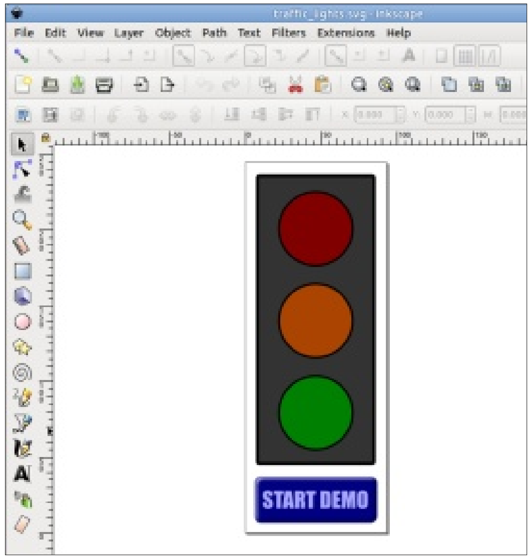
Written by Mark Crutch

Inkscape - Part 87

Last time, we used a linked JavaScript file to create an SVG file containing a circle that changes color when a button is pressed, when loaded into a web browser. This time, we'll extend that simple example to show how the combination of SVG and JavaScript is ideal for animated demonstrations, by implementing a set of traffic lights. Our lights are the sort of thing that might appear on an educational site, or in a museum: they'll cycle through the sequence of colors (red, red and amber, green, amber, red) once when the button is clicked. But first, we need some traffic lights:


traffic_lights.svg - Inkscape
File Edit View Layer Object Path Text Filters Extensions Help
START DEMO

For simplicity, I've based this on the file I created for the previous instalment. That means the red light already has an ID ("redCircle"), and the JS file is already linked. I've duplicated the red circle to create the amber and green ones, giving them corresponding IDs of "amberCircle" and "greenCircle". Because the JS is linked, and the red circle has the same ID as last time, loading the file into a web browser draws the lights okay, and clicking the button toggles the red light to green and back. Clearly there's more work to be done, but at least the fundamentals are in place.

For this demo, we want the dull colored lights to turn bright at the correct times. Let's forget about the timing for now, and deal with the colors first. With a variation on our existing code, we could easily set each light to a specific color by targeting it using its ID, then setting the "style.fill" property directly. A better approach, in this case, is to use classes. We can set a class for each light onto some ancestor object, and use CSS to apply the right fill. Since classes can be combined, we don't need a "red-and-amber" class; we can just set the "red" and "amber" classes at the same time.

But before we get too far ahead of ourselves, we need to set some default colors in CSS, so that we can override them later using classes. Open the file in a text editor, and find the <style> section (or add one, as a child of the <svg> element, if there isn't one already). Put in some ID selectors, with the base colors you want to use – something like this:

```
<style>
  #redCircle {
    fill: #800000;
  }

  #amberCircle {
    fill: #aa4400;
  }

  #greenCircle {
    fill: #008000;
  }
</style>
```

Don't worry if there's already content in your <style> block, such as the rules used to style the button – just add the new code to the end. Don't forget that you also have to remove the "fill" properties from the style attributes on the <circle> elements, otherwise they'll just override anything set in the <style> block. A good test is to change all the colors in the CSS to "blue" and reload your page – if you still see red, amber or green then you have an override entry on the elements themselves.

Now, we need to add the colors that we want to use when each light is turned on. It's just another set of three styles added to the end of the <style> block, before the closing tag:

```
  ...

  .red #redCircle {
    fill: ff0000;
  }

  .amber #amberCircle {
    fill: #ff6600;
  }

  .green #greenCircle {
    fill: #00dd00;
  }
</style>
```

Each of these rules is similarly structured, and can be read as “set this fill color for the element with a specific ID, but only if one of its ancestors has a specific class”. With this method we can set classes of “red”, “amber” and “green” on some ancestor element of the lights, such as the parent layer, or even on the root <svg> element, in order to activate the lights. So let’s do that…

We’ve already seen how to use document.querySelector() with an ID to retrieve a particular element. To add our classes to the Inkscape layer would simply be a case of finding the right ID for the relevant <g> element. But to demonstrate a different approach, we’ll get a reference to the root <svg> element instead, then add a class to that. If you followed along last time you should already have a JavaScript file with a buttonPressed() function. Let’s replace the content of that function with this:

```
function buttonPressed() {
  const svg =
document.documentElement;

svg.classList.toggle("red");
}
```

The document.documentElement property returns the root element of an XML or XML-alike document. In the case of an SVG file, it returns the <svg> element; for an HTML document, it returns the <html> element, and so on. So you can see that this simple two-line function will toggle the “red” class on the <svg> element, and our CSS is written such that this should alter the color of the red circle. Save the code, reload the image in your browser, and check that clicking the button does, indeed, toggle the shade of red for the first circle.

Replace the word “red” with “amber”, save the file, reload the web page, and click the button again. Then do the same with “green” as the class name. Make sure that each light works as expected before moving on.

Our traffic light sequence includes one step in which two lights must be illuminated at once – requiring us to set two classes. In an ideal world, the classList.toggle() method would be flexible enough to take a parameter of “red amber”, and toggle both classes. But we’re stuck in a less than ideal world – one in which the classList methods all work with a single class at a time – so to toggle both classes requires the method to be called twice:

```
function buttonPressed() {
  const svg =
document.documentElement;

svg.classList.toggle("red");

svg.classList.toggle("amber")
;
}
```

In our demo we don’t actually want to toggle lights on and off – we just want to set a fixed selection of lights for each step, without having to also turn off lights from the previous step, or call the same method multiple times. Using the classList interface actually makes life more difficult for us, when all we want to do is set the “class” attribute to a specific value. Luckily for us, browsers provide a function for setting the value of an attribute. It’s got the sensible name of setAttribute() and its arguments are the name of the attribute to set, and the value to set it to. Let’s use it to turn on both the red and amber lights:

```
svg.setAttribute("class",
"red amber");
```

If you try this in your code you’ll find that you can turn the lights on, but as we’re no longer using a toggling function you can’t turn them off again without reloading the page. But we’re not really interested in toggling – we want a sequence of particular lights. For that, however, we need a little foray into the history of JavaScript…

Back in the early days of the web, JavaScript was executed as part of the same “thread” as the browser code itself. This meant that the browser would effectively hand over control to the script, and couldn’t update its UI, or respond to input, until the JS code relinquished that control. You might remember the bad old days when a rogue web page could hang the browser, preventing you from doing anything else with either the page itself or the browser UI. So, JavaScript doesn’t contain any instructions to pause execution of the script, as doing so would block the browser entirely. That means we can’t sequence our lights with something as simple as this pseudo-code:

```
svg.setAttribute("class",
"red");
pause(3000);
svg.setAttribute("class",
"red amber");
pause(3000);
svg.setAttribute("class",
"green");
...
```

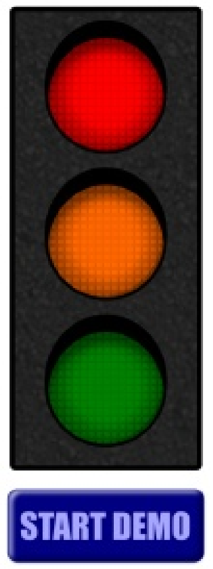
Instead JavaScript has a function called setTimeout(). This is a mechanism for queuing up a function call for later on. It doesn't pause execution of the current function, but asks the browser to run another function after at least a certain amount of time has passed. It takes two parameters: a function or reference to a function, and the minimum timeout in milliseconds. With this, we can write a series of functions that call each other in sequence, to create our demo:

```
function buttonPressed() {
  const svg =
document.documentElement;
  svg.setAttribute("class",
"red");
  setTimeout(redAmber, 3000);
}

function redAmber() {
  const svg =
document.documentElement;
  svg.setAttribute("class",
"red amber");
  setTimeout(green, 3000);
}
```

```
function green() {
  const svg =
document.documentElement;
  svg.setAttribute("class",
"green");
}
```

That covers the first three steps of the sequence. I'll leave it as an exercise for the reader to extend it to the full five steps described at the start of the article.



With that, our traffic lights demo is almost complete. Clicking the button will begin the sequence, which will stop automatically at the end. All that's left is a little more work in Inkscape to make the lights look better. Provided you don't change the CSS we added, or remove the link to the JS file, you're free to tweak the design as much as you like without fear of the interactivity being broken. Here's my result, partway through its cycle, after a little work in Inkscape with a few gradients and some lines.

Comparing this to the first image in this article shows just how much impact a little extra design work can achieve. But if I'd done all that design work first and only then started adding JavaScript, I would have had to deal with a much more complex SVG file for the few manual edits required. If you possibly can, it's usually better to focus on the core aspects of your animations or interactions, and get those working on a simple version of your file first.

If you're careful, editing your file in Inkscape shouldn't break your code and interactions. But because work like this requires flitting back and forth between Inkscape and a text editor, both working on the same file and therefore able to interfere with each other's contributions, I strongly advise taking very frequent backups as you work.

Next time, we'll have a quick look at some other options for manipulating SVG using JavaScript, moving beyond a few changes of fill and stroke color into altering other aspects of your images.

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

contents ^

Written by Mark Crutch

Inkscape - Part 88

So far in this series, we've used some JavaScript to change the fill or stroke color of an object in an SVG file when loaded in a web browser. But JavaScript in SVG is the same language, powered by the same engine, as JavaScript in HTML. That makes it a powerful tool for doing far more than just tweaking some colors.

First, a quick reminder of the structure of an XML tag, of the sort you might find in an SVG file:

```
<tagName id="uniqueID"
attributeName="attributeValue
">textContent</tagName>
```

Let's look at each part of this individually:
• tagName – The name of the tag or element. In SVG, this might be a 'g' for a group, or 'rect' for a rectangle or square, for example.
• id – This is just an attribute that happens to be named 'id', but the rules of XML dictate that IDs must be unique within a document. That makes them handy for targeting with the querySelector() function.
• attributeName – Each tag may have zero or more attributes which contain additional data associated with the element. In XML languages, these always take the form of attributeName="attributeValue", whereas HTML (confusingly) allows for some attributes that have no value associated with them. Each attributeName must be unique within the element, but may appear many times across different elements. The attributeValue will vary depending on what the attribute is actually used for.
• textContent – This is not so common in XML. Usually, an element will contain zero or more child elements before the closing tag (the </tagName> in this example), but a few elements allow for plain text to be included. In SVG, the most common cases are <text> and <tspan> elements, where the plain text holds the text string that will be rendered.

There are also a couple of variations to be aware of. Self-closing tags take the form <tagName … />. By definition these can have no children or text content. XML documents also make use of namespaces, which are defined in the main tag for the document (e.g. the <svg> tag), and may then crop up appended to tags and attributes with a colon. You won't see these often: usually a default namespace is declared, in which case namespaces need to be added only to tags and attributes that are from 'foreign' XML languages.

The theory is fine, but let's see how these parts manifest themselves with yet another super-simplified SVG file:

```
<svg
xmlns="http://www.w3.org/2000
/svg" viewBox="0 0 100 100">
  <text id="text" x="50"
y="50" text-anchor="middle">
    This is <tspan
id="ts1">some</tspan> <tspan
id="ts2">SVG text</tspan>
  </text>
</svg>
```

Breaking this down, we have an <svg> tag containing a <text> tag with some further content. The <svg> tag has a couple of attributes. The first defines the default namespace, and is required so that the browser knows this is a document conforming to the W3C's SVG spec, and not some other type of file that happens to have a tag name called 'svg'. The second attribute sets up the coordinate space we'll be using in this file – I usually stick with "0 0 100 100" for my hand-created files, as I can then treat my values as percentages within the image.

The <text> tag also has some attributes. The ID is self-explanatory. The others set the 'anchor point' for the text to the middle of the image (50, 50), and indicate that the anchor point should be in the middle of the text (i.e. the text is centered, not left- or right-aligned).

Finally the <text> tag contains a mixture of text content and a couple of <tspan> elements with IDs, which will allow us to specifically target those parts of the text via JavaScript.

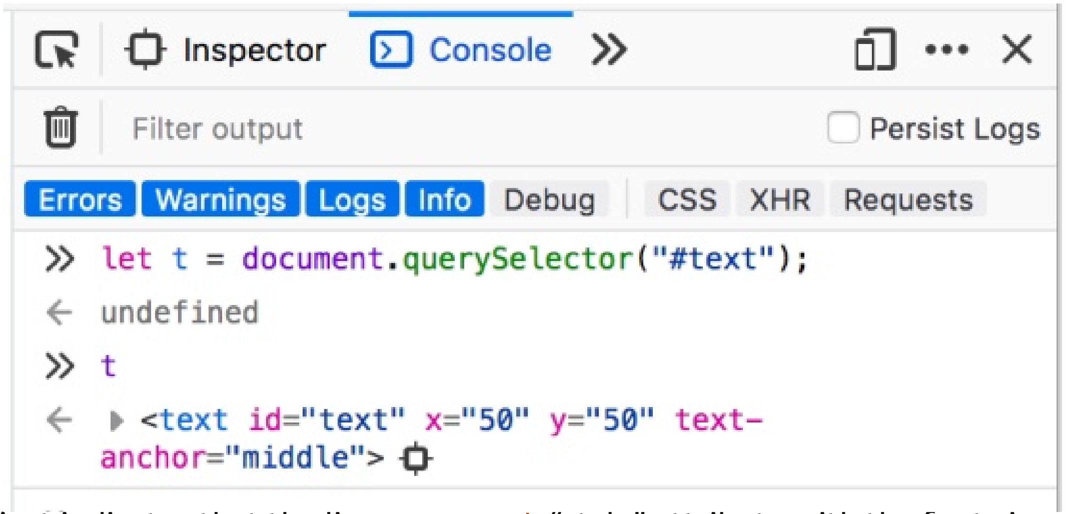
Save the file and load it into a web browser – preferably Firefox or Chrome, as they have better developer tools than most others.

