Working with the gridSVG Coordinate System

Simon Potter

September 3, 2012

Introduction

When exporting **grid** graphics as SVG, all information about the coordinate system used by **grid** is lost from the image as everything is turned into pixels. This document describes how gridSVG exports additional information during this process to retain the coordinate system.

To demonstrate this, we will show how to add points to an exported plot, both from within R and also, less permanently, within a JavaScript and SVG capable web browser.

Firstly, consider the following code which is a simple plot containing two items of interest. Firstly, a viewport is created which has scales defined for each of its axes. Secondly, points are added to the plot using native units. We then write this out to SVG in a file called `"pointsPlot.svg"` which can be viewed in a web browser.

```
> library(gridSVG)
```

```
> panelvp <- plotViewport(xscale = c(0, 20),
+                         yscale = c(0, 20),
+                         name = "panelvp")
> pushViewport(panelvp)
> grid.xaxis()
> grid.yaxis()
> grid.points(x = runif(10, 5, 15),
+             y = runif(10, 5, 15),
+             pch = 16,
+             name = "datapoints")
> popViewport()
> gridToSVG("pointsPlot.svg")
```

The challenge is to now modify this plot so that we can add extra information, such as new data points. As the SVG file was exported, all of the locations on the plot were transformed into pixels. This means that in our SVG file, none of

the axis scales exist, and the locations of points are no longer native coordinates, but absolutely positioned pixels.

Recent changes in gridSVG have enabled us to retain this information by exporting viewport metadata in the form of JSON. This enables us to be able to retain viewport locations and scales so that we can now transform pixel locations to native coordinates, and vice versa.

The following fragment shows the coordinates file that is exported by gridSVG. It is exported in the form of a JavaScript statement that assigns an object literal to a variable, `"gridSVGCoords"`.

```
var gridSVGCoords = {
 "ROOT": {
 "x":      0,
"y":      0,
"width":    432,
"height":    432,
"xscale": [      0,    432 ],
"yscale": [      0,    432 ],
"inch":     72
},
"panelvp": {
 "x":  59.04,
"y":  73.44,
"width": 342.72,
"height": 299.52,
"xscale": [      0,     20 ],
"yscale": [      0,     20 ],
"inch":     72
}
};
```

This shows all of the information available to gridSVG, which includes the viewport location and dimensions in terms of SVG pixels. Also included are the axis scales and how many pixels gridSVG thought an inch was as it was writing out the SVG file.

Browser-based Modification

We can modify the plot using the information described earlier by executing JavaScript code to insert SVG elements representing points into the plot. To start off we first load the image into the browser. What this does is loads the SVG image, and executes any JavaScript code that is referenced or included by the image. By default gridSVG exports coordinate information to a JavaScript file, along with a utility JavaScript file that contains functions useful for working with gridSVG graphics. In particular, the utility code includes functions that enable us to do unit conversion in the browser, e.g. from `native` to `npc` or to `inches`.

Because gridSVG must perform some name manipulation to ensure that SVG element `id`s are unique, a couple of JavaScript utility functions require introduction. Firstly, although not stricly necessary, if we know the name of the viewport, we can find out which viewport path it belonged to by calling `grobViewport()`.

```
>>> grobViewport("datapoints");
"panelvp.1"
```

We see that the viewport name is not exactly what we chose in R, but suffixed with a numeric index. To drop the suffix, we can call `baseViewportPath()` to get the original name of the viewport path. This will be identical to the viewport name that we assigned in R, or whatever **grid** assigned it in the case that a name was unspecified.

```
>>> baseViewportPath("panelvp.1");
"panelvp"
```

Now that we can query the viewport name, we know which viewport to draw into and the SVG element that we can add elements to. However, the issue remains that we really want to be able to use `native` units in the browser, rather than SVG pixels. To remedy this, unit conversion functions have been created. These functions are:

- `viewportConvertX`
- `viewportConvertY`
- `viewportConvertWidth`
- `viewportConvertHeight`

The first two conversion functions take three parameters, the viewport you want the location of, the size of the unit, and what type of unit it is. These functions return a number which represents the location in terms of SVG pixels.

The second two conversion functions take an additional parameter which is the type of unit that you want to convert to. This means we can convert between `inches`, `native` and `npc` in the browser without requiring an instance of R available, so long as we stick to our existing viewports.

