Reimplementing `grid` grobs

December 24, 2003

The original implementation of `grid` grob objects uses external references (`http://www.stat.uiowa.edu/ luke/R/simpleref.html`). This means that when you create a grob, you actually get a "pointer" to the object. If you copy a grob, you get a copy of the pointer, but you are still pointing at the same object. A simple demonstration:

```
> mygrob <- grid.lines()
> unclass(mygrob)
[[1]]
<pointer: 0x8f26e08>

> copyofmygrob <- mygrob
> unclass(copyofmygrob)
[[1]]
<pointer: 0x8f26e08>
```

This implementation was originally chosen because it made it possible to have a user-level R object point to the same actual grob as was stored on the display list. grid places a copy of drawn grobs on its display list; if I retain a copy of a grob myself and edit that copy, I can effectively modify the objects on the display list (so the output gets modified).

A side effect of this implementation is that user-level grobs violate the normal behaviour of R objects. Most objects copy by value, but grobs copy by reference; if you modify a grob (e.g., using `grid.edit`), you change all copies of that grob. In the following example, the `gp` slot of `copyofmygrob` starts off as an empty list, but we modify it to have a `col="green"` specification *by editing* `mygrob`.

```
> grid.get(copyofmygrob, "gp")
list()
attr(,"class")
[1] "gpar"

> grid.edit(mygrob, gp=gpar(col="green"))
NULL

> grid.get(copyofmygrob, "gp")
$col
[1] "green"

attr(,"class")
[1] "gpar"
```

A new implementation of grobs

I propose to change the implementation of grobs so that they behave just like normal R objects (i.e., copy by value). This will involve making a distinction between a user-level grob *description*

and a *drawn* grob (on the display list). The former is a user-level object containing a set of parameters describing the structure and appearance of a grob (just like the object pointed to by current grobs). The latter is a system-level record of a grob that has been drawn on a device (it will contain all of the original description, but may also contain additional information such as its bounding box on the device). Grobs will have an extra "name" slot and this will be used to provide access to grobs on the display list. So rather than having a pointer reference to objects that have been drawn, I would have the ability to refer to drawn objects by name.

This implementation parallels the new implementation of `grid` viewports. These are now divided into user-level viewport objects (a description of a rectangular region) and system-level *pushed* viewports (a record of a rectangular region that has been created on a device). Viewports have a name slot and this is used to identify a particular pushed viewport.

Basic reasons against grobs-by-reference

It has already been noted that grobs currently violate standard R semantics; this is inconsistent and could cause users some nasty surprises. In addition, it is not currently possible to save/load grobs. This should in theory be solvable by implementing a "finalizer" and "initializer" for grobs, but a tougher one is the issue of how to reestablish multiple references to the same grob. In the first example above, if `mygrob` and `copyofmygrob` are saved and then reloaded, we would like them to both point to the same grob again (there are much more complex examples where it becomes much more important that this is true). This sort of saving of common references is probably much harder to achieve (although it is already solved for function environments??).

A potential use for the copy-by-reference behaviour of grob is in the reuse of grobs. For example, a multi-plot image could use exactly the same grob for the x-axis of all plots. This would mean that all axes could be edited simultaneously by simply editing the shared grob. Unfortunately, this turns out to be a pretty dumb way of achieving this sort of thing (and I had learned this before so I'm doubly stupid for repeating the mistake); this sort of sharing forces the axes (in this case) to be *identical*. What is far more useful is the ability to have graphical objects share some features (but not all). Another way to say this is that you want to be able to modify a particular characteristic of several grobs at once.

Basic Reasons for grobs-by-value

The obvious ones are that the problems with the current implementation go away. Grobs would obey normal R semantics and grobs could be saved and reloaded (without any extra effort). Common references could also be saved and restored because references would be in the form of a grob name; the same name refers to the same grob.

Having grobs share only some features would be possible if `grid.edit` is modified to accept grob names; for example, it would be possible to do things like edit a particular feature of all grobs whose names match a regular expression. Something like:

```
grid.edit(".*xaxis.*", at=1:5)
```

Reasons against grobs-by-value

I think there are two biggies. First of all, for users, it would be a very large change in the behaviour of grobs. This may not be such a big issue because I believe the number of people directly using `grid` is very small (most use it via `lattice`). Also, of those few who use `grid` directly, I think possibly only one (M Kondrin) is making use of `grid` grobs; I think all others just use `grid` functions for their side-effect of producing graphical output (and ignore the grobs that are produced).

The second major problem is that the new implementation would lead to a lot more copying of objects. In the first example above, only the pointer is copied; if grobs copied by value then the entire grob objects would be copied. This would lead to greater memory requirements and a speed penalty. The size of this effect would have to be investigated.

