The gridSVG Package

by Paul Murrell and Simon Potter

Abstract The **gridSVG** package can be used to generate a **grid**-based R plot in an SVG format, with the ability to add special effects to the plot. The special effects include animation, interactivity, and advanced graphical features, such as masks and filters. This article provides a basic introduction to important functions in the **gridSVG** package and discusses the advantages and disadvantages of **gridSVG** compared to similar R packages.

Introduction

The SVG graphics format (Dengler et al., 2011) is a good format for including plots in web pages because it is a vector format (so it scales well) and because it offers features for animation and interactivity. SVG also integrates well with other important web technologies such as HTML and JavaScript.

It is possible to produce a static R plot in an SVG format with the built-in `svg()` function (from the **grDevices** package), but the **gridSVG** package (Murrell and Potter) provides an alternative way to generate an SVG plot that allows for creating animated and interactive graphics.

There are two types of graphics functions in R: functions based on the default **graphics** package and functions based on the **grid** graphics package. As the package name suggests, the **gridSVG** package only works with a plot that is drawn using the **grid** graphics package. This includes plots from several important graphics packages in R, such as **lattice** (Sarkar, 2008) and **ggplot2** (Wickham, 2009), but **gridSVG** does not work with all plots that can be produced in R.

This article demonstrates basic usage of the **gridSVG** package and outlines some of the ways that **gridSVG** can be used to produce graphical results that are not possible in standard R graphics. There is also a discussion of other packages that provide ways to generate dynamic and interactive graphics for the web and the strengths and weaknesses of **gridSVG** compared to those packages.

Basic usage

The following code draws a **lattice** multi-panel plot (see Figure 1).

```
> library(lattice)

> dotplot(variety ~ yield | site, data = barley, groups = year,
          key = simpleKey(levels(barley$year), space = "right"),
          subset = as.numeric(site) < 4, layout = c(1, 3))
```

The `grid.export()` function in **gridSVG** converts the current (**grid**) scene on the active graphics device to an SVG format in an external file.

```
> library(gridSVG)

> grid.export("lattice.svg")
```

This SVG file can be viewed directly in a browser (see Figure 2) or embedded within HTML as part of a larger web page.

This usage of **gridSVG**, to produce a static SVG version of an R plot for use on the web, offers no obvious benefit compared to the built-in `svg()` graphics device. However, the **gridSVG** package provides several other functions that can be used to enhance the SVG version of an R plot.

A simple example

In order to demonstrate, with code, some of the distinctive features of **gridSVG**, we introduce a simple **grid** scene that is inspired by the Monty Hall problem.[1]

```
> library(grid)
```

[1] http://en.wikipedia.org/wiki/Monty_Hall_problem

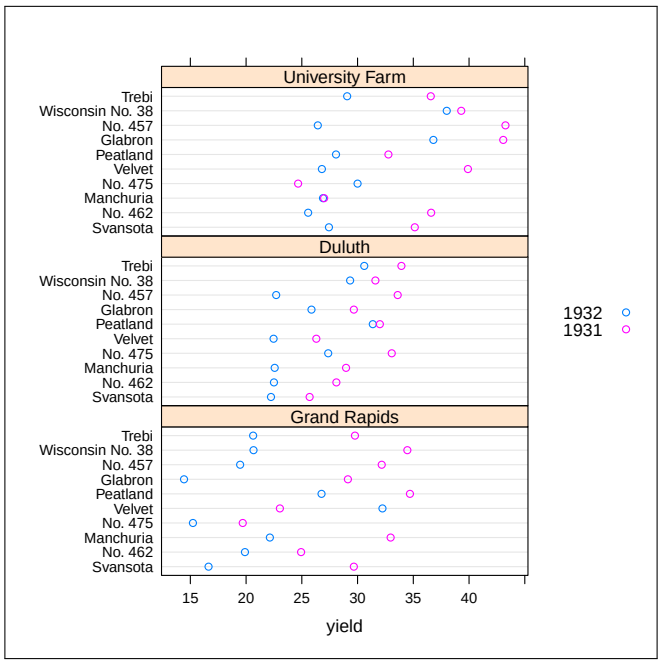

Figure 1: A **lattice** multi-panel plot drawn on the standard `pdf()` graphics device.

