A Redesign of `grob` Extensibility (Oct/Nov 2012)

*These changes (to **grid**) were committed to `r-devel` as `r61044`; plus a small fix added in `r61059`.*

R developers can currently extend the basic `grob` class from the **grid** package by writing methods for one or more of the following generic functions:

- `drawDetails()`, to customize what gets drawn by the `grob` *at drawing time.*
- `preDrawDetails()`, to customize the *drawing context* for the `grob` (typically, pushing additional viewports).
- `postDrawDetails()`, to clean up after `preDrawDetails()`.

These generic function "hooks" are called as part of the standard drawing behaviour of `grobs`, which means that new types of `grob`s can be created without having to write complete `grid.draw()` methods (which is good because the `grid.draw()` methods for the core **grid** `grob`s—`grob`s and `gTree`s—contain all sorts of nasty code that is best left as a hidden implementation detail). Having these hooks into the standard `grid.draw()` function also allows new `grob`s to benefit from important basic `grob` behaviour such as automatic pushing/popping of the `vp` slot and automatic setting/unsetting of the `gp` slot.

There are at least two reasons for wanting to build a `grob` only at drawing time: either the `grob` needs to know the physical properties of the drawing context before it can decide its dimensions or location (e.g., an `"xaxis"` `grob` needs to know the viewport scale *at drawing time* in order to create its tick marks and labels); or the `grob` consists of non-trivial content *and* the grob can be updated over time *and re*building the `grob` contents is complex or costly, so you only want to do it once when you actually draw, rather than every time the `grob` is updated (this is the situation for **gtable**).

There are also `x|y|width|heightDetails()` functions, to specify how to correctly determine the dimensions and location of a `grob`, but those are not part of this review.

There are two main problems with the generic functions `drawDetails()`, `preDrawDetails()`, and `postDrawDetails()`:

1. they do not provide *enough* flexibility, so developers end up having to write new `grid.draw()` methods after all (one specific example of this problem is that `preDrawDetails()` only gets called *after* the `vp` slot of a grob has already been pushed). This is bad because `grid.draw()` methods tend to be nasty (see above).

2. these functions are currently used *both* to create new viewports or grobs to draw *and* to push those viewports or draw those grobs, *without* returning the viewports or grobs that were created. This leads to two issues: there are other situations when it is useful to know about the context that a grob sets up (e.g., when determining the width/height of a grob for `"grobwidth"` units) or the content that a grob draws (e.g., when the **gridSVG** package tries to reproduce the output from a `grob`) and the `drawDetails()`, `preDrawDetails()`, and `postDrawDetails()` functions cannot be (re)used to obtain this information; there are also situations where it would be useful to retain information about what a `grob` draws (e.g., this is what a `grid.force()` function needs to know if it wants to replace the original `grob` with the output that the `grob` would draw).

This redesign is motivated by several goals:

1. Make it easier for developers to create new types of `grobs` (e.g., for the **gtable** package). (i.e., avoid having to write `grid.draw()` methods.)
2. Make it easier for developers of new packages to work with (or cope with) new grobs. For example, the **gridSVG** package currently has to write a `primToDev()` method for any `grob` that has its own `drawDetails()` method (and the `primToDev()` method is usually pretty much a copy of the `drawDetails()` method. So the goal is less code duplication and more code reuse.
3. Make it possible to convert a `grob` that only produces its output at drawing time into a static `grob` (e.g., for use with `grid.edit()`). This would allow a `grid.force()` function (and that would greatly simplify **gridSVG**, which would just "force" everything and then export using basic `grob` and `gTree` methods).

The fundamental idea behind the redesign is to introduce new hook functions that roughly correspond to the existing `drawDetails()` and `preDrawDetails()` functions, but these new hooks are *only* responsible for generating (and returning) a new context for drawing (viewports) or new content to draw (grobs) and the actual enforcement of that context and the actual drawing is part of the default `grob` behaviour.

Implementation

Two new generic hook functions have been added to **grid**: `makeContext()` and `makeContent()`.

The `makeContext()` function has one argument, which is a `grob`, and methods should return a `grob` as the result. This function is designed to provide an opportunity for a `grob` to modify or enhance its `vp` slot (i.e., control the generation of the drawing context for the `grob` at drawing time).

The `makeContent()` function is similar (take a `grob` and return a `grob`), but it is designed to provide an opportunity for a `grob` to modify itself in terms of what will be drawn (i.e., control the generation of drawing content at drawing time).

For basic `grob`s, the `makeContent()` function should return just a simple `grob`.

For `gTree`s, the `makeContext()` function could also be used to modify the `childrenvp` slot (i.e., the drawing context for children of the `gTree`) and the `makeContent()` function can be used to modify the `children` slot of the `gTree`, so it should return a `gTree`.

During drawing of a `grob`, the `makeContext()` function is called *before* enforcing the `grob`'s `vp` slot (and before enforcing the `childrenvp` slot for `gTree`s). This means that a `makeContext()` method should modify the `vp` slot (and/or the `childrenvp` slot) in order to affect the drawing context.

The `makeContent()` function is called *before* the `drawDetails()` function (and before drawing the `children` of a `gTree`). This means that a `makeContent()` method should generate a standard `grob` primitive (or a `gTree` with `children`) in order to affect what is drawn.

NOTE that rather than *increasing* the number of hook functions in **grid**, the idea is that these two hooks *replace* the three old hooks `drawDetails()`, `preDrawDetails()`, and `postDrawDetails()`. See page 35 for comments on transitioning from and backward compatibility with existing code.

Example 1: a `makeContent()` method

An example of a `gTree` that generates its content at drawing time is an `"xaxis"` object (with `at=NULL`; because then the axis must calculate its tick marks and labels based on its drawing context at drawing time).

Here is the old `drawDetails()` method for the `"xaxis"` class ...

```
drawDetails.xaxis <- function(x, recording=TRUE) {
  if (is.null(x$at)) {
    x$at <- grid.pretty(current.viewport()$xscale)
    x <- addGrob(x, make.xaxis.major(x$at, x$main))
    x <- addGrob(x, make.xaxis.ticks(x$at, x$main))
    x <- updateXlabels(x)
    x <- applyEdits(x, x$edits)
    for (i in childNames(x))
      grid.draw(getGrob(x, i))
  }
}
```

... and here is the new `makeContent()` method (this *replaces* the `drawDetails()`, which is removed) ...

```
> grid:::makeContent.xaxis
```

```
function (x)
{
    if (is.null(x$at)) {
        x$at <- grid.pretty(current.viewport()$xscale)
        x <- addGrob(x, make.xaxis.major(x$at, x$main))
        x <- addGrob(x, make.xaxis.ticks(x$at, x$main))
        x <- updateXlabels(x)
        x <- applyEdits(x, x$edits)
    }
    x
}
<bytecode: 0x31dbee0>
<environment: namespace:grid>
```

The code is very similar, the only real difference being that the `makeContent()` method only generates the content to draw, but does not draw it, and the modified `"xaxis"` `grob` is returned.

The result of drawing an `"xaxis"` object is unchanged. In the old set up, the `drawDetails()` method would get called to generate *and draw* the content of the axis; in the new set up, the `makeContent()` method just *generates* the content of the axis (by adding the content as children of the `gTree`) and the standard drawing behaviour for `gTree`s takes care of the actual drawing.