From the previous articles, you already know how to add JavaScript to your SVG file, either directly in Inkscape or by linking to an external JS file, but we won't be doing that today. For the rest of this article, we're going to rattle through a few ways you can affect your SVG, but we'll do so within the browser's developer tools. Any of these commands or techniques can be added to your own JavaScript if you want to create something less ephemeral.

Press F12 or use the menu to open your browser's developer tools. Somewhere along the top should be a row of tabs (though they're not always clearly styled as such). Make sure you have the "Console" tab selected. If the panel is already filled with text, find the button in the console's toolbar to clear it, for clarity. Click inside the console area to give it the focus, and type the following (followed by the Enter key):

```
var t =
document.querySelector("#text
");
```

The console will display the string "undefined" at this point. That's nothing to worry about, it



```
>> let t = document.querySelector("#text");
<- undefined
>> t
<- ▸ <text id="text" x="50" y="50" text-
  anchor="middle">
```

just indicates that the line you entered didn't return a value. But what it has done is find the element with an ID of "text" and assign it to the variable "t". You can confirm that by typing the letter "t" on its own, then pressing Enter. The console should show a representation of the <text> element, looking something like that shown above.

Let's use some JavaScript we already know to reduce the size of the font a little. Type this into the console:

```
t.style.fontSize = "10px";
```

The SVG content should react as soon as you press the Enter key. Type the letter "t" again and you'll see that the element now has a "style" attribute with the font-size property set. Notice that we set "fontSize" in JS, but the CSS in the attribute shows "font-size". If you tried to use the latter in JavaScript, it would be interpreted as trying to subtract the "size" variable from the "font" variable, and would throw an error. As a general rule, any CSS property containing embedded hyphens is available as a JavaScript property by removing the hyphens and capitalising the first letter of all but the first word.

Breaking down the line above, you know that "t" is a JavaScript representation of our XML node. The browser exposes various properties and methods (functions tied to a specific object) on that node, including the "style" property. This property, in turn, has a "fontSize" property, which we've set to a value of "10px". But the browser treats the "style" property a little differently to most JavaScript properties, and instead also applies any changes to the "style" attribute in the XML. In this instance, it doesn't matter whether you change the attribute or the property – but that's not usually the case.

To change most attributes, therefore, you can't just set a correspondingly named JavaScript property. Instead, you have to use the setAttribute() method that we've looked at previously. Here's how we might move the text up a little:

```
t.setAttribute("y", 20);
```

Type "t" again to see the XML, and you'll notice the "y" attribute now has a value of "20". We can also retrieve that value using the getAttribute() method:

```
t.getAttribute("y");
// Returns "20"
```

Remembering that the y-axis in SVG runs from the top of the screen to the bottom, you might be inclined to try some code like this to move the text down by 10 units:

```
var yPos =
t.getAttribute("y");
t.setAttribute("y", yPos +
10);
```

Gah! Where did the text go!? Actually it's still there, but it's been positioned so far down in the image that it's dropped out of the 100x100 viewBox, so isn't visible. But why is that, when we just wanted to adjust the value from 20 to 30?

The problem is that XML is a text-based system, and doesn't really have a concept of different data types. All attributes are therefore text strings, regardless of the value you put in, so our call to getAttribute() returns the string "20", not the number 20. JavaScript then tries to be 'helpful' by determining that we're trying to 'add' the number 10 to the string "20". Since you can't add a number to a string, it automatically converts the number into a string ("10"), then concatenates the two, to give a result of "2010". That's the value we end up putting into the attribute in our setAttribute() call, so our text ends up being moved to a y-position of 2010 units!

We can fix this by converting the value returned from getAttribute() into a number. We only want an integer value, so the parseInt() function is the tool to use – but there is also a parseFloat() if you need to deal with decimal fractions. parseInt() has a second parameter for the number base that you should always provide (with a value of 10 for a decimal conversion) to avoid some rare-but-odd corner case bugs when converting certain strings. Entering the following lines into the console should get us the result we were looking for:

```
t.setAttribute("y", 20);
var yPosNumeric = 0;
yPos = t.getAttribute("y");
yPosNumeric = parseInt(yPos,
10);
t.setAttribute("y",
yPosNumeric + 10);
```

You can run the last three lines repeatedly to move your text down by 10 units each time.

Now we know how to get and set attributes, but you can also remove them entirely. This will get rid of the "style" attribute we indirectly created earlier, returning the text to its 'natural' size:

```
t.removeAttribute("style");
```

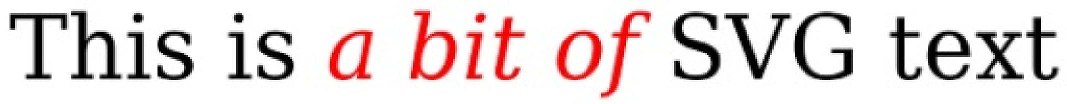
There's no equivalent createAttribute() call - setting the value of a non-existent attribute using setAttribute() will automatically create it. Let's get our style back by manipulating the attribute rather than the property:

```
t.setAttribute("style",
"font-size: 10px;");
```

As well as working with attributes, you can also dynamically change the text content of an element. Let's type a few lines into the console to alter the first <tspan>:

```
var ts1 =
document.querySelector("#ts1"
);
ts1.style.fill = "#ff0000";
ts1.style.fontStyle =
"italic";
ts1.textContent = "a bit of";
```

[relevant image shown above]

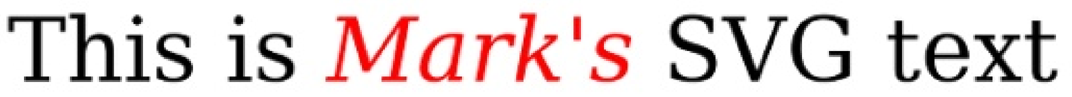


Being able to change the text content via JavaScript opens up a world of possibilities, including images with descriptions that can be switched between different languages, or ones that populate with data requested from a server somewhere such as live graphs and stock tickers. That degree of sophistication is a little beyond this series, but here's a trivial example that prompts the user to enter their name, then updates the text on the page accordingly:

```
ts1.textContent =
prompt("What is your name?")
+ "'s";
```

[relevant image shown below]

Modifying the properties, attributes and text content of existing elements is useful, but to have complete control over a document we also need to be able to add and remove elements using JavaScript. The removal part is trivial, provided you can get a



reference to the element using querySelector() or some other mechanism. Let's delete our first <tspan> entirely:

```
ts1.remove();
```

Adding a new element to the page can be trivially easy, or it can be rather convoluted. Let's start with the easy method, by adding another <tspan> to the <text> element, which is still assigned to our "t" variable:

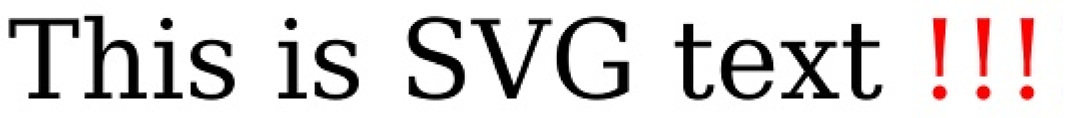
```
t.innerHTML += '<tspan
id="ts3" style="fill:
red;">!!!</tspan>';
```

Even though we're working on an SVG file, which is a form of XML document, we still have to use the "innerHTML" property. This returns all the descendants of the specified node as a string – basically a string of HTML (or XML in this case) much like the ones you type into a text editor. The "+=" operator essentially retrieves a value, adds or concatenates something to it, and puts the result back into the same place. In our case it has the effect of appending a new <tspan> to the end of the existing content.

Let's do something similar, but with a more complex approach…

```
var ns =
"http://www.w3.org/2000/svg";
var newTS =
document.createElementNS(ns,
"tspan");
newTS.id = "ts4";
newTS.setAttribute("style",
"fill: blue");
newTS.textContent = "!!!";
t.appendChild(newTS);
```

That's a lot more lines to explain:
• We set up a variable, "ns", that will hold our SVG namespace. Usually this is done once at the top of the JavaScript so you can use it in multiple places.
• We create a new <tspan> element. If you've ever done this in HTML, you might be familiar with document.createElement(), but, in the XML world, we need to use a namespace-aware equivalent, createElementNS(), and pass the namespace as the first parameter.
• We give the element an ID to make it easier to get hold of later. We could have used setAttribute() for this, but the browser has an implicit mapping between the property and attribute in this case, in the same manner as we saw earlier with the 'style' property.
• Now we can set an attribute on the new element. We would need to repeat a line like this for each attribute we wish to set.
• We've created a <tspan>, so we won't see much unless we also give it some text content.
• Finally, we append it as a child of the object referred to by the "t" variable – our <text> element.



Clearly that's a lot more typing than the innerHTML version, so why would you ever want to take this approach? Precisely because it's verbose, splitting the element, attributes and text content into separate lines, it lends itself to some types of looping or manipulation that can otherwise become unwieldy when using just a single chunk of text.

Consider trying to plot a graph using SVG. Each point on the graph might be represented by a <circle> requiring several attributes: x, y, r and fill, for example. These values will be determined by some data source, and may need to be manipulated to get them into the right format for SVG. All of that is a little easier to arrange, and can lead to clearer code, if you deal with each attribute separately. Certainly it can be done with the "innerHTML" approach, but as the code and SVG content become more complex, an approach that relies on building and manipulating strings can become harder to follow, and less robust.

Next time, we'll build on the techniques used in this instalment, to further investigate ways to manipulate the individual elements in an SVG document through JavaScript.

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

Written by Mark Crutch

Inkscape - Part 89

Last time, we looked at some different ways to use JavaScript to modify your SVG file dynamically in a web browser. We concluded with a verbose way to create a new SVG element, set its attributes, and append it to an existing element. In this instalment, we'll build on those same ideas to do even more with our elements – so go ahead and re-read last month's column if you need a refresher before we plough on.

Our test file last time was made up primarily of an SVG <text> element – picked because it's one of the few SVG elements that has text content within it, and I wanted to demonstrate how you might go about getting and setting such content. Most SVG elements, however, have either no content or only other elements as their children, so that's the sort of structure we'll focus on this time. Here's the SVG file you'll need to create as a starting point:

```
<svg

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

viewBox="0 0 100 100">

</svg>
```

Okay, that's admittedly a pretty terse SVG file, even by the standards of this series, but that's because it doesn't actually have any content. Instead we're going to create all the content dynamically, using the browser's developer tools as we did last time. So save the file, load it into a browser, open the developer tools, switch to the Console tab, and clear any existing messages. Phew! Now that we're ready to proceed, let's begin by creating a square using the 'simple' approach from last time:

```
var svg =
document.querySelector("svg")
;

svg.innerHTML = '<rect
id="s1" x="10" y="10"
width="50" height="50"
fill="red" />';
```

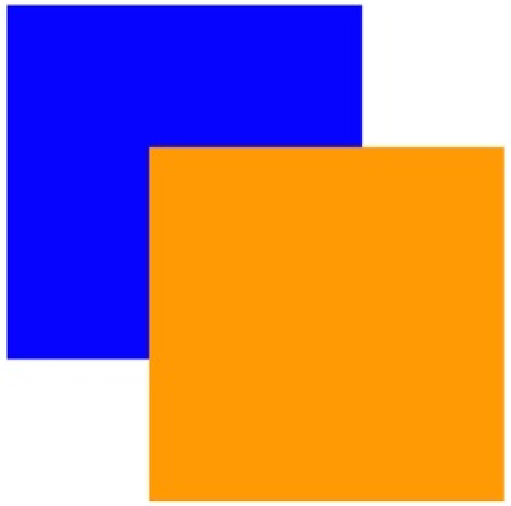
This 'innerHTML' approach is simple, and can create complex nested structures, but it doesn't return a 'handle' that we can use to further manipulate the created content. What if we now want our red square to be blue? We'll need to do something like this, relying on the fact that we gave the square an ID:

```
var square1 =
document.querySelector("#s1")
;

square1.setAttribute("fill",
"blue");
```

The more long-winded approach we took last time gives us a JavaScript object representing our element which we then use to set all the attributes. But we can hang on to that handle to use later on, if we wish. Let's add another red square, this time using the verbose method (below):

```
var ns = "http://www.w3.org/2000/svg";
var square2 = document.createElementNS(ns, "rect");
square2.id = "s2";
square2.setAttribute("x", 30);
square2.setAttribute("y", 30);
square2.setAttribute("width", 50);
square2.setAttribute("height", 50);
square2.setAttribute("fill", "orange");
svg.appendChild(square2);
```



Now if we want to change the color of the second square, we can simply use the same 'square2' variable we used when creating it, even though it's now been added to the page:

```
square2.setAttribute("fill",
"yellow");
```

I'll spare you a screenshot – I'm sure you can guess what it looks like now.

