Working with the gridSVG Coordinate System

Simon Potter

September 21, 2012

Introduction

grid is an alternative graphics system to the traditional base graphics system provided by R. Two key features of **grid** distinguish it from the base graphics system, graphics objects (grobs) and viewports.

Viewports are how **grid** defines a drawing context and plotting region. All drawing occurs relative to the coordinate system within a viewport. Viewports have a location and dimension and set scales on the horizontal and vertical axes. Crucially, they also have a name so we know how to refer to them.

Graphics objects store information necessary to describe how a particular object is to be drawn. For example, a **grid** `circleGrob` contains the information used to describe a circle, in particular its location and its radius. As with viewports, graphics objects also have names.

The task that gridSVG performs is to translate viewports and graphics objects into SVG equivalents. In particular, the exported SVG image retains the naming information on viewports and graphics objects. The advantage of this is we can still refer to the same information in **grid** and in SVG. In addition, we are able to annotate **grid** grobs to take advantage of SVG features such as hyperlinking and animation.

The gridSVG Coordinate System

When exporting **grid** graphics as SVG, instead of positioning within a viewport, all drawing occurs within a single pixel-based coordinate system. This document describes how gridSVG exports additional information during this process to retain the original.

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.

```
R> library(gridSVG)
```

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

```
<g id="datapoints">
  <defs>
    <symbol id="gridSVG.pch16" viewBox="-5 -5 10 10" overflow="visible">
      <circle cx="0" cy="0" r="3.75"/>
    </symbol>
  </defs>
  <use id="datapoints.1" xlink:href="#gridSVG.pch16" x="152.25" y="261.94" width="8.62" he
  <use id="datapoints.2" xlink:href="#gridSVG.pch16" x="208.15" y="201.59" width="8.62" he
  <use id="datapoints.3" xlink:href="#gridSVG.pch16" x="208.01" y="177.8" width="8.62" hei
  <use id="datapoints.4" xlink:href="#gridSVG.pch16" x="294.57" y="247.77" width="8.62" he
  <use id="datapoints.5" xlink:href="#gridSVG.pch16" x="194.26" y="163.45" width="8.62" he
  <use id="datapoints.6" xlink:href="#gridSVG.pch16" x="158.24" y="220.05" width="8.62" he
  <use id="datapoints.7" xlink:href="#gridSVG.pch16" x="174.84" y="173.57" width="8.62" he
  <use id="datapoints.8" xlink:href="#gridSVG.pch16" x="234.85" y="199.1" width="8.62" hei
  <use id="datapoints.9" xlink:href="#gridSVG.pch16" x="173.06" y="279.91" width="8.62" he
  <use id="datapoints.10" xlink:href="#gridSVG.pch16" x="252.3" y="189.73" width="8.62" he
</g>
```

The previous SVG fragment shows what gridSVG has translated the points to. Note that the locations (`x` and `y`) and dimensions (`width` and `height`) of each of the points have been translated into absolute locations in SVG. They no longer relate to the axis scales in their viewport, instead they are SVG pixels.

Recent changes in gridSVG have enabled us to keep viewport information by exporting viewport metadata in the form of JSON, a structured data format. 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.1": {
 "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. This JavaScript object contains a list of viewport names, with each viewport name associated with its metadata. This metadata includes the viewport location and dimensions in terms of SVG pixels. Also included are the axis scales, along with the resolution that the viewport was exported at. The resolution simply represents the number of pixels that span an inch.

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 ids 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().

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

We see that the viewport name is not exactly what we chose in R, but suffixed with a numeric index. 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:

```
JS> viewportConvertX("panelvp.1", 3, "native");
110.45
JS> viewportConvertY("panelvp.1", 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, and will be demonstrated in **Section XXXXXX**. A JavaScript library of particular significance that can assist greatly in manipulating SVG images in the browser is d3.js. d3.js will feature prominently in our more complex example.

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. As gridSVG automatically loads the XML package, all of the functionality from the XML package is readily available to us.

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

```
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.

```
R> # Getting the object representing our viewport that contains
R> # our data points
R> panel <- getNodeSet(svgdoc,
```

```
R+                          "//svg:g[contains(@id, 'panelvp')]",
R+                          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`.

```
R> # Reading in, cleaning up and importing the coordinate system
R> 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`.

```
R> 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 $(3, 14)$.

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

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

An advantage of being able to modify the image, or even to generate the image immediately is that it opens up the possibility of serving dynamic SVG images over the web. Such an example is demonstrated in **Section XXXXX** using the Rook package.