The following code generates a simple scene that contains an x-axis to show that this still works (see Figure 1).

```
> xag <- xaxisGrob(name="test")
```

```
> x <- runif(10)
> y <- runif(10)
```

```
> # Draw an axis with 'at=NULL', so the ticks get generated
> # on-the-fly
> grid.newpage()
> grid.rect(gp=gpar(col="grey"))
```

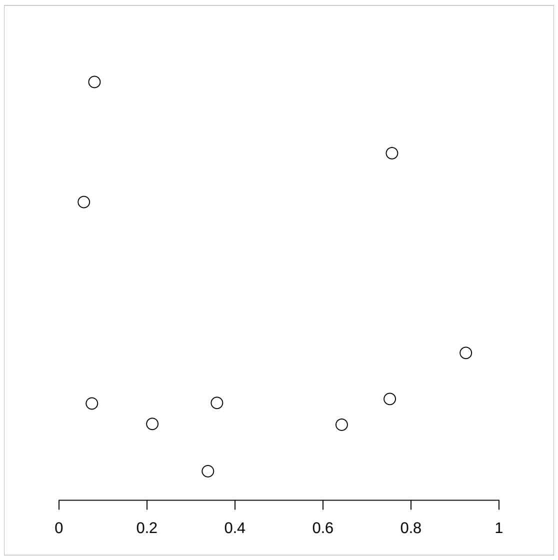

Figure 1: A simple **grid** scene that contains an x-axis.

```
> pushViewport(viewport(width=.8, height=.8,
+                       name="plot"))
> grid.draw(xag)
> grid.points(x, y, name="points")
```

Forcing and reverting a grid scene

For `grob`s that only generate their content at drawing time, the content is not recorded on the **grid** display list, so the content is not directly editable.

We can demonstrate this for the simple scene above with the `grid.ls()` function. There is an `"xaxis"` `gTree` (called `test`) on the display list, but the tick marks and labels that have been drawn are not visible (they only exist during drawing).

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

```
rect[GRID.rect.2]
xaxis[test]
points[points]
```

While we can modify the high-level description of the x-axis (see Figure 2) ...

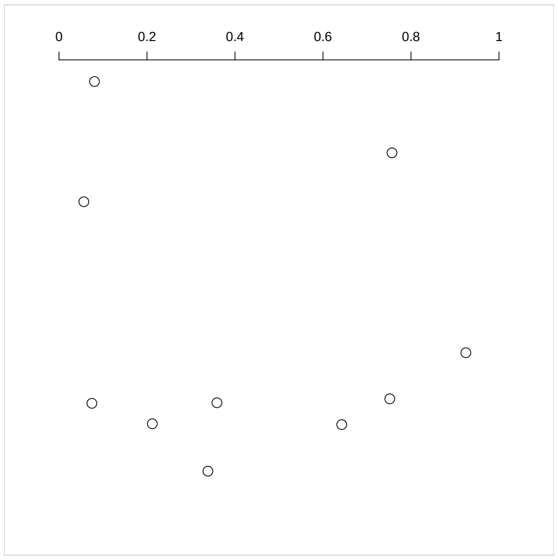

Figure 2: A high-level modification of the simple **grid** scene that contains an x-axis.

```
> grid.edit("test", main=FALSE)
```

... we cannot directly access the children of the `gTree`.[1]

```
> grid.edit("ticks", gp=gpar(col="red"))
```

```
Error in editDLfromGPath(gPath, specs, strict, grep, global, redraw) :
  'gPath' (ticks) not found
```

The new `grid.force()` function makes use of the new `makeContent()` hook to allow a scene like this to be "forced" so that `grob`s and `gTree`s are replaced with their on-the-fly content (if they generate their content on-the-fly).

The following code does not alter the appearance of the scene, but it does make the children of the `"xaxis"` accessible.

```
> grid:::grid.force()
```

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

[1]The `"xaxis"` class does provide the ability to modify its children via an `edits` argument, but that approach to solving this problem has not seen broad up-take, plus it does not provide the other benefits that the new `makeContent()` approach can provide.

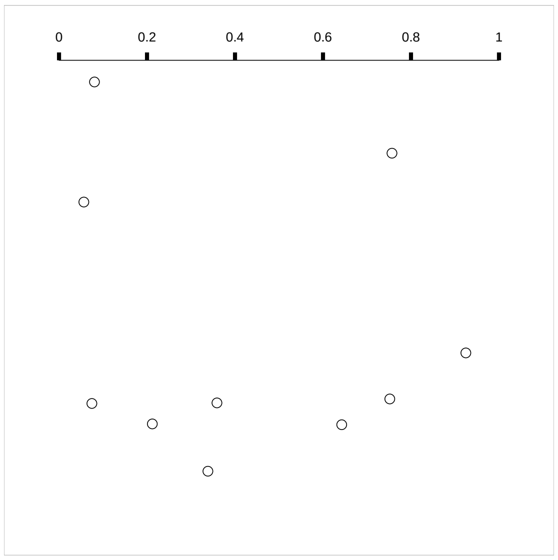

Figure 3: A low-level modification of the simple **grid** scene that contains an x-axis (once the scene has been "forced").

```
rect[GRID.rect.4]
forcedgrob[test]
  lines[major]
  segments[ticks]
  text[labels]
points[points]
```

So now we can directly modify the content of the axis (see Figure 3).

```
> grid.edit("ticks", gp=gpar(lwd=5, lineend="butt"))
```

On the downside, the x-axis is no longer actually an `"xaxis"` object—it is just a plain `gTree`—so the following edit has no effect.

```
> grid.edit("test", main=TRUE)
```

However, another new function, `grid.revert()` allows the previous "force" to be reverted (see Figure 4; notice that the low-level editing is *lost*).

```
> grid:::grid.revert()
```

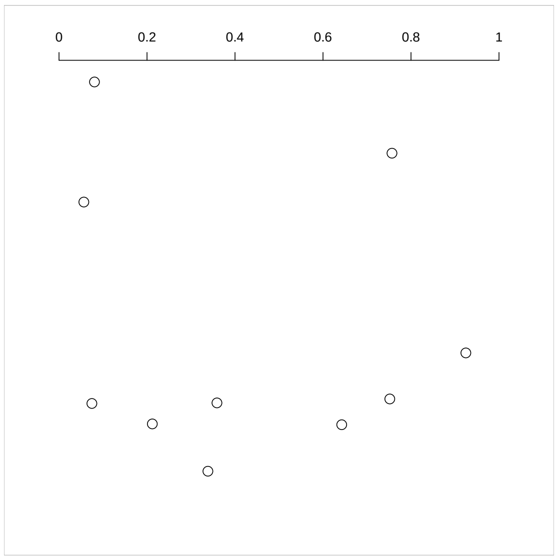

Figure 4: The simple **grid** scene that contains an x-axis after being "reverted" (after being "forced").

