Debugging grid Graphics

Paul Murrell and Velvet Ly

Abstract A graphical scene that has been produced using the **grid** graphics package consists of grobs (graphical objects) and viewports. This article describes functions that allow the exploration and inspection of the grobs and viewports in a **grid** scene. This is useful for adding more drawing to a scene that was produced using **grid** and for understanding and debugging the **grid** code that produced a scene.

Introduction

The **grid** graphics package for R contains features that are intended to assist in the creation of flexible, complex graphical scenes, such as the plots that are produced by **lattice** and **ggplot2**.

Two particularly important features are *viewports*, which represent rectangular regions on the page for drawing into, and *grobs*, which represent shapes that have been drawn onto the page.

To illustrate these **grid** concepts, the following code draws a simple scene consisting of a narrow "strip" region atop a larger "panel" region, with a rectangle boundary drawn for each region and the top region shaded grey (see Figure 1).

```
> library(grid)

> stripVP <- viewport(y=1,
+                     height=unit(1, "lines"),
+                     just="top",
+                     name="stripvp")
> panelVP <- viewport(y=0,
+                     height=unit(1, "npc") -
+                            unit(1, "lines"),
+                     just="bottom",
+                     name="panelvp")

> pushViewport(stripVP)
> grid.rect(gp=gpar(fill="grey80"),
+           name="striprect")
> upViewport()
> pushViewport(panelVP)
> grid.rect(name="panelrect")
> upViewport()
```

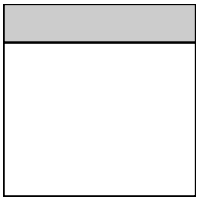

Figure 1: A scene consisting of two viewports, with a rectangle drawn in each.

One benefit that accrues from using viewports to draw the scene in Figure 1 is that, once the scene has been drawn, the viewports can be revisited to add further drawing to the scene. For example, the following code revisits the "strip" region and adds a text label (see Figure 2).

```
> downViewport("stripvp")
> grid.text("strip text", name="striptext")
> upViewport()
```

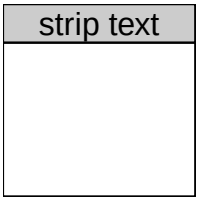

Figure 2: The scene from Figure 1 with text added to the top viewport.

One benefit that accrues from the fact that **grid** creates grobs representing the shapes in a scene is that, after the scene has been drawn, it is possible to modify elements of the scene. For example, the following code modifies the text that was just drawn in the strip region so that it is dark green, italic, and in a serif font (see Figure 3).

```
> grid.edit("striptext",
+           label="modified text",
+           gp=gpar(col="darkgreen",
+             fontface="italic",
+             fontfamily="serif"))
```

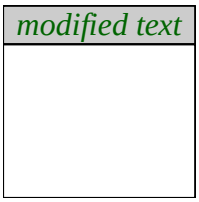

Figure 3: The scene from Figure 2 with the text modified to be dark green, italic, and serif.

The following code shows that it is also possible to remove objects from a scene — this returns the scene to its original state (Figure 1) by removing the text that we had added above.

```
> grid.remove("striptext")
```

The importance of names

The ability to navigate within viewports in a scene and the ability to modify grobs within a scene both depend upon being able to unambiguously specify a particular viewport or grob.

All viewports and grobs have a *name*, so specifying a particular viewport or grob is simply a matter of specifying the relevant viewport name or grob name.

In the simple example above, this is not a difficult task because we have the code that created the scene so we can see the names that were used. However, when a scene has been generated by someone else's code, for example, a call to a **lattice** plotting function, it may not be very easy to determine the name of a viewport or grob.[1]

Pity the developer

Problems can also arise when we want to develop new functions that draw scenes using **grid**. In this case, knowing the names of viewports and grobs is not the problem because we have created the names. Instead, the problem is knowing where on the page the viewports and grobs have ended up. The result of running error-ridden **grid** code can be a confusing jumble of drawing output. In this case, it is useful to be able to identify where on the page a particular viewport or grob has been drawn.