So far, we've mostly revised the content from last month, but in doing so you've got a nice arrangement of two squares, a yellow one on top of a blue one. But why are they in that order? Why isn't the blue one on top? You might think it's because we created the blue one first, and the yellow one second – and to some extent you'd be right. But there is a little more to it than that.

The reason isn't that we created the squares in a particular chronological order, but rather that they ended up in the XML structure in a particular document order. When we added the second square we used the appendChild() method, which inserts it as the last child of the selected parent, so our XML structure ends up looking roughly like this:

```
<svg>
     <rect id="s1" />
     <rect id="s2" />
</svg>
```

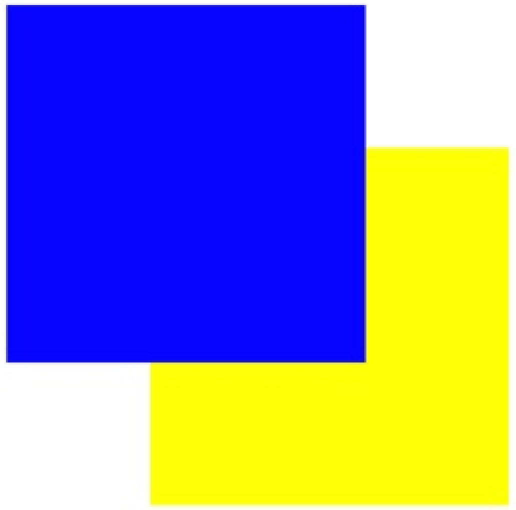
The blue <rect>, with ID "s1" is first in the document, so it gets drawn first. The yellow <rect> ("s2") is second in the document, so gets drawn second. SVG uses what's called the "painter's model" in which later objects in the document are painted on top of earlier objects – so the yellow square is rendered on top of the blue square.

If you're familiar with HTML and CSS, you might imagine that you could override this ordering using the "z-index" CSS property. Unfortunately, that approach doesn't work for SVG. The SVG2 spec does add z-index but, as with many of the useful additions in SVG2, no browser yet supports it. Currently, if you want to put things into a particular stacking order, you have no choice but to rearrange the content of your SVG document.

So how would we go about putting the blue square on top of the yellow one? It's a two step operation: first we remove the blue square from the document, but keep it hanging around in memory; then we insert it back into the document, at the end. Given that we've already assigned the blue <rect> to the "square1" variable, we can use these two lines of JavaScript to achieve our goal:

```
square1.remove();

svg.appendChild(square1);
```



So now we know how to add an object to the top of the image, and how to move an object from a lower level up to the top. What about inserting a new object at the top of the document, so that it appears right at the bottom of the stack? If appendChild() adds it to the end of the document, surely insertChild() will put it at the start?

```
var square3 = document.createElementNS(ns, "rect");
square3.id = "s3";
square3.setAttribute("x", 40);
square3.setAttribute("y", 40);
square3.setAttribute("width", 50);
square3.setAttribute("height", 50);
square3.setAttribute("fill", "pink");
svg.insertChild(square3);
```

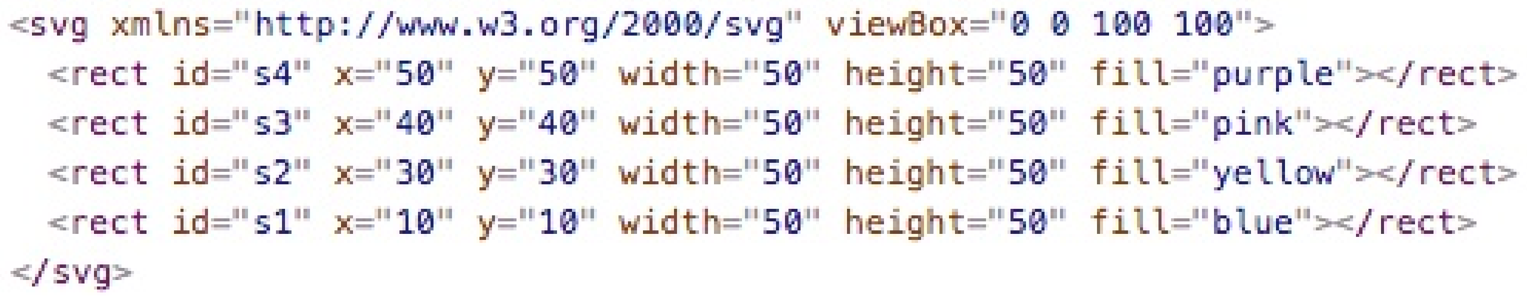
Uh-oh! That's not so good. The problem is that there's no XML method called insertChild(), regardless of how much sense it would make. Instead, you have to insert your node into the document before another reference node – in other words, you have to specify that you want to insert it before the existing first child. Given that our yellow square is first in the document, and we already have a handle to it in our "square2" variable, we can run the following JavaScript line in the console to inject "square3" into the "svg" parent, before "square2":

```
svg.insertBefore(square3,
square2);
```

This is great when you've already got a handle to the first child element, but that's not always the case. Perhaps it was inserted dynamically by some other code, or you've just lost track of which element is which. You can always append a new node at the end of the parent's list of children, so it would be useful to have an equivalent bit of code to insert a new node at the start of the list. Every XML element has a "firstElementChild" property that can be used to retrieve a handle to its first child (skipping any text content) without needing to know anything more about it. We can use this to insert another element at the bottom of the stack (top right).

```
var square4 = document.createElementNS(ns, "rect");
square4.id = "s4";
square4.setAttribute("x", 50);
square4.setAttribute("y", 50);
square4.setAttribute("width", 50);
square4.setAttribute("height", 50);
square4.setAttribute("fill", "purple");
svg.insertBefore(square4, svg.firstElementChild);
```

There's still a bit of a gap between the yellow and the blue squares. Time to insert an element in the middle of the list of child nodes. As a reminder of where we stand at the moment, switching to the "Inspector" (Firefox) or "Elements" (Chrome) tab in the developer tools will show you the current state of your XML document:



```
<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 100 100">
  <rect id="s4" x="50" y="50" width="50" height="50" fill="purple"></rect>
  <rect id="s3" x="40" y="40" width="50" height="50" fill="pink"></rect>
  <rect id="s2" x="30" y="30" width="50" height="50" fill="yellow"></rect>
  <rect id="s1" x="10" y="10" width="50" height="50" fill="blue"></rect>
</svg>
```

To mix things up a little, we're not going to create a brand new square this time – instead we'll create a copy of an existing one. In browser terms, we're going to create a "clone" of the node, but don't confuse it with Inkscape's concept of clones – the two are completely different things (Inkscape's "clones" are actually implemented as SVG <use> elements). First, let's clone our purple square – which we've still got assigned to the "square4" variable – and assign the clone to a highly-imaginatively named variable:

```
var square5 =
square4.cloneNode(true);
square5.id = "s5";
square5.setAttribute("x",
20);
square5.setAttribute("y",
20);
```

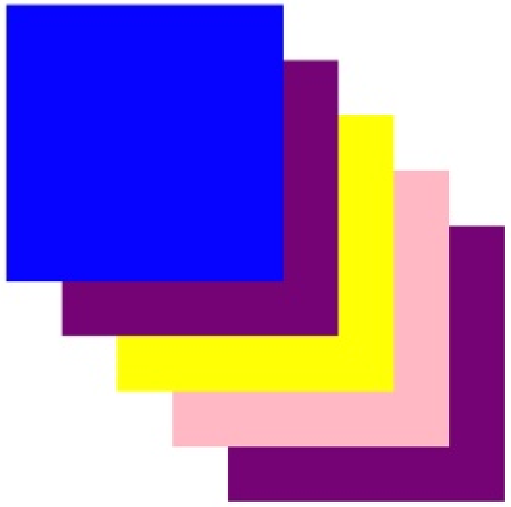
All we needed to do was call the cloneNode() method of the node we wish to duplicate. The "true" parameter ensures that we clone not only the node itself, but any descendants it may have – if we had passed "false" instead, we would get only a duplicate of the node itself. In this case, the results are identical, since our <rect> has no children. But consider cloning a <text> or <g> element, where the content inside is just as important as the node itself, and you can see why passing "true" is usually the safest option.

You'll also note that I've changed the ID of the cloned element. We're about to put it back into the same document and, although browsers don't enforce it, the XML rules specifically prohibit duplicate IDs in a single document. For the sake of correctness, therefore, we change the ID while the cloned node is still just a fragment in memory that hasn't yet been inserted into the SVG.

To insert the clone, we're just going to use the same insertBefore() method we used earlier. But, this time, our reference element (the one we're inserting before) will be the blue <rect>. Of course we could use the reference we already have to it (square1), or get a fresh reference using document.querySelector(), but instead, we're going to do something more generic. We've already seen a generic way to insert before the first child, now we're going to write some equally generic code to insert just before the last child:

contents ^

```
svg.insertBefore(square5,
```



```
svg.lastElementChild);
```

As a reminder, the last child is the one that's drawn on top (the blue square), so inserting just before the last child actually puts the cloned purple square below the blue square in the z-order.

The "firstElementChild" and "lastElementChild" properties are useful shortcuts, but you don't always want to use the first or last child as your reference point. For more general purpose requirements, XML nodes have a "children" property, which returns a collection of all the children. A "collection", for what it's worth, can be described as an object that's a bit like an array except different enough to be annoying. So don't expect to have access to all the array methods, but you can read the "length" property, and reference individual child nodes using a square bracket syntax:

```
// How many child nodes are
there?

console.log(svg.children.leng
th);


// Remove the third one

svg.children[2].remove();


// Remove the penultimate
node, regardless of how many
there are

svg.children[svg.children.len
gth – 2].remove();
```

Remembering that array (and collection) indexes start at zero, it should be clear why the third child has an index of 2. For the same reason the index of the last child is always "children.length – 1", so the penultimate node will be "children.length – 2". Of course if there are fewer than two child nodes present, this call will fail – a real program would have to check the length first, before trying to remove the penultimate child.

Consider all this in terms of Inkscape: when you move things up and down in the z-order within Inkscape, or move entire layers up and down, what you're actually doing is removing nodes from the document and reinserting them at a different position. If you've got multiple items selected, or a group or layer containing lots of other items, they all have to be removed and reinserted. If you ever find yourself wondering why Inkscape is taking a long time to paste something, now you've got an idea of how involved this process actually is!

With the JS you've learnt so far, you have enough knowledge to write some code that will move objects around the canvas, as well as up and down in the z-order. You can use document.querySelector() to get a JS handle to an element in your drawing, and setAttribute() to dynamically change its parameters. Next time, we'll look at how you might use some of these features to animate your SVG image.

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

Written by Mark Crutch

Inkscape - Part 90

Over the past few months we've looked at ways to dynamically modify your SVG content when it's running in a web browser, using JavaScript. By making these changes over a period of time, we can effectively use JS to animate our SVG files.

We have looked at animation in this series before: in part 75 we looked at some simple CSS animations, then in parts 76 and 77 we moved on to SMIL animation. At the time I commented that SMIL was something of a dead-end technology, largely due to Microsoft never implementing it in any browser. But times have changed, and Microsoft have effectively given up on developing their own browser engine: shortly Microsoft Edge will begin using the same engine as Chrome, so should gain SMIL support as a side-effect. Whether or not this is enough to turn the tide for SMIL remains to be seen. Personally I think it's a great technology for animating SVG particularly, but the browser vendors seem to have settled on CSS animations as the way to go – even though that technology still doesn't cover all the use-cases that SMIL can handle.

So we're slightly stuck in limbo. SMIL offers huge power, but its time may be short. CSS animation is less powerful, but widely supported. However with JavaScript we can sort-of get the best of both worlds: as much power and flexibility as we need, in a way that has excellent cross-browser support.

Of course things aren't all rosy. When using SMIL or CSS animations you essentially take a pretty hands-off approach to things. Your input is a simple instruction: "Animate this object from A to B, over 5 seconds". You don't need to work out how the position of the object changes from one frame to the next, or worry about your animation imposing a heavy load on the machine. Instead you simply let the browser handle all the intermediate calculations – and the browser's code is a lot faster at handling those things than anything you could write for yourself in JavaScript.