Now we are back to the situation that the children of the x-axis are no longer acessible ...

```
> grid.edit("ticks", gp=gpar(col="red"))
```

```
Error in editDLfromGPath(gPath, specs, strict, grep, global, redraw) :
  'gPath' (ticks) not found
```

... but we can once again make high-level changes to the axis (see Figure 5).

```
> grid.edit("test", main=TRUE)
```

Example 2: a `makeContext()` method

An example of a `gTree` that customises its drawing context, at drawing time, is a `"frame"` object (because it wants to push a viewport with a layout for its cells to occupy).

Here are the old `preDrawDetails()` and `postDrawDetails()` methods for the `"frame"` class ...

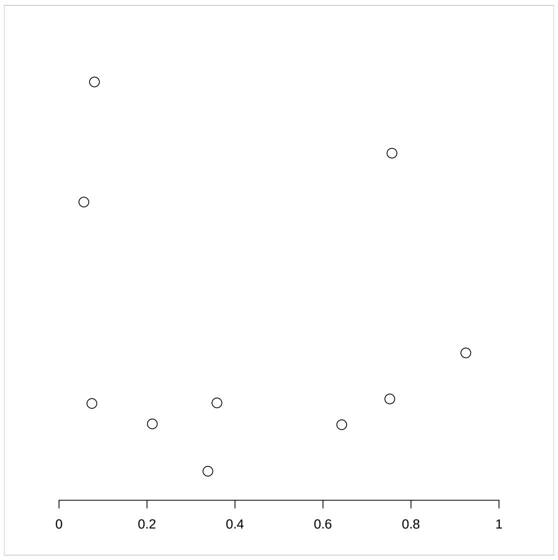

Figure 5: The simple **grid** scene that contains an x-axis after being "reverted" (after being "forced") and then modified.

```
preDrawDetails.frame <- function(x) {
  if (!is.null(x$framevp))
    pushViewport(x$framevp, recording=FALSE)
}

postDrawDetails.frame <- function(x) {
  if (!is.null(x$framevp))
    upViewport(recording=FALSE)
}
```

... and here is the new `makeContext()` method (this *replaces* the `preDrawDetails()` and `postDrawDetails()` methods, which are removed; the large and complex `gridList()` method is also be removed) ...

```
> grid:::makeContext.frame
```

```
function (x)
{
    if (!is.null(x$framevp)) {
        if (!is.null(x$vp)) {
            x$vp <- vpStack(x$vp, x$framevp)
        }
        else {
            x$vp <- x$framevp
        }
    }
    x
}
<bytecode: 0x3a24858>
<environment: namespace:grid>
```

The code is very similar, the only difference being that the `makeContext()` method only sets up the viewports to push, but does not push them (and we are no longer responsible for popping them afterwards) and it returns the modified `"frame"` grob.

The result of drawing a frame is unchanged. In the old set up, the `preDrawDetails()` method would get called to generate *and push* the `framevp` viewport to set up the correct drawing context (and the `postDrawDetails()` method would get called to *pop* the viewport after drawing; in the new set up, the `makeContext()` method updates the `vp` slot to include the `framevp` viewport, then the standard behaviour for `gTree`s takes care of pushing the viewports before drawing and popping them afterwards.

A new, and very similar, `makeContext()` method has been added for `"cellGrob"` objects, again leading to the removal of methods for `preDrawDetails()`, `postDrawDetails()`, and `gridList()`.

The following code generates a simple frame and draws it to show that drawing still works (see Figure 6).

```
> fg <- frameGrob(name="gf")
> fg <- packGrob(fg,
+                rectGrob(gp=gpar(fill="grey")),
+                width=unit(1, "null"))
> fg <- packGrob(fg,
+                textGrob("hi there"),
+                side="right")
```

```
> grid.newpage()
> grid.rect(gp=gpar(col="grey"))
> grid.draw(fg)
```

Benefits for `grid.ls()`

A `"frame"` `grob` does not generate content at drawing time, but it does generate drawing *context* at drawing time and this needs to be reflected in the output of

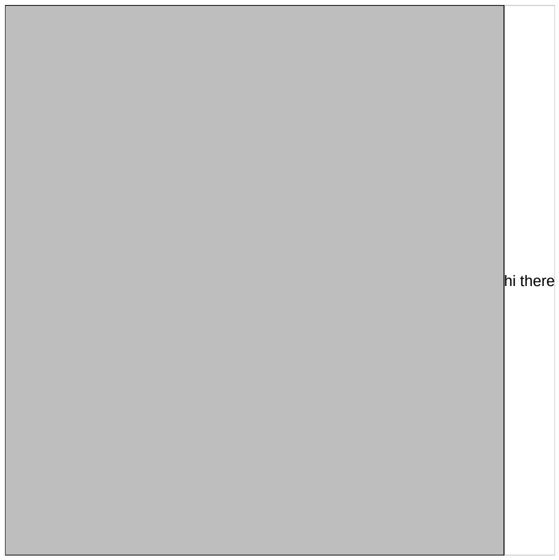

Figure 6: A simple **grid** scene that contains a frame (packed with a rectangle and a piece of text).

`grid.ls()`.

Prior to the `makeContext()` function, this meant that objects that customised the drawing context at drawing time, like `"frame"`s, had to have a special `gridList()` method. Now, `grid.ls()` can call `makeContext()` to determine the correct drawing context, *without* the need for special `gridList()` methods.

The following output shows that listing the viewports in this scene still works, *without* the need for `gridList()` methods, because the standard `grid.ls()` behaviour for `grob`s and `gTree`s now uses `makeContext()` to determine the correct drawing context for listing.

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

```
viewport[ROOT]
  rect[GRID.rect.13]
  viewport[GRID.VP.1]
    frame[gf]
      viewport[GRID.VP.4]
        cellGrob[GRID.cellGrob.9]
          rect[GRID.rect.8]
        upViewport[1]
      viewport[GRID.VP.3]
```

```
        cellGrob[GRID.cellGrob.11]
          text[GRID.text.10]
        upViewport[1]
    upViewport[1]
```

Example 3: a grob example

The "roundrect" class is an example of a simple grob (not a gTree) that calculates what to draw at drawing time (because it needs physical dimensions to know how to draw its corners).

The "roundrect" class had a preDrawDetails() method that set up a viewport where the rectangle is going to be drawn (with corresponding postDrawDetails() ...

```
preDrawDetails.roundrect <- function(x) {
  pushViewport(viewport(x$x, x$y, x$width, x$height, just=x$just),
               recording=FALSE)
}

postDrawDetails.roundrect <- function(x) {
  popViewport(recording=FALSE)
}
```

... and a drawDetails() method that generated a "polygon" grob to draw in that viewport ...

```
drawDetails.roundrect <- function(x, recording) {
    boundary <- rrpoints(x)
    grid.Call.graphics(L_polygon, boundary$x, boundary$y,
                       list(as.integer(seq_along(boundary$x))))
}
```

These methods have been replaced with a makeContext() method to build the viewport ...

```
> grid:::makeContext.roundrect
```

```
function (x)
{
    rrvp <- viewport(x$x, x$y, x$width, x$height, just = x$just,
        name = "rrvp")
    if (!is.null(x$vp)) {
        x$vp <- vpStack(x$vp, rrvp)
    }
    else {
        x$vp <- rrvp
    }
    x
}
<bytecode: 0x39f2fe0>
<environment: namespace:grid>
```

... and a `makeContent()` method to build the polygon ...

```
> grid:::makeContent.roundrect
```

```
function (x)
{
    boundary <- rrpoints(x)
    polygonGrob(boundary$x, boundary$y, name = x$name, gp = x$gp,
        vp = x$vp)
}
<bytecode: 0x3a6abe0>
<environment: namespace:grid>
```

Once again, the new methods are very similar to the old, though one important detail about the `makeContent()` method is that it must set the `gp` and `vp` slots of the `"polygon"` `grob` based on the values of these slots in the `"roundrect"` object (after calling the `makeContext()` method). This is necessary so that the `postDraw()` code will correctly undo the set up performed in `makeContext()`.

