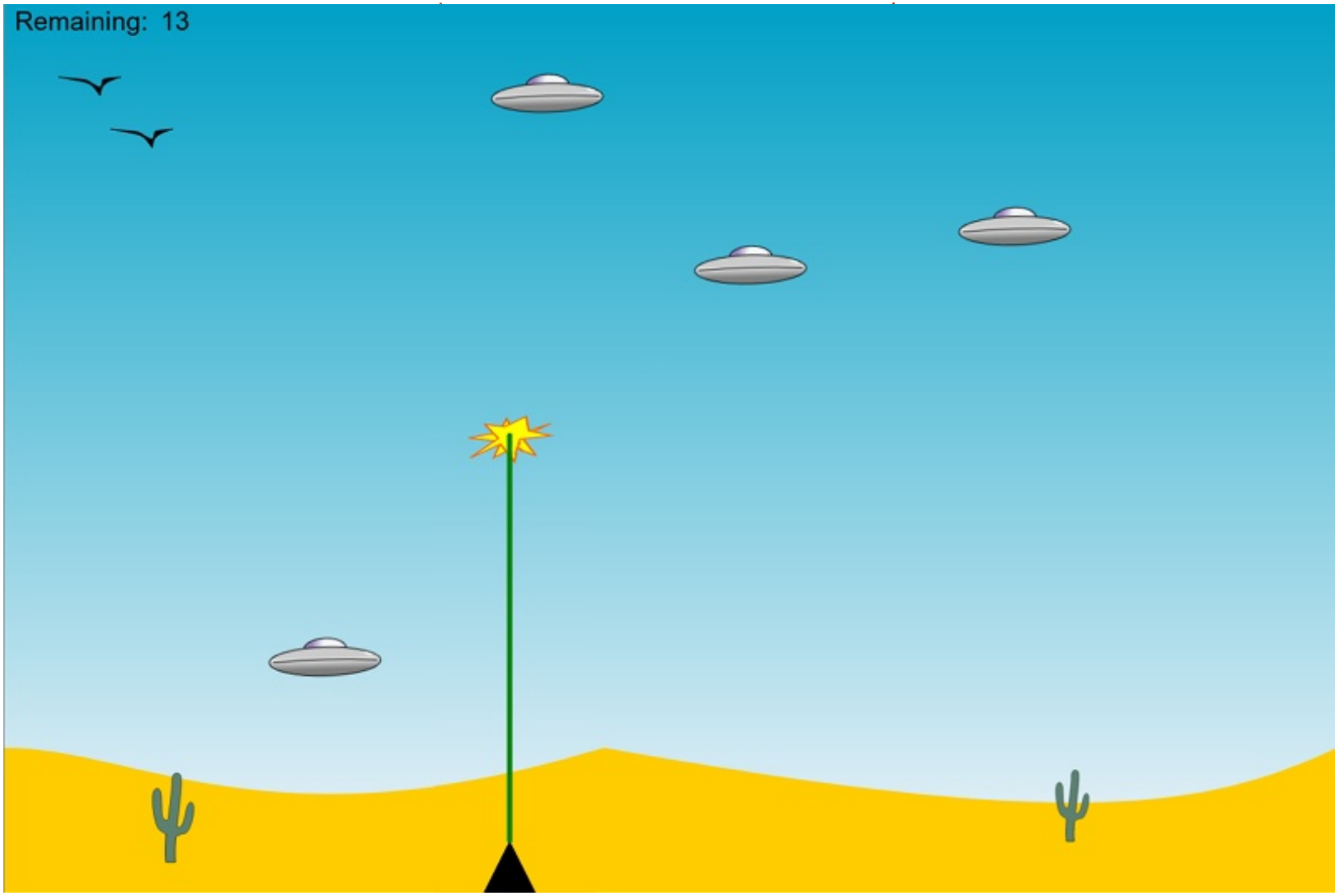


You may be wondering about that orange path. The final step in designing an animation like this is typically to hide the animation path somehow. I usually move the path down in the z-stack behind everything else, or change its stroke color or opacity to make it transparent. Even when it's transparent you can still get to it using Inkscape's View ! Display Mode ! Outline option, if you do need to make some later changes. Although this simple example uses just a single curved path segment, the animation path can be as complex as you like with loops, twists, curves and sharp corners, so being able to tweak it graphically in Inkscape can be invaluable.

One final thing to note is that although my test animation ran smoothly in both Chrome and Firefox when the SVG file was loaded directly, referencing it via

an <img> tag in a web page resulted in a choppy animation in Firefox.

I'll leave you with a little SMIL anecdote: back in 2011 I made use of SMIL for an Easter egg in one of my webcomics, to animate a UFO flying over the scene. The animation path itself is seemingly jerky and erratic, but digging into the file in Inkscape reveals that the path actually encodes a URL. Visiting that address shows a small demo of what SMIL can do: by using some JavaScript to dynamically add and modify SVG and SMIL elements, I wrote a simple Space Invaders style game that runs in the browser. JavaScript handles the game logic, with SMIL responsible for ensuring that the flying saucers move smoothly around the sky. For the time being, at least, it runs in all the major browsers, except Microsoft's…



These couple of articles have just provided a brief introduction to SMIL. With browser support waning, it's unlikely to ever fulfill its early promise of allowing interactive animations in a way that can be safely used online anywhere a simple image is allowed. As is too often the case, it appears that corporate politics has killed a promising technology.