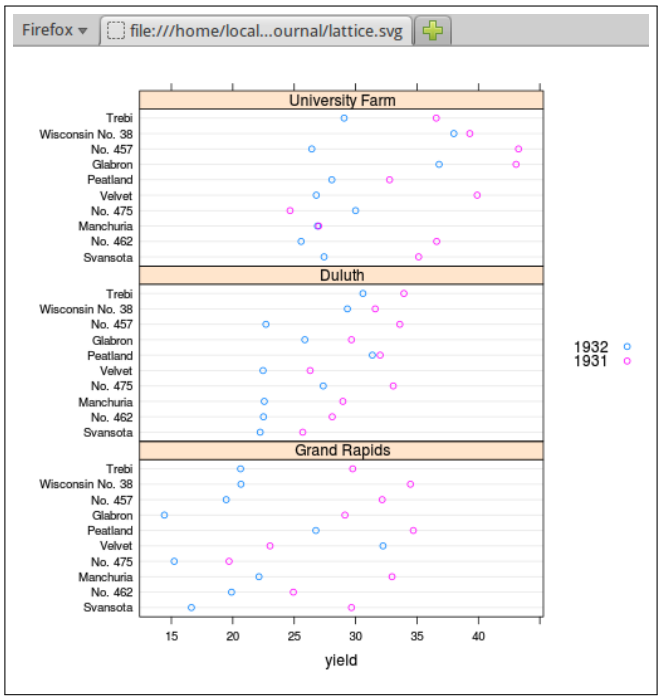

Figure 2: The **lattice** plot from Figure 1 exported to an SVG file by **gridSVG** and viewed in Firefox. This demonstrates that a static R plot can be converted to an SVG format with **gridSVG** for use on the web.

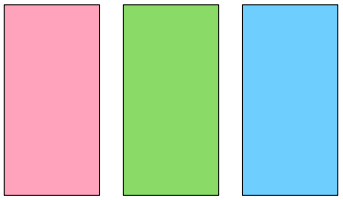

Figure 3: A diagram of the Monty Hall problem, drawn using **grid**. Hidden behind each rectangle is either the word "goat" or the word "car").

The scene consists of three words, "goat", "goat", and "car", drawn in random order across the page, with an opaque rectangle drawn on top of each word.

In relation to the Monty Hall problem, the three rectangles represent three "doors", behind which are hidden two goats and a car. A "contestant" must choose a door and then he or she gets the "prize" behind that door. However, after the contestant has chosen a door, a "game show host" opens one of the other doors to reveal a "goat" and the contestant gets the opportunity to change to the remaining unopened door or stick with the original choice. Should the contestant stick or switch?[2]

The following code produces the scene and the result is shown in Figure 3. The main drawing code is wrapped up in a function so that we can reuse it later on.

```
> text <- sample(c("goat", "goat", "car"))
> cols <- hcl(c(0, 120, 240), 80, 80)

> MontyHall <- function() {
      grid.newpage()
      grid.text(text, 1:3/4, gp = gpar(cex = 2), name = "prizes")
      for (i in 1:3) {
          grid.rect(i/4 - .1, width=.2, height=.8, just = "left",
                    gp = gpar(fill = cols[i]), name = paste0("door", i))
      }
  }

> MontyHall()
```

The code in the `MontyHall()` function makes use of the fact that **grid** functions allow names to be associated with the objects in a scene. In this case, the three rectangles in this scene have been given names—`"door1"`, `"door2"`, and `"door3"`—and the text has been given the name `"prizes"`.

The **grid** function `grid.ls()` can be used to display the names of all objects in a scene.

```
> grid.ls(fullNames = TRUE)

text[prizes]
rect[door1]
rect[door2]
rect[door3]
```

These names will be used later to identify the rectangles so that we can modify them to generate special effects.

Hyperlinks

The `grid.hyperlink()` function from the **gridSVG** package can be used to add hyperlinks to parts of a **grid** scene. For example, the following code adds a link to each door so that clicking on a door (while viewing the SVG version of the scene in a browser) leads to a Google Image Search on either "car" or "goat" depending on what is behind the door. The first argument to `grid.hyperlink()` is the name of the **grid** object with which to associate the hyperlink. The `href` argument provides the actual link and the `show` argument specifies how to show the target of the link (`"new"` means open a new tab or window).

