Examples of the Use of grid grobs

Paul Murrell

February 12, 2004

The document "Reimplementing grid grobs" discusses reasons for changing the way graphical objects are implemented in grid. These changes are available in an experimental version of grid and this document provides a number of examples that demonstrate the sorts of things that are possible in this experimental version.

First of all, the changes have no impact on grid code that just produces output. For example, the following works exactly as before:

```
> example(Grid)


Grid> grid.show.layout(grid.layout(4, 2, heights = unit(rep(1,
    4), c("lines", "lines", "lines", "null")), widths = unit(c(1,
    1), "inches")))

Grid> grid.show.viewport(viewport(x = 0.6, y = 0.6, w = unit(1,
    "inches"), h = unit(1, "inches")))

Grid> grid.multipanel(vp = viewport(0.5, 0.5, 0.8, 0.8))
viewport[ROOT]
```

There is a new distinction between grobs which are just stored in user-level R objects and grobs which represent drawn output (i.e., grobs on the display list. There is a naming convention that grid.*() functions are (mainly) used for their side-effect of producing output or modifying existing output (they create/affect grobs on the display list). Functions of the form *Grob() are used for their return value; the grob that they create/modify. For example, the following creates a grob and then modifies it, but performs absolutely no drawing; this is purely manipulating a description of a graphical object.

```
> gl <- linesGrob()
> gl <- editGrob(gl, gp = gpar(col = "green"))
```

The next example produces output. As previously, a grob is returned, but that grob is just a description of the output that was drawn and has no direct link to the output. A link to the output is possible using the grob's name. In order to

access a grob which represents drawn output (i.e., a grob on the display list), you must specify a gPath. The gPath should be created using the gPath() function for writing scripts, but in interactive use, it is possible to specify the gPath directly as a string. The code below shows both approaches.

```
> grid.newpage()
> grid.lines(name = "lines")
> grid.edit(gPath("lines"), gp = gpar(col = "pink"))
> grid.edit("lines", gp = gpar(col = "red"))
```

Complex graphical objects are provided by the gTree class. A gTree is a grob which may have other grobs as children. The xaxis and yaxis grobs provided by grid are examples of gTrees; the children of an axis include a lines grob for the tick-marks and a text grob for the tick-mark labels. The function childNames() can be used to list the names of the children of a gTree. When dealing with these hierarchical objects, more complex gPaths can be used to access children of a gTree. In the following example, an x-axis is drawn, then the xaxis itself is edited to modify the locations of the tick-marks, then the xaxis's text child is edited to modify the location of the labels on the tick-marks.

```
> grid.newpage()
> pushViewport(viewport(w = 0.5, h = 0.5))
> grid.rect(gp = gpar(col = "grey"))
> grid.xaxis(name = "myxaxis")
> grid.edit("myxaxis", at = 1:4/5)
> grid.edit(gPath("myxaxis", "labels"), y = unit(-1, "lines"))
```

This next example extends the idea a step further to edit the child of a child of a gTree. It also shows the use of the gTree function to construct a simple gTree (this is just creating an instance of the gTree class – it is also possible to extend the gTree class in order to provide specialised behaviour for drawing and other things; more on this later). Finally, the example demonstrates how gPaths of depth greater than 1 can be specified directly as a string.

```
> grid.newpage()
> pushViewport(viewport(w = 0.5, h = 0.5))
> myplot <- gTree(name = "myplot", children = gList(rectGrob(name = "box",
+     gp = gpar(col = "grey")), xaxisGrob(name = "xaxis")))
> grid.draw(myplot)
> grid.edit("myplot::xaxis", at = 1:10/11)
> grid.edit("myplot::xaxis::labels", label = round(1:10/11, 2))
> grid.edit("myplot::xaxis::labels", y = unit(-1, "lines"))
```

The semantics of "grobwidth" units are slightly changed in the new implementation. Existing code will still work, but the grobs provided as data are no longer "pointers". This means that modifications to the grob will not be reflected in the evaluation of the unit. The following example just shows that "grobwidth" units still work (but if you modify gt it will have no effect on the width of the drawn rectangle).

```
> grid.newpage()
> gt <- grid.text("Hi there")
> grid.rect(width = unit(1, "grobwidth", gt))
```

