What's in a Name?

The Importance of Naming grid Grobs

by Paul Murrell

Abstract Any shape that is drawn using the **grid** graphics package can have a name associated with it. If a name is provided, it is possible to access, query, and modify the shape after it has been drawn. These facilities allow for very detailed customisations of plots and also for very general transformations of plots that are drawn by packages based on **grid**.

When a scene is drawn using the **grid** graphics package in R (R Development Core Team, 2011), a record is kept of each shape that was used to draw the scene. This record is called a *display list* and it consists of a list of R objects, one for each shape in the scene. For example, the following code draws several simple shapes: some text, a circle, and a rectangle (see Figure 1).

```
> library(grid)

> grid.text(c("text", "circle", "rect"),
+           x=1:3/4, gp=gpar(cex=c(3, 1, 1)))
> grid.circle(r=.25)
> grid.rect(x=3/4, width=.2, height=.5)
```

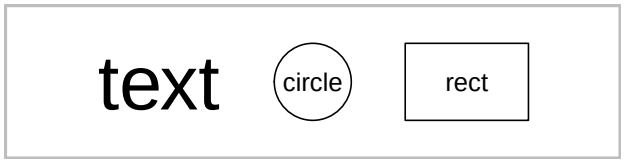

Figure 1: Some simple shapes drawn with **grid**.

The following code shows the contents of the display list for this scene. There is an object for each shape that we drew. The output below shows what sort of shape each object represents and it shows a name for each object (within square brackets). In the example above, we did not specify any names, so **grid** made some up.

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

text[GRID.text.5]
circle[GRID.circle.6]
rect[GRID.rect.7]
```

It is also possible to explicitly name each shape that we draw. The following code does this by specifying the `name` argument in each function call (the resulting scene is the same as in Figure 1) and shows that the objects on the display list now have the names that we specified.

```
> grid.text(c("text", "circle", "rect"),
+           x=1:3/4, gp=gpar(cex=c(3, 1, 1)),
+           name="leftText")
> grid.circle(r=.25, name="middleCircle")
> grid.rect(x=3/4, width=.2, height=.5,
+           name="rightRect")


> grid.ls(fullNames=TRUE)


text[leftText]
circle[middleCircle]
rect[rightRect]
```

Furthermore, **grid** provides functions that allow us to access and modify the objects on the display list. For example, the following code modifies the circle in the middle of Figure 1 so that its background becomes grey (see Figure 2). We select the object to modify by specifying its name.

```
> grid.edit("middleCircle", gp=gpar(fill="grey"))
```

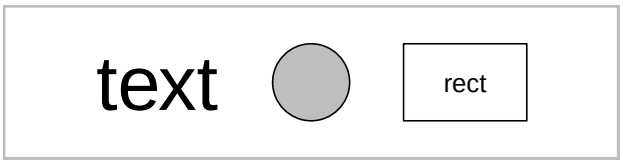

Figure 2: The simple shapes from Figure 1 with the middle circle modified so that its background is grey.

The purpose of this article is to discuss why it is useful to provide explicit names for the objects on the **grid** display list. We will see that several positive consequences arise from being able to identify and modify the objects on the display list.

Too many arguments

This section discusses how naming the individual shapes within a plot can help to avoid the problem of having a huge number of arguments or parameters in a high-level plotting function.

The plot in Figure 3 shows a *forest plot*, a type of plot that is commonly used to display the results of a meta-analysis. This plot was produced using the `forest()` function from the **metafor** package (Viechtbauer, 2010).

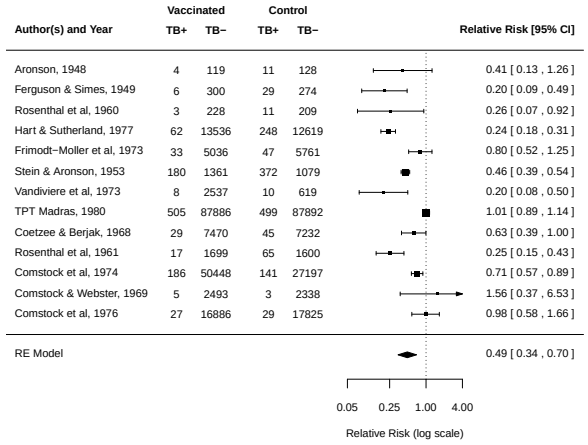

Figure 3: A forest plot produced by the `forest()` function from the **metafor** package.

This sort of plot provides a good example of how statistical plots can be composed of a very large number of simple shapes. The plot in Figure 3 consists of many different pieces of text, rectangles, lines, and polygons.

High-level functions like `forest()` are extremely useful because, from a single function call, we can produce many individual shapes and arrange them in a meaningful fashion to produce an overall plot. However, a problem often arises when we want to *customise* individual shapes within the plot.

For example, a post to the R-help mailing list in August 2011 asked for a way to change the colour of the squares in a forest plot because none of the (thirty-three) existing arguments to `forest()` allowed this sort of control. The reply from Wolfgang Viechtbauer (author of **metafor**) states the problem succinctly:

> "The thing is, there are so many different elements to a forest plot (squares, lines, polygons, text, axes, axis labels, etc.), if I would add arguments to set the color of each element, things would really get out of hand ...
>
> ... what if somebody wants to have a different color for *one* of the squares and a different color for the other squares?"

The reality is that it is impossible to provide enough arguments in a high-level plotting function to allow for all possible modifications to the low-level shapes that make up the plot. Fortunately, an alternative is possible through the simple mechanism of providing names for all of the low-level shapes.

In order to demonstrate this idea, consider the **lattice** plot (Sarkar, 2008) that is produced by the following code and shown in Figure 4.