The following code generates a `"roundrect"` and draws it to show that drawing still works (see Figure 7).

```
> grid.newpage()
> grid.rect(gp=gpar(col="grey"))
> grid.roundrect(width=.8, height=.8, name="rr")
```

Implications for `grid.ls()`

One interesting side-effect of the new set up is that the viewport that a `"roundrect"` pushes in `makeContext()` is now visible to `grid.ls()`.

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

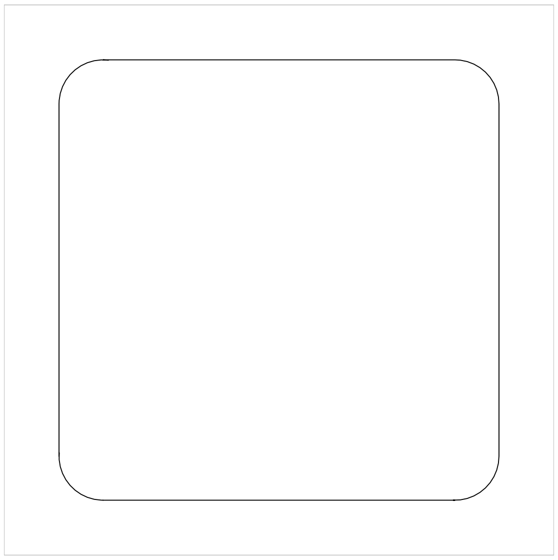

Figure 7: A simple **grid** scene that contains a `"roundrect"`.

```
viewport[ROOT]
  rect[GRID.rect.15]
  viewport[rrvp]
    roundrect[rr]
    upViewport[1]
```

This is an accurate reflection of the situation because the default behaviour for cleaning up a drawing context is to call `upViewport()` rather than `popViewport()`. So that viewport that got pushed when the `"roundrect"` was drawn is available to revisit (see the code below and Figure 8).

```
> downViewport("rrvp")
> grid.rect()
```

Implications for `grobX()`

A `"roundrect"` presents an interesting case for determining the dimensions or locations on the boundary of the `grob` (via functions like `grobX()`). This is a case where these calculations must take into account the drawing context for the `grob`; the viewport that was set up to draw a `"roundrect"` must be set up again when calculating, say, the left edge of the roundrect. This still works because the `makeContext()` hook has been made part of the standard pre-drawing context

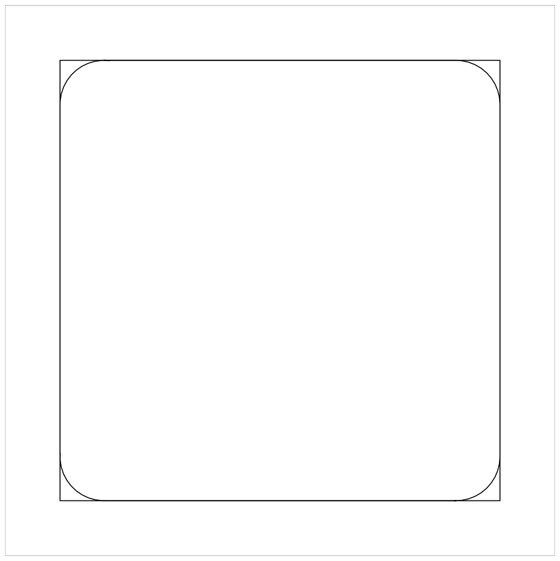

Figure 8: A simple **grid** scene that contains a `"roundrect"` after revisiting the viewport that was set up to draw the round rect and adding another rectangle.

set up (though some changes to C code were required to take account of the fact that the pre-drawing set up now *returns* a modified `grob`).

The following code demonstrates that these calculations still work (see Figure 9).

```
> grid.newpage()
> grid.rect(gp=gpar(col="grey"))
> grid.roundrect(width=.8, height=.8, name="rr")
> grid.circle(grobX("rr", "west"),
+             r=unit(2, "mm"),
+             gp=gpar(fill="black"))
```

Another `grob` example

The `"beziergrob"` class is a simple example of a custom `grob` that generates another sort of `grob` (in this case an `"xspline"`) at drawing time. Thus it had a very simple `drawDetails()` method ...

```
drawDetails.beziergrob <- function(x, recording=TRUE) {
    drawDetails(splinegrob(x))
```

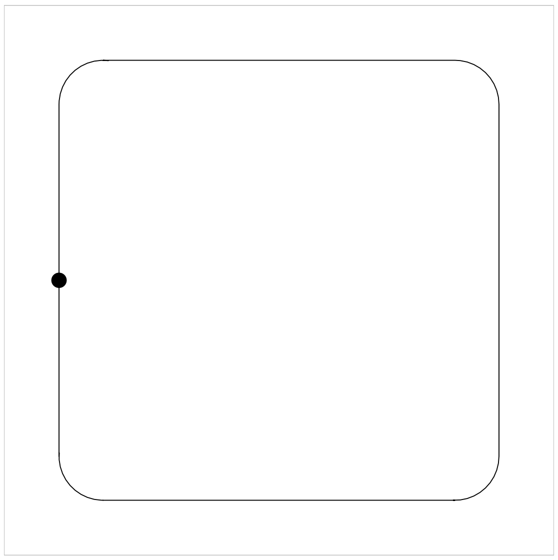

Figure 9: A simple **grid** scene that contains a `"roundrect"` with a dot drawn at the left edge of the `"roundrect"`.

```
}
```

A new `makeContent()` method is just as straightforward (though a small change was required to `splinegrob()` to make sure that it set the `vp` slot on the returned `"xspline"`, so that `postDraw()` clean up works correctly).

```
> grid:::makeContent.beziergrob

function (x)
{
    splinegrob(x)
}
<bytecode: 0x34a3e50>
<environment: namespace:grid>
```

The following code shows that drawing still works (see Figure 10) ...

```
> x <- c(0.2, 0.2, 0.8, 0.8)
> y <- c(0.2, 0.8, 0.8, 0.2)
> grid.bezier(x, y,
+             gp=gpar(lwd=3, fill="black"),
```

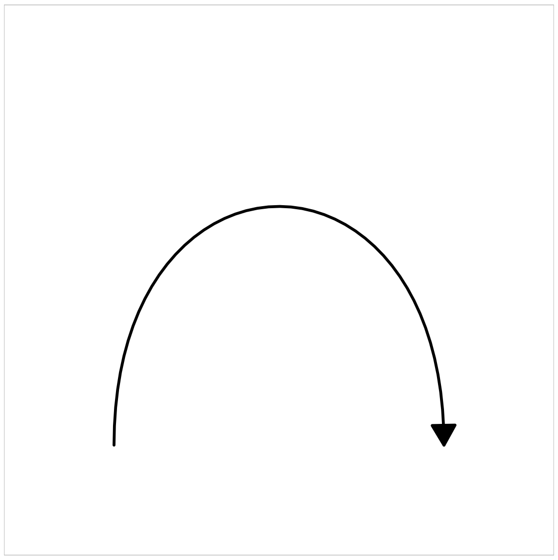

Figure 10: A simple **grid** scene that contains a `"beziergrob"`.

```
+              arrow=arrow(type="closed"),
+              name="testbezier")
```

... but we gain the fact that we can "force" these `"beziergrob"` `grob`s to be `"xspline"` `grob`s (which helps in places like **gridSVG**, which then only have to worry about handling `"xspline"` `grob`s and can ignore the existence of `"beziergrob"` `grob`s).