In order to allow "pointers" to grobs within "grobwidth", it is possible to specify a gPath rather than a grob as the data for a "grobwidth" unit. The following example modifies the previous example to use such a "pointer".

```
> grid.newpage()
> gt <- grid.text("Hi there", name = "sometext")
> grid.rect(width = unit(1, "grobwidth", "sometext"))
> grid.edit("sometext", label = "Something different")
```

One issue in the evaluation of "grobwidth" units has always been establishing the correct "context" for a grob when determining its width (if a grob has a viewport in its vp slot then that viewport gets pushed before the grob is drawn; that viewport should also be pushed when determining the width of the grob). This has been pretty awkward in the past and is slightly cleaner in the new implementation. The old drawDetails() generic has been split into preDrawDetails(), drawDetails(), and postDrawDetails() (suggestions for better names welcome!). The idea is that pushing and popping of viewports should occur in the pre and post generics, and any actual drawing happens in the main drawDetails() generic. This allows the code that calculates a grob width to call the preDrawDetails() in order to establish the context in which the grob would be drawn before calculating its width. The following example shows a test case; a grob is created (extending to a new class to allow specific methods to be written), and methods are provided which establish a particular context for drawing the grob. These methods are used both in the drawing of the grob and in the calculation of the grob's width (when drawing a bounding rectangle).

```
> grid.newpage()
> mygrob <- grob(name = "mygrob", cl = "mygrob")
> preDrawDetails.mygrob <- function(x) {
+     pushViewport(viewport(gp = gpar(fontsize = 20)))
+ }
> drawDetails.mygrob <- function(x, recording = TRUE) {
+     grid.draw(textGrob("hi there"), recording = FALSE)
+ }
> postDrawDetails.mygrob <- function(x) {
+     popViewport()
+ }
> widthDetails.mygrob <- function(x) {
+     unit(1, "strwidth", "hi there")
+ }
> grid.draw(mygrob)
> grid.rect(width = unit(1, "grobwidth", mygrob))
```

This next example shows a slightly different test case where the standard `preDrawDetails()` and `postDrawDetails()` methods are used, but the `grob` does have a `vp` slot so these methods do something. Another interesting feature of this example is the slightly more complex `gTree` that is created. The `gTree` has a `childrenvp` specified. When the `gTree` is drawn, this viewport is pushed and then "up"ed before the children of the `gTree` are drawn. This means that the children of the `gTree` can specify a `vpPath` to the viewport they should be in. This allows the parent `gTree` to create a suite of viewports and then children of the `gTree` select which one they want – this can be more efficient than having each child push and pop the viewports it needs, especially if several children are drawn within the same viewport. Another, more realistic example of this is given later.

```
> grid.newpage()
> mygtree <- gTree(name = "mygrob", childrenvp = viewport(name = "labelvp",
+     gp = gpar(fontsize = 20)), children = gList(textGrob("hi there",
+     name = "label", vp = "labelvp")), cl = "mygtree")
> widthDetails.mygtree <- function(x) {
+     unit(1, "grobwidth", getGrob(x, "label"))
+ }
> grid.draw(mygtree)
> grid.rect(width = unit(1, "grobwidth", mygtree))
```

The "frames and packing" facilities in `grid` have always involved working with `grob`s rather than just producing output, so they are affected quite significantly by the changes. In particular, the behaviour of `grid.pack()` and `grid.place()` are quite different because they now only affect drawn output and consequently only allow a `gPath` for the frame argument. Constructing a description of a `frame` `grob` must now be done via `packGrob()` and `placeGrob()`. The following example shows the construction of a simple frame consisting of two equal-size columns.

```
> grid.newpage()
> fg <- frameGrob(layout = grid.layout(1, 2))
> fg <- placeGrob(fg, textGrob("Hi there"), col = 1)
> fg <- placeGrob(fg, rectGrob(), col = 2)
> grid.draw(fg)
```