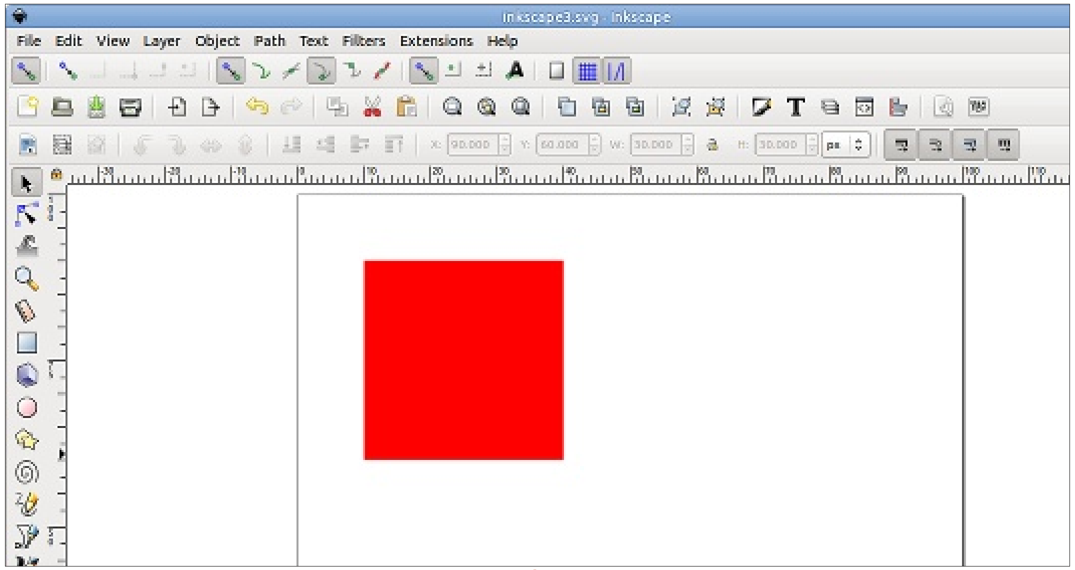
So JS animation gives you flexibility, but at the expense of performance. Most of the time that won't matter: the JavaScript engine in a modern browser is heavily optimised, so just moving an element or two around the screen isn't likely to impose much of a burden. But if you start to animate a large number of objects, especially on a mobile device, you might find that your animations aren't as smooth as they could be with the other technologies.

Enough of the pros and cons, on with the code! Once again we'll do all this in the browser's developer tools, so you'll need a super simple SVG file to start with the code shown below.

```
<svg
     xmlns="http://www.w3.org/2000/svg"
     viewBox="0 0 100 100">
     <rect id="s1" x="10" y="10" width="30" height="30"
fill="red" />
</svg>
```

If you were to load that file into Inkscape, it should look like the image below. The page boundary is a square of 100×100 units, as defined in the viewBox attribute. The square itself is positioned with its top left corner at 10 units down, and 10 units across from the origin (the top-left of the page in SVG). Remember these units are not pixels – the image will actually be scaled to fit the available space in the browser window. By using a 100×100 viewBox it can be convenient to think of the values as percentages, but in reality it's better to treat them as proportions or ratios, as that mental model works regardless of the viewBox size.

We're going to animate the "x" attribute from its starting value of 10 up to a value of 90. Because the square is 30 units wide, this will


inkscape3.svg - Inkscape
File Edit View Layer Object Path Text Filters Extensions Help

leave it hanging off the right-hand side of the screen when the animation finishes. I've done this to demonstrate a key difference between animating the content of an SVG file, and animating a <div> or other box in an HTML page: in the latter case the page width will grow and a horizontal scroll bar will appear (unless you specifically prevent that behaviour). With an SVG file, anything outside the viewBox simply isn't rendered, making it easier to have animations that start or end 'off-screen'. Think of it a little like a theatre stage, with your props and characters moving to and from the wings.

Load that file into your browser then in the developer console (F12) we'll get a handle to it assigned to a variable for use later on.

```
var s1 =
document.querySelector("#s1")
;
```

As a reminder, here's how we can change the x coordinate using JS. Run it to confirm that the box shifts over a little.

```
s1.setAttribute("x", 20);
```

Essentially our animation will consist of running a line like this repeatedly. JavaScript is a pretty poor language for timing-sensitive tasks such as this, but it does offer a few rudimentary functions that will be good enough for our needs. We've already seen the most basic of these a few months ago, when we used setTimeout() to turn some traffic lights on and off in sequence. The following (shown above) will shift the square over, after a delay of 2000ms (2 seconds).

```
var endX = 90;
var currentX = 0;
var delta = 10;

function moveSquare() {
        if (currentX > endX) currentX = 0;
        currentX = currentX + delta;
        s1.setAttribute("x", currentX);
};

setTimeout(moveSquare, 2000);
```

Notice what we did there? Instead of just setting the attribute directly I've created a global variable ("currentX"), then added 10 to it before we use the computed result. I also have an "if" statement to set the value back to zero if it gets too big, ensuring the square doesn't keep moving to infinity. How does this help us to create an animation? Now we can call that function repeatedly to make the square move by 10 units each time. Copy and paste the following lines as a single block, to make sure they all run before the 2s initial delay is up:

```
setTimeout(moveSquare, 2000);
setTimeout(moveSquare, 2100);
setTimeout(moveSquare, 2200);
setTimeout(moveSquare, 2300);
setTimeout(moveSquare, 2400);
setTimeout(moveSquare, 2500);
```

Well I don't think Pixar has got anything to worry about, but it's definitely animated. Creating a long list of setTimeout() calls isn't great though. Fortunately JavaScript has a related function, setInterval(), which does the same thing as an infinite list of setTimeout() calls. It returns a unique number that can be used with the clearInterval() method to stop the process when you're bored with looking at a jerkily moving square.

```
var i =
setInterval(moveSquare, 100);
```

```
// Some time later…
clearInterval(i);
```

Now that we don't have to type a setTimeout() function for every frame of our animation, we can make things move a little more smoothly by reducing the delta, and reducing the time between function calls accordingly:

```
var delta = 1;
var i =
setInterval(moveSquare, 10);

// Some time later…
clearInterval(i);
```

Still a bit fast for you? Increase the delay in the setInterval() call. Not fast enough? You can reduce the delay further, but browsers clamp setTimeout() and setInterval() to a lower limit, so it probably won't have much effect. Instead you can increase the delta value so the square moves two or three units at a time. Or 4.25 if you want – neither JS nor SVG require everything to be integers.

For a long time setTimeout() and setInterval() were the only practical way to run some JavaScript at the sort of regular intervals needed for animation. Over the past few years, however, browsers have gained more support for technologies needed to run games – 2D bitmap canvases, 3D graphics and a little thing called requestAnimationFrame(). In case the name didn't give it away, this function is intended to make animation a little easier. Specifically it does so by calling a function just before the browser is about to display the next frame on screen (typically about 60 times per second). It's like a setTimeout() where the delay is automatically set to to an optimal value by the browser.

In human terms 1/60 of a second is a pretty short delay, so to test this in the console you will probably want to increate the "delta" variable again, so that it's more obvious that your square moves.

```
var delta = 10;

requestAnimationFrame(moveSqu
are);
```

Now you might be expecting me to introduce an equivalent animation function to setInterval(), but I'm afraid I'll have to disappoint you. There's no function that will repeatedly run a function just before each frame is redrawn. But

```
Var endX = 90;
var currentX = 10;
function animSquare() {
        if (currentX <= endX) {
                currentX = currentX + 1;
                s1.setAttribute("x", currentX);
                requestAnimationFrame(animSquare);
        }
};

// Start the animation running
requestAnimationFrame(animSquare);
```

what we can do is to call requestAnimationFrame() again from inside the animation function.

We'll create a different animation function (shown above) for this next example. It's similar to the previous one, except that it always moves the square by 1 unit, and stops when it reaches the right-hand side, rather than looping back round. This latter change is mainly so that the code doesn't keep running indefinitely, otherwise it gets a bit tricky to proceed with the rest of this tutorial. Notice that we call requestAnimationFrame() to start the function running, but we also call this from within the "if" block, firing off another trip to our animation function just before the next frame is rendered.

Now we've got an animation running, but we don't really have any control over it. The duration of the animation will depend on how frequently requestAnimationFrame() fires in your browser, and all we can do is change "+ 1" to a different value to make the square move more or less on each iteration. But really that's all we need to be able to do. So long as we know the exact time that the function is called, we can calculate how far into the animation we are, and therefore what position the square should be at. To achieve this, the function that is called by getAnimationFrame() receives a single parameter: a high-resolution timestamp.

Let's work on a practical example: suppose we want our animation to take 10 seconds. At 60 frames per second that's about 600 movements our square will make, each of them a fraction of a unit. We could divide the total distance by 600 to calculate the exact amount of movement, but that will fail if we run the code on something that refreshes at 30 or 120 frames per second, or if some frames get dropped due to the load on the machine. A better idea is to track how long has passed since the start of the animation, and use that to calculate where the object should be. We can then set the coordinates to the new value, before we fire off another requestAnimationFrame() for the next step in the animation.

The parameter that gets passed to our function is a value in milliseconds since the document was loaded. We don't really care about that specific point in time – we need to know how long the animation itself has been running. What we need to do, therefore, is to record the timestamp the first time our function is called. On subsequent calls we can subtract that value from the latest timestamp to work out how far along the animation timeline we have progressed.

Let's start by initialising a few variables. We'll create a variable to hold our starting position, setting it to 10. Next we have a "duration" (in milliseconds) to hold the time we want our animation to run for, and "endX" for the X coordinate we want to end up with. Putting these into variables makes it easier to modify the animation to run at a different speed or cover a different distance. Finally we'll include a "startTime" variable, with an "undefined" value initially, into which we'll store a copy of the timestamp we receive the first time our animation code is called.

```
var startX = 10
var endX = 90;
var currentX = 0;
var duration = 10;
var startTime = undefined;
```

Now for our reworked animation function. The main animation code is fairly similar to the previous incarnation, except that we work with the timestamp that is passed to the function in order to calculate the new X value. The first time the function is called we don't need to update the X coordinate – by definition we're at the start of the animation – so we just store the current timestamp, then queue up another call before the next frame, then exit. On subsequent calls, the startTime is no longer "undefined" so this part of the function is skipped entirely.

By storing the initial timestamp outside the function, we can calculate how long the animation has been running. Since the two timestamps are in milliseconds, we'll divide the result by 1000 to convert it to seconds. Since we know the total time for the animation, a quick division will give the proportion of the animation that has passed: we can then multiply that value by the total distance to travel in order to calculate the new X position for that moment in time. Code is shown below.

Running this should produce a smooth animation that takes 10s to complete. Re-run the previous block of "var" lines to reset everything, then the final requestAnimationFrame() call to kick it off again. Try changing the values in the variables to alter the distance the square moves, or the time it takes to perform the animation. In every case you should find that the animation is, if not smooth, at least a lot smoother than you saw with

```
function animSquare(ts) {
        if (startTime === undefined) {
                startTime = ts;
                 requestAnimationFrame(animSquare);
                return;
        }


        var runningTime = (ts - startTime) / 1000;
        var proportion = runningTime / duration;


        if (currentX <= endX) {
                currentX = startX + (endX * proportion);
                s1.setAttribute("x", currentX);
                requestAnimationFrame(animSquare);
        }
};


// Start the animation running
requestAnimationFrame(animSquare);
```

contents ^

setTimeout() and setInterval().

Creating all these variables outside a function (so-called "global" variables) is generally seen as bad form in the programming world. It also makes it tricky to animate more than one thing, as they'll all potentially be sharing the same global variables. A better approach is to encapsulate all the variables in a single JavaScript object, then attach that to the SVG element you're trying to manipulate. Here's the code above rewritten to work in this way (shown right).

```
s1.animProperties = {
        attribute: "x",
        start: 10,
        end: 90,
        current: 0,
        duration: 10,
        startTime: undefined
}

function animSquare(ts) {
        props = s1.animProperties;

        if (props.startTime === undefined) {
                props.startTime = ts;
                 requestAnimationFrame(animSquare);
                return;
        }

        var runningTime = (ts - props.startTime) / 1000;
        var proportion = runningTime / props.duration;

        if (props.current <= props.end) {
                props.current = props.start + (props.end * proportion);
                s1.setAttribute(props.attribute, props.current);
                requestAnimationFrame(animSquare);
        }
};

// Start the animation running
requestAnimationFrame(animSquare);
```

Notice that I've removed the "X" from the end of the parameter names, and created a new "attribute" entry with a value of "x". This starts to make the code more generic: you could change the "x" to a "y" in order to animate movement in the vertical direction, or "r" to animate the radius of a circle.

For a truly generic solution you could turn this JS object into an array of objects. The animation code would loop over each entry in the array, allowing you to animate more than one attribute at a time – essential if you want your images to move at an angle, for example. I'll leave it as a challenge for the reader to implement this.

At the moment our animation is also strictly linear: the attributes are changed at a constant rate over time. More "natural" animation can be achieved with rates that vary – accelerating and decelerating over the course of the movement. The maths to produce such effects, referred to as an "easing function", is well outside the scope of this series. That alone is a good reason why CSS animation, SMIL, and third-party animation libraries are usually a better option than rolling your own JS animations from scratch. But for simple animations, or just your own education, it's good to see how the same code you might use to dynamically modify your SVG in discrete steps, can also be put to work to achieve continuous effects.