Further changes accompanying the new implementation

All of the `grid.*` functions that currently produce output and return grobs would be split into a `*Grob` function to create and return a grob and a `grid.*` function which produces the output. The latter would simply become something like:

```
grid.lines <- function(<lots of args>) {
  gl <- linesGrob(<lots of args>)
  grid.draw(gl)
}
```

This change is to make it easier just to create a grob (description) without producing any output (i.e., instead of having to do `grid.lines(..., draw=FALSE)` you just do `linesGrob(...)`.

Continuing the parallel with the new design of `grid` viewports, a new class would be created called a `gTree`. This would extend the grob class and allow a grob to have children. This new `gTree` would replace the current collection grob (forcing a [internal] reimplementation of xaxis, yaxis, and frame grobs). Having children of a grob identifiable by name would allow for a cleaner implementation of the grid.edit interface for selecting sub-components of a grob; a list of sub-component names could be replace by a single `gPath` argument (`gPath`s would be similar to `vpPath`s). Also, this would require a useful interface for modifying the children of a `gTree`; something like `addChild` and `removeChild` functions.

A `gTree` would also have a `children.vpTree` slot. The standard vp slot would work as normal (push it before drawing the `gTree` and pop it afterwards), but the `children.vpTree` would be pushed and then an appropriate `upViewport` call made. This would create a viewport context for the grobs which are children of the `gTree` so that the children could have vp slots which simply name an appropriate viewport within the `children.vpTree`.

grobwidth and grobheight units

In the original implementation, grobwidth units contain a copy of the relevant grob (which points to a shared grob object). When the unit needs to be evaluated, the grob is asked for its width. Complexities arise because the grob may have a non-`NULL` vp slot; some effort is expended to ensure that the width of the grob (which is itself a unit) gets evaluated within the proper context.

It is assumed that the grob in a grobwidth unit is at the same level within the `vpTree` (any viewports in the vp slot are taken to be relative to the current position of the unit within the `vpTree`). This is not necessarily true – the grob could be anywhere in the `vpTree` – so inconsistent situations can occur.

With the new implementation, a grobwidth unit could be based upon a grob (description), in which case the grob is assumed to be at the same level within the `vpTree` (and this is consistent; even if there is another grob within the `vpTree` based on the same description, it is not exactly the same grob so the meaning of the grobwidth unit is clear). The grobwidth unit could also be based upon a `gPath` (possibly just a grob name) – this would correspond to the original implementation, being a (declarative) pointer to a grob. When the unit needs to be evaulated, the `gPath` could be resolved to find a grob and everything carries on as before. Again there is consistency because we only try to resolve the `gPath` by looking below our current position within the `vpTree`.

Procedural versus object-oriented graphics

There would still be two ways to use `grid` graphics. One way is to perform a series of viewport pushes and pops and grob drawing operations to build up an image piece by piece. Something like:

```
pushViewport(viewport(w=0.5, h=0.5))
grid.rect()
popViewport()
```

This sort of approach is useful for "sketching" an image on-the-fly (without having to predesign the entire thing). Another approach is to create a complete description of an image first, then draw the whole thing. Something like:

```
myimage <- gTree(rectGrob(name="r1", vp="vp1"),
                 children.vp=viewport(w=0.5, h=0.5, name="vp1"))
grid.draw(myimage)
```

The usefulness of the latter approach is that it provides a simple way to save/load a complete graphical component (it makes graphical "templates" possible). Note that this "image" can be used as part of some larger image; this sort of approach is how graphical images should ultimately be defined if they are to be used by others.

There are two important requirements to satisfy in order to maintain the two approaches to creating graphics: first of all, anything that can be done procedurally must be possible via an object-oriented (declarative) approach (an important example is the ability to specify the *order* in which grobs are drawn); secondly, following on from the first, it should be possible to generate an object-oriented description from an image that has been created procedurally. The latter would allow the display list to be "captured" as a single `gTree`; or the display list could be created as a `gTree` in the first place (deleting a grob from the current image could simply be a matter of calling removeChild with the appropriate name).

A `gTree` could have a vector of names which specifies the drawing order of its children. Having such an explicit ordering would allow for implementing things like "bring to top", "push to back" (within a `gTree` at least); i.e., you could explicitly manipulate the drawing order of grobs.

Backwards-compatibility: the impact on existing code

There should be no effect on code which simply relies on the output produced by `grid`. This basically means the use of graphical primitives (`grid.lines()`, `grid.text()`, ...), graphical components (`grid.xaxis()`, `grid.yaxis()`), and `pushViewport`, `popViewport`, `upViewport`, `downViewport`, and `seekViewport`.

Code which manipulates grobs would have to make some changes. For example, `grid.pack` and `grid.place` behave differently so the construction of keys in lattice would require some modification. The syntax and semantics of `grid.edit()` would change considerably; this may affect M Kondrin's Digrace package. I am unsure of the effect on Frank Harrell's Hmisc and Design and Nicholas Lewin Koh's code.