Display Lists in **grid**

Paul Murrell

October 19, 2015

A display list is a record of drawing operations. It is used to redraw graphics output when a graphics window is resized, when graphics output is copied from one device to another, and when graphics output is edited (via `grid.edit`).

There are two display lists that can be used when working with **grid**. R's graphics engine maintains a display list and **grid** maintains its own display list. The former is maintained at the C code level and records both base graphics output and **grid** graphics output. The latter is maintained at the R code level and only records **grid** output.

In standard usage, the graphics engine's display list is used to redraw when a window is resized and when copying between devices. **grid**'s display list is used for redrawing when editing **grid** output.

There are two main problems with this standard usage:

1. The graphics engine display list only records graphics output; none of the calculations leading up to producing the output are recorded. This particularly impacts on plots which perform calculations based on the physical dimensions of the device – an example is the legend function which performs calculations in order to arrange the elements of the legend. The effect can be seen from any example which uses the legend function. Try running `example(legend)` then resize the device (make it quite tall and thin or quite wide and fat); the legend will start to look pretty sick.

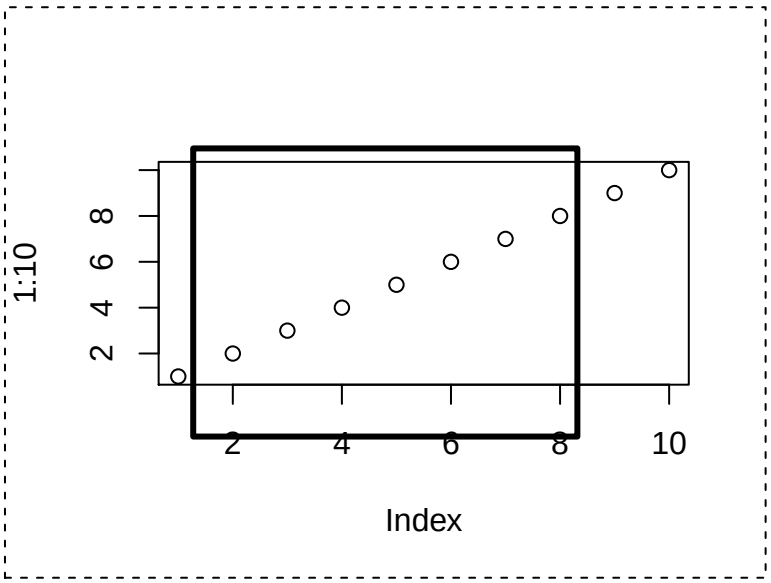
   NOTE: that this is a problem with the graphics engine display list – it is not specific to **grid**. In fact, much of **grid**'s behaviour is protected from this problem because things like **grid** units are "declarative" and will be re-evaluated on each redraw. However, there are situations where **grid** output can be afflicted, in particular, whenever the `convertUnit()` function (or one of its variants) is used (the help file for `convertUnit()` gives an example).

   A situation where this problem becomes very relevant for **grid** output is when the **gridBase** package is used. This is a situation where lots of calculations are performed in order to align base and grid output, but these calculations are not recorded on the graphics engine display list, so if the device is resized the output will become very yukky.

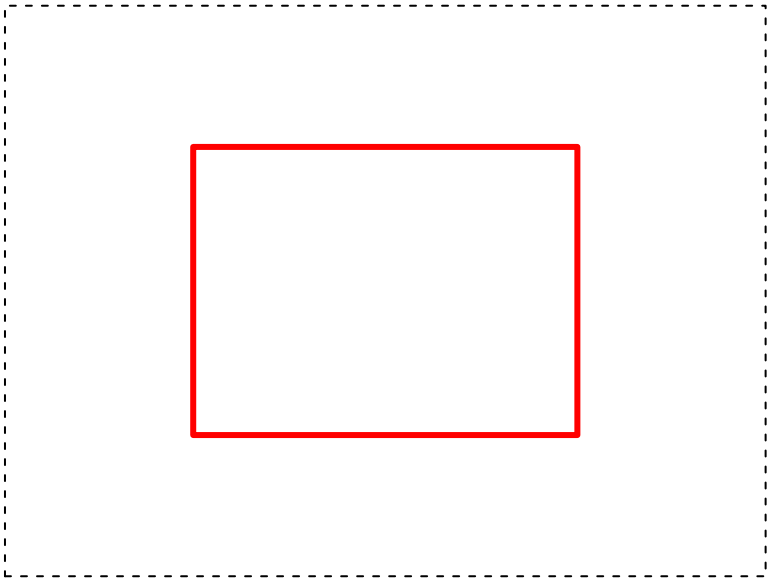
2. **grid**'s display list does not record base graphics output[1] so if both base and **grid** output appear on the same device then the result of editing will not redraw the base output. The following code provides a simple example:

[1] This is not quite true; it is possible to include base graphics output on the **grid** display list as we will see later.

```
> plot(1:10)
> par(new = TRUE)
> grid.rect(width = 0.5, height = 0.5, gp = gpar(lwd = 3), name = "gr")
>
```




```
> grid.edit("gr", gp = gpar(col = "red", lwd = 3))
```



After the `grid.edit`, the rectangle has been redrawn, but the base plot has not.