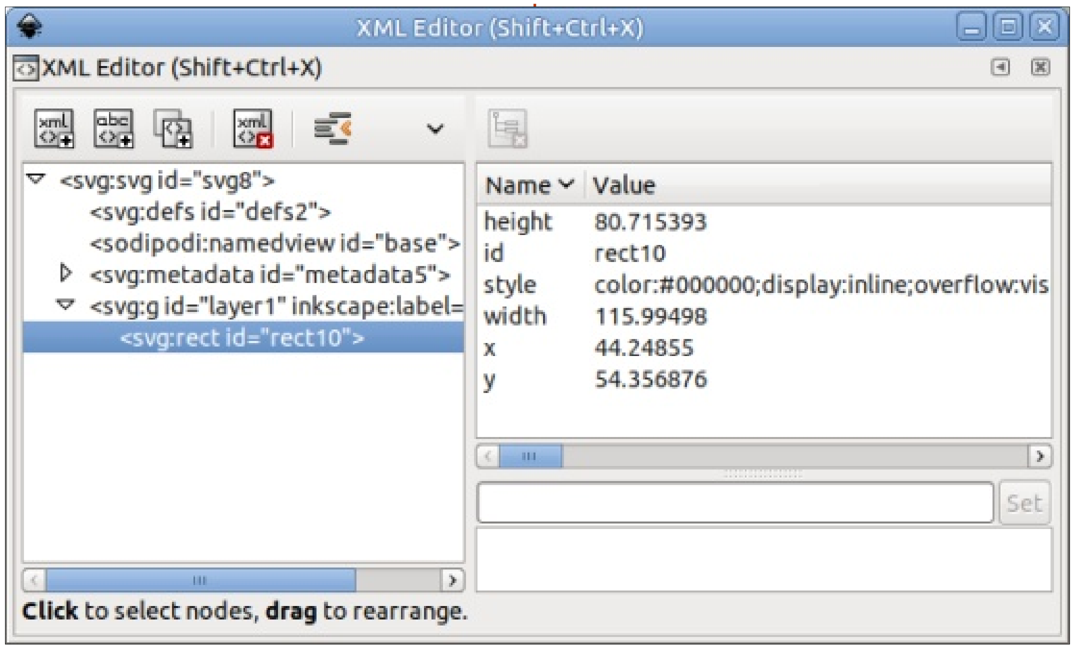
Mark uses Inkscape to create three webcomics, 'The Greys', 'Monsters, Inked' and 'Elvie', which can all be found at http://www.peppertop.com/

Written by Mark Crutch

Inkscape - Part 91

Over the past few months, we've looked at ways to manipulate SVG in a web browser, using JavaScript, culminating in some simple animation. The code we used last time let us move an object horizontally on screen by manipulating its "x" attribute. By extension, we could also do the same with the "y" attribute to move it vertically. This time we'll look at an alternative method of moving objects up, down, left and right, using an attribute that also opens up the ability to rotate or skew the element at the same time. The attribute in question is "transform", and it crops up all the time in Inkscape documents, so let's start by looking at how it's used there.

Open Inkscape, with a fresh, blank document. Draw a simple square or rectangle – it doesn't matter about the fill and stroke at this point, though it will be useful if you can actually see and interact with it. Now open the XML Editor dialog (Edit > XML Editor...).


XML Editor (Shift+Ctrl+X)
XML Editor (Shift+Ctrl+X)
Set
Click to select nodes, drag to rearrange.

```
▽ <svg:svg id="svg8">
    <svg:defs id="defs2">
    <sodipodi:namedview id="base">
  ▷ <svg:metadata id="metadata5">
  ▽ <svg:g id="layer1" inkscape:label=
      <svg:rect id="rect10">
```

Name	Value
height	80.715393
id	rect10
style	color:#000000;display:inline;overflow:vis
width	115.99498
x	44.24855
y	54.356876

By now, the right-hand side of this dialog should make some sense to you. You can see the usual collection of attributes that you would expect to find on a <rect>, together with their values – including the familiar "x" and "y" attributes.

Leaving the XML editor open, you should also open the Inkscape Preferences (Edit > Preferences...). Navigate to the Behaviour > Transforms screen, and ensure that the "Store transformation" option is set to "Optimised".

You should now have these two dialogs open, together with your main Inkscape window. Arrange things so that you can easily get to the main canvas whilst seeing what happens in the XML editor. Select the rectangle in your document, then use the cursor keys to move it around a little. Notice that your "x" and "y" attributes in the XML editor change as you do so.

Now return to the Inkscape Preferences dialog and switch the "Store transformation" option to "Preserved". Repeat the exercise of moving the rectangle around using the cursor keys, once again keeping an eye on the XML editor.

You should notice that, this time, the "x" and "y" attributes remain unchanged. Instead a new "transform" attribute is added, with a value that takes the form "translate(x,y)". If you move your rectangle only horizontally, your "translate" will have only a single parameter – if it's omitted entirely, the "y" parameter is considered to be zero.

That's all very interesting, but what have we actually achieved? One important point is that we've now switched from using absolute coordinates to relative ones. Instead of setting absolute coordinates ("x" and "y" attributes), we're combining an absolute starting point (the "x" and "y" attributes) together with a relative offset (the "x" and "y" parameters in the translate() function). Think back to our animation from last time: we had to keep track of the current "x" value and add our offset to it each time. By manipulating a "transform" attribute instead, we just have to set the offset directly, simplifying our code. It no longer matters what the coordinates were previously, we need to set only the translate() to the right values for the amount of time that has elapsed in our animation. It also leaves the original coordinates untouched, so moving the object back to its starting position is easier. This also makes our code more reusable: we can apply the same animation to various objects, each with their own "x" and "y" attributes, and therefore all separately positioned, despite sharing the same relative movement.

Another key use for the transform attribute is in combination with groups. The SVG <g> element doesn't have its own "x" and "y" attributes, so moving a group of objects (as one) would require code to update the attributes of each and every element in the group on each step of the animation. By setting a transform attribute on the group, you can obtain the same effect with far less work.

So the transform attribute is a useful, if not essential, way of moving objects around in your drawing. But it offers more than that. Consider how you might incorporate rotation into your animation. If you're animating a path – and you're sufficiently mathematically astute – you could recalculate the coordinates of each node and handle in the path. But us mere mortals need an easier way to manage such tasks and the transform attribute offers that capability.

To see how it works, revert your rectangle back to a point where there's no transform attribute showing in the XML editor, and change the "Store transformation" setting back to "Optimised". With the selection tool active, click the rectangle a second time to switch to the rotate and skew handles. Use the corner arrows to rotate the rectangle and you should see a transform attribute appear, but this time with a value of "rotate(r)", where "r" is the amount of rotation in degrees. By holding the Ctrl key, you can make the value jump between the steps defined in Inkscape's preferences, or release the key for free rotation of your shape.

Notice that rotating the object not only adds a transform() with the rotation amount, but also changes the "x" and "y" values. Once again, set "Store transformation" to "Preserved". Now the transform() function has three parameters: the rotation angle and the x and y coordinates of the center of rotation.

There are four other functions that can be used in the transform attribute:

```
scale(x, y)
skewX(a)
skewY(a)
matrix(a, b, c, d, e, f)
```

I've called these out separately because of the way Inkscape treats them. The scale() function simply increases or decreases the size of the object, stretching it if the x and y values are not the same. As with translate() the y parameter is optional.

skewX() and skewY() transform your element in the same way as the skew handles in the Inkscape UI. They each take a value, in degrees, that specifies the angle of skew.

These three functions map fairly obviously to the select tool in Inkscape, which can be used to stretch, scale and skew an object. So you might expect to see these functions appear in the XML editor as you manipulate your object. Instead you'll get the matrix() function appearing.

Without going into too much detail, the matrix() function lets you supply a series of six numbers that are used to fill the first two rows in a 3×3 matrix. This is used via standard mathematical matrix multiplication to map the original coordinates of the object to the transformed coordinates. In non-mathematical terms, a single matrix can not only produce the same output as all the other transformation functions, but can also produce output that is equivalent to any combination or mixture of them. Want to skew, rotate, scale and translate all at once? A matrix() transform will do the trick.

Working out the six numbers that need to be passed to the matrix() function is not for the faint-hearted. They don't correspond to simple values such as x, y and rotation. So although Inkscape likes to use the matrix()

contents ^

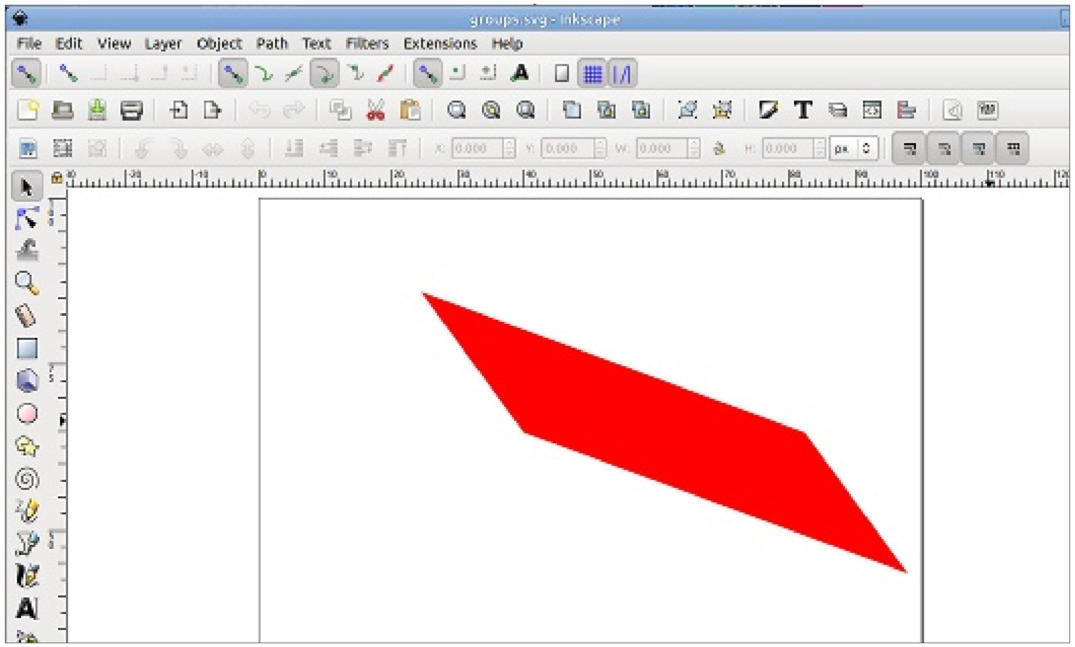
function internally, it's probably not something you want to be manipulating in JavaScript. Fortunately there are a couple of ways to work with the individual functions, rather than being forced to combine everything into a single matrix().

The first is simply to wrap your objects in SVG groups (the <g> element), and apply a separate transformation to each one. Here's how an SVG file might look (top right) if we took this approach to both skew and rotate a square:

```
<svg
  xmlns="http://www.w3.org/2000/svg"
  viewBox="0 0 100 100">

  <g transform="skewX(60)">
    <g transform="rotate(45)">
      <rect
        x="10" y="10"
        width="30" height="30"
        fill="red"
      />
    </g>
  </g>
</svg>
```

When loaded into Inkscape the result looks like this:


groups.svg - Inkscape
File Edit View Layer Object Path Text Filters Extensions Help

Opening the XML editor shows that the transform on the outer group remains intact – it's still a skewX() and doesn't get automatically converted to a matrix() when loading it into Inkscape. As soon as you make a change via the GUI, however, the transform's value will be replaced with a matrix(). If you just want to change the value in the existing function (eg. changing the angle of skew in this case), then you can make the modification in the XML editor. But remember that a <g> doesn't have its own x and y attributes, so even something as trivial as moving the object slightly will mean Inkscape converts the attribute to a matrix() that combines the skew with the translate.