```
> library(lattice)

> xyplot(mpg ~ disp, mtcars)
```

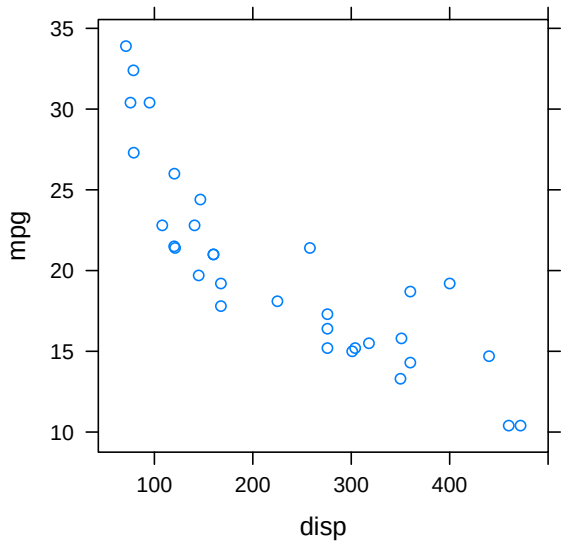

Figure 4: A simple **lattice** scatterplot.

This plot is simpler than the forest plot in Figure 3, but it still contains numerous individual shapes. Anyone familiar with the **lattice** package will also know that it can produce plots of much greater complexity; in general, the **lattice** package faces a very difficult problem if it wants to provide an argument in its high-level functions to control every single shape within any of its plots.

However, the **lattice** package also provides names for everything that it draws. The following code shows the contents of the **grid** display list after drawing the plot in Figure 4.

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

rect[plot_01.background]
text[plot_01.xlab]
text[plot_01.ylab]
segments[plot_01.ticks.top.panel.1.1]
segments[plot_01.ticks.left.panel.1.1]
text[plot_01.ticklabels.left.panel.1.1]
segments[plot_01.ticks.bottom.panel.1.1]
text[plot_01.ticklabels.bottom.panel.1.1]
segments[plot_01.ticks.right.panel.1.1]
points[plot_01.xyplot.points.panel.1.1]
rect[plot_01.border.panel.1.1]
```

Because everything is named, it is possible to access any component of the plot using the low-level **grid** functions. For example, the following code modifies the x-axis label of the plot (see Figure 5).

```
> grid.edit("plot_01.xlab",
+           label="Displacement",
+           gp=gpar(fontface="bold.italic"))
```

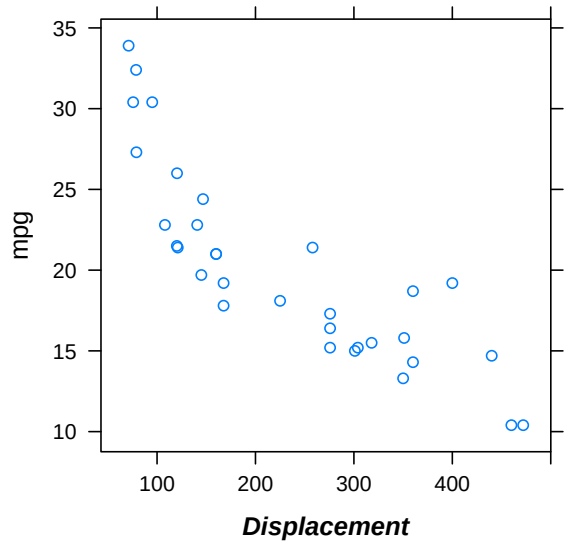

Figure 5: The **lattice** plot from Figure 4 with the x-axis modified using low-level **grid** functions.