Pity the student

Even when the author of a piece of **grid** code knows exactly what the code is doing, and the code is behaving correctly, it can be difficult to convey to other people the relationship between the **grid** code and the output that it produces on a page. This is another situation where it can be useful to provide a visual cue about the location on the page of abstract concepts such as viewports and grobs and the relationships between them.

This article describes a number of functions that are provided by the grid package to help identify what viewports and grobs have been used to create a scene and track exactly where each viewport and grob has been drawn on the page.

The `grid.ls()` function

A simple listing of the names of all grobs in a scene can be produced using the `grid.ls()` function. For example, the following code lists the grobs in Figure 1, which consists of just two rectangles called `"striprect"` and `"panelrect"`

```
> grid.ls()

striprect
panelrect
```

The `grid.ls()` function can also be used to list viewports in the current scene, via the `viewports` argument and the `fullNames` argument can be specified to print further information in the listing so that it is easier to distinguish viewports from grobs. The following code produces a more complete listing of the scene from Figure 1 with both viewports and grobs listed. Notice that the names are indented to reflect the fact that some viewports are nested within others and also to reflect the fact that the grobs are drawn within different viewports.

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

viewport[ROOT]
  viewport[stripvp]
    rect[striprect]
    upViewport[1]
  viewport[panelvp]
    rect[panelrect]
    upViewport[1]
```

This function is useful for at least viewing the names of all grobs and viewports in a scene and it gives some indication of the structure of the scene. Even for a complex scene, such as a **lattice** multipanel conditioning plot it is possible, if a little tedious, to identify important components of the scene.

The `showGrob()` function

The `showGrob()` function displays the names of the grobs in a scene by labelling them on the current scene. By default, a semitransparent rectangle is drawn to show the extent of each grob and the name of the grob is drawn within that rectangle. For example, the following code labels the grobs in the simple scene from Figure 1. The resulting labelled scene is shown in Figure 4 — there are two rectangles called `"striprect"` and `"panelrect"`.

```
> showGrob()
```

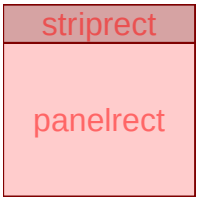

Figure 4: The scene from Figure 1 with labelling added by the `showGrob()` function to show the locations and names of the grobs used to draw the scene.

In more complex scenes, it is common for several grobs to overlap each other so that this sort of labelling becomes very messy. Later sections

[1] The **lattice** package does provide some assistance in the form of the `trellis.vpname()` and `trellis.grobname()` functions.

will demonstrate how to cope with that complexity using other functions and other arguments to the `showGrob()` function.

The `showViewport()` function

The `showViewport()` function performs a similar task to `showGrob()` except that it labels the viewports in a scene. Again, the labelling consists of a semitransparent rectangle and the name of the viewport. For example, the following code labels the viewports in the scene from Figure 1, which has a narrow viewport called `"stripvp"` on top and a larger viewport called `"panelvp"` below.

```
> showViewport()
```

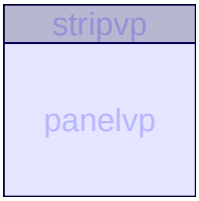

Figure 5: The scene from Figure 1 with labelling added by the `showViewport()` function to show the locations and names of the viewports that were used to draw the scene.

In more complex scenes, it is common for viewports to overlap each other, so the default output from `showViewport()` is less legible. Later sections will describe solutions to this problem using further arguments to `showViewport()` as well as different debugging functions.

The gridDebug package

The **gridDebug** package provides some additional tools for debugging **grid** output.

The `gridTree()` function draws a *scene graph* from a **grid** scene, using the **graph** and **Rgraphviz** packages. This is a node-and-edge graph that contains a node for each grob and each viewport in the current **grid** scene. The graph has an edge from each child viewport to its parent viewport and an edge from each grob to the viewport within which the grob is drawn. The nodes are labelled tihe the name of the corresponding grobs and viewports. For example, the following code produces a scene graph for the simple scene in Figure 1. The scene graph is shown in Figure 6.