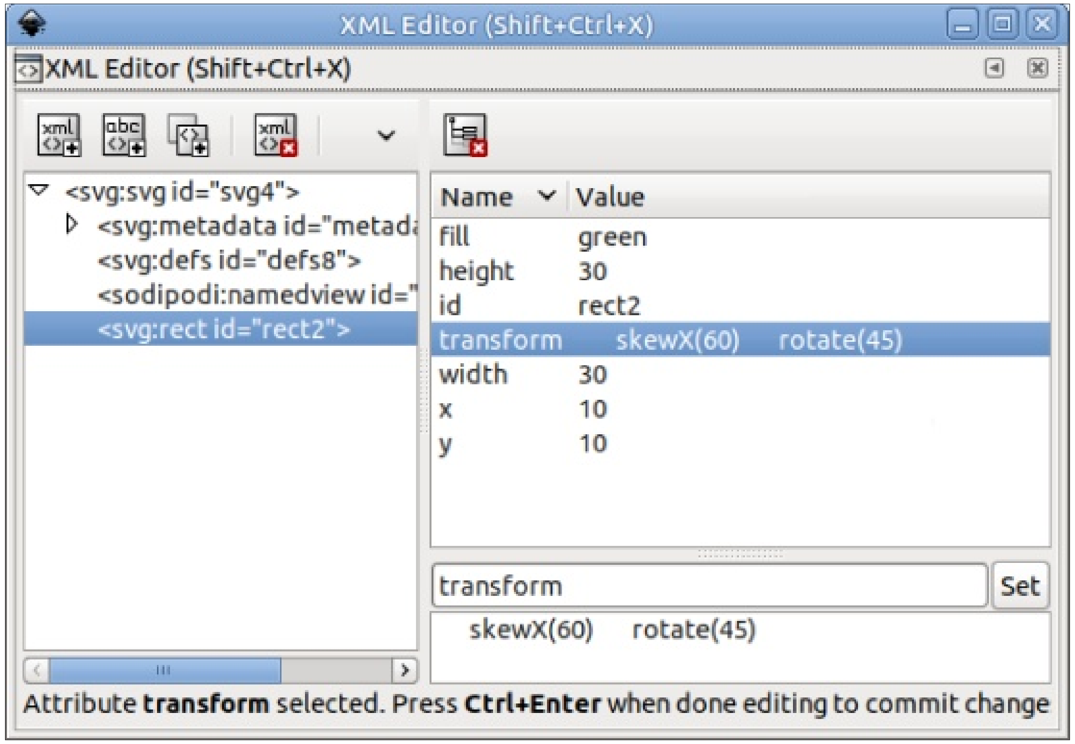
This is an important thing to be aware of. It's easy to set up a file for animation with some nicely hand-coded transform attributes, then absent-mindedly open it in Inkscape to make a minor change, only to find that your hand-coded values have all been replaced with matrix() functions instead.

There is a second way to apply multiple transformations to an object: you just list them all in a single transform() element. Here's a version of the previous file, but this time there's no need for the <g> elements, since the transformation can be applied directly to the square (below).

```
<svg
  xmlns="http://www.w3.org/2000/svg"
  viewBox="0 0 100 100">


  <rect
    transform="
      skewX(60)
      rotate(45)
    "
    x="10" y="10"
    width="30" height="30"
    fill="green"
  />
</svg>
```

Note that the transform attribute is now a list of transformations to apply. The white-space isn't important: I've listed the functions one-per-line for clarity, but you could just put them onto a single line with a space character between them. When viewed in Inkscape, they appear in the XML editor on a single line, with every space and tab included between them, but none of the carriage returns:


XML Editor (Shift+Ctrl+X)

Name	Value
fill	green
height	30
id	rect2
transform	skewX(60) rotate(45)
width	30
x	10
y	10

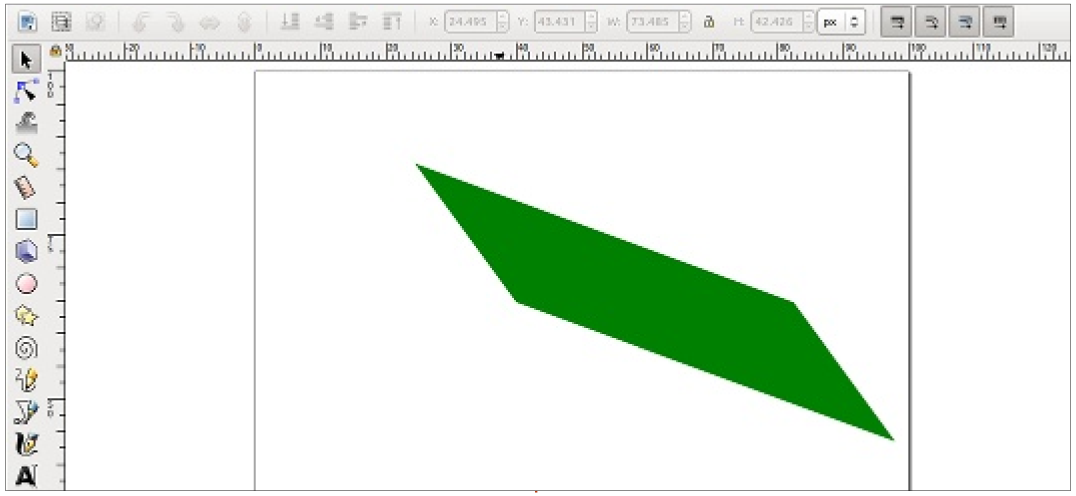
```
<svg:svg id="svg4">
  <svg:metadata id="metad
  <svg:defs id="defs8">
  <sodipodi:namedview id="
  <svg:rect id="rect2">
```

transform

skewX(60)   rotate(45)

Attribute **transform** selected. Press **Ctrl+Enter** when done editing to commit change

Once again, there's no real surprise with the appearance of the file when it's loaded into Inkscape (see above).



Yet again, with this approach we face the same issue of Inkscape's desire to convert the value into a single matrix(). This time we do have x and y attributes (since we're working on the <rect> itself), so you might think that a "Store transformation" setting of "Optimised" might leave the transform untouched and just update the coordinates when you move the object around. Unfortunately, even in this case you'll find that a matrix() comes along and tramples over everything, in addition to the x and y attributes changing.

It's annoying that Inkscape doesn't offer a third option beyond "Optimised" and "Preserved". A "Verbose" option that stores transforms in a more human-friendly form. Instead of a composite matrix() function, you would get a list of separate functions in the attribute. Moving would add or update the translate(), rotating would add or update the rotate(), and so on. For anyone planning to manipulate their SVG file with code, the advantages of this approach would be huge.

As it stands, for most Inkscape users the internal details of how objects are moved, rotated, scaled and skewed is irrelevant. "Preserved" or "Optimised" has no bearing on how you work with elements in the GUI, or how the image is rendered in a web browser. If you do wish to alter the transform attribute using JavaScript, then there's a slight advantage to "Preserved" – but only if the x and y attributes are set correctly in the first

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 92

Last time, we looked at the "transform" attribute, particularly with regard to its frequent appearances in Inkscape files. In this instalment, and the next, we're going to combine this information with some of the JavaScript we've covered in previous months to show how you might animate the moving, turning, skewing and scaling of an SVG element.

Once again we'll start off with a very simple SVG file (top right).

As usual, we've got a viewbox of 100×100 units, but, this time, I've added a background <rect> to make it clearer where the extents of the canvas are when the file is loaded into a web browser. The element we're going to animate is the red rectangle, which is inside a group (<g>). The group has a transform attribute on it, containing a series of functions that will be applied to its content. In this case, they each have no effect – a rotation of 0 doesn't turn the shape at all, whilst a scale factor of 1 leaves it at its original size. Strictly speaking, they're not necessary at all at this point, but they are present to show you the format of the string we'll be creating with our animation code.

The rectangle itself bears a little explanation. In an SVG document, the y-axis runs down the page, and the x-axis runs from left to right. So the origin – the 0,0 point in the image – is at the top-left of the page. The position of a <rect> is also based on its top-left corner. Therefore, setting the "x" and "y" attributes to 0 would position the rectangle at the top left of the screen. But, for this code, I want to pretend that the "x" and "y" coordinates correspond to the center of the rectangle. This means offsetting its position by half the width to the left, and half the height upwards. Since it's 20 units on each side, the transform attribute with a translate(-10, -10) does the job nicely. Basically, we've moved the rectangle's reference point from the top-left corner to the center. With that manipulation in place, using 50 for the "x" and "y" coordinates puts it right in the

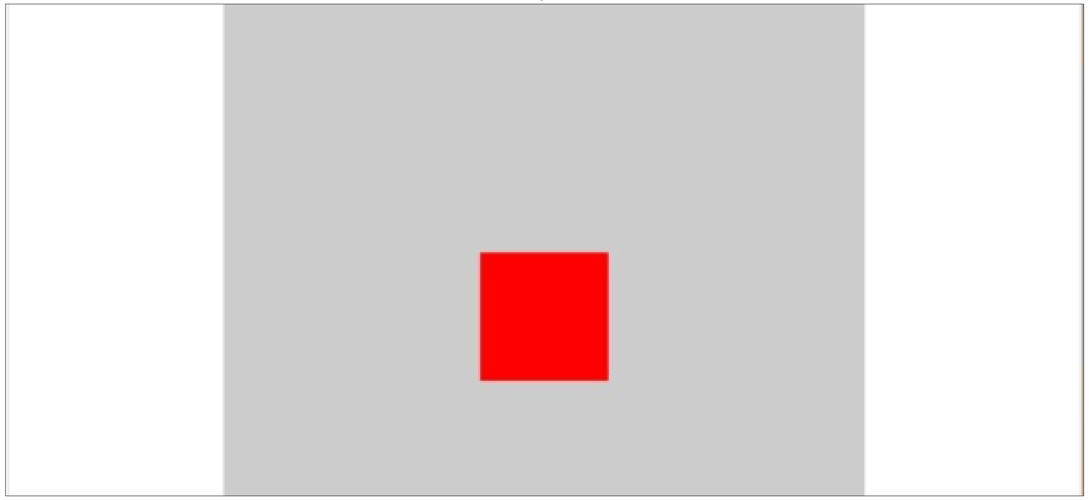
```
<svg
  xmlns="http://www.w3.org/2000/svg"
  viewBox="0 0 100 100">

  <script href="transform.js" />

  <rect x="0" y="0"
    width="100" height="100"
    fill="#ccc" />

  <g id="g1"
    transform="
      translate(0,0)
      scale(1)
      rotate(0, 50, 50)
      skewX(0)
      skewY(0)">

    <rect
     transform="translate(-10,-10)"
      x="50" y="50"
      width="20" height="20"
      fill="red" />
  </g>
</svg>
```



middle of the image.

There’s one other line in the file – a <script> tag that references an external document. For the sake of simplicity, we’re going to keep our code in a separate JavaScript file, avoiding problems with character escaping, and meaning that we don’t need to make any further changes to the SVG file. The only requirement is that the JS file is kept in the same directory as the SVG file, since I’ve used only a filename, not a relative path or absolute URL. The JS file itself, “transform.js”, contains just a single line at this point:

```
alert("Hello World");
```

By loading the page in a web browser we can now easily confirm that we have a red square in the middle of a grey square, and that a message pops up from our JavaScript file to indicate that it’s being loaded correctly. Now we can move on to some real code.

Based on the approach from part 90 of this series, we’re going to create a single function that updates the transform element for each frame that the browser renders. The function will receive a timestamp, and use that to determine how long the animation has been running, and therefore what values should be put into the transform element’s functions for that particular point in time.

To begin with, we’ll just animate one of the properties: rotation. Replace the alert() call in the JavaScript file with this (right).

```
let group;
requestAnimationFrame(initialise);

function initialise(ts) {
  group = document.querySelector("#g1");

  group.animProperties = {
    startTime: ts,
    rotationDuration: 5   // Time to turn 360 degrees
  }


  // Start the animation running
  requestAnimationFrame(animate);
}


function animate(ts) {
  const props = group.animProperties;


  const runningTime = (ts - props.startTime) / 1000;
  const anglePerSecond = 360 / props.rotationDuration;
  const totalAngle = anglePerSecond * runningTime;
  const angle = totalAngle % 360;


  group.setAttribute("transform", `
    rotate(${angle}, 50, 50)
  `);


  requestAnimationFrame(animate);
}
```

Much of this looks similar to code we’ve seen previously, but there are enough differences to warrant a step-by-step walkthrough.

We start by declaring a global variable called “group”. Previously, we’ve used the “var” keyword to do this, but modern JS has mostly replaced that with “let” (for variables that will change), and “const” (for those that won’t). We’ve used “var” when working in the console as it won’t throw an error if you try to run the same line twice – as “const” would do. But, as we’re creating a separate JS file here, we’ll stick to convention. In this case, the “group” variable will eventually hold a reference to the <g> element, but as that’s not necessarily available as soon as the page loads, we’ll declare it using “let” and update the value later.

Later doesn’t take long to arrive. The very next line of code causes our initialise() function to run just before the next frame is drawn, giving the browser time to render the content so that our <g> element actually exists in the document structure before we use it. The initialise() function itself does these things:
• Set our “group” global variable as a reference to the <g> element.
• Set up some JS properties on the element. First is the initial timestamp, which will be used as the basis for calculating the

timings in the animate() function.
• The only other property we're creating for now is the time it should take for the square to do one rotation, in seconds.
• Finally, we have another call to getAnimationFrame() which will start the actual animation running.

Now we get to the animation code itself, in the form of the animate() function. We begin by getting a reference to the animation properties we set up previously, and storing it with a more convenient name. We can use "const" instead of "let" here as the value we assign doesn't get changed within this function.