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

```
beziergrob[testbezier]
```

After a `grid.force()`, the `"beziergrob"` becomes an `"xspline"`.

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

```
forcedgrob[testbezier]
```

```
> class(grid.get("testbezier"))
```

```
[1] "forcedgrob" "xspline"    "grob"       "gDesc"
```

A gTree example

The "curve" class is an interesting case. The old `drawDetails()` method looks straightforward ...

```
drawDetails.curve <- function(x, ...) {
    grid.draw(calcCurveGrob(x, x$debug))
}
```

... and the `makeContent()` method is also straightforward ...

```
> grid:::makeContent.curve

function (x)
{
    calcCurveGrob(x, x$debug)
}
<bytecode: 0x37dcb58>
<environment: namespace:grid>
```

... however, this hides the fact that the result of `calcCurveGrob()` is sometimes a `glist`, not just a single `grob`. That will cause a problem for drawing because a "curve" is just a simple `grob`, and the drawing behaviour for a `grob` is to just call `drawDetails()` on the result of `makeContent()` (and that will draw NOTHING for a `glist`).

The complete solution in this case requires making a "curve" a `gTree` *and* modifying `calcCurveGrob()` so that it returns a `gTree`. That way `calcCurveGrob()` can return one, two, or more children AND the children will all get drawn because the drawing behaviour for a `gTree` is to draw all of its children (after the call to `makeContent()`).

The following code just shows that drawing a "curve" still works (see Figure 11) ...

```
> grid.curve(.2, .2, c(.2, .8), .8,
+            gp=gpar(lwd=3, fill="black"),
+            arrow=arrow(type="closed"),
+            name="curve")
```

... and if we `grid.force()` the scene we can see that the result is a `gTree`, which can have more than one child.

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

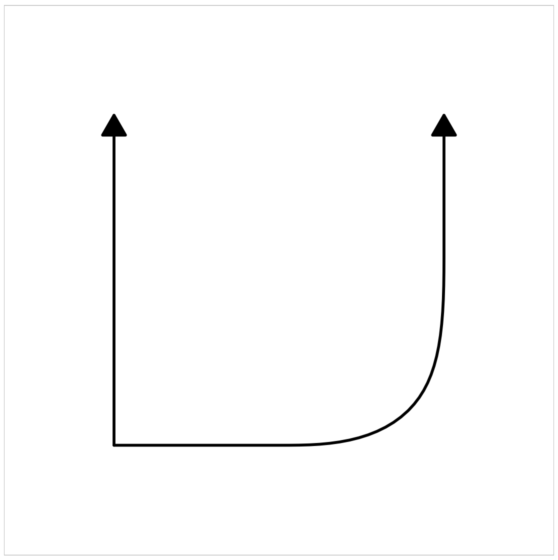

Figure 11: A simple **grid** scene that contains a `"curve"` `grob`.

```
curve[curve]
```

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

```
forcedgrob[curve]
  segments[segment]
  xspline[xspline]
```

Yet another `grob` example

The `"functiongrob"` class really is a simple case. The `drawDetails()` method was simple ...

```
drawDetails.functiongrob <- function(x, ...) {
    xy <- genXY(x)
    grid.lines(xy$x, xy$y, default.units=x$units)
}
```

... and the `makeContent()` method is simple (the only change being to make sure that the returned `grob` has the right `vp` slot) ...

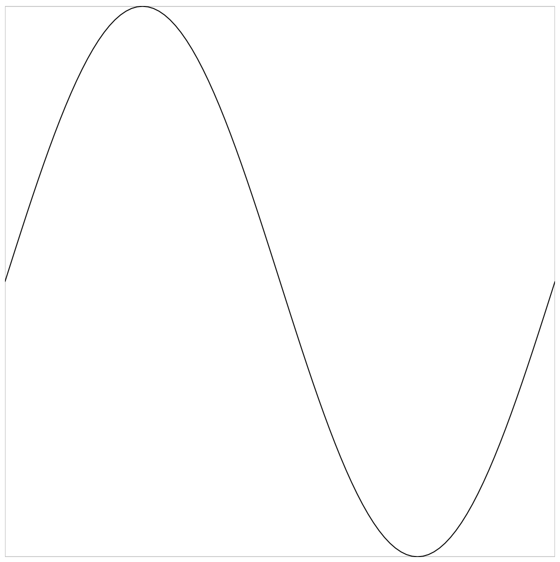

Figure 12: A simple **grid** scene that contains a `"function"` `grob`.

```
> grid:::makeContent.functiongrob
```

```
function (x)
{
    xy <- genXY(x)
    linesGrob(xy$x, xy$y, default.units = x$units, name = x$name,
        gp = x$gp, vp = x$vp)
}
<bytecode: 0x2eb14f0>
<environment: namespace:grid>
```

The following code shows that a `"functiongrob"` stills draws properly (see Figure 12) ...

```
> pushViewport(viewport(xscale=c(0, 2*pi), yscale=c(-1, 1)))
> grid.function(function(x) list(x=x, y=sin(x)),
+               name="testfunction")
```

... and we can "force" a `"functiongrob"` into its underlying `"lines"` grob ...

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

```
functiongrob[testfunction]
```

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

forcedgrob[testfunction]

> class(grid.get("testfunction"))

[1] "forcedgrob" "lines"      "grob"       "gDesc"
```

A tricky `grob` example

The `"recordedGrob"` class cannot be reliably converted to the new design.

The old `drawDetails()` method shows that at drawing time, a `"recordedGrob"` executes arbitrary R code. This class is designed purely for producing graphical output as a side-effect. Thus it keeps its `drawDetails()` method, you can never directly access the `grob`s that it draws, and you cannot "force" a `"recordedGrob"`.

```
drawDetails.recordedGrob <- function(x, recording) {
  eval(x$expr, x$list, getNamespace("grid"))
}
```

For completeness, a new `grob` class (actually it's a `gTree` class) has been created to provide an analogous `grob` that works with a `makeContent()` method. This class is called `"delayedgrob"` and it has exactly the same arguments as a `"recordedGrob"`, but with the additional requirement that the `expr` should return a `grob` (or a `glist`). The `makeContent()` method for this class is shown below.