That particular modification of a **lattice** plot could easily be achieved using arguments to the high-level `xyplot()` function, but the direct access to low-level shapes allows for a much wider range of modifications.

For example, the following code generates a more complex multipanel **lattice** barchart.

```
> barchart(yield ~ variety | site, data = barley,
+          groups = year, layout = c(1,6),
+          stack = TRUE,
+          ylab = "Barley Yield (bushels/acre)",
+          scales = list(x = list(rot = 45)))
```

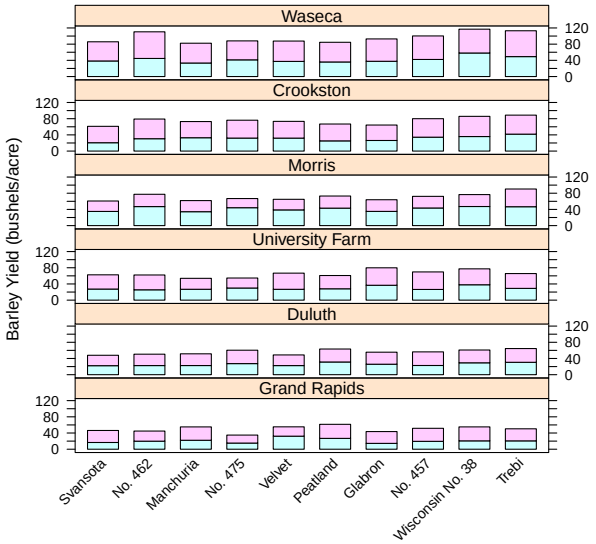

Figure 6: A complex multipanel **lattice** barchart.

There are too many individual shapes in this plot to show the full display list here, but all of the shapes have names and the following code makes use of those names to perform a more sophisticated plot modification: highlighting the sixth set of bars in each panel of the barchart (see Figure 7).

```
> grid.edit("barchart.pos.6.rect",
+           grep=TRUE, global=TRUE,
+           gp=gpar(lwd=3))
```

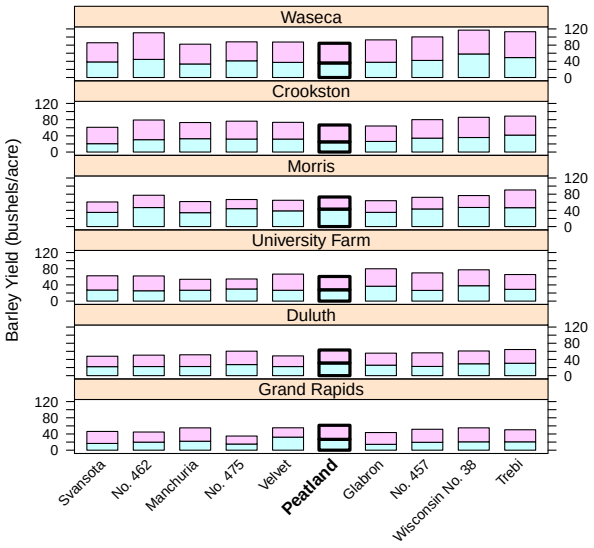

Figure 7: The barchart from Figure 6 with the sixth set of bars in each panel highlighted.

It would not be reasonable to expect the high-level `barchart()` function to provide an argument that allows for this sort of customisation, but, because **lattice** has named everything that it draws, `barchart()` does not need to cater for every possible customisation. Low-level access to individual shapes can be used instead bceause individual shapes can be identified by name.

Post-processing graphics

This section discusses how naming the individual shapes within a plot allows not just minor customisations, but general transformations to be applied to a plot.

The R graphics system has always encouraged the philosophy that a high-level plotting function is only a starting point. Low-level functions have always been provided so that a plot can be customised by *adding* some *new drawing* to the plot.

The previous section demonstrated that, if every shape within a plot has a label, it is also possible to customise a plot by *modifying* the *existing shapes* within a plot.

However, we can go even further than just modifying the existing parameters of a shape. In theory, we can think of the existing shapes within a picture as a basis for more general post-processing of the image.