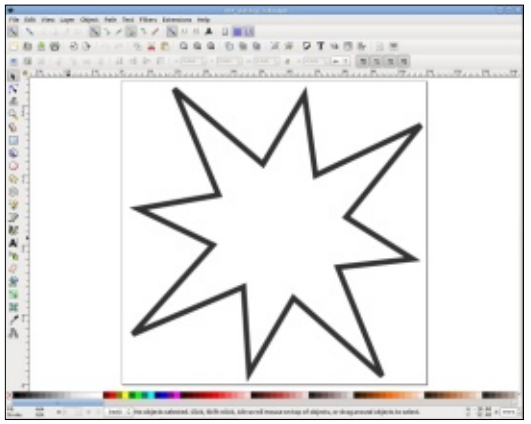
Mark uses Inkscape to create three webcomics, 'The Greys', 'Monsters, Inked' and 'Elvie', which can all be found at http://www.peppertop.com/

Inkscape - Part 78

A quick summary: over the past few instalments, we've looked at a couple of animation techniques for SVG files. First was to use the animation capabilities of CSS, which has better support across browsers but is limited to animating those properties that can be exposed via CSS (i.e. the ones that will work in the "style" attribute). This covers a lot of possibilities, but does not include many of the core attributes that are found in SVG elements, such as their coordinates or path definitions. For some cases – such as transforms – enough of SVG's capabilities have been added to CSS for such restrictions to be nothing more than an annoyance that can be worked around. For others, however, CSS simply isn't up to the task.

That's where SMIL comes in. It's an older specification for animation, created around the same time as the original SVG specs, and does provide the ability to animate arbitrary attributes in SVG. It can animate the "d" attribute that makes up the shape of a path, for example – a task that can only otherwise be accomplished in the browser via JavaScript. But although SMIL is more powerful than CSS in this regard, it has never been adopted in any Microsoft browser, leading to its inexorable decline and a likely deprecation in the browsers that do support it.

Given this situation, we won't look any further at SMIL, but there are still a few tricks that can be performed with CSS that are worthy of further investigation. This time, we're going to look at an increasingly popular animation technique which is really the result of taking a simple feature of SVG and pushing it in ways it wasn't really intended for. It's commonly referred to as "line animation", although the line itself doesn't change shape (remember, that can't be done with CSS yet). It's also sometimes referred to as a "self-drawing" image, which is a bit more descriptive: this technique results in a path that appears to be drawn on the screen in the same way as you would draw it with a pen.



The first thing we'll need is a path to animate. It should not have a fill, but will require a stroke. For this demonstration I've used Inkscape's star tool to create an eight pointed star, then converted it to a path and roughly moved every second point inwards to give more of a cartoon gunshot appearance. I've given it a stroke-width of 2px and saved it using the "Optimised SVG" option in order to strip out most of the excess baggage from the file, just so that this tutorial isn't filled with several pages of code.

Opening the file in a text editor reveals a structure like that shown below.

As you can see, it's a pretty simple file. Even the <g> isn't really needed, but it's easier to leave it in place with its own transform, rather than try to manipulate the coordinates of the path to compensate for its removal.

The secret to this animation technique is in the fact that SVG has a nice, simple, CSS-animatable way of drawing dashed and dotted strokes, rather than solid ones. The next step, therefore, is to introduce some dashes, defined in CSS. Note that you'll probably have

```
<svg xmlns="http://www.w3.org/2000/svg">
  <g transform="translate(-10 -10)">
    <path d="m29 13 32 …"
      style="fill:none;stroke-width:2;stroke:#333; …"
    />
  </g>
</svg>
```

to also remove the corresponding properties from the path's "style" attribute if you saved as a normal Inkscape file.

Save the file (top right) and load it directly into a web browser, and you should see something like this:

The values we set in the stroke-dasharray property are used to determine the lengths of the drawn segments and gaps along the path. With our value of "10 2 3 5" the stroke will be drawn as a line segment of 10 units, followed by a gap of 2 units, then another line of 3 units and a gap of 5 units. The sequence then automatically repeats, so the next line segment is 10 units long and the next gap is 2. It runs around the path uninterrupted by corners, so a line segment (or gap) of 10 might appear as 3 units on one side of a corner and 7 on the other – look at the leftmost corner of the shape for a good example of this. If you follow the line round, taking that into account, you can see that the stroke dashes do follow the pattern we've set.

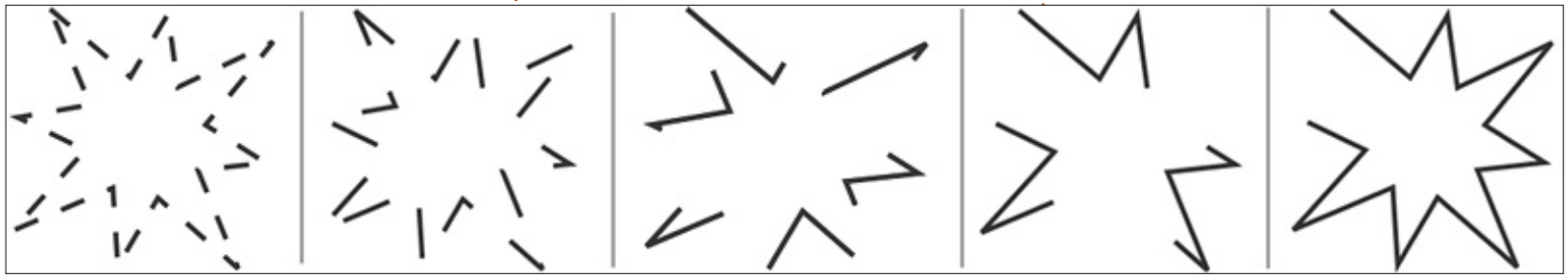
This automatic repeating of values in the stroke-dasharray property produces a couple of neat features. The first is that any odd number of entries is effectively the same as writing a value that is twice as long, with the line and gap values swapped in the second half. So a value of "10 5 10" is the same as writing "10 5 10 10 5 10" – in other words a long line, short gap, long line then a long gap, short line, long gap. Notice that the length of the lines in the first half make up the length of the gaps in the second, and vice versa.