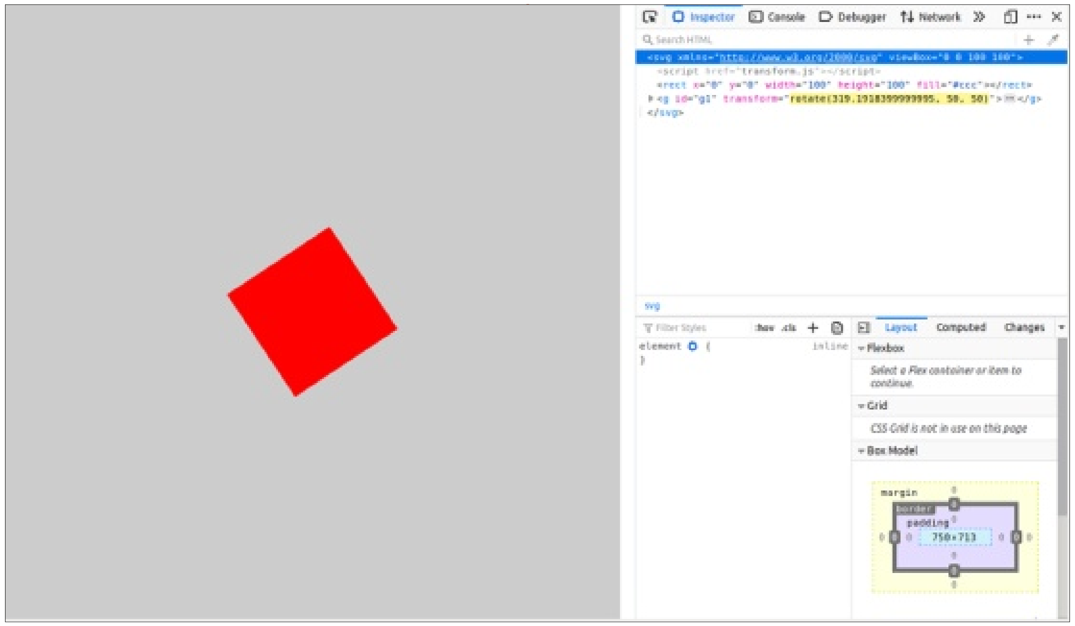
The next group of lines just calculates the value, in degrees, that we need to rotate the square by. We get the amount of time the animation has been running for, by subtracting the initial timestamp from the current one, then divide the value by 1000 to convert from milliseconds to seconds. By dividing 360 by the desired rotation time we find the amount of rotation we need to perform every second; multiplying that value by the amount of time we've been running for gives a total value for the number of degrees to rotate by.

After the first rotation has completed, the calculated value will be larger than 360. That's not actually a problem – the browser will happily do the right thing for you in this case – but I prefer to be a little explicit about what's happening. That explains the last line of this block, where we use the modulus operator (%) to get the value that remains after dividing the total angle by 360. This has the effect of normalising the rotation angle so it never goes above 360, which can make it easier to see what's happening if you need to log the value out, or if you view it live in the developer tools.

The penultimate line uses setAttribute() to update our transform attribute with a new value. The value itself is a template string, delimited by backticks (`…`). They're not always as easy to spot in code as the more usual quotes and double-quotes, so make sure you don't overlook them. Within a template string any content inside a ${} block will be evaluated as JavaScript, and the result will be placed into the string. For our purposes, this means we can use ${angle} to inject the value of the "angle" variable into the string, without having to perform a lot of concatenation. We're using the three-value form of rotate() in order to provide coordinates for the center of rotation (50, 50) – without it the square will rotate about the top-left corner of the page.

The last line simply queues up another call to the animate() function, as we've seen previously. Load the SVG file into a web browser and, if everything is correct, you should see the square spinning around in the middle of the page. Press F12 in the browser to open the developer tools, and select the tab labelled "Inspector" (Firefox) or "Elements" (Chrome/Chromium) – it's usually the leftmost tab on the bar. You should see the structure of your SVG file, with the <g> element visible, and the value of the transform attribute updating as the square rotates. Notice how the first number never goes above 360; try replacing the ${angle} tag with ${totalAngle} and see what difference it makes to the rotation and to the attribute's value.

Rotation is pretty straightforward because we have to deal with only an ever increasing number. If we exceed a full rotation then we either normalise the number, or let the browser do



```
<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 100 100">
  <script href="transform.js"></script>
  <rect x="0" y="0" width="100" height="100" fill="#ccc"></rect>
▸ <g id="g1" transform="rotate(319.1918399999995, 50, 50)">…</g>
</svg>
```

```
element { 
}
```

contents ^

it for us. The other transform functions are a little more tricky: skewX and skewY expect a value between -90 and +90 (though the extreme ends of the range distort the object so much that they're not very useful); translate can take any number, but there's only a limited range that makes sense within the confines of our 100×100 viewbox; scale has a similar practical limit. For all these transform functions, therefore, we want to animate back and forth between two values. This means creating three properties for each thing we want to animate, for the lower limit, upper limit and duration. Here's how the group.animProperties object might be extended to also include skewX, for example (note the addition of a comma after the rotationDuration property, as this is no longer the last item in the object). Shown top right.

To go with the new property, we'll also need an extra group of lines in the animation function, just after the corresponding lines for rotation, but before the call to setAttribute() (bottom left).

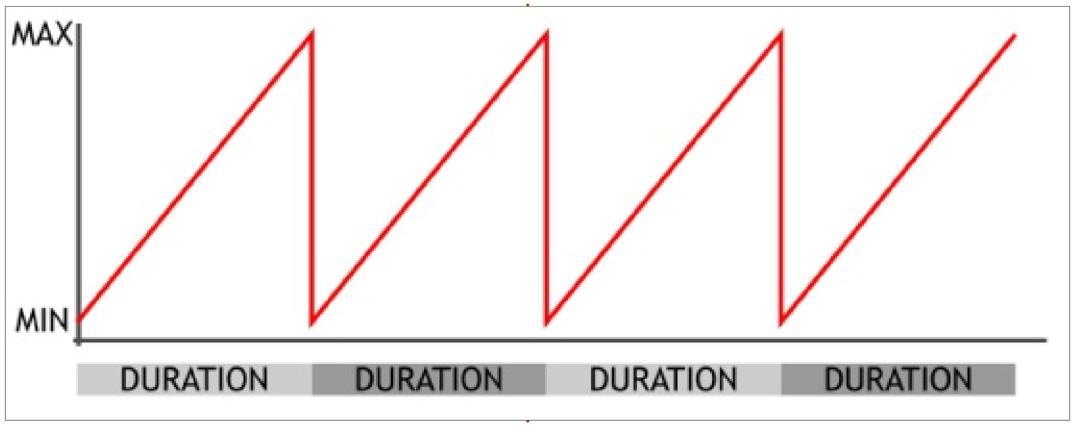
```
group.animProperties = {
  startTime: ts,
  rotationDuration: 5,    // Time to turn 360 degrees

  skewXDuration: 10,
  skewXMin: -60,
  skewXMax: 60
}
```

We start by assigning props.skewDuration to a local variable, for no other reason than it gets used a lot, so we've given it a more convenient name. The second line subtracts the minimum value property from the maximum, to give us the total amount of possible skew. We'll use this to work out what the current skew amount should be at any given timestamp.

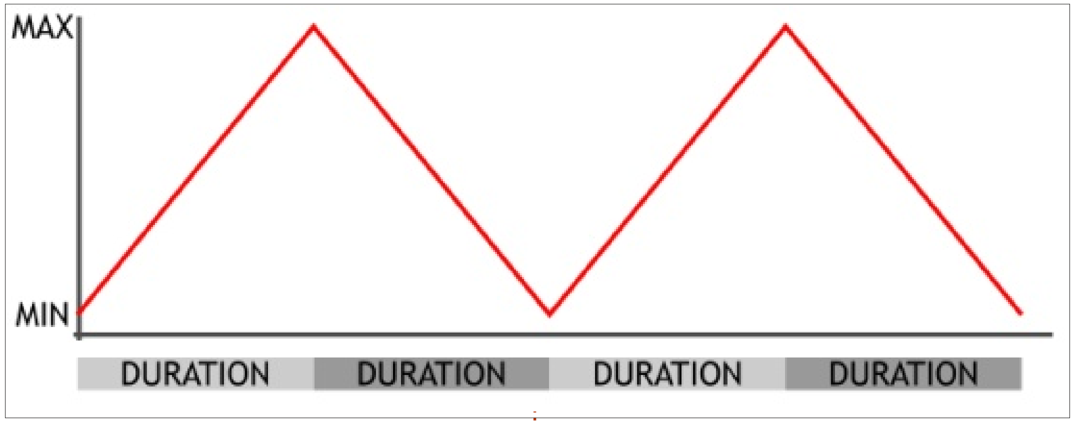
The third line calculates the "position" along the animation for the current timestamp. We do this by taking the running time (calculated earlier, in the previous block), dividing it by the duration for this animation, then taking the remainder. This gives us a value that runs from zero to the duration value, then jumps back to zero before ramping up again on each iteration. Rather than running from zero to "duration" it's more useful if we adjust this value to be a decimal from 0 to 1, which is achieved by dividing by the total duration.

If we were to comment out the next few lines and jump to the last one, we would find that the animation cycles repeatedly from the minimum value to the maximum, jumping straight back to the minimum on each iteration. Plotting the values over time results in a "sawtooth" chart.



For our animation, however, we want the value to transition linearly both up and down, without the sudden jump between iterations. What we want is a triangle wave:

```
const skewXDur = props.skewXDuration;
const skewXRange = props.skewXMax - props.skewXMin;
let skewXPosition = (runningTime % skewXDur) / skewXDur;
const skewXDirection = runningTime % ( skewXDur * 2);
if (skewXDirection > skewXDur) {
  skewXPosition = 1 - skewXPosition;
}
const skewXAmount = (skewXRange * skewXPosition) + props.skewXMin;
```



As you can see, on odd numbered iterations we want the animation to proceed as usual, but on the even numbered ones we want the position value to step downwards rather than upwards. In the code above this is done by creating a “skewXDirection” variable which holds the modulus of the current running time when divided by twice the duration. This value will ramp up from zero at the start of an odd iteration, through the duration value at the end of the odd iteration, continuing up to twice the duration value at the end of the subsequent even iteration. We’ve created another sawtooth wave, but this time running from zero to duration×2 over the course of two iterations.

The “if” statement that follows checks to see if this direction value is greater than the duration: if it is then we must be on an even cycle. In that case the “skewXPosition” variable (which, if you recall, ranges from 0 to 1) is subtracted from 1, so as the animation progresses the final position value first steps upwards, then steps downwards, before the cycle repeats in a triangle wave pattern.

The last step in calculating the actual value is to multiply the current position in the cycle by the total range of the animation, then add the minimum value to move the final result into the right range of numbers.

Phew! That was a lot to take in for a few lines of code. If you find it easier to follow, try adding some console.log() lines amongst the code so you can see how the values change in the developer tools.

```
group.setAttribute("transform", `
  rotate(${angle}, 50, 50)
  skewX(${skewXAmount})
`);
```

With our final value calculated, the last step is to update the transform attribute to hold both the rotate() and the skewX() functions. Extend the previous template string to this:

Loading the file into the browser, you should see the square being skewed as it rotates. But you’ll also notice that our simple, constrained rotation in the middle of the screen has turned into a whirling dervish that swoops out of the bounds of our image before flying back in and then setting off into the distance once more. Next time we’ll discuss why this is happening, and finish this little animation by adding scaling and translation. In the meantime why not test your own understanding of this code by adding the necessary lines to make the shape also skew in the Y direction, at a different frequency to the skewX() effect.

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

contents ^

Written by Mark Crutch

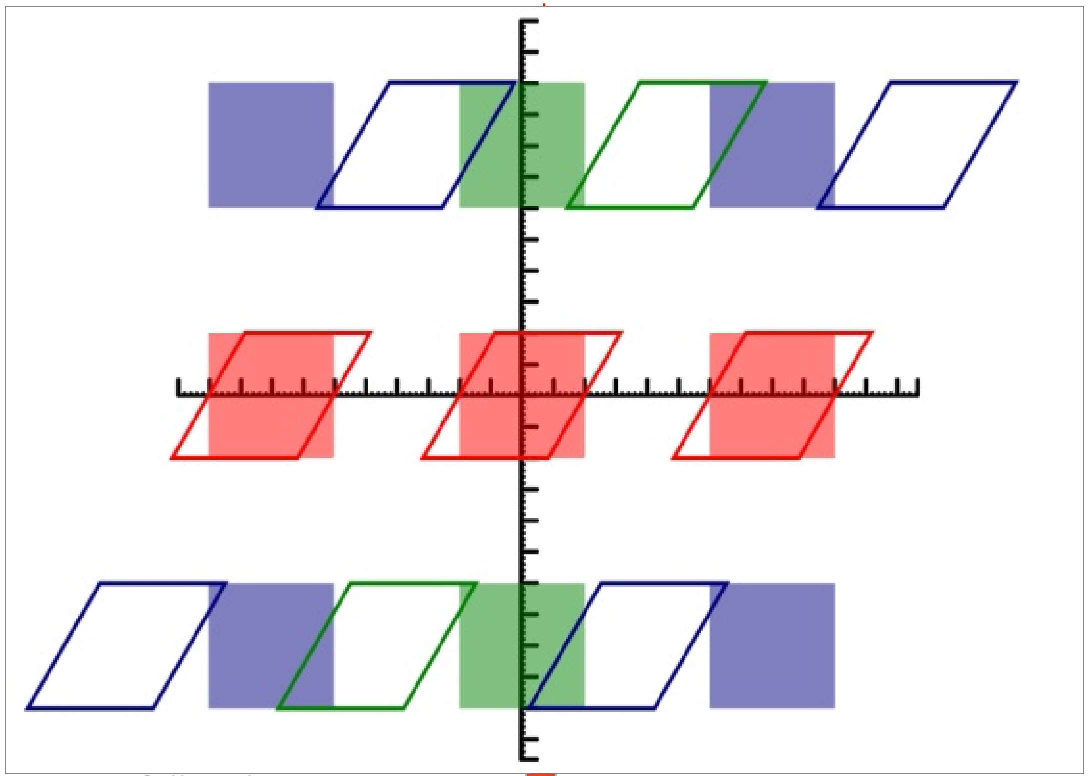
Inkscape - Part 93

Last time, we began work on an SVG file which uses JavaScript to animate the "transform" attribute of an object when loaded in a web browser. We'd got as far as animating the rotate() and skewX() functions, and you'd been left with the challenge of adding skewY() to the mix. Hopefully you worked out that this was mostly a case of copying the existing code, and replacing "X" with "Y". Specifically:

• Create three new properties in the animProperties object: skewYDuration, skewYMin, skewYMax.
• Insert another block of code to calculate the current value of the skewY() function, given the supplied timestamp. This is literally a copy of the skewX code with the letter replaced throughout, resulting in a skewYAmount variable at the end.
• Add another line to the setAttribute() call to include the skewY() function in the transform attribute, passing the value of the skewYAmount variable via the template string.