```
> library(gridDebug)
```

```
> gridTree()
```

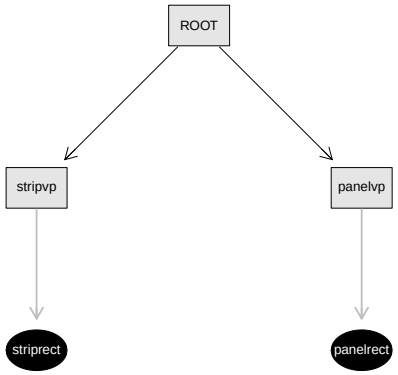

Figure 6: A node-and-edge graph of the scene from Figure 1. Both viewports are direct descendants of the `ROOT` viewport and one grob is drawn in each viewport.

This graph shows that the two viewports have both been pushed directly beneath the `ROOT` viewport (they are siblings) and that each grob has been drawn in a separate viewport.

One advantage of this function is that it is unaffected by overlapping grobs or viewports. The main downside is that node labels become very small as the scene becomes more complex.

More complex scenes

We will now consider a more complex scene and look at how the various debugging functions can be adapted to cope with the additional complexity. As an example, we will look at a plot produced by the `histogram()` function from the **lattice** package (see Figure 7).

```
> library(lattice)
```

```
> histogram(faithful$eruptions)
```

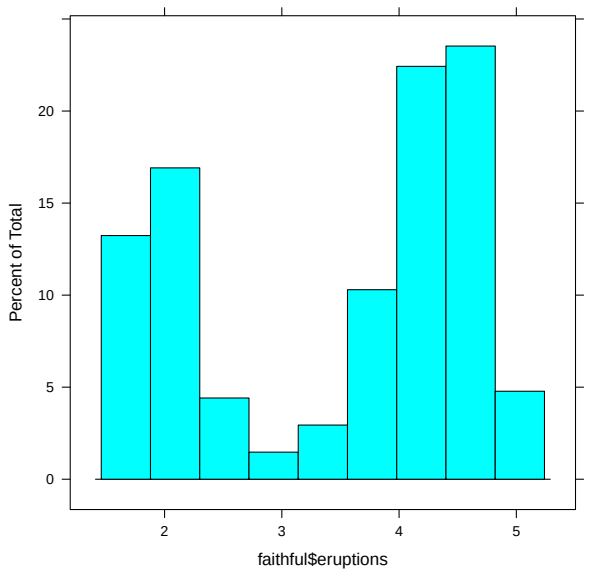

Figure 7: A more complex **grid** scene consisting of a simple plot produced by the `histogram()` function from the **lattice** package.

The `grid.ls()` function

For more complex scenes, the number of viewports and grobs can make it difficult to consume the listing from `grid.ls()` and, as viewports and grobs become nested to greater depths, simple indenting can be insufficient to convey the nesting clearly.

One solution is to specify a different formatting function via the `print` argument to the `grid.ls()` function. For example, the following code lists all grobs and viewports from Figure 7, but with only one line for each grob. The nesting of viewports is shown by listing the full viewport path to each grob. Figure 8 shows the resulting output.

```
> grid.ls(viewports=TRUE, print=grobPathListing)
```

Another solution is to capture (rather than just print) the result from `grid.ls()`. This is a list object containing a lot of information about the current scene and it can be processed computationally to answer more complex questions about the scene (see Figure 9).

```
> sceneListing <- grid.ls(viewports=TRUE,
+                         print=FALSE)

> do.call("cbind", sceneListing)
```

The `showGrob()` function

In a more complex scene, it is common for grobs to overlap each other, which can result in a messy labelling from the `showGrob()` function. Another problem is that text grobs do not label well because the labelling text is hard to read when overlaid on the text that is being labelled. One possible solution is to vary the graphical parameters used in the labelling. For example, the following code sets the fill colour for the grob bounding rectangles to be opaque (see Figure 10).