The other feature – and the one we're interested in here – is that you can collapse the value down to a single number to get equal line and gap lengths. A lone value of "10" expands to "10 10", giving you a repeating series of lines that are 10 units long, followed by gaps that are 10 units long. In my example file, that gives you something like this:

```
<svg xmlns="http://www.w3.org/2000/svg">
  <style>
    path {
      stroke-dasharray: 10 2 3 5;
    }
  </style>

  <g transform="translate(-10 -10)">
    <path … />
  </g>
</svg>
```

Now consider what happens if we increase the value. As the number goes up, so the visible line segments get longer – but so do the gaps. Below is a sequence of screenshots of the same image with the stroke-dasharray value set to 10, 20, 50, 100 and 500.

By the time we get to 500 our first visible line segment almost covers the entire path. If we were to continue increasing the value until it is the same as the path length, it would effectively render it the same as if we didn't have the stroke-dasharray at all. And that's exactly what we want to do!



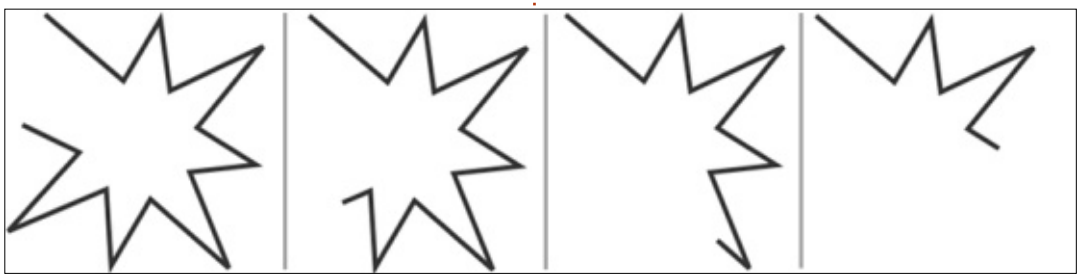
You might be thinking it's a little pointless to add a CSS property, then set it to a value that appears to have no effect, but remember that as well as the drawn line, there's an equivalent length of gap that's not visible because it extends beyond the end of the path. By adding another CSS property, however, we can adjust the start point of the drawn dashes, bringing our gap into play.

The stroke-dashoffset property tells the browser to adjust its calculations to begin further into your stroke-dasharray sequence (note that the "px" here isn't the same as screen pixels; rather it's in the units of the SVG file, which are likely to be different to screen pixels due to scaling). If your dasharray is "10 5" and you set the offset to 3px then the first segment drawn will be only 7 units long, then a gap of 5, then a line of 10 and the rest continues as normal.

In practice this means that setting a positive value here will "swallow" that much of the start of the stroke array, whereas a negative value will "push" the stroke array further along the path.

By adjusting the value of stroke-dashoffset we can "slide" the drawn portion of the stroke along the path. Here's what it looks like with values of 0px, 100px, 200px, 300px. Notice that progressively larger numbers swallow more of the path, showing more of the gap at the end.

As you know, setting the dasharray to the same value as the length of the path will draw the entire path. But also setting the dashoffset to that value will then swallow the drawn segment, leaving only the gap (which is also the length of the path). In other words, our path will disappear completely.



```
<style>
  path {
    stroke-dasharray: 500;
    stroke-dashoffset: 0px;
  }
</style>
```

With the drawn segment completely swallowed, we're at the starting point for our animation: our path is not visible on the screen. By decreasing the dashoffset value we can make the drawn segment creep onto the screen, following the shape of the path. All we need to do, therefore, is to animate the stroke-dashoffset value (see part 75 if you need a reminder of the animation syntax):

Reload the file in your browser and you should see the effect we're looking for – and because it's all just CSS animation embedded directly in the SVG file, it will even work when the file is pulled into a web page via an <img> tag.

Just a minute! Where did that value of 575 for the dasharray and dashoffset come from? As you've probably realised, that's the total length of my path. You could theoretically get it from Inkscape via the Extensions > Visualise Path > Measure Path… extension, but that throws a Python error on my system. Alternatively you could just use trial and error: increasing the value of the dasharray until it just fills the path (which is quite

```
<style>
    path {
      stroke-dasharray: 575;
      stroke-dashoffset: 575px;
      animation-name: drawPath;
      animation-duration: 5s;
      animation-fill-mode: both;
    }

    @keyframes drawPath {
      100% { stroke-dashoffset: 0px; }
    }

  </style>
```

```
<svg xmlns="http://www.w3.org/2000/svg"
  onload="alert(document.querySelector('path').getTotalLength());">
  <style>
  …
```

easily done using the browser developer tools, if you're familiar with them).

The simplest option, though, is to ask the browser to do the hard work for you by invoking a little JavaScript. There is a method on the <path> object called getTotalLength() which will return the calculated length of the path. You can call it via the developer console in the browser, or modify your <svg> element to call the method when the file loads and display the value on screen. Here's an example that will work for a file with a single path:

Remember that JavaScript won't run when the document is loaded via the <img> tag, so to use the getTotalLength() method you will have to load the SVG file directly into the browser. You only need to do it once, then you can simply hard-code the value into the <style> block and remove the JavaScript code entirely.