As an example of how we might use these functions, we can find out where the coordinates $(3, 14)$ are in the main panel by running the following code:

```
>>> viewportConvertX("panelvp", 3, "native");
110.45
>>> viewportConvertY("panelvp", 14, "native");
283.1
```

We now know that the location of $(3, 14)$ in SVG pixels is $(110.45, 283.1)$. Using this information we can insert a new point into our plot at that location. To do this requires a bit of knowledge of JavaScript, and knowledge of the SVG DOM. To demonstrate this, a red SVG circle is going to be inserted at $(3, 14)$, in the plot using JavaScript.

```
// Getting the element that contains all existing points
var panel = document.getElementById("panelvp.1");

// Creating an SVG circle element
var c = document.createElementNS("http://www.w3.org/2000/svg", "circle");

// Setting some SVG properties relating to the appearance
// of the circle
c.setAttribute("stroke", "rgb(255,0,0)");
c.setAttribute("fill", "rgb(255,0,0)");
c.setAttribute("fill-opacity", 1);
c.setAttribute("r", 4);

// Setting the location of our points via the gridSVG
// conversion functions
c.setAttribute("cx", viewportConvertX("panelvp", 3, "native"));
c.setAttribute("cy", viewportConvertY("panelvp", 14, "native"));

// Adding the point to the same "viewport" as the existing points
panel.appendChild(c);
```

When running this code in the browser we see the new point. More complex demonstrations and usage of gridSVG utility functions are possible, but are outside of the scope of this document. However, a JavaScript library of particular significance that can assist greatly in manipulating SVG images in the browser is d3.js.

All changes to an SVG image via JavaScript are lost when the image is reloaded. To modify the image programmatically while also saving the state we need to use a tool other than JavaScript.

Modification via the XML package

In order to reproduce the effect of the JavaScript example earlier, we will be making use of the XML package in order to modify our SVG image. The first step is to load the required packages:

```
> library(gridSVG)
> library(XML)
```

We then need to parse the image, so that it is represented as a document within R.

```
> svgdoc <- xmlParse("pointsPlot.svg")
```

We know that the name of the viewport we are looking for has the exported name of "panelvp.1". An XPath query can be created to collect this viewport.

```
> # Getting the object representing our viewport that contains
> # our data points
> panel <- getNodeSet(svgdoc,
+                     "//svg:g[contains(@id, 'panelvp')]",
+                     c(svg="http://www.w3.org/2000/svg"))[[1]]
```

Now, we need to read in the JavaScript file that contains the coordinates information. However, some cleanup is needed because the code is designed to be immediately loaded within a browser, and is thus not simply JSON. We need to clean up the data so that it is able to be parsed by fromJSON.

```
> # Reading in, cleaning up and importing the coordinate system
> jsonData <- readCoordsJS("pointsPlot.svg.coords.js")
```

We now have valid JSON in the form of a character vector. Using this, we can initialise a coordinate system in R by utilising both gridSVGCoords and fromJSON.

```
> gridSVGCoords(fromJSON(jsonData))
```

Now that a coordinate system is initialised we are able convert coordinates into SVG pixels. This means we can create a <circle> element and correctly position it using "native" units at $(3, 14)$.

```
> # Creating an SVG circle element to insert into our image
> # that is red, and at (3, 14)
> circ <- newXMLNode("circle",
+                    parent = panel,
+                    attrs = list(cx = viewportConvertX("panelvp", 3, "native"),
+                                 cy = viewportConvertY("panelvp", 14, "native"),
+                                 r = 4,
+                                 stroke = "red",
+                                 fill = "red",
+                                 "fill-opacity" = 1))
```

Note that we have used the viewportConvert* functions to position the circle at the correct locations. This is because the same functions that are available in JavaScript are also available in SVG.

This point has been inserted into the same SVG group as the rest of the points by setting the "parent" parameter to the object representing the viewport group.

The only thing left to do is write out the new XML file with the point added.

```
> # Saving a new file for the modified image
> saveXML(svgdoc, file = "newPointsPlot.svg")
```

```
[1] "newPointsPlot.svg"
```

The new SVG image is located at "newPointsPlot.svg" and when loaded into the browser shows the new point. The appearance of the plot should be identical to the modifications we made using JavaScript, except these modifications are permanent and are able to be distributed to others.