```
> showGrob(gp=gpar(fill=rgb(1, .85, .85)))
```

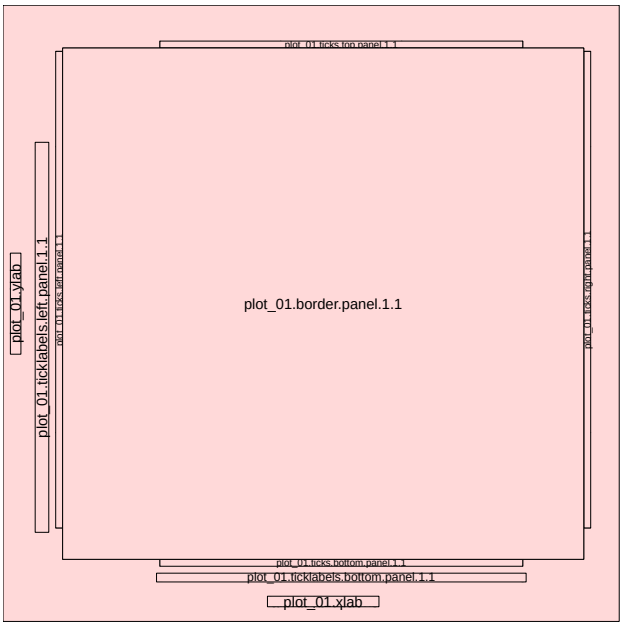

Figure 10: The **lattice** plot from Figure 7 with labelling added by the `showGrob()` function to show the locations and names of the grobs that were used to draw the scene.

One problem with this solution is that some overlapping grobs are not visible at all. To solve this, the `gPath` argument can be used to specify a particular grob to label. The following code uses this approach to label just the rectangle grob called `"plot_01.histogram.rect.panel.1.1"` (the rectangle grob that draws the histogram bars; see Figure 11).

```
> showGrob(gPath="plot_01.histogram.rect.panel.1.1")
```

```
ROOT                                                    | plot_01.background
ROOT::plot_01.toplevel.vp::plot_01.xlab.vp              | plot_01.xlab
ROOT::plot_01.toplevel.vp::plot_01.ylab.vp              | plot_01.ylab
ROOT::plot_01.toplevel.vp::plot_01.strip.1.1.off.vp     | plot_01.ticks.top.panel.1.1
ROOT::plot_01.toplevel.vp::plot_01.strip.left.1.1.off.vp | plot_01.ticks.left.panel.1.1
ROOT::plot_01.toplevel.vp::plot_01.strip.left.1.1.off.vp | plot_01.ticklabels.left.panel.1.1
ROOT::plot_01.toplevel.vp::plot_01.panel.1.1.off.vp     | plot_01.ticks.bottom.panel.1.1
ROOT::plot_01.toplevel.vp::plot_01.panel.1.1.off.vp     | plot_01.ticklabels.bottom.panel.1.1
ROOT::plot_01.toplevel.vp::plot_01.panel.1.1.off.vp     | plot_01.ticks.right.panel.1.1
ROOT::plot_01.toplevel.vp::plot_01.panel.1.1.vp         | plot_01.histogram.baseline.lines.panel.1.1
ROOT::plot_01.toplevel.vp::plot_01.panel.1.1.vp         | plot_01.histogram.rect.panel.1.1
ROOT::plot_01.toplevel.vp::plot_01.panel.1.1.off.vp     | plot_01.border.panel.1.1
```

Figure 8: A listing of the grobs and viewports from Figure 7 produced by `grid.ls()`.