[2] An exercise for the reader is to determine which door conceals the car based on the R code and figures presented in this article.

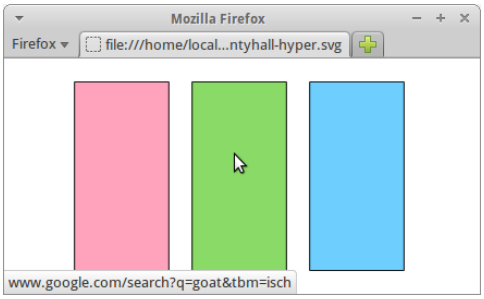

Figure 4: The Monty Hall image, with a hyperlink on each door. The mouse is hovering over the middle door and the browser is showing the hyperlink target in the bottom-left of its window. If we click the mouse, we will navigate to a Google Image Search for the word "goat".

```
> library(gridSVG)

> links <- c("http://www.google.com/search?q=car&tbm=isch",
             "http://www.google.com/search?q=goat&tbm=isch")
> for (i in 1:3) {
      grid.hyperlink(paste0("door", i),
                     href = links[match(text[i], c("car", "goat"))],
                     show = "new")
  }
```

After running this code, the scene is completely unchanged on a normal graphics device, but if we use `grid.export()` to convert the scene to SVG, we end up with an image that contains hyperlinks. Figure 4 shows the result, with the mouse hovering over the middle door; at the bottom-left of the browser window, we can see from the hyperlink that there is a goat behind this door.

```
> grid.export("montyhall-hyper.svg")
```

Animation

The function `grid.animate()` from **gridSVG** allows us to animate the features of shapes in a **grid** scene. For example, the following code draws the Monty Hall scene again and then animates the width of the middle door so that it slides open (to reveal the word "goat"). The first argument to `grid.animate()` is the name of the object to animate. Subsequent arguments specify which feature of the object to animate, in this case `width`, plus the values for the animation. The `duration` argument controls how long the animation will last.

```
> MontyHall()
> goatDoor <- grep("goat", text)[1]
> grid.animate(paste0("door", goatDoor), width = c(.2, 0), duration = 2)

> grid.export("montyhall-anim.svg")
```

Again, no change is visible on a normal R graphics device, but if we export to SVG and view the result in a browser, we see the animation (see Figure 5).

Advanced graphics features

The **gridSVG** package offers several graphics features that are not available in standard R graphics devices. These include non-rectangular clipping paths, masks, fill patterns and fill gradients, and filters (Murrell and Potter, 2013). This section demonstrates the use of a mask on the Monty Hall scene.

A mask is a greyscale image that is used to affect the transparency (or alpha-channel) of another image: anywhere the mask is white, the masked image is fully visible; anywhere the mask is black, the masked image is invisible; and anywhere the mask is grey, the masked image is semitransparent.

The following code uses standard **grid** functions to define a simple scene consisting of a white cross on top of a grey circle on a white background, which we will use as a mask (see Figure 6). Any **grid** scene can be used to create a mask; in this case, we use the `gTree()` function from **grid** to create a graphical object that is a collection of several other graphical objects.

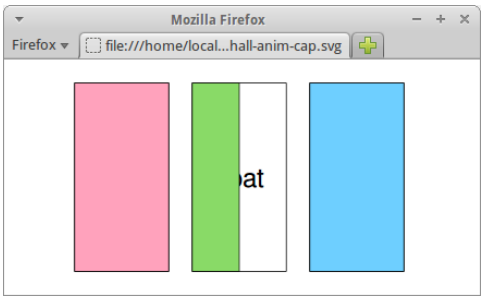

Figure 5: The Monty Hall image, with the middle "door" animated so that it slides open (to reveal the word "goat").