```
> grid:::makeContent.delayedgrob

function (x)
{
    grob <- eval(x$expr, x$list, getNamespace("grid"))
    if (is.grob(grob)) {
        children <- gList(grob)
    }
    else if (is.gList(grob)) {
        children <- grob
    }
    else {
```

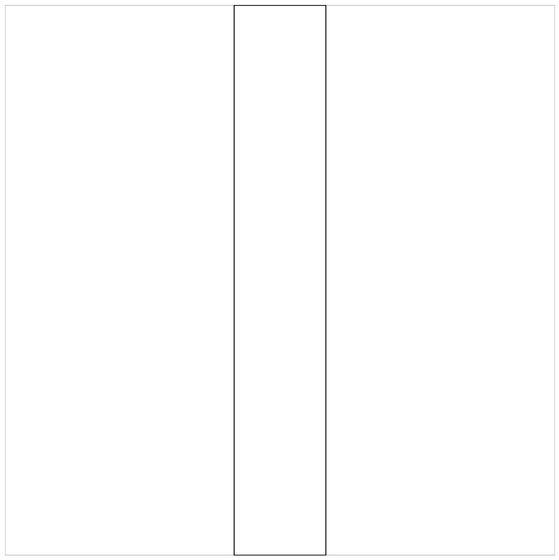

Figure 13: A simple **grid** scene that contains a `"delayedgrob"`.

```
        stop("'expr' must return a grob or gList")
    }
    x <- setChildren(x, children)
    x
}
<bytecode: 0x34aa268>
<environment: namespace:grid>
```

An example usage of this class is demonstrated in the following code. The expression performs a calculation (which needs to happen at drawing time and be recalculated on redraw) and then generates a `grob` (see Figure 13).

```
> grid.delay({
+              w <- convertWidth(unit(1, "inches"), "npc")
+              rectGrob(width=w, name="delayedrect")
+            },
+            list(),
+            name="delayed")
```

The generated `grob` can be made visible and accessible if we "force" the `"delayedgrob"` ...

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

```
delayedgrob[delayed]
```

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

```
forcedgrob[delayed]
  rect[delayedrect]
```

... (although the result only draws a rectangle 1 inch wide in the context when `grid.force()` was called, e.g., as long as the graphics device is not resized).

This is essentially a slightly simpler way of creating a new `grob` with a `makeContent()` method, without having to write a special constructor or `makeContent()` method.

Implications for gridSVG

The **gridSVG** package can take advantage of the new design by simply calling `grid.force()` before exporting a **grid** scene because the result of `grid.force()` will tend to be vanilla `grob`s and `gTree`s and so **gridSVG** only has to know how to export vanilla `grob`s and `gTree`s.

This reduces the complexity of the **gridSVG** package by removing several special-case methods (e.g., `primToDev.xaxis()`, `primToDev.yaxis`, `grobToDev.roundrect()`, `grobToDev.frame()`, and `grobToDev.cellGrob()`).

There are also flow on effects for **gridSVG** because other packages do not have to write their own special `primToDev()` or `grobToDev()` methods to work with **gridSVG** (e.g., see section below on **grImport**).

*The changes described in this section are available in the `R2.16` branch of the **gridSVG** package on R-Forge (`https://r-forge.r-project.org/projects/gridsvg/`).*

Implications for grImport

```
> library(grImport)
```

A number of `grob`s that are created when drawing an imported vector image using **grid** have a `drawDetails()` method (because they need to figure out the physical size of the region that they are drawing into). This provides opportunities for `makeContent()` methods instead.

A `"picstroke"` object is a line from an image that needs to figure out its line thickness and line style at drawing time, hence the old `drawDetails()` method ...

```
drawDetails.picstroke <- function(x, recording) {
    lwd <- convertWidth(unit(x$lwd, "native"), "bigpts",
                        valueOnly=TRUE)
    lty <- fixLTY(x$lty, x$lwd)
    grid.polyline(x$x, x$y,
                  default.units=x$default.units,
                  id.lengths=x$id.lengths,
                  gp=gpar(lwd=lwd, lty=lty, col=x$col, fill=NA))
}
```

This is an interesting case because it needs to set `gp` (part of the drawing context) at drawing time. This cannot be easily converted to a `makeContent()` method that just calls `polygonGrob()` instead of `grid.polygon()` because the `gp` settings in the resulting `grob` will not be enforced.

The solution is for a `"picstroke"` object to be a customised `gTree` (not just a customised `grob`). Then the `makeContent()` method can call `polygonGrob()` to generate a *child* for the `gTree`. This child will then be drawn from scratch, which includes enforcing its `gp` settings.

```
> grImport:::makeContent.picstroke

function (x)
{
    lwd <- convertWidth(unit(x$lwd, "native"), "bigpts", valueOnly = TRUE)
    lty <- fixLTY(x$lty, x$lwd)
    child <- polylineGrob(x$x, x$y, default.units = x$default.units,
        id.lengths = x$id.lengths, gp = gpar(lwd = lwd, lty = lty,
            col = x$col, fill = NA))
    setChildren(x, gList(child))
}
<environment: namespace:grImport>
```

Very similar changes are made for `"symbolstroke"` and `"symbolfill"` objects (both need to generate a `grob` representing multiple copies of the imported image at drawing time) and `"picturetext"` objects (need to determine text size at drawing time).

These changes make no difference to the output that **grImport** functions like `grid.picture()` produce, but they do mean that it is possible to “force” a scene that contains **grImport** objects to reduce the scene to standard **grid** objects, which means that it is possible for the **gridSVG** package to export a scene that contains **grImport** objects without having to write any special code.

The following code demonstrates a simple scene consisting of an imported “flower” image (see Figure 14) and shows that forcing the scene results in standard **grid** objects.

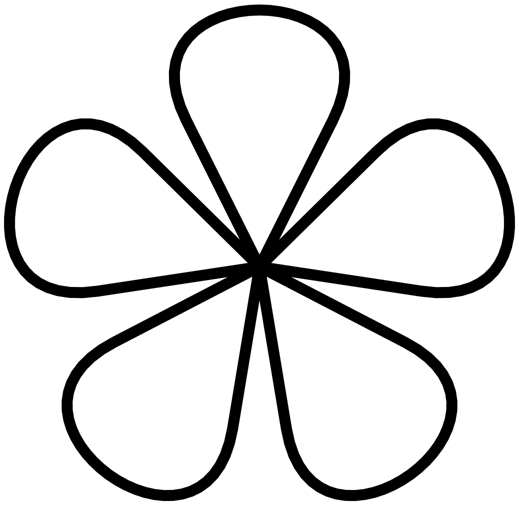

Figure 14: A simple **grid** scene that contains a "picstroke" object from the **grImport** package.