This next example constructs a slightly fancier frame using packing.

```
> grid.newpage()
> pushViewport(viewport(layout = grid.layout(2, 2)))
> drawIt <- function(row, col) {
+     pushViewport(viewport(layout.pos.col = col, layout.pos.row = row))
+     grid.rect(gp = gpar(col = "grey"))
+     grid.draw(fg)
+     upViewport()
+ }
```

```
> fg <- frameGrob()
> fg <- packGrob(fg, textGrob("Hi there"))
> fg <- placeGrob(fg, rectGrob())
> drawIt(1, 1)
> fg <- packGrob(fg, textGrob("Hello again"), side = "right")
> drawIt(1, 2)
> fg <- packGrob(fg, rectGrob(), side = "right")
> drawIt(2, 2)
```

In order to allow frames and packing to make use of "pointers" to grobs (so that a frame can be made to expand or contract if the contents are edited), there is a `dynamic` argument to `packGrob()` (and `grid.pack()`). The following extends the previous example to show how this might be used. Another feature of this example is the demonstration of "non-strict" searching that occurs in the `grid.edit()` call; the `grob` called `"midtext"` is not at the top-level, but is still found. Something like `grid.get("midtext", strict=TRUE)` would fail.

```
> grid.newpage()
> fg <- frameGrob()
> fg <- packGrob(fg, textGrob("Hi there"))
> fg <- placeGrob(fg, rectGrob())
> fg <- packGrob(fg, textGrob("Hello again", name = "midtext"),
+     side = "right", dynamic = TRUE)
> fg <- packGrob(fg, rectGrob(), side = "right")
> grid.draw(fg)
> grid.edit("midtext", label = "something much longer")
```

There have been a few examples already which have involved creating a `gTree`. This next example explicitly demonstrates this technique. A `gTree` is created with two important components. The `childrenvp` is a `vpTree` consisting of a `"plotRegion"` viewport to provide margins around a plot and a `"dataRegion"` viewport to provide x- and y-scales. The `"dataRegion"` gets pushed within the `"plotRegion"` and both are pushed and then "up"ed before the children are drawn. The `children` of the `gTree` are an `xaxis` and a `yaxis` both drawn within the `"dataRegion"`, and a `rect` drawn around the border of the `"plotRegion"`. A further feature of this example is the use of the `addGrob()` and `removeGrob()` functions to modify the `gTree`. The first modification involves adding a new child to the `gTree` which is a set of points drawn within the `"dataRegion"`. The second modification involves adding another set of points with a different symbol (NOTE that this second set of points is given a name so that it is easy to identify this set amongst the children of the `gTree`). The final modification is to remove the second set of points from the `gTree`.

```
> grid.newpage()
> pushViewport(viewport(layout = grid.layout(2, 2)))
> drawIt <- function(row, col) {
+     pushViewport(viewport(layout.pos.col = col, layout.pos.row = row))
+     grid.rect(gp = gpar(col = "grey"))
```

```
+     grid.draw(gplot)
+     upViewport()
+ }
> gplot <- gTree(x = NULL, y = NULL, childrenvp = vpTree(plotViewport(c(5,
+     4, 4, 2), name = "plotRegion"), vpList(viewport(name = "dataRegion"))),
+     children = gList(xaxisGrob(vp = "plotRegion::dataRegion"),
+         yaxisGrob(vp = "plotRegion::dataRegion"), rectGrob(vp = "plotRegion")))
> drawIt(1, 1)
> gplot <- addGrob(gplot, pointsGrob(vp = "plotRegion::dataRegion"))
> drawIt(1, 2)
> gplot <- addGrob(gplot, pointsGrob(name = "data1", pch = 2, vp = "plotRegion::dataRegion
> drawIt(2, 1)
> gplot <- removeGrob(gplot, "data1")
> drawIt(2, 2)
```