```
                name gDepth vpDepth gPath                                    vpPath        type
1                ROOT      0       0                                                  vpListing
2   plot_01.background      0       1                                      ROOT grobListing
3  plot_01.toplevel.vp      0       1                                      ROOT   vpListing
4      plot_01.xlab.vp      0       2                    ROOT::plot_01.toplevel.vp   vpListing
5         plot_01.xlab      0       3 ROOT::plot_01.toplevel.vp::plot_01.xlab.vp grobListing
6                    1      0       3 ROOT::plot_01.toplevel.vp::plot_01.xlab.vp vpUpListing
7      plot_01.ylab.vp      0       2                    ROOT::plot_01.toplevel.vp   vpListing
8         plot_01.ylab      0       3 ROOT::plot_01.toplevel.vp::plot_01.ylab.vp grobListing
9                    1      0       3 ROOT::plot_01.toplevel.vp::plot_01.ylab.vp vpUpListing
10   plot_01.figure.vp      0       2                    ROOT::plot_01.toplevel.vp   vpListing
```

Figure 9: The raw result that is returned by a `grid.ls()` call for the scene in Figure 7. Only the first 10 lines of information is shown.

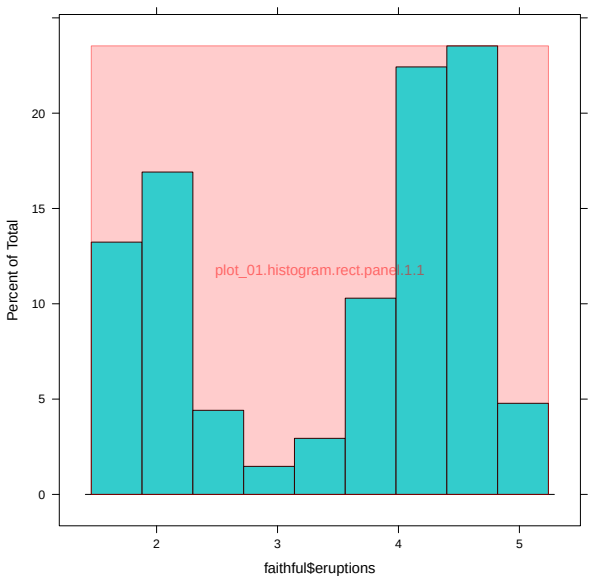

Figure 11: The **lattice** plot from Figure 7 with labelling added by the `showGrob()` function to show the location of grob `"plot_01.histogram.rect.panel.1.1"`.

The `showViewport()` function

In complex scenes, it is also very common for viewports to overlap each other. It is possible to display just a specific viewport with `showViewport()`, by supplying a viewport path as the first argument, but another option is to draw all viewports separately via the `leaves` argument. The following code demonstrates this approach and the result is shown in Figure 12.

```
> showViewport(newpage=TRUE, leaves=TRUE,
+              col="black")
```

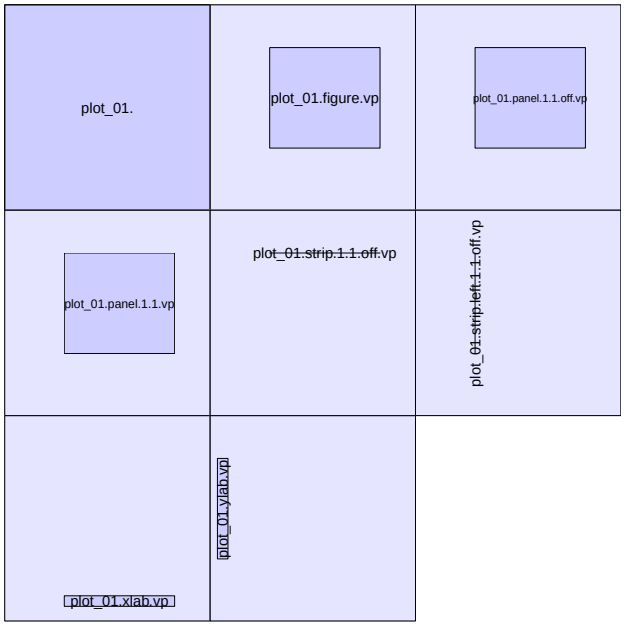

Figure 12: The result of calling `showViewport()` to display the viewports used to draw the scene in Figure 7, with each viewport displayed on its own in a separate "mini page" to overcome the fact that several viewports overlap each other.

The `gridTree()` function

One advantage of the `gridTree()` function is that it is immune to the overlap of grobs and viewports in a scene. This is because this sort of display emphasizes the conceptual structure of the scene rather than reflecting the location of grobs and viewports on the page.

The following code produces a scene graph for the **lattice** plot from Figure 7 and the result is shown in Figure 13.

```
> gridTree()
```

One problem that does arise with the `gridTree()` function is that the grob and viewport names, which are used to label the nodes of the scene graph, can become too small to read.

The following code demonstrates this problem with an example plot from the **ggplot2** package. The plot is shown in Figure 14 and the scene graph generated by `gridTree()` is shown in Figure 15.