Figure 6: Using a mask on an image. The picture on the left shows a white cross on top of a grey circle on a white background. This is used as a mask on the rectangles in the Monty Hall image on the right. The effect is to create a semitransparent window in the middle door (through which we can glimpse the word "goat").

```
> circleMask <- gTree(children = gList(rectGrob(gp = gpar(col = NA, fill = "white")),
                          circleGrob(x = goatDoor/4, r=.15,
                                     gp = gpar(col = NA, fill = "grey")),
                          polylineGrob(c(0, 1, .5, .5),
                                       c(.5, .5, 0, 1),
                                       id = rep(1:2, each = 2),
                                       gp = gpar(lwd = 10, col = "white"))))
```

The next code shows how this crossed circle on a white background can be used as a mask to affect the transparency of one of the rectangles in the Monty Hall scene. The functions `mask()` and `grid.mask()` are from **gridSVG**. The `mask()` function takes a **grid** object (as generated above) and turns it into a mask object. The `grid.mask()` function takes the name of a **grid** object to mask, plus the mask object produced by `mask()`.

```
> MontyHall()
> grid.mask(paste0("door", goatDoor), mask(circleMask))

> grid.export("montyhall-masked.svg")
```

The effect of the mask is shown in Figure 6.

Interactivity

The `grid.garnish()` function in the **gridSVG** package opens up a broad range of possibilities for enhancing a **grid** scene, particularly for adding interactivity to the scene.

A simple example is shown in the code below. Here we are adding tooltips to each of the doors in the Monty Hall scene so that hovering the mouse over a door produces a label that shows what is behind the door (see Figure 7). The first argument to `grid.garnish()` is the name of the object to modify. Subsequent arguments specify SVG attributes to add to the object; in this case, we add a `title` attribute, which results in a tooltip (in some browsers).

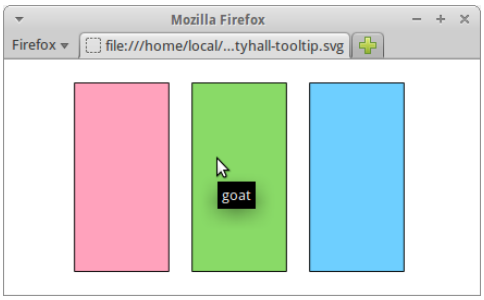

Figure 7: The Monty Hall image with tooltips added to each door. The mouse is hovering over the middle door, which results in a tooltip being displayed to show that there is a "goat" behind this door.

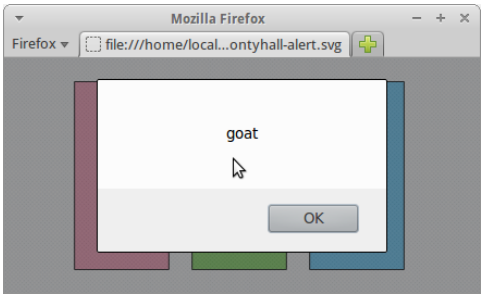

Figure 8: The Monty Hall image with interactivity. The mouse has just been clicked on the middle door, which has resulted in an alert box popping up to show that this door has a "goat" behind it.

```
> MontyHall()
> for (i in 1:3) {
      grid.garnish(paste0("door", i), title = text[i])
  }

> grid.export("montyhall-tooltip.svg")
```

The `grid.garnish()` function can also be used to associate JavaScript code with an object in the scene. The following code shows a simple example where clicking on one of the rectangles pops up an alert box showing what is behind that door (see Figure 8). The attribute in this example is `onclick`, which is used to define an action that occurs when the object is clicked with the mouse (in a browser).

```
> MontyHall()
> for (i in 1:3) {
      grid.garnish(paste0("door", i),
                   onclick = paste0("alert('", text[i], "')"))
  }

> grid.export("montyhall-alert.svg")
```

For more complex interactions, it is possible to include JavaScript code within the scene, using the `grid.script()` function, so that an event on an object within the scene can be associated with a JavaScript function call to perform a more sophisticated action. The code below shows a simple example where clicking on one of the rectangles in the Monty Hall scene will call the JavaScript function `open()` to "open" the door (by making the rectangle invisible; see Figure 9). The `open()` function is defined in a separate file called `"MontyHall.js"` (shown in Figure 10).

```
> MontyHall()
> for (i in 1:3) {
      grid.garnish(paste0("door", i), onclick = "open(evt)")
  }
> grid.script(file = "MontyHall.js")

> grid.export("montyhall-js.svg")
```