As an example, one thing that we can do is to query the existing components of a plot to determine the position or size of an existing component. This means that we can position or size new drawing in relation to the existing plot. The following code uses this idea to add a rectangle around the x-axis label of the plot in Figure 4 (see Figure 8). The `grobWidth()` function is used to calculate the width of the rectangle from the width of the x-axis label. The `downViewport()` function is used to make sure that we draw the rectangle in the right area on the page.[1]

```
> xyplot(mpg ~ disp, mtcars)

> rectWidth <- grobWidth("plot_01.xlab")

> downViewport("plot_01.xlab.vp")
> grid.rect(width=rectWidth + unit(2, "mm"),
+           height=unit(1, "lines"),
+           gp=gpar(lwd=2),
+           name="xlabRect")
```

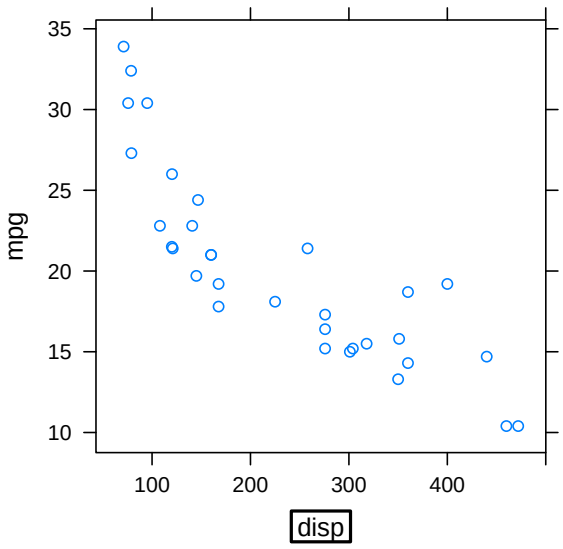

Figure 8: The **lattice** plot from Figure 4 with a rectangle added around the x-axis label.

The display list now contains an new rectangle object, as shown below.

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

rect[plot_01.background]
text[plot_01.xlab]
text[plot_01.ylab]
segments[plot_01.ticks.top.panel.1.1]
segments[plot_01.ticks.left.panel.1.1]
text[plot_01.ticklabels.left.panel.1.1]
segments[plot_01.ticks.bottom.panel.1.1]
text[plot_01.ticklabels.bottom.panel.1.1]
segments[plot_01.ticks.right.panel.1.1]
points[plot_01.xyplot.points.panel.1.1]
rect[plot_01.border.panel.1.1]
rect[xlabRect]
```

Importantly, the new object depends on the size of the existing x-axis label object within the scene. For example, if we edit the x-axis label again, as below, the rectangle will grow to accommodate the new label (see Figure 9).

```
> grid.edit("plot_01.xlab",
+           label="Displacement",
+           gp=gpar(fontface="bold.italic"))
```

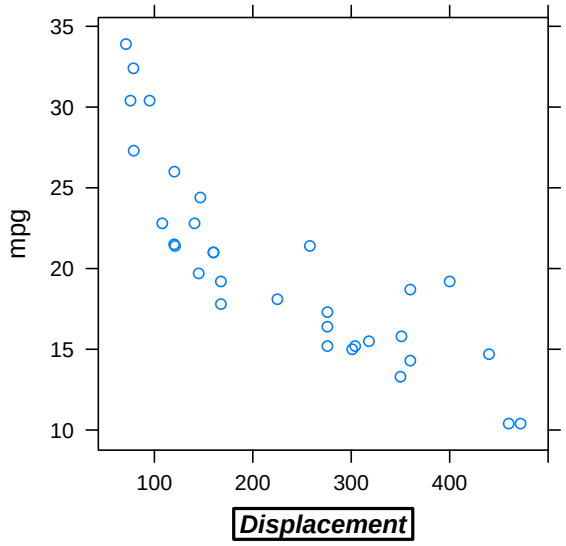

Figure 9: The **lattice** plot from Figure 4 with a rectangle added around the *modified* x-axis label.

A more extreme example of post-processing is demonstrated in the code below. In this case, we again query the existing x-axis label to determine its width, but this time, rather than adding a rectangle, we *replace* the label with a rectangle (in effect, we "redact" the x-axis label; see Figure 10).