```
> flower <- readPicture("flower.xml")
```

```
> grid.picture(flower)
```

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

```
picture[GRID.picture.35]
  picstroke[GRID.picstroke.36]
  picstroke[GRID.picstroke.37]
  picstroke[GRID.picstroke.38]
  picstroke[GRID.picstroke.39]
  picstroke[GRID.picstroke.40]
```

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

```
forcedgrob[GRID.picture.46]
  forcedgrob[GRID.picstroke.47]
    polyline[GRID.polyline.57]
  forcedgrob[GRID.picstroke.48]
```

```
    polyline[GRID.polyline.58]
  forcedgrob[GRID.picstroke.49]
    polyline[GRID.polyline.59]
  forcedgrob[GRID.picstroke.50]
    polyline[GRID.polyline.60]
  forcedgrob[GRID.picstroke.51]
    polyline[GRID.polyline.61]
```

The changes described in this section are available in version 0.9 of the ***grImport*** *package on R-Forge (`https://r-forge.r-project.org/projects/grimport/`).*

Implications for gridGraphviz

When drawing graph edges, the **gridGraphviz** package generates special `"edgegrob"` objects, with a special `drawDetails()` method, to draw the edges of the graph.

It turns out that there is no need for the content of the edges to be generated at drawing time—they can be generated as a normal `gTree`—so the only change to this package was to remove the `drawDetails()` method (no need to add a `makeContent()` method).

The changes described in this section are available in version 0.2 of the ***gridGraphviz*** *package on R-Forge (`https://r-forge.r-project.org/projects/gridgraph/`).*

Implications for gridDebug

The **gridDebug** package produces **grid** scene graphs using **gridGraphviz**, including an option to export the scene graph as SVG with tooltips. This means exporting graph edges from **gridGraphviz**, which consist of Bezier curves and straight lines. This meant special `primToDev()` methods to get the export working, but now bezier curves are automatically handled by **gridSVG** (via `grid.force()`) and **gridGraphviz** no longer has special `"edgegrob"` `grob`s, so *both* of the `primToDev()` methods can be removed.

The changes described in this section are available in the `R2.16` branch of the ***gridDebug*** *package on R-Forge (`https://r-forge.r-project.org/projects/griddebug/`).*

Implications for gtable

The **gtable** package has to work pretty hard, via `grid.draw()` methods and `preDrawDetails()` methods, to get the behaviour it wants. The especially hard part was getting a `"gTableChild"` object to push its `wrapvp` *before* its own standard `vp` (so that the `"gTableChild"` was positioned correctly within its parent's layout; the `wrapvp` is a viewport with `layout.pos.col` and `layout.pos.row` settings).

The `grid.draw()` methods and `preDrawDetails()` methods have been completely replaced with (much simpler and easier to understand) `makeContext()` and `makeContent()` methods.

A `"gtable"` now includes its `layoutvp` as part of the drawing context set up ...

```
> gtable:::makeContext.gtable
```

```
function (x)
{
    layoutvp <- viewport(layout = gtable_layout(x), name = x$name)
    if (is.null(x$vp)) {
        x$vp <- layoutvp
    }
    else {
        x$vp <- vpStack(x$vp, layoutvp)
    }
    x
}
<environment: namespace:gtable>
```

A `"gtable"` now generates a list of `"gTableChild"` objects as its content (children) ...

```
> gtable:::makeContent.gtable
```

```
function (x)
{
    children_vps <- mapply(child_vp, vp_name = vpname(x$layout),
        t = x$layout$t, r = x$layout$r, b = x$layout$b, l = x$layout$l,
        clip = x$layout$clip, SIMPLIFY = FALSE)
    x$grobs <- mapply(wrap_gtableChild, x$grobs, children_vps,
        SIMPLIFY = FALSE)
    setChildren(x, do.call("gList", x$grobs[order(x$layout$z)]))
}
<environment: namespace:gtable>
```

A `"gTableChild"` now includes its `wrapvp` as part of the drawing context set up (with the `wrapvp` coming *before* its own `vp`) ...

```
> gtable:::makeContext.gTableChild
```

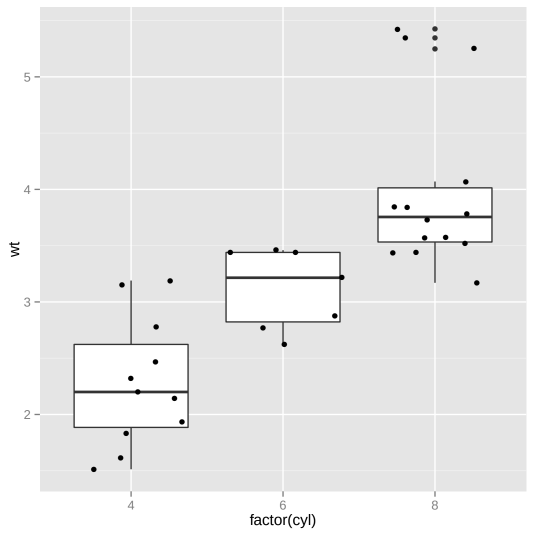

Figure 15: A plot produced by `qplot()` from the **qqplot2** package (which uses the **gtable** package for layout).

```
function (x)
{
    if (is.null(x$vp)) {
        x$vp <- x$wrapvp
    }
    else {
        x$vp <- vpStack(x$wrapvp, x$vp)
    }
    NextMethod()
}
<environment: namespace:gtable>
```

The following code demonstrates that drawing still works as normal (via a **ggplot2** plot; see Figure 15).

```
> library(ggplot2)
```

```
> qplot(factor(cyl), wt, data = mtcars, geom=c("boxplot", "jitter"))
```

The benefit of this change is not just tidier and cleaner code. It also means that it is now possible to "force" a `"gtable"` object. For example, the following code shows that only the top-level "gtable" is visible after drawing the plot in Figure 15.

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

```
gtable[layout]
```

However, after a call to grid.force(), the entire contents of the plot are visible ...

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

```
forcedgrob[layout]
  forcedgrob[background.1-5-6-1]
  forcedgrob[spacer.4-3-4-3]
  forcedgrob[panel.3-4-3-4]
    gTree[grill.gTree.235]
      rect[panel.background.rect.228]
      polyline[panel.grid.minor.y.polyline.230]
      polyline[panel.grid.major.y.polyline.232]
      polyline[panel.grid.major.x.polyline.234]
    gTree[boxplots.gTree.222]
      gTree[geom_boxplot.gTree.200]
        segments[GRID.segments.193]
        gTree[geom_crossbar.gTree.199]
          gTree[geom_polygon.gTree.197]
            polygon[GRID.polygon.196]
          segments[GRID.segments.198]
      gTree[geom_boxplot.gTree.209]
        segments[GRID.segments.202]
        gTree[geom_crossbar.gTree.208]
          gTree[geom_polygon.gTree.206]
            polygon[GRID.polygon.205]
          segments[GRID.segments.207]
      gTree[geom_boxplot.gTree.220]
        points[geom_point.points.211]
        segments[GRID.segments.213]
        gTree[geom_crossbar.gTree.219]
          gTree[geom_polygon.gTree.217]
            polygon[GRID.polygon.216]
          segments[GRID.segments.218]
    points[geom_jitter.points.224]
    zeroGrob[panel.border.zeroGrob.225]
```