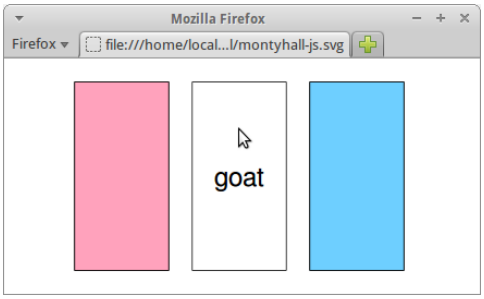

Figure 9: The Monty Hall image with more interactivity. The mouse has just been clicked on the middle door, which has resulted in the middle door becoming invisible, thereby revealing a "goat" behind the door.

```
open = function(e) {
    e.currentTarget.setAttribute("visibility", "hidden");
}
```

Figure 10: The JavaScript code used in Figure 9 that defines the `open()` function to "open" a door by making the rectangle invisible.

A more complex demonstration

The previous section kept things very simple in order to explain the main features of **gridSVG**. In this section, we present a more complex example which involves adding interactivity to a **lattice** multi-panel plot. The following code generates the **lattice** plot from Figure 1.

```
> dotplot(variety ~ yield | site, data = barley, groups = year,
          key = simpleKey(levels(barley$year), space = "right"),
          subset = as.numeric(site) < 4, layout = c(1, 3))
```

Because the **lattice** package is built on **grid**, and because the **lattice** package names all of the objects that it draws,[3] there are names for every object drawn in this plot. The following code uses the **grid** function `grid.grep()`[4] to show some of the named objects in this plot (in this case, all objects that have a name that contains `"xyplot.points"`). These are the objects that represent the data symbols within the **lattice** plot.

```
> grid.grep("xyplot.points", grep = TRUE, global = TRUE)

[[1]]
plot_01.xyplot.points.group.1.panel.1.1

[[2]]
plot_01.xyplot.points.group.2.panel.1.1

[[3]]
plot_01.xyplot.points.group.1.panel.1.2

[[4]]
plot_01.xyplot.points.group.2.panel.1.2

[[5]]
plot_01.xyplot.points.group.1.panel.1.3

[[6]]
plot_01.xyplot.points.group.2.panel.1.3
```

The following code uses `grid.garnish()` to add event handlers to the objects that represent the points in the plot, so that JavaScript functions are called whenever the mouse moves over a point and whenever the mouse moves off the point again.

[3] http://lattice.r-forge.r-project.org/Vignettes/src/naming-scheme/namingScheme.pdf
[4] Introduced in R version 3.1.0.

```
function highlight(evt) {
  var element = evt.currentTarget;
  var id = element.id;
  var index = id.substring(id.search(/[.][0-9]+$/) + 1, id.length);
  for (var panel=1;panel<4;panel++) {
    for (var group=1;group<3;group++) {
      var selid =
        'plot_01.xyplot.points.group.'+group+'.panel.1.'+panel+'.1.'+index;
      var dot = document.getElementById(selid);
      dot.setAttribute("stroke-width", "6");
    }
  }
}

function unhighlight(evt) {
  var element = evt.currentTarget;
  var id = element.id;
  var index = id.substring(id.search(/[.][0-9]+$/) + 1, id.length);
  for (var panel=1;panel<4;panel++) {
    for (var group=1;group<3;group++) {
      var selid =
        'plot_01.xyplot.points.group.'+group+'.panel.1.'+panel+'.1.'+index;
      var dot = document.getElementById(selid);
      dot.setAttribute("stroke-width", "1");
    }
  }
}
```

Figure 11: The JavaScript code that defines the `highlight()` and `unhighlight()` functions to implement linked selection of points for the **lattice** plot in Figure 12.

```
> numPoints <- length(levels(barley$variety))
> grid.garnish("xyplot.points", grep = TRUE, global = TRUE, group = FALSE,
               onmouseover = rep("highlight(evt)", numPoints),
               onmouseout = rep("unhighlight(evt)", numPoints),
               "pointer-events" = rep("all", numPoints))
```