```
> xyplot(mpg ~ disp, mtcars)

> xaxisLabel <- grid.get("plot_01.xlab")
> grid.set("plot_01.xlab",
+          rectGrob(width=grobWidth(xaxisLabel) +
+                           unit(2, "mm"),
+                   height=unit(1, "lines"),
+                   gp=gpar(fill="black"),
+                   name="plot_01.xlab"))
```

[1] This `downViewport()` works because the **grid** viewports that **lattice** creates to draw its plots all have names too!

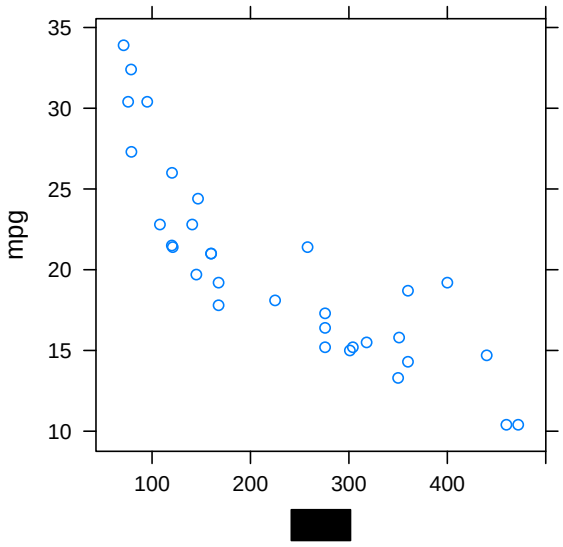

Figure 10: The **lattice** plot from Figure 4 with the x-axis label redacted (replaced with a black rectangle).

The display list now consists of the same number of objects as in the original plot, but now the object named `"plot_01.xlab"` is a *rectangle* instead of text (see below).

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

rect[plot_01.background]
rect[plot_01.xlab]
text[plot_01.ylab]
segments[plot_01.ticks.top.panel.1.1]
segments[plot_01.ticks.left.panel.1.1]
text[plot_01.ticklabels.left.panel.1.1]
segments[plot_01.ticks.bottom.panel.1.1]
text[plot_01.ticklabels.bottom.panel.1.1]
segments[plot_01.ticks.right.panel.1.1]
points[plot_01.xyplot.points.panel.1.1]
rect[plot_01.border.panel.1.1]
```

The artificial examples shown in this section so far have been deliberately simple in an attempt to make the basic concepts clear, but the ideas can be applied on a much larger scale and to greater effect. For example, the **gridSVG** package (Murrell, 2011) uses these techniques to transform static R plots into dynamic and interactive plots for use in web pages. It has functions that modify existing objects on the **grid** display list to add extra information, like hyperlinks and animation, and it has functions that transform each object on the **grid** display list to SVG code. The following code shows a simple demonstration where the original **lattice** plot is converted to an SVG document with a hyperlink on the x-axis label. Figure 11 shows the SVG document in a web browser.

```
> xyplot(mpg ~ disp, mtcars)

> library(gridSVG)

> url <- "http://www.mortality.org/INdb/2008/02/12/8/document.pdf"

> grid.hyperlink("plot_01.xlab", href=url)
> gridToSVG("xyplot.svg")
```

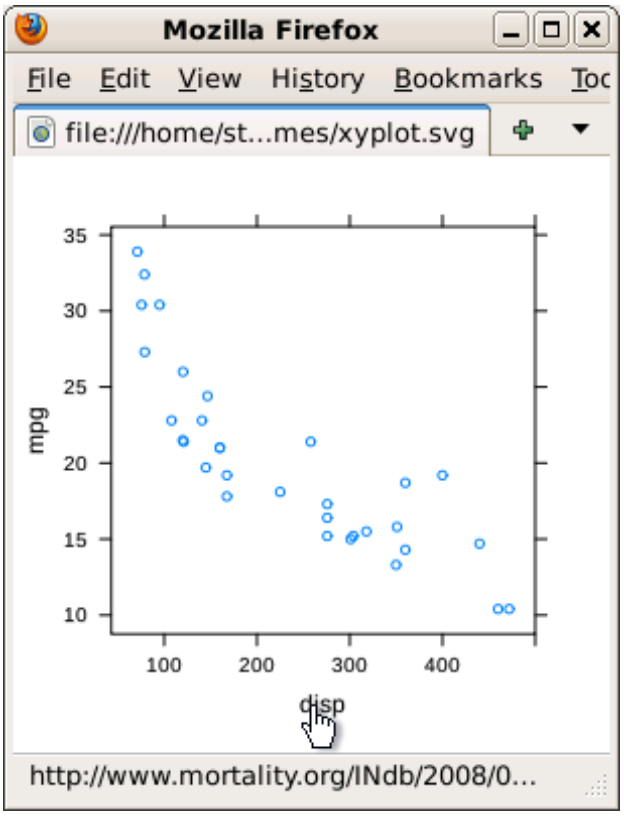

Figure 11: The **lattice** plot from Figure 4 transformed into an SVG document with a hyperlink on the x-axis label.