```
> library(ggplot2)

> qplot(faithful$eruptions, binwidth=.5)
```

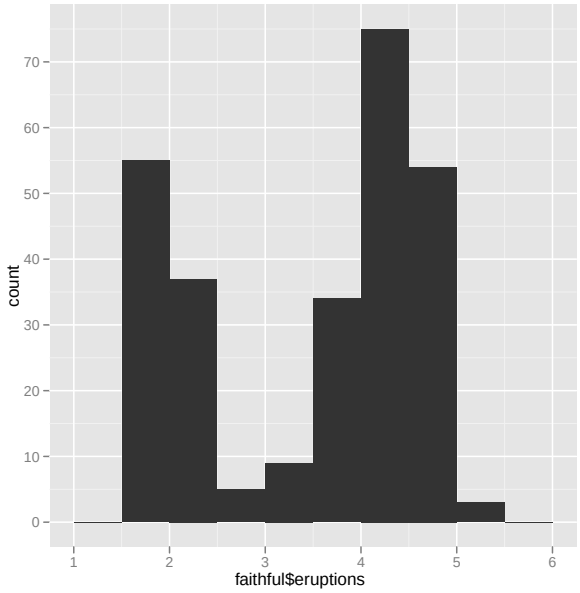

Figure 14: A more complex **grid** scene consisting of a simple plot produced by the `qplot()` function from the **ggplot2** package.

Although it is impossible to read the names of individual grobs and viewports on this graph, it is still interesting to compare the structure of this scene with the graph from the **lattice** plot in Figure 13. The graph clearly shows that the **lattice** package uses two levels of viewports, but only simple grobs, while the **ggplot2** package has a single, relatively complex, gTree that contains numerous other grobs, gTrees and viewports.

Interactive tools

The problem of unreadable labels on a scene graph may be alleviated by using the `gridTreeTips()` function, from the **gridDebug** package. This makes use of the **gridSVG** package to produce an SVG version of the scene graph with simple interaction added so that, when the mouse hovers over a node in the scene graph, a tooltip pops up to show the name of the node. Figure 16 shows an example of the output from this function (as viewed in Firefox).

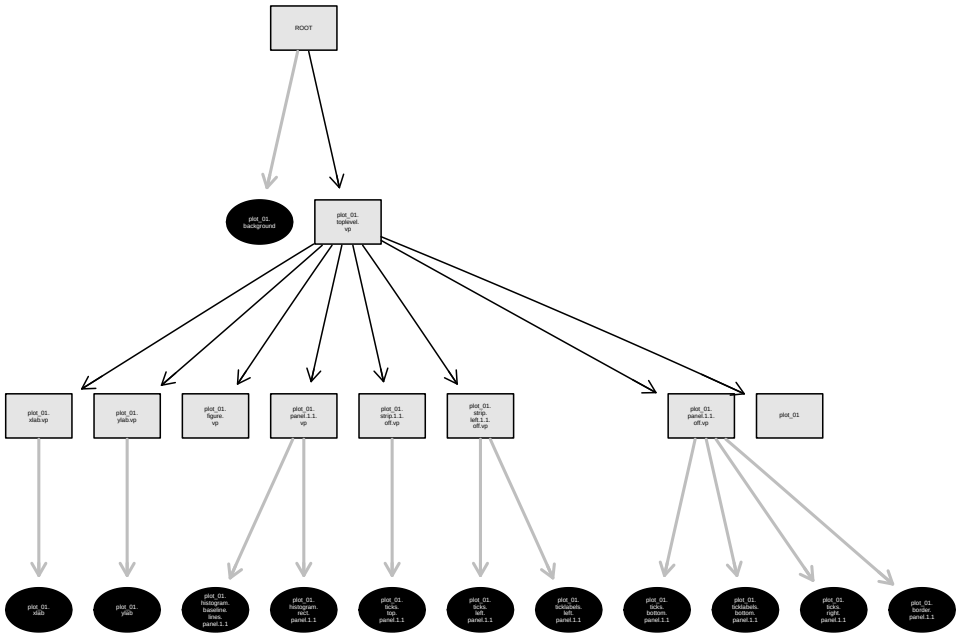

Figure 13: A node-and-edge graph of the scene from Figure 7

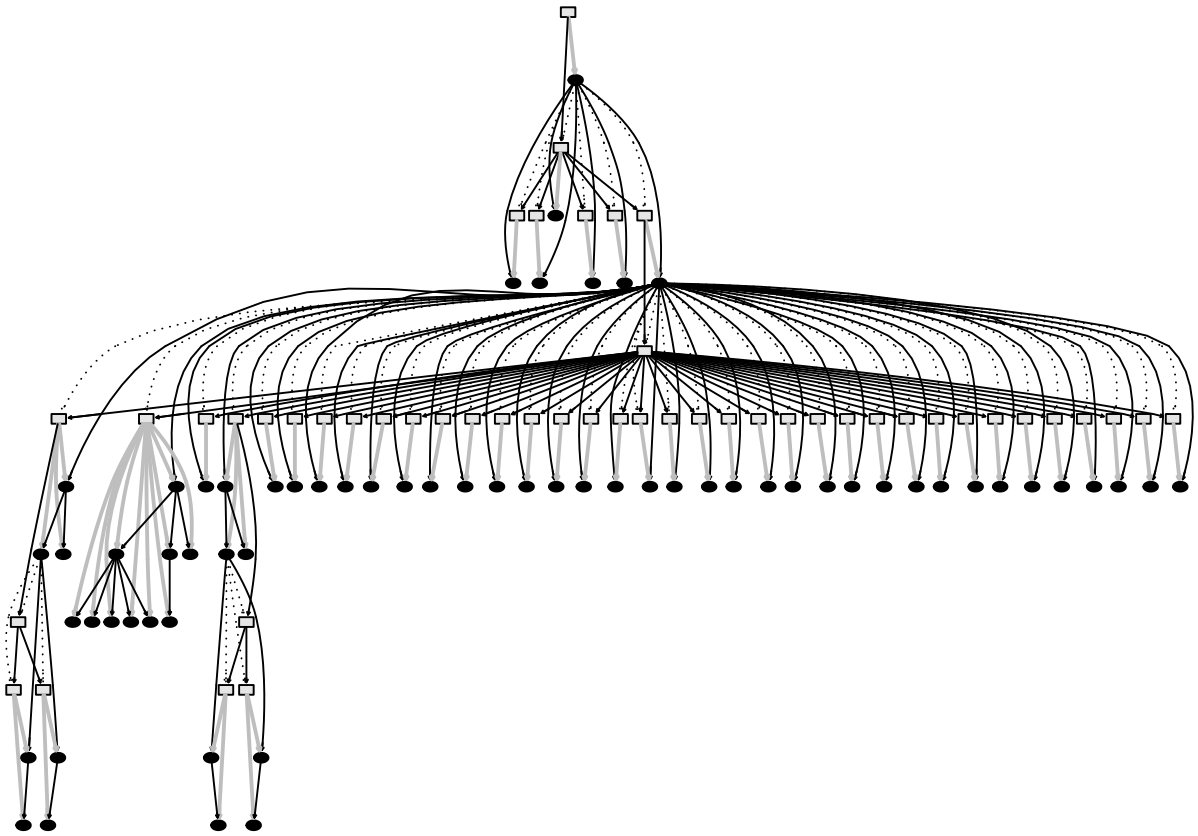

Figure 15: A node-and-edge graph of the scene from Figure 14