This use of `grid.garnish()` differs from the previous simple examples because it has an effect on several **grid** objects, rather than just one. The `grep` and `global` arguments specify that the name, `"xyplot.points"`, should be treated as a regular expression and the garnish will affect all objects in the scene with a name that matches that pattern. Furthermore, each **grid** object that matches represents several data symbols, so the `group` argument is used to specify that the garnish should be applied to each individual data symbol. Because, for each object, the garnish is being applied to multiple data symbols, we must provide multiple values, which explains the use of `rep()` for the `onmouseover`, `onmouseout`, and `pointer-events` arguments.

The JavaScript code that defines the event handlers `hightlight()` and `unhighlight()` is shown in Figure 11. A detailed explanation of this code is beyond the scope of this article, but it should be clear that these functions are relatively simple, just looping over the two groups in each panel, and over the three panels, to highlight (or unhighlight) all points that share the same index.

This JavaScript code is added to the plot using `grid.garnish()`, and then the whole scene is exported to SVG with `grid.export()`.

```
> grid.script(file = "lattice-brush.js")
> grid.export("lattice-brush.svg")
```

A snapshot of the final result is shown in Figure 12, with the mouse over one point and all related points highlighted.

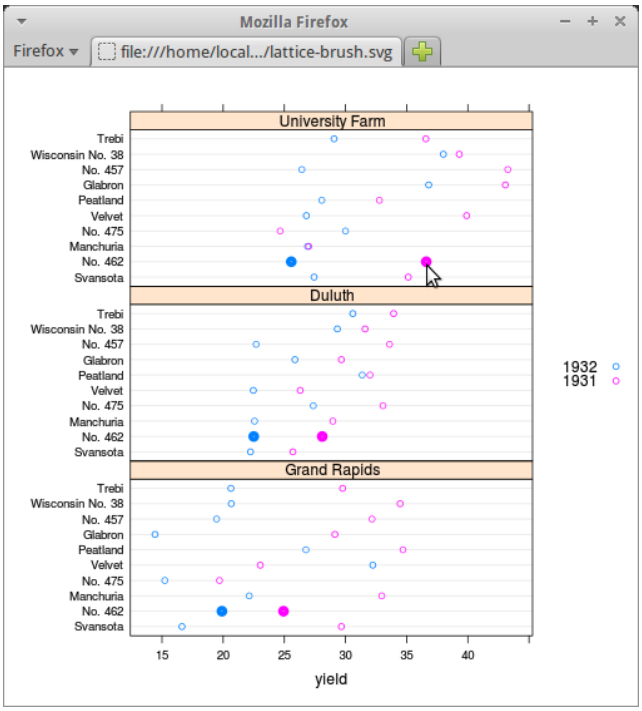

Figure 12: The **lattice** plot from Figure 1 with interaction added. Moving the mouse over a point highlights the point and highlights every other "related" point in all panels of the plot.

Limitations

The **gridSVG** package provides an opportunity to produce more sophisticated, more dynamic, and more interactive R plots compared to the standard R graphics devices. However, there are some strict limitations on what can be achieved with this package.

First of all, the package only works for plots that are based on the **grid** graphics system. This includes some major graphics packages, such as **lattice** and **ggplot2**, but excludes a large amount of graphics functionality that is only available in the default **graphics** package or packages that build on **graphics**. Given a plot from a function that is not based on **grid**, the **gridSVG** package will only produce a blank SVG file.

Another limitation is that **gridSVG** does not generate any JavaScript code itself. This means that anything beyond the most basic interactivity will require the user to write JavaScript code, which imposes a burden on the user in terms of both time and knowledge.

Another point that has only briefly been acknowledged in the example R code so far is that the **gridSVG** functions that add special features to a **grid** scene (such as hyperlinks and animation) rely heavily on the ability to *identify* specific components of a **grid** scene. The Monty Hall examples all rely on the fact that the rectangles that are drawn to represent doors each have a name—`"door1"`, `"door2"`, and `"door3"`—and the code that adds hyperlinks or animation identifies the rectangles by using these names. This means that **gridSVG** is dependent upon an appropriate naming scheme being used for any **grid** drawing (Murrell, 2012). This requirement is met by the **lattice** package and, to a lesser extent by the **ggplot2** package, but cannot be relied on in general.