Saving calculations on the graphics engine display list and saving base graphics on the grid display list

Both of the problems described in the previous section can be avoided by using a `drawDetails()` method in **grid**. When a **grid** grob is drawn, the `drawDetails` method for that

grob is called; if calculations are put within a `drawDetails` method, then the calculations will be performed every time the grob is drawn.

This means that it is possible, for example, to use `convertUnit()` and have the result consistent across device resizes or copies[2]. This next piece of code is an example where the output becomes inconsistent when the device is resized. We specify a width for the rectangle in inches, but convert it (gratuitously) to NPC coordinates – when the device is resized, the NPC coordinates will no longer correspond to 1".

```
> grid.rect(width = convertWidth(unit(1, "inches"), "npc"))
```

The next piece of code demonstrates that, if we place the calculations within a `drawDetails` method, then the output remains consistent across device resizes and copies.

```
> drawDetails.myrect <- function(x, recording) {
+     gr <- rectGrob(width = convertWidth(unit(1, "inches"), "npc"))
+     grid.draw(gr)
+ }
> grid.draw(grob(cl = "myrect"))
```

The next example shows that a `drawDetails()` method can also be used to save base graphics output on the **grid** display list. This example uses **gridBase** to combine base and **grid** graphics output. Here I replicate the last example from the **gridBase** vignette – a set of base pie charts within **grid** viewports within a base plot. In this case, I can produce all of the grobs required in the normal manner – their locations and sizes are not based on special calculations[3].

```
> x <- c(0.88, 1.00, 0.67, 0.34)
> y <- c(0.87, 0.43, 0.04, 0.94)
> z <- matrix(runif(4*2), ncol = 2)
> maxpiesize <- unit(1, "inches")
> totals <- apply(z, 1, sum)
> sizemult <- totals/max(totals)
> gs <- segmentsGrob(x0 = unit(c(rep(0, 4), x),
+                              rep(c("npc", "native"), each = 4)),
+                    x1 = unit(c(x, x), rep("native", 8)),
+                    y0 = unit(c(y, rep(0, 4)),
+                              rep(c("native", "npc"), each = 4)),
+                    y1 = unit(c(y, y), rep("native", 8)),
+                    gp = gpar(lty = "dashed", col = "grey"))
> gr <- rectGrob(gp = gpar(col = "grey", fill = "white", lty = "dashed"))
```

What is important is that I place the calls to the **gridBase** functions within the `drawDetails` method so that they are performed every time the grob is drawn *and* the calls to the base graphics functions are in here too so that they are called for every redraw.

```
> drawDetails.pieplot <- function(x, recording) {
+     plot(x$x, x$y, xlim = c(-0.2, 1.2), ylim = c(-0.2, 1.2), type = "n")
```

[2] In each of the examples that follow, you should execute the example code, resize the device to see any inconsistency, then close the device before trying the next example.

[3] The example is wrapped inside a check for whether the **gridBase** package is installed so that the code will still "run" on systems without **gridBase**.

```
+     vps <- baseViewports()
+     pushViewport(vps$inner, vps$figure, vps$plot, recording = FALSE)
+     grid.draw(x$gs, recording = FALSE)
+     for (i in 1:4) {
+         pushViewport(viewport(x = unit(x$x[i], "native"),
+                               y = unit(x$y[i], "native"),
+                               width = x$sizemult[i]*x$maxpiesize,
+                               height = x$sizemult[i]*x$maxpiesize),
+                      recording = FALSE)
+         grid.draw(x$gr, recording = FALSE)
+         par(plt = gridPLT(), new = TRUE)
+         pie(x$z[i, ], radius = 1, labels = rep("", 2))
+         popViewport(recording = FALSE)
+     }
+     popViewport(3, recording = FALSE)
+ }
```

The "pieplot" is created by assembling the component grobs into a collective grob of the appropriate class; the `drawDetails` method takes care of actually producing the output.

```
> if (suppressWarnings(require("gridBase", quietly = TRUE))) {
+ grid.draw(grob(x = x, y = y, z = z,
+                maxpiesize = maxpiesize, sizemult = sizemult,
+                gs = gs, gr = gr, cl = "pieplot"))
+ }
```

The output from this example can be resized safely; **grid** handles all of the redrawing, and performs all of the actions within the `drawDetails` method for each redraw, including redrawing the base graphics output!

As a final example, we will harness the **grid** display list purely to achieve consistency in base graphics output. The following reproduces the last example from the `legend()` help page, but produces output which can be resized without the legend going crazy.

```
> drawDetails.mylegend <- function(x, recording) {
+     x <- 0:64/64
+     y <- sin(3*pi*x)
+     plot(x, y, type = "l", col = "blue",
+          main = "points with bg & legend(*, pt.bg)")
+     points(x, y, pch = 21, bg = "white")
+     legend(.4,1, "sin(c x)", pch = 21, pt.bg = "white", lty = 1, col = "blue")
+ }
> grid.draw(grob(cl = "mylegend"))
```