There is also an alternative to specifying the actual length of the path. You could add a "pathLength" attribute to the path, set it to a value of your choosing, then treat that value as the total length in the CSS. This basically tells the browser "I want to pretend this path is 100 units long, even though we both know it's not, so can you just do the maths for me as needed". A good compromise is to use JavaScript to find the actual length, then round it to the next whole number and set that value in the pathLength attribute and the CSS. That way any browser that understands pathLength will use it to give you a precise animation, and those that don't are still using a value that's close enough to work in most cases.

There's one final thing to note with this approach. So far I've used the version of the technique you'll most commonly find documented online, but, according to SVG expert Amelia Bellamy-Royds, the definition of stroke-dashoffset in the specification is vague enough that not all browsers behave identically. She suggests an alternative of forgetting dashoffset entirely and just animating stroke-dasharray instead. In this case, you need to include two numbers in the property as you need both the line and the gap to change together. My animation code then becomes

```
<style>
  path {
    stroke-dasharray: 0 575;
    animation-name: drawPath;
    animation-duration: 5s;
    animation-fill-mode: both;
  }

  @keyframes drawPath {
    100% { stroke-dasharray: 575 0; }
  }

</style>
```

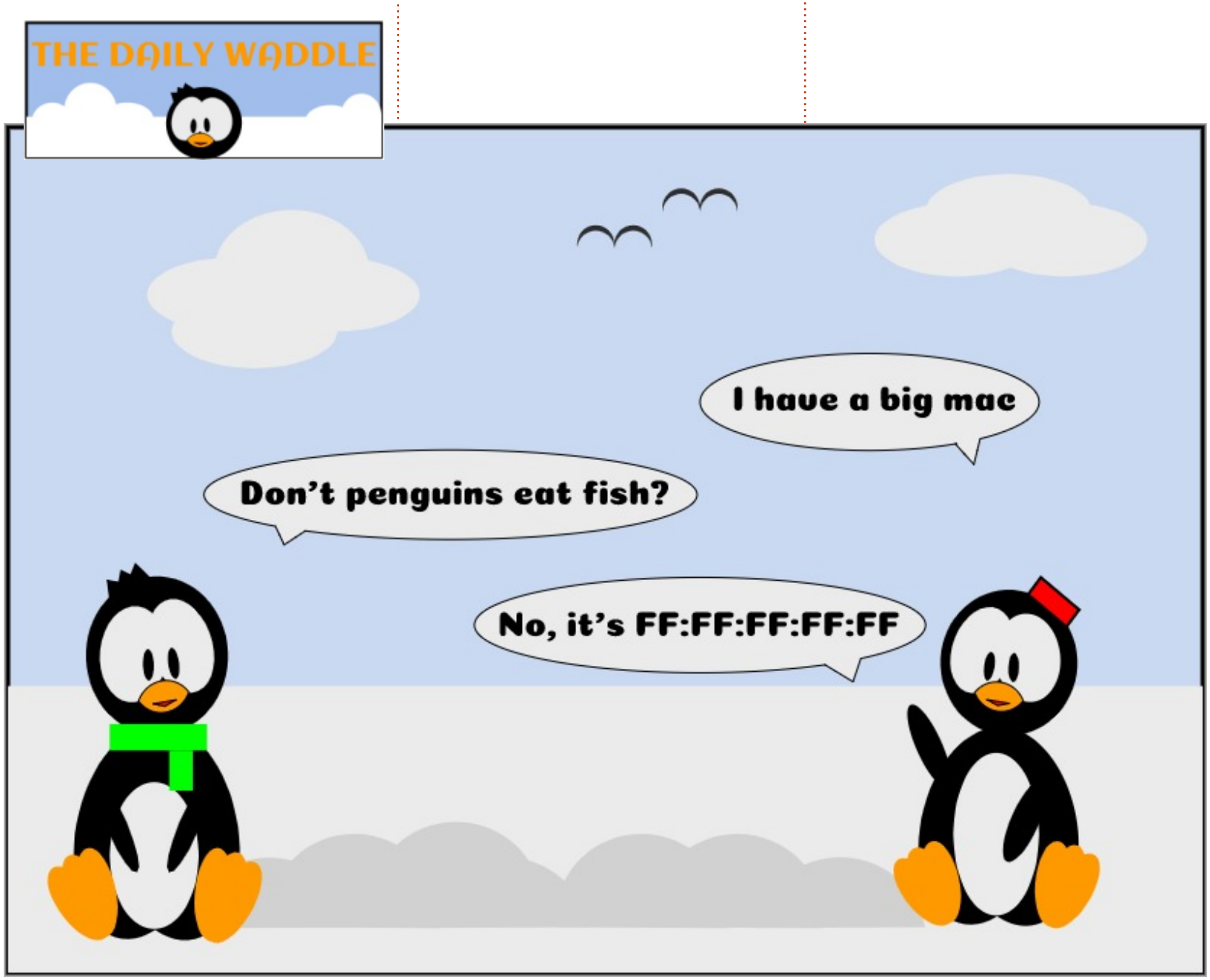
that shown below.

If you want to play around with this effect, but have the computer do a little more of the hard work for you, a discussion of this technique over on inkscapeforum.com led to one user creating an Inkscape extension that will add the CSS animation code for you. You can find the extension at https://gitlab.com/Moini/ink_line_animator/ and follow the original thread at http://www.inkscapeforum.com/viewtopic.php?f=5&t=33721

Just animating a star being drawn is a little dull, but does show you the basic approach. Although the star contains no curves, this method works equally well with any shape of path. An easy step on from this, for example, would be to replace the star with a path for some handwritten text, to produce a "self-writing" effect. Be aware, however, that sharp transitions can cause rendering artefacts depending on the miter limit. In the case of my star, for example, the corner at the start/end of the line flashes on and off as the rest of the line is drawn. It's often better to use rounded corners and

end caps to avoid this, which also helps to reinforce the illusion of the line being drawn by a pen or pencil.