Alternative approaches

The **gridSVG** package provides one way to produce dynamic and interactive versions of R plots for use on the web, but there are several other packages that provide alternative routes to the same destination. This section discusses the differences between **gridSVG** and several other packages that have similar goals.

The **animation** package (Xie, 2013) provides a convenient front-end for producing animations in various formats (some of which are appropriate for use on the web), but the approach is frame-based (draw lots of separate images and then stitch them together to make an animation). The advantage of an SVG-based approach to animation is that the animation is declarative, which means that the

animation can be described more succinctly and efficiently and the resulting animation will often appear smoother. On the other hand, the **animation** package will work with any R graphics output; it is not restricted to just **grid**-based output.

The **SVGAnnotation** package (Nolan and Temple Lang, 2012) performs a very similar role to **gridSVG**, by providing functions to export R plots to an SVG format with the possiblity of adding dynamic and interactive features. One major advantage of **SVGAnnotation** is that it will export R plots that are based the standard **graphics** package (as well as plots that are based on **grid**). **SVGAnnotation** also provides some higher-level functions that automatically generate JavaScript code to implement specific sorts of more complex interactivity. For example, the `linkPlots()` function can be used to generate linked plots, where moving the mouse over a data symbol in one plot automatically highlights a corresponding point in another plot. The main disadvantage of **SVGAnnotation** is that it works with the SVG that is produced by the built-in `svg()` device, which is much less structured than the SVG that is generated by **gridSVG**. That is not a problem if the functions that **SVGAnnotation** provides do everything that we need, but it makes for much more work if we need to, for example, write our own JavaScript code to work with the SVG that **SVGAnnotation** has generated.

Another package that can export R graphics output to SVG is the **RSVGTipsDevice** package (Plate, 2011). This package creates a standard R graphics device, so it can export any R graphics output, but it is limited to adding tooltips and hyperlinks. This package also requires the tooltips or hyperlinks to be added at the time that the R graphics output is produced, rather than after-the-fact using names to refer to previously-drawn output. This makes it harder to associate tooltips or hyperlinks with output that is produced by someone else's code, such as a complex **lattice** plot.

A number of packages, including **rCharts** and **googleVis** (Vaidyanathan, 2013; Gesmann and de Castillo, 2011), provide a quite different approach to producing dynamic and interactive plots for the web. These packages outsource the plot drawing to JavaScript libraries such as NVD3, highcharts, and the Google Visualisation API (Novus, 2012; Highsoft AS, 2013; Google, 2013). The difference here is that the plots produced are not R plots. The advantage is that very little R code is required to produce a nice result, provided the JavaScript library can produce the style of plot and the sort of interactivity that we want.

Another approach to interactivity that is implemented in several packages, notably **shiny** (RStudio Inc., 2013), involves running R as a web server and producing new R graphics in response to user events in the browser. The difference here is that the user typically interacts with GUI widgets (buttons and menus) outside the graphic and each user event generates a completely new R graphic. With **gridSVG**, the user can interact directly with elements of the graphic itself and all of the changes to the graphic occur in the browser with no further need of R.

In summary, using the **gridSVG** package is appropriate if we want to add advanced graphics features to a **grid**-based R plot, or if we want to add dynamic or interactive elements to a **grid**-based R plot, particularly if we want to produce a result that is not already provided by a high-level function in the **SVGAnnotation** package. An approach that holds some promise is to generate SVG content using **gridSVG** and then manipulate that content by adding JavaScript code based on a sophisticated JavaScript library such as d3 (Bostock et al., 2011) and Snap.svg (Baranovskiy, 2014).

Availability

The **gridSVG** package is available from CRAN. The code examples in this article are known to work for **gridSVG** versions 1.3 and 1.4 using Firefox 28.0 on Ubuntu 12.04.