One of the original motivations for the new implementation was the ability to save/load grobs. This next example provides a simple demonstration that this now works. It is also a nice demonstration that grobs really do copy like normal R objects now.

```
> gplot <- gTree(x = NULL, y = NULL, childrenvp = vpTree(plotViewport(c(5,
+     4, 4, 2), name = "plotRegion"), vpList(viewport(name = "dataRegion"))),
+     children = gList(xaxisGrob(vp = "plotRegion::dataRegion"),
+         yaxisGrob(vp = "plotRegion::dataRegion"), rectGrob(vp = "plotRegion")))
> save(gplot, file = "gplot1")
> gplot <- addGrob(gplot, pointsGrob(vp = "plotRegion::dataRegion"))
> save(gplot, file = "gplot2")
> grid.newpage()
> pushViewport(viewport(layout = grid.layout(1, 2)))
> pushViewport(viewport(layout.pos.col = 1))
> load("gplot1")
> grid.draw(gplot)
> popViewport()
> pushViewport(viewport(layout.pos.col = 2))
> load("gplot2")
> grid.draw(gplot)
> popViewport()
```

This next example just demonstrates that it is possible to use a gPath to access the children of a gTree when editing. This is the editGrob() equivalent of an earlier example that used grid.edit(). One useful application of this API is the ability to modify the appearance of quite precise elements of a large, complex graphical object by editing the gp slot of a child (of a child ...) of a gTree.

```
> myplot <- gTree(name = "myplot", children = gList(rectGrob(name = "box",
+     gp = gpar(col = "grey")), xaxisGrob(name = "xaxis")))
> myplot <- editGrob(myplot, gPath = "xaxis", at = 1:10/11)
> myplot <- editGrob(myplot, gPath = "xaxis::labels", label = round(1:10/11,
```

```
+     2))
> myplot <- editGrob(myplot, gPath = "xaxis::labels", y = unit(-1,
+     "lines"))
> grid.newpage()
> pushViewport(viewport(w = 0.5, h = 0.5))
> grid.draw(myplot)
```

The API for accessing children of a `gTree` or any drawn `grob` has been cleaned up considerably. The following example demonstrates the use of the `getGrob()` and `grid.get()` (along with `gPath`s) to access `grob`s.

```
> myplot <- gTree(name = "myplot", children = gList(rectGrob(name = "box",
+     gp = gpar(col = "grey")), xaxisGrob(name = "xaxis")))
> getGrob(myplot, "xaxis")
> myplot <- editGrob(myplot, gPath = "xaxis", at = 1:10/11)
> getGrob(myplot, "xaxis::labels")
> grid.newpage()
> pushViewport(viewport(w = 0.5, h = 0.5))
> grid.draw(myplot)
> grid.get("myplot")
> grid.get("myplot::xaxis")
> grid.get("myplot::xaxis::labels")
```

There is also now an API for (re)setting children of a `gTree` or any drawn `grob`. This is not intended for general user use, but provides a simple way for developers to perform modifications to the structure of a `gTree` by doing something like ...

```
grob <- getGrob(<spec>)
<modify grob>
setGrob(<spec>, grob)
```

This approach is used in the new implementation of packing and placing grobs. The following example shows some simple usage of the `setGrob()` and `grid.set()` functions to replace children of a `gTree` with different `grob`s. NOTE that currently such replacement can only occur if the name of the new `grob` is the same as the name of the old `grob`.

```
> myplot <- gTree(name = "myplot", children = gList(rectGrob(name = "box",
+     gp = gpar(col = "grey")), xaxisGrob(name = "xaxis")))
> myplot <- setGrob(myplot, "xaxis", rectGrob(name = "xaxis"))
> grid.newpage()
> pushViewport(viewport(w = 0.5, h = 0.5))
> grid.draw(myplot)
> grid.set("myplot::xaxis", xaxisGrob(name = "xaxis", at = 1:3/4))
> grid.set("myplot::xaxis::labels", textGrob(name = "labels", x = unit(1:3/4,
+     "native"), y = unit(-1, "lines"), label = letters[1:3]))
> myplot <- setGrob(grid.get("myplot"), "xaxis::labels", circleGrob(name = "labels"))
```

```
> grid.newpage()
> pushViewport(viewport(w = 0.5, h = 0.5))
> grid.draw(myplot)
```