Naming schemes

The basic message of this article is straightforward: name everything that you draw with **grid**. However, deciding what names to use—deciding on a *naming scheme*—is not necessarily so easy.

The approach taken in the **lattice** package is to attempt to reflect the structure of the plot in the naming scheme. For example, everything that is drawn within a panel region has the word `"panel"` in its name, along with a suffix of the form *i.j* to identify the panel row and column.

The decision may be made a lot easier if a plot is drawn from gTrees rather than simple grobs, because the gTrees reflect the plot structure already and names for individual components can be chosen to reflect just the "local" role of each plot component. The naming scheme in the **ggplot2** package (Wickham, 2009) is an example of this approach.

In addition to the code developer deciding on a naming scheme, the code user also faces the problem of how to "discover" the names of the components of a plot.

From the developer side, there is a responsibility to document the naming scheme (for example, the **lattice** naming scheme is described on the packages's R-Forge web site[2]). It may also be possible to provide a function interface to assist in constructing the names of objects (for example, the `trellis.grobname()` function in **lattice**).

From the user side, there are tools that help to display the names of objects in the current scene. This article has demonstrated the `grid.ls()` function, but there is also a `showGrob()` function, and the **gridDebug** package (Murrell and Ly., 2011) provides some more tools.

Discussion

In summary, if we specify an explicit name for every shape that we draw using **grid**, we allow low-level access to every object within a scene. This allows us to make very detailed customisations to the scene, without the need for long lists of arguments in high-level plotting functions, and it allows us to query and transform the scene in a wide variety of ways.

An alternative way to provide access to individual shapes within a plot is to allow the user to simply select shapes on screen via a mouse. How does this compare to a naming scheme?

Selection using a mouse works well for some sorts of modifications (see, for example, the **playwith** package; Andrews, 2010), but providing access to individual shapes by name is more efficient, more general, and more powerful. For example, if we write code to make modifications, referencing objects by name, we have a record of what we have done, we can easily automate large numbers of modifications, we can share our modification techniques, and we can express more complex modifications (like "highlight every sixth bar").

Another alternative way to provide detailed control over a scene is simply to modify the original R code that drew the scene. Why go to the bother of naming objects when we can just modify the original R code?

If we have written the original code, then modifying the original code may be the right approach. However, if we draw a plot using someone else's code (for example, if we call a **lattice** function), we do not have easy access to the code that did the drawing. Even though it is possible to see the code that did the drawing, understanding it and then modifying it may require a considerable effort, especially when that code is of the size and complexity of the code in the **lattice** package.

A parallel may be drawn between this idea of naming every shape within a scene and the general idea of *markup*. In a sense, what we are aiming to do is to provide a useful label for each meaningful component of a scene. Given tools that can select parts of the scene based on the labels, the scene becomes a "source" that can be transformed in many different ways. When we draw a scene in this way, it is not just an end point that satisfies our own goals. It also creates a resource that others can make use of to produce new resources. When we write code to draw a scene, we are not only concerned with producing an image on screen or ink on a page; we also allow for other possible uses of the scene in ways that we may not have anticipated.

Acknowledgements

Thanks to Wolfgang Viechtbauer for useful comments on an early draft of this article.

Bibliography

F. Andrews. **playwith**: *A GUI for interactive plots using GTK+*, 2010. URL http://CRAN.R-project.org/package=playwith. R package version 0.9-53.

P. Murrell. **gridSVG**: *Export grid graphics as SVG*, 2011. R package version 0.7-0.

P. Murrell and V. Ly. **gridDebug**: *Debugging Grid Graphics*, 2011. R package version 0.2.

R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2011. URL http://www.R-project.org/. ISBN 3-900051-07-0.

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

W. Viechtbauer. Conducting meta-analyses in R with the metafor package. *Journal of Statistical Software*, 36(3):1–48, 2010. URL http://www.jstatsoft.org/v36/i03/.

H. Wickham. *ggplot2: elegant graphics for data analysis*. Springer New York, 2009. ISBN 978-0-387-98140-6. URL http://had.co.nz/ggplot2/book.

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

[2] http://lattice.r-forge.r-project.org/documentation.php