Support for SVG varies between browsers, for example Chrome 34.0 on Ubuntu 12.04 does not produce the tooltips in Figure 7. Several web sites provide summary tables of supported SVG features.[5] Differences between browser JavaScript engines is another potential source of variation. Nevertheless, all major browsers now provide native support of at least basic SVG features, several mature and stable JavaScript libraries are available to abstract away browser differences, and the situation is constantly improving.

Online versions of the figures in this article are available from `http://www.stat.auckland.ac.nz/~paul/Reports/gridSVGrjV2/`. Further documentation and examples for **gridSVG** are available from `https://www.stat.auckland.ac.nz/~paul/R/gridSVG/`.

[5] `http://caniuse.com/svg`
`http://en.wikipedia.org/wiki/Comparison_of_layout_engines_%28Scalable_Vector_Graphics%29`

Acknowledgements

We would like to thank the anonymous reviewers for many helpful comments that lead to improvements in this article.

Bibliography

D. Baranovskiy. Snap.svg, 2014. URL http://snapsvg.io/. [p142]

M. Bostock, V. Ogievetsky, and J. Heer. D3: Data-Driven Documents. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2011. URL http://vis.stanford.edu/papers/d3. [p142]

P. Dengler, D. Jackson, C. Lilley, J. Fujisawa, C. McCormack, E. Dahlström, A. Grasso, J. Ferraiolo, D. Schepers, and J. Watt. Scalable vector graphics (SVG) 1.1 (second edition). W3C recommendation, W3C, Aug. 2011. http://www.w3.org/TR/2011/REC-SVG11-20110816/. [p133]

M. Gesmann and D. de Castillo. googleVis: Interface between R and the Google Visualisation API. *The R Journal*, 3(2):40–44, December 2011. URL http://journal.r-project.org/archive/2011-2/RJournal_2011-2_Gesmann+de~Castillo.pdf. [p142]

Google. Google Visualization API, 2013. URL https://developers.google.com/chart/interactive/docs/reference. [p142]

Highsoft AS. Highcharts JS, 2013. URL http://www.highcharts.com/. [p142]

P. Murrell. What's in a Name? *The R Journal*, 4(2):5–12, dec 2012. URL http://journal.r-project.org/archive/2012-2/RJournal_2012-2_Murrell.pdf. [p141]

P. Murrell and S. Potter. *gridSVG: Export grid graphics as SVG*. R package version 1.4-0. [p133]

P. Murrell and S. Potter. Advanced SVG Graphics from R. Technical Report 2013-7, Department of Statistics, The University of Auckland, 2013. URL http://stattech.wordpress.fos.auckland.ac.nz/2013-7-advanced-svg-graphics-from-r/. [p136]

D. Nolan and D. Temple Lang. Interactive and animated scalable vector graphics and R data displays. *Journal of Statistical Software*, 46(1):1–88, 1 2012. ISSN 1548-7660. URL http://www.jstatsoft.org/v46/i01. [p142]

Novus. NVD3.js : Re-usable charts for d3.js, 2012. URL http://nvd3.org/. [p142]

T. Plate. *RSVGTipsDevice: An R SVG graphics device with dynamic tips and hyperlinks*, 2011. URL http://CRAN.R-project.org/package=RSVGTipsDevice. R package version 1.0-4. [p142]

RStudio Inc. *shiny: Web Application Framework for R*, 2013. URL http://CRAN.R-project.org/package=shiny. R package version 0.3.0. [p142]

D. Sarkar. *Lattice: Multivariate Data Visualization with R*. Springer-Verlag, New York, 2008. URL http://lmdvr.r-forge.r-project.org. ISBN 978-0-387-75968-5. [p133]

R. Vaidyanathan. *rCharts: Interactive Charts using Polycharts.js*, 2013. R package version 0.4.2. [p142]

H. Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag, New York, 2009. ISBN 978-0-387-98140-6. URL http://had.co.nz/ggplot2/book. [p133]

Y. Xie. animation: An R package for creating animations and demonstrating statistical methods. *Journal of Statistical Software*, 53:1–27, 2013. URL http://www.jstatsoft.org/v53/i01/. [p141]

Paul Murrell
The University of Auckland
Auckland
New Zealand paul@stat.auckland.ac.nz

Simon Potter
The University of Auckland
Auckland
New Zealand simon@sjp.co.nz