If you're prepared to spend more time hand-crafting your animations there's no reason why you shouldn't produce a self-drawing SVG image consisting of multiple paths, each animated separately using delays to ensure that they appear in the correct order. Once your outline is drawn, some more delayed animations could fade in the fill colours, erase some of the lines, or cross-fade to a raster image. With time and effort, this technique can produce some spectacular results, and all just by moving a dashed line around.



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

Inkscape - Part 79

There's a part of the Inkscape user interface that is so common, and so taken for granted, that you've probably never given much thought to it: the page border. Yet this seemingly mundane part of the display plays a vital role when it comes to creating SVG files for use online. By understanding what it represents, and how to manipulate it, you will open up some techniques that can make your SVG files far more versatile, even if they're only being included via an <img> tag.

By default, creating a new document in Inkscape will create an A4 page (210x297mm), displaying a thin outline to represent the page border, and a drop shadow to make it look a little more like a page of paper than a simple rectangle. I can't say for certain if A4 is used as a default everywhere, or if there is a locale dependency that creates US Letter pages in the USA, for example, but either way, you get a default size and visible page border. Changing the page size is done via the File > Document Properties (Ctrl-Alt-D) dialog: you can either select one of the predefined page sizes or enter a custom width and height, with your choice of units. The bottom of the dialog also provides options to show or hide the page border and to display it without the drop shadow, if you prefer.

I've seen a few video tutorials where the presenter heads straight to this dialog to turn off the page border, but I think that's usually a mistake. If you're designing for print then having an idea of how your work fits into the page is essential. But the border is just as important for web work, as anything that is drawn outside it won't be rendered by the browser. If you turn the border off, there's a real danger that parts of your design might inadvertently fall outside it, preventing them from rendering as expected.

The non-display, or non-printing, of content outside the page border, can be a blessing. If your design has to bleed off the edge of the page, it can be essential. You can also use the area outside the page to store rough designs, notes, spare elements, or the source objects for clones (particularly those with unset fills) – anything which you want to keep with your image, but don't want to be visible in the final product. I've often used this capability to include Easter eggs in my comic strips – even to the extent of holding extra panels or even entire extra strips that can be found only by opening the original source file in Inkscape.

For many uses, the page size can natively be set in the Document Properties dialog, with no need to concern yourself any further. This sets the "width" and "height" attributes on the main <svg> which determines the default size that your image will be drawn in the browser. For an icon design, for example, you might set the dimensions to 32px by 32px, and that's the size it will be rendered. But what happens when you want to use a different scale inside your drawing? Your drawing might be in metres or even miles, yet you still want it displayed at a reasonable size in the browser. For this, there is the viewBox attribute.

The viewBox attribute is a list of four numbers, representing the x and y coordinates of the origin in the drawing, and the width and height of the drawing in "user units". Let's look at a couple of examples:

```
<svg …
     width="100"
     height="200"
     viewBox="0 0 100 200"
     … >
```

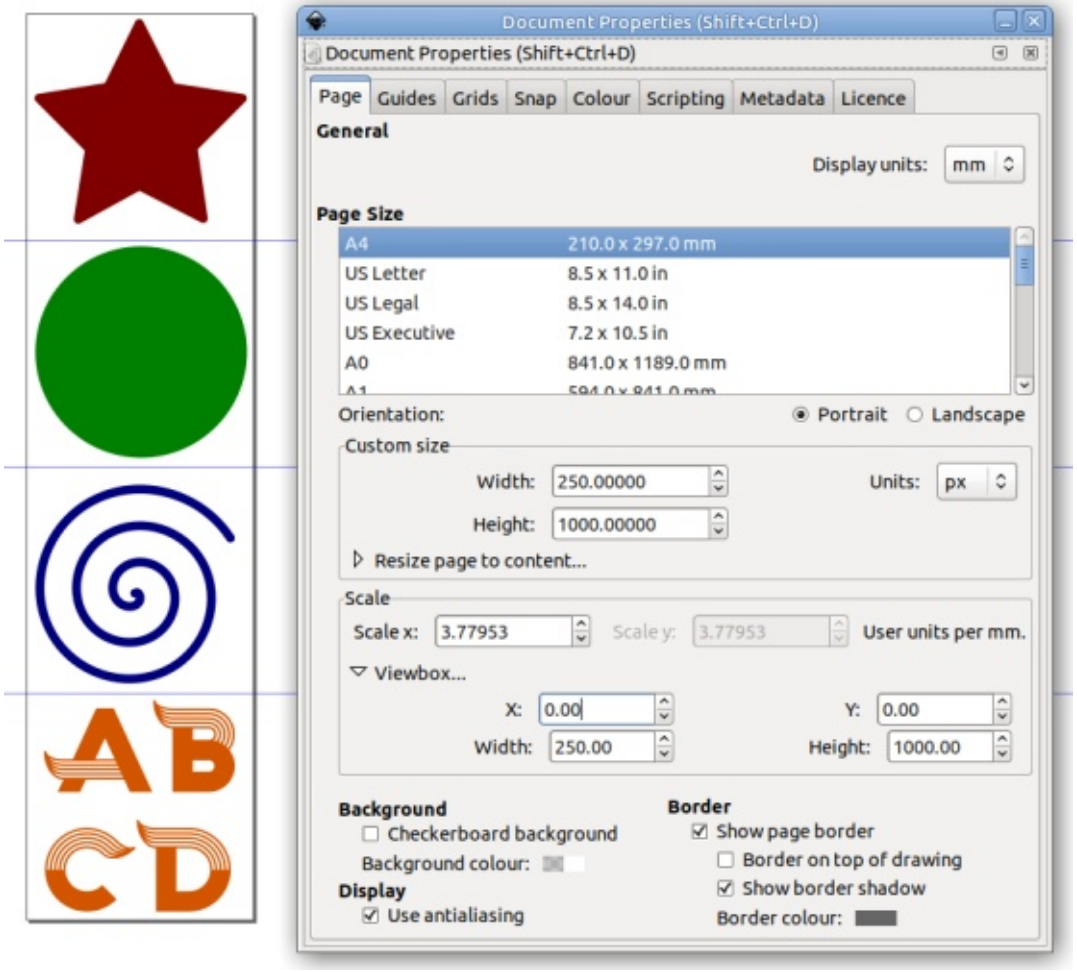
This one's simple. The width and height of the image will default to 100px by 200px, and the coordinates in the drawing are mapped to the image on a 1:1 basis. If you draw a rectangle that's 100 units wide and 200 units tall, it will fill the available space in the browser window (assuming you position it at the top left of your drawing). Let's try another:

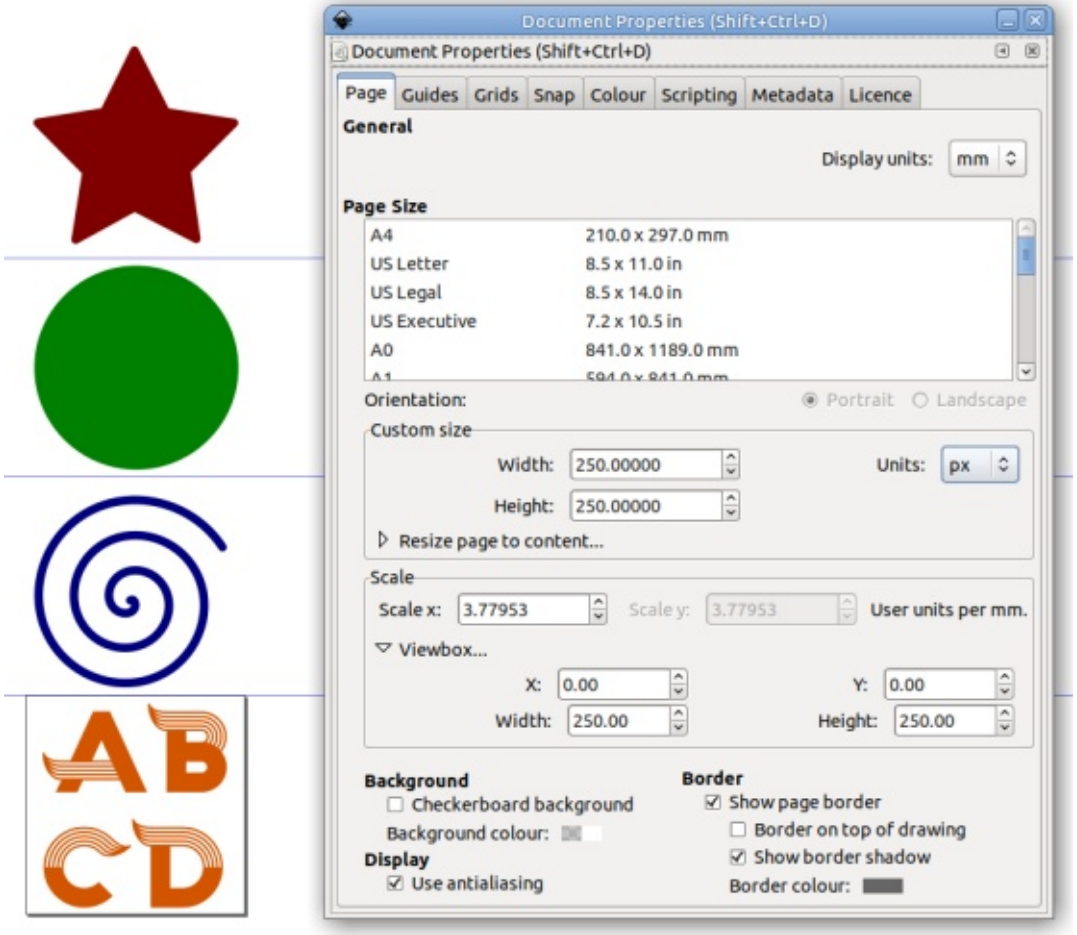
```
<svg …
     width="100"
     height="200"
     viewBox="0 0 500 1000"
     … >
```

Again the image will render to a size of 100px by 200px in the browser. But the viewBox defines a different coordinate system. Now 500 units in the drawing map to 100px in the browser. If you draw a rectangle that's 100 units wide and 200 units tall this time, it will occupy only one corner of the image (actually being drawn as 20px by 40px in the browser). To fill the image, you would need to draw a rectangle that's 500 units by 1000 units.

Changing the x and y values lets you move the origin of your coordinate system. It lets you say "the origin for the browser (i.e. the point that's used as the top left of the image) should actually be 100 units down and 50 units across in my drawing". Another way to think of it is that the viewBox lets you create a viewport into your drawing of a specific size and location: everything inside the viewport will be scaled up or down to fill the image in the browser; everything outside the viewport will be cropped and left un-drawn.