This next example just shows more complex use of the add/remove facilities for modifying grobs. Again, `addGrob()` and `removeGrob()` are for constructing descriptions of graphical objects and `grid.add()` and `grid.remove()` are for modifying drawn output. Of particular note are the last two lines involving `grid.remove()`. The first point is that there are multiple `grob`s on the display list with the same name. The example only affects the first one it finds; this could easily be extended to affect the display list "globally" (for children of `gTree`s, there cannot be multiple children with the same name so the issue does not arise). The last line is interesting because it actually erases the `grob` named `"plot1"` from the display list altogether (well, the first instance on the display list of a `grob` called `"plot1"` anyway).

```
> drawIt <- function(row, col) {
+     pushViewport(viewport(layout.pos.col = col, layout.pos.row = row))
+     grid.rect(gp = gpar(col = "grey"))
+     grid.draw(gplot)
+     upViewport()
+ }
> gplot <- gTree(name = "plot1", childrenvp = vpTree(plotViewport(c(5,
+     4, 4, 2), name = "plotRegion"), vpList(viewport(name = "dataRegion"))),
+     children = gList(xaxisGrob(name = "xaxis", vp = "plotRegion::dataRegion"),
+         yaxisGrob(name = "yaxis", vp = "plotRegion::dataRegion"),
+         rectGrob(name = "box", vp = "plotRegion")))
> grid.newpage()
> pushViewport(viewport(layout = grid.layout(2, 2)))
> drawIt(1, 1)
> grid.add("plot1", pointsGrob(0.5, 0.5, name = "data1", vp = "plotRegion::dataRegion"))
> grid.add("plot1::xaxis", textGrob("X Axis", y = unit(-2, "lines"),
+     name = "xlab"))
> grid.edit("plot1::xaxis::xlab", y = unit(-3, "lines"))
> gplot <- grid.get("plot1")
> gplot <- addGrob(gplot, gPath = "yaxis", textGrob("Y Axis", x = unit(-3,
+     "lines"), rot = 90, name = "ylab"))
> drawIt(1, 2)
> gplot <- removeGrob(gplot, "xaxis::xlab")
> drawIt(2, 1)
> grid.remove("plot1::data1")
> grid.remove("plot1")
```

The next example is just a `grid.place()` and `grid.pack()` equivalent of an earlier example involving `placeGrob()` and `packGrob()`. The interesting feature is that each action is reflected in the output as it occurs.

```
> grid.newpage()
> grid.frame(name = "myframe", layout = grid.layout(1, 2))
```

```
> grid.place("myframe", textGrob("Hi there"), col = 1)
> grid.place("myframe", rectGrob(), col = 2)
> grid.newpage()
> grid.frame(name = "frame2")
> grid.pack("frame2", textGrob("Hi there"))
> grid.place("frame2", rectGrob())
> grid.pack("frame2", textGrob("Hello again"), side = "right")
> grid.pack("frame2", rectGrob(), side = "right")
```

One concern about the new implementation is the speed penalty that will incurred due to extra copying. The next two examples provide a couple of very simple test cases in case the effect is obviously unacceptable. The code is slower, in some cases by up to 50This is not especially noticeable for a single plot, but may become a bit painful in a batch job and it makes dynamic graphics even less conceivable. On the other hand, this new design is the way I want to go so my approach is: wait to see if people complain and if they do, try to make this design go faster.

The first example involves a plot with a large number of points and the second is just a benchmarking of the standard example output.

```
> myplot <- gTree(name = "myplot", children = gList(rectGrob(name = "box",
+     gp = gpar(col = "grey")), xaxisGrob(name = "xaxis")))
> myplot <- addGrob(myplot, pointsGrob(name = "data", x = runif(10000),
+     y = runif(10000)))
> grid.newpage()
> pushViewport(viewport(w = 0.5, h = 0.5))
> grid.draw(myplot)
> grid.edit("myplot::xaxis", at = 1:3/4)
> grid.edit("myplot::data", pch = ".")
```

```
> system.time(for (i in 1:10) example(Grid))
```

That's all for now! :)