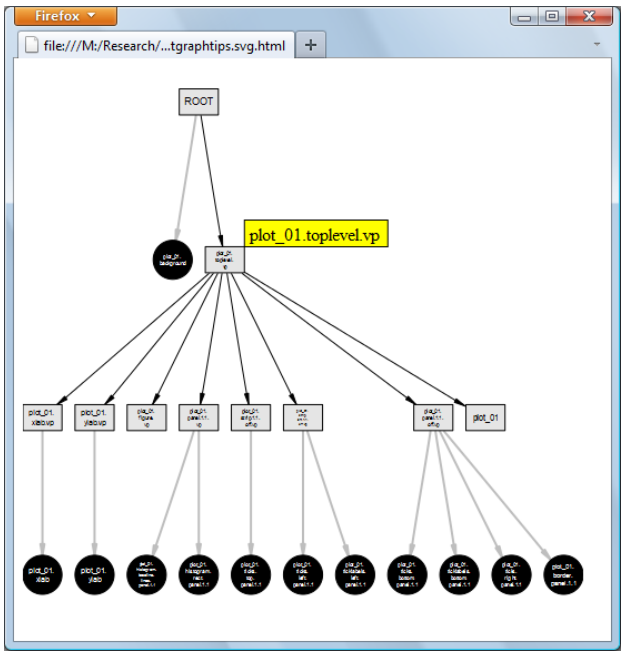

Figure 16: A node-and-edge graph of the scene from Figure 14 in SVG format so that, when the mouse hovers over a node on the graph, a tooltip shows the name of the node. The mouse is hovering over the node for the viewport called `"plot_01.toplevel.vp"`.

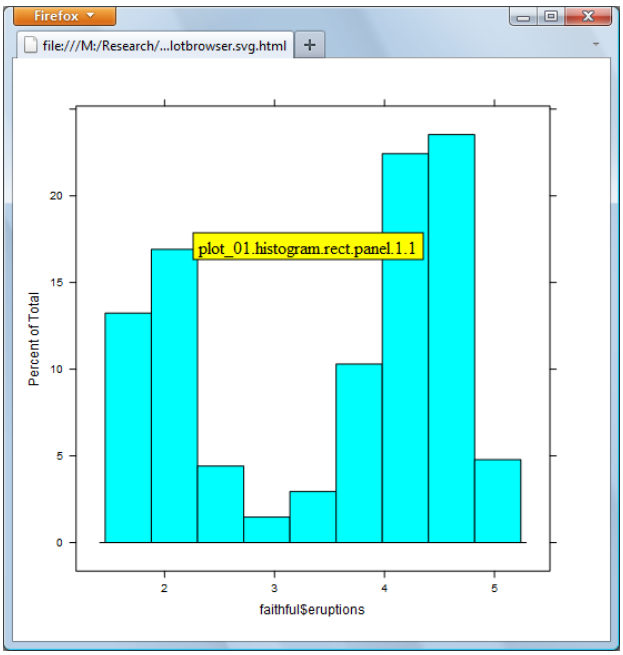

Figure 17: The scene from Figure 14 in SVG format so that, when the mouse hovers over a grob in the scene, a tooltip shows the name of the grob. The mouse is hovering over one of the bars in the histogram, which corresponds to the grob called `"plot_01.histogram.rect.panel.1.1"`.

Another function from the **gridDebug** package, which also makes use of **gridSVG**, is the `grobBrowser()` function. This takes any **grid** scene and produces an SVG version of the scene that also contains tooltips. In this case, whenever the mouse hovers over a grob in the scene, a tooltip pops up to show the name of the grob. Figure 17 shows an example of the output from this function (as viewed in Firefox).

Tools in other packages

The **playwith** package also provides some tools for exploring the grobs in a **grid** scene. The `showGrobsBB()` function produces a similar result to `showGrob()` and `identifyGrob()` allows the user to click within a normal R graphics device to identify grobs. If the click occurs within the bounding box of a grob then the name of that grob is returned as the result. The result may be several grob names if there are overlapping grobs.

Conclusions

This article has described several tools that assist with the debugging of **grid** graphics code, whether that is trying to understand someone else's code, trying to understand your own code, or trying to explain **grid** code to someone else.

The tools provide various ways to view the names of grobs and viewports that were used to draw a scene, the relationships between the grobs and viewports, and where those grobs and viewports end up when drawn on the page.

Each of the tools has various weaknesses, so it may be necessary to use them in combination with each other in order to gain a complete understanding of a complex scene.

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.

R. Gentleman, E. Whalen, W. Huber, and S. Falcon. **graph**: *A package to handle graph data structures*, 2010. URL http://CRAN.R-project.org/package=graph. R package version 1.28.0.

J. Gentry, L. Long, R. Gentleman, S. Falcon, F. Hahne, D. Sarkar, and K. Hansen. **Rgraphviz**: *Provides plotting capabilities for R graph objects*, 2010. R package version 1.23.6.

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.

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

Velevt Ly
Department of Statistics
The University of Auckland
New Zealand
kly004@aucklanduni.ac.nz