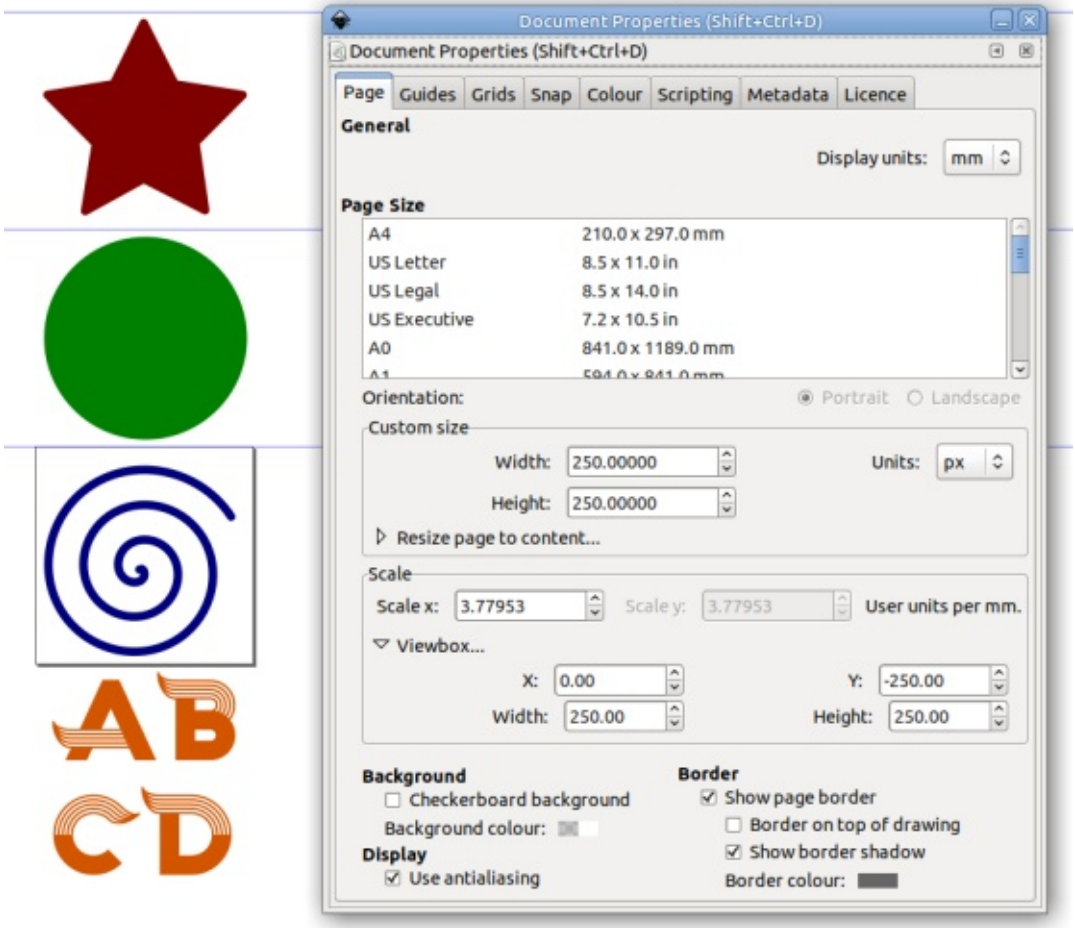
For use in a web page, this capability to render only part of the image lets us perform a rather neat trick. By changing the viewBox values, we can selectively display subsections of the file, letting us store multiple images in one file. This reduces the number of network requests needed by your page, in turn speeding up your site.





Consider this collection of four images – a star, a circle, a spiral path, and some text in the rather wonderful free font, Trump Grotesk Bold. I've drawn them in four sections of a tall, thin page that is 250px wide by 1000px tall (so each element occupies 250px by 250px). With the viewBox set to "0 0 250 1000", we have a 1:1 mapping when the image is drawn in the browser. You can pretty much ignore the "Scale x" and "Scale y" values – they get set automatically by Inkscape as you change the viewBox fields.

As you might expect, when I save this image and load it into the browser, I see all four elements, taking up a space of 250px by 1000px. But look what happens if I change both the page height and the viewBox height to 250:



As you can see, the page border only surrounds the text. If I save the page and load it into the browser, all I get is a 250px by 250px image showing the text element.

If I now set the "y" value for the viewBox to -250, thus moving the viewport upwards, only the spiral appears in the page. Saving the file and loading it in a browser now only renders the spiral, hiding the other three elements.

I'm sure you've worked out by now that setting the "y" value to -500 will put the page around the circle, whilst -750 puts it around the star. Now let's just remind ourselves what the HTML <img> tag looks like to render this image:

```
<img src="views.svg"></img>
```

Well that's pretty terse and to the point. It just tells the browser to show the "views.svg" file using the width and height set in the SVG file, and displaying the default viewport set by the viewBox attribute. But we can append a little magic to the filename to tell the browser to override the default viewbox:

```
<img
src="views.svg#svgView(viewBo
x(0, -750, 250, 250))"></img>
```

By changing the viewBox values in the URL we can therefore select a specific region of the image to display. In this case, it allows us to choose between one of several sub-images, making this approach ideal for files that contain multiple icons or logos. An alternative is to use viewBox values that focus on a particular part of your design, or cause some section to be zoomed in. That gives you the possibility of showing, for example, an overview and a detail view, both taken from the same image.

Hard-coding the viewBox dimensions into the URLs does have one significant problem: if your image changes, such that elements are swapped or moved, you also need to update the HTML or CSS file containing the URLs. SVG has an answer to this problem as well: named views.

Named views are, as you might have guessed, a way of giving a particular set of viewBox values a

name – that can then be referenced from elsewhere. Unfortunately, Inkscape has no specific support for them, except to expose the underlying code in the XML editor. But the syntax is straightforward enough that they're easy to add via a text editor. They can go pretty much anywhere in the SVG file, but as they're not visible objects in their own right, I prefer to keep them in the <defs> section where things such as filter and gradient definitions live. Named views can be thought of as viewBox definitions, so this location makes sense to me. Here's an example of the top section of the SVG file above, once I've added named views for each of the objects to my <defs> (below).