I also left you with something of a puzzle: with the addition of the skewX() function, the square doesn't just rotate and skew in the middle of the screen as you might expect; instead it moves wildly in and out of the browser window as it rotates. Adding the skewY() function simply exacerbates the problem. Why does it do that? And how can we get the behaviour we expected?

The reason is quite simple: whereas the rotate() function has an optional pair of parameters for setting the center of rotation, there's no equivalent for the skew functions. Skewing takes place relative to a baseline, rather than a single point, but there's no generic skew() function that lets you specify this baseline via two sets of coordinates. Instead, there are only two possible baselines available: the x-axis (via the skewX() function) and the y-axis (via the skewY() function). The graph below shows the effect of skewing along the x-axis for the same size of object positioned in a variety of places (bottom left)



The red squares along the x-axis all skew "in-place", resulting in the outline shapes displayed. The green and blue squares – colored to avoid confusion where they overlap – move to the left and right as a result of the skewing process. It doesn't take much thought to realise that the amount of movement to the left and right is actually proportional to the distance from the x-axis, so even a small angle of skew can quickly move a shape by a large distance if it's located far from the axis. And, of course, these rules also apply for the skewY() function, but rotated by 90°.

You may recall that we did some deliberate manipulation of our object in order to position it in the middle of the screen (50, 50). Unfortunately, all that work means that skewing the shape also

pushes it around. The problem is that we have two conflicting requirements:
• We want the object centered at (50, 50) for display.
• We need the object to be centered at (0¸ 0) in order to skew it.

There are a few ways to solve this conundrum:
• Use a translate() function to move the object to (0, 0); then skew it; then use another translate() function to move it back again.
• Change the x and y attributes of the object so that it starts at position (0, 0). After skewing, add a translate() function to move it to (50, 50).
• Use a matrix() function rather than the skewX()/skewY() functions, as this can combine skewing and translating into a single call.

I'm going to immediately dismiss the last option, as it requires far too much mathematics. But it does, perhaps, explain why Inkscape always uses matrix() rather than the named skew functions. The first option is probably the clearest in terms of what's happening, but it results in the position of the object being calculated three times: once with the initial x and y values set to 50; once when it's translated back to (0, 0) prior to the skew; then a final time when an inverse translation is applied to put the skewed version back into place.

The best compromise for this project is the second approach. Simply changing the x and y values in the SVG, however, means that the source file no longer holds the "true" values for the coordinates, so if the JavaScript fails to load for some reason the square will be positioned at the top left, rather than in the middle of the screen. A slight modification to this approach – and the one we'll take here – is to leave the SVG file untouched, but change the x and y values to zero at the start of the JS file. That way, if the script fails to load you still get a stationary fallback image with the square in the right place, but if everything loads correctly, the JS immediately changes the object's coordinates to make for less work in the rest of the code. The first step is therefore to modify the start of the initialise() function to get a handle to the <rect> inside the group, and reset the attributes (shown top right).

```
function initialise(ts) {
  group = document.querySelector("#g1");
  const rect = group.querySelector("rect");
  rect.setAttribute("x", 0);
  rect.setAttribute("y", 0);
```

We use group.querySelector() to get the rectangle as it limits the search to descendants of the group object (compared with document.querySelector() which would search the whole document), and it makes it easy to replace the argument with an ID or class selector when working with a more complex drawing, or with a different element selector if we change the content of the group to be a different type of object (e.g. a <path> or an <image> instead of the <rect>). Once we've got a handle to the element, it's back to our old friend setAttribute() to set the new values we want.

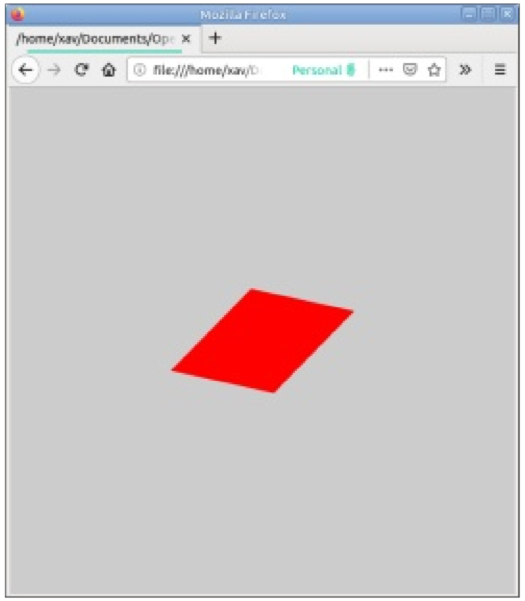
Reloading the page at this point shows that, if anything, we've made the problem worse! Now the square is rotating over an even wider range than before, spending most of its time out of bounds. The reason for this is also a simple one: remember that we used the three parameter version of the rotate() transformation function, so even though our object is centered at (0, 0) it's still rotating around (50, 50). Now we can pare that function down to the single value version, and the line where we set the value of the transform attribute becomes this (bottom right).

Reload the page and we've got what we expected: a square that rotates and skews in-place at the top-left of the screen. Our last step is to move it back to (50, 50) with an extra translate() step inserted to the start of the list, whose values are hard-coded (next page, top right).

Note that the transformations

```
group.setAttribute("transform", `
  rotate(${angle})
  skewX(${skewXAmount})
  skewY(${skewYAmount})
`);
```

are actually applied in reverse order: first the skewY(), then skewX(), then rotate() then finally translate(). When all we had was rotate and skew functions the order made little difference, but adding the translate() makes a huge difference. Put it at the end of the list, and we're back to the same problem with the square zooming around as it skews. At the start of the list, however, we've got a nicely controlled square, rotating and skewing whilst never leaving the middle of the screen.



To complete our set of transform functions, let's add a scale(), so that our square also grows and shrinks. This function can take one or two parameters to indicate the scale factor: if only one is provided then the object is scaled equally in both the x and y directions; if you wish to scale the two directions differently, then you have to provide two parameters.

Note, however, that there's no parameter for the center point of the scaling operation. As with the skew functions, your object has to be positioned with its center at (0, 0) if you don't want it to move as well as change size. Since we've already handled this problem for skewing, we just need to ensure that our scale() function is put into the transform attribute after the translate(), to ensure that the scale operation is performed before the translation.

To make our animation more interesting we're going to animate the change in x and y scale separately, over different time periods but within the same range of 0.1 (one tenth of the width or height) to 3.0 (treble the width or height). We'll encompass these parameters as two more sets of properties in the group.animProperties object (middle right).

```
group.setAttribute("transform", `
  translate(50, 50)
  rotate(${angle})
  skewX(${skewXAmount})
  skewY(${skewYAmount})
`);
```

```
group.animProperties = {
    rotationDuration: 5,
    ...
    skewYMax: 50,

    scaleXDuration: 3,
    scaleXMin: 0.5,
    scaleXMax: 2,

    scaleYDuration: 7,
    scaleYMin: 0.5,
    scaleYMax: 2
  }
```

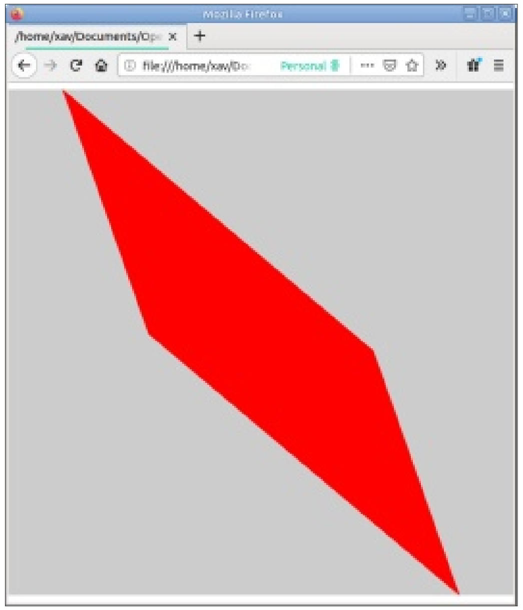
Like the skew functions we also want to animate from the minimum to the maximum, then back again – as opposed to continuously going in one direction as we did with rotate(). We therefore need a couple of blocks of code to calculate the relevant value at any given time point, changing direction after each period. Below is the code for the x-axis scaling – compare it to the equivalent block for skewX() from last month and you should be able

```
const scaleXDur = props.scaleXDuration;
const scaleXRange = props.scaleXMax - props.scaleXMin;
let scaleXPosition = (runningTime % scaleXDur) / scaleXDur;
const scaleXDirection = runningTime % (scaleXDur * 2);
if (scaleXDirection > scaleXDur) {
  scaleXPosition = 1 - scaleXPosition;
}
const scaleXAmount = (scaleXRange * scaleXPosition) + props.scaleXMin;
```

to reproduce code for the y-axis scaling yourself.

And, of course, we need to add our scale() function and two new parameters to the transform attribute (top right).

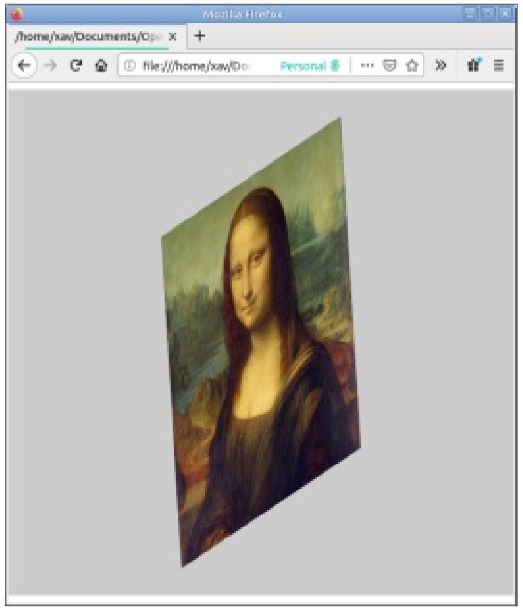
At last we have our object rotating, skewing and scaling, all while centered in the browser window – though a static screenshot doesn’t really do it justice.



I’m going to finish this month with a couple of exercises for you to try, which build on the animation we’ve created over these previous few articles:

• Our final transform attribute has a fixed translate() function to position the square in the middle of the screen. Why not add another two sets of parameters to also animate the x and y position, causing the square to move around the window a little as well. Setting min and max values either side of 50 means you can replace the hard-coded coordinates in the existing translate(). Or you could have a negative min and positive max, then use the values in a second translate() function – but be careful of the ordering!

• Try replacing the contents of the group with something else. It could be a more interesting single shape, such as a star or more faceted



```
group.setAttribute("transform", `
  translate(50, 50)
  rotate(${angle})
  skewX(${skewXAmount})
  skewY(${skewYAmount})
  scale(${scaleXAmount}, ${scaleYAmount})
`);
```

polygon, but it could also be any other Inkscape drawing – with multiple shapes and colors. Simply replacing the <rect> with an <image> element makes for an interesting effect, reminiscent of the kind of thing that required a Hollywood budget back in the 1980s.

The most important thing to remember is that this animation code – and the JS that we used previously in this series – are just examples to get you going. There’s no reason why you can’t create an animation that messes with the transform attribute whilst at the same time altering the fill and stroke, or manipulating the “d” attribute of a <path> in order to change the shape being drawn. With an understanding of how to change attributes and properties from JS you can create interactive or animated SVG images that go way beyond the frame-based limitations of an animated GIF. It’s a bit of a cliché, but the only real limit is your own imagination.

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