In this example, the viewBox attribute in the <svg> element is set to show the text content, but I could equally have set it to show all four objects, just a couple of them, a smaller part of one of the objects, or any other rectangular space in the image. This is the viewBox that will be used by default if nothing else is specified in the document's URL.

In the <defs> section, you can see that I've also defined four <view> elements. Each of these has an ID that will be used to reference them later, together with a viewBox attribute. The IDs aren't special: I've called them "textView", "spiralView" and so on, simply to make it clear what they're showing, but I could

```
<!DOCTYPE html>


<html>
<head>
  <title>SVG viewBox tutorial</title>
</head>


<body>
  <div>
    <img src="views.svg#svgView(viewBox(0, -250, 250, 500))"></img>
    <img src="views.svg#circleView"></img>
    <img src="views.svg#starView" style="width: 100px;"></img>
    <img src="views.svg#svgView(viewBox(0, 0, 140, 125))"
        style="width: 300px;"></img>
  </div>

  <br />

  <div>
    <p>Look! <img src="views.svg#starView" style="width: 1em;"></img>
    A single SVG file <img src="views.svg#spiralView"
        style="width: 1em;"></img> used for
    <img src="views.svg#circleView"
        style="width: 1em;"></img> four different inline icons!
    <img src="views.svg#svgView(viewBox(138, 0, 140, 125))"
        style="width: 1.5em; vertical-align: bottom;"></img>
    </p>
  </div>
</body>
</html>
```

```
<svg …
     width="100"
     height="200"
     viewBox="0 0 250 250"
     … >
     <defs id="defs">
            <view id="textView" viewBox="0 0 250 250" />
        <view id="spiralView" viewBox="0 -250 250 250" />
        <view id="circleView" viewBox="0 -500 250 250" />
        <view id="starView" viewBox="0 -750 250 250" />
     </defs>
<!-- Rest of SVG file follows… -->
```

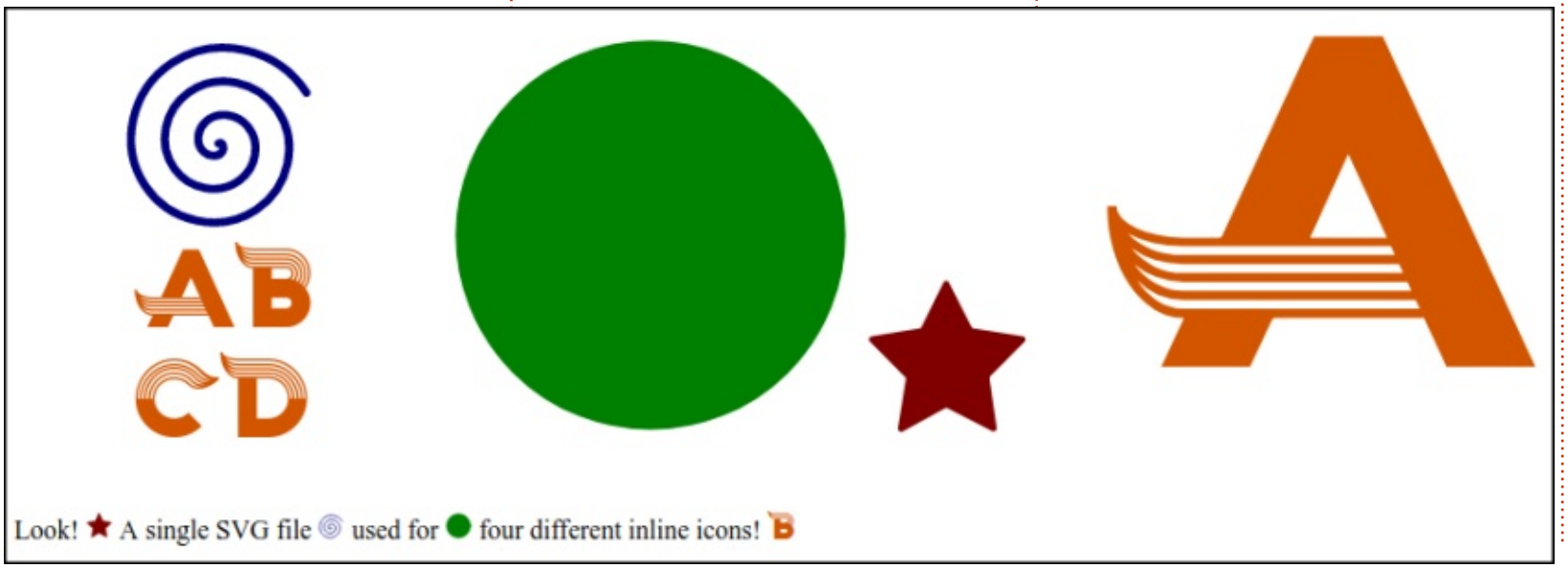
equally have gone for “fred”, “wilma”, “barney” and “betty” – had I wished to. The only requirements are that they are valid XML IDs, and are unique within the document. If in doubt, stick to plain text with no punctuation and you should be okay.

Using your named view in an <img> tag is trivially straightforward – you just have to set the appropriate ID as the fragment identifier (the bit after the “#” character in a URL):

```
<img
src="views.svg#spiralView"></
img>
```

Of course there’s nothing to stop you referencing the same image more than once in a web page, with a different fragment identifier each time. You can also mix and match named views, the “svgView()” syntax, and the default viewBox. In this way, a single SVG image can easily be used to provide a whole host of icons and other images for your page. To finish off, here’s an example of an HTML document that uses the SVG image from this tutorial:

You should now be able to understand how our single SVG image is used multiple times to give the final result:



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