```
forcedgrob[axis-l.3-3-3-3]
  zeroGrob[axis.line.y.zeroGrob.245]
  forcedgrob[axis]
    forcedgrob[axis.1-1-1-1]
    forcedgrob[axis.1-2-1-2]
forcedgrob[axis-b.4-4-4-4]
  zeroGrob[axis.line.x.zeroGrob.239]
  forcedgrob[axis]
    forcedgrob[axis.1-1-1-1]
    forcedgrob[axis.2-1-2-1]
forcedgrob[xlab.5-4-5-4]
forcedgrob[ylab.3-2-3-2]
forcedgrob[title.2-4-2-4]
```

This means that we can edit pieces of the plot and it means that **gridSVG** can export **ggplot2** plots without having to write special methods.

*The changes described in this section are available in a fork of the **gtable** package on github (`https://github.com/pmur002/gtable`).*

Summary of tips for new `grob` classes

- Create a `gTree` class - it's just easier. Only create a `grob` class if you really know what you are doing and enjoy pain; in particular, if your custom `grob` wants to generate more than one `grob` at drawing time, use a `gTree` (and modify the children of the `gTree`).
- ONLY modify `vp` (and `childrenvp` for a `gTree`) in a `makeContext()` method.
- ONLY modify content (including `children` for a `gTree`) in a `makeContent()` method.
- Make sure that `makeContext()` and `makeContent()` are cumulative; in particular, make sure that the `grob` returned by `makeContent()` ends up with the same `vp` slot as the `grob` returned by `makeContext()` (otherwise the drawing context clean up is going to leave a mess).
- A `makeContent()` method for a `grob` should return a `grob` with a `drawDetails()` method (typically a **grid** primitive). [because the only thing that is going to happen after the call to `makeContent()` is a call to `drawDetails()`]
- A `makeContent()` method for a `gTree` should return a `gTree` with the `grobs` to draw as `children` of the `gTree`. [because what will happen after the call to `makeContent()` is that `grid.draw()` will be called on the `children` of the `gTree`]

Recursive content

`drawDetails()` methods could call `grid.draw()` and start from the top again (i.e., call other `drawDetails()` methods, which allows high level `grob`s to call intermediate level `grob`s, which call primitive `grob`s (which do the actual drawing).

Can a `makeContent()` method generate a `grob` that has a `makeContent()` method?

I don't think so ...

... and that may be bad ...

... **UNLESS** you use a `gTree`, in which case, the children get `grid.draw()`n, which means calling `makeContent()` (among other things), so you can get recursion (high level building on intermediate level) after all!

Recursive "forcing"

A `makeContent()` method for a `grob` should not produce a `grob` that itself has a `makeContent()` method because `grid.force()` will not recurse.

This should be ok for `gTree`s though because the children of a "forced" `gTree` are themselves "forced".

Why do some `grob`s still have `drawDetails()`?

The `makeContent()` hook provides an alternative way for defining *customised* `grob`s. The fundamental **grid** primitives still define and use `drawDetails()` methods.

This makes sense (I think) because `drawDetails()` methods are really actions at drawing time that only have side-effects (of drawing stuff usually). Creating a customised `grob` has become a matter of defining a `makeContent()` method, which is essentially a conversion of the custom `grob` into a basic **grid** primitive (or a `gTree` containing **grid** primitives). Ulitmately, something has to actually get drawn, and that is the job of the **grid** primitives.

So this redesign consists of looking at existing `grob`s and deciding whether they are basic primitives (in which case they keep their `drawDetails()` method so that they will draw something at drawing time) or whether they need to be converted at drawing time into basic primitives (via a new `makeContent()` method).

The `"clip"` primitive is a slightly interesting case. It does not *draw* anything, but it is still something that needs a `drawDetails()` method because its only action at drawing time is to produce the side-effect of modifying the **grid** clipping region.

The `"labelgrob"` class is *really* interesting. This is used in `showGrob()` when adding debugging labelling to a **grid** scene. The exceptional feature about this is that debugging labelling is designed NOT to be recorded on the **grid** display list. In other words, a `"labelgrob"` really is designed to only have the side-effect of producing graphics output. It is not *supposed* to be visible to `grid.ls()` and it is not *supposed* to be able to be "forced". So this is another case where a `drawDetails()` method is actually appropriate.

Options not pursued (and why)

One early idea was to simply provide a single new hook that allows an entire rebuild of a `grob` at drawing time (both drawing context and content to draw). This would be very simple and provide ultimate flexibility, but it does not separate drawing context from content to draw. For example, it does not make it easy to write the code that evaluates `"grobwidth"` units because, at best, you have to reconstruct the entire grob content in order to just pick out the new drawing context. Another problem situation is a `grob` that wants its drawing context correct so that it can determine dimensions or locations of its content; in this case, the single hook would have to generate the drawing context *and enforce it* in order to generate the content correctly. This would make the single hook more complex than necessary, especially in cases where a `grob` does not want to do anything special with the drawing context, just generate special content!

In summary, it was deemed more useful to have separate hooks for generated context and for generating content so that developers could provide just one of those if necessary and so that other functions that need to query a `grob` can just request one of context or content if necessary.

Mixing viewports and `vpPaths`

A prerequisite for the new design is that a new hook that generates a new drawing context must be able to not only *add* new viewports, but also *combine* new viewports with the existing `vp` slot of a `grob`.

The canonical example comes from **gtable** (which has a similar goal to `frames` in **grid**). The basic set up is that we want to add a child `grob` to a parent `grob`, where the parent has a layout and the child will occupy some part of the layout. This means that the parent has to provide a viewport with a layout and the child has to provide a viewport that is located within that parent layout *and that child viewport has to be the first viewport pushed by the child* (otherwise the layout positioning within the parent layout will not work!). If the child has a `vp` slot, the new drawing context for the child has to create a new viewport (to position the child within the parent layout) and *then* apply the child's pre-existing `vp` viewport.

This "combining" of viewports is handled fine by the concept of `vpStack`, `vpList`, and `vpTree` objects *as long as the child's* **`vp`** *slot is a viewport.* However, the child's `vp` slot could also be a `vpPath` and that was not previously supported.

This combination of viewports with `vpPath`s is now allowed within `vpStack`, `vpList`, and `vpTree` objects. There are situations where such a construct does not really make sense, but the existing viewport checking should handle this (by reporting an error). However, in other situations, this is a perfectly reasonable thing to do.

```
vp <- viewport(width=.5, height=.5, name="a")

grid.newpage()
pushViewport(vp)
upViewport()

# Go down to an existing viewport then push a
# further viewport before drawing
grid.rect(vp=vpStack(vpPath("a"), vp),
          gp=gpar(col="red"))

# Push a viewport and then visit it!
# (a bit pathological)
grid.newpage()
grid.rect(vp=vpList(vp, vpPath("a")))
```

This change was committed to `r-devel` as `r60838`.

Default `vp` behaviour is `upViewport()`

A consequnce of this redesign is that the only (recommended) way to modify the drawing context for a `grob` (the viewports that a `grob` pushes before drawing), is to modify the `vp` slot of the `grob` (via a `makeContext()` method).

The default behaviour of the `vp` slot, prior to this redesign, was to call `pushViewport()` before drawing the `grob` and then `popViewport()` after drawing the `grob` (unless the `vp` slot was a `vpPath`, in which case the behaviour was a `downViewport()` before and an `upViewport()` after).

That behaviour for the `vp` slot means that it is impossible to customise the drawing context for a `grob` and have the custom drawing context retained after the `grob` has been drawn (the drawing context is lost because of the `popViewport()`).

This lead to a decision to *change* the default behaviour for `grob`s so that `pushViewport()` is called before drawing the `grob` and *`upViewport()` is always called after drawing the `grob`.*

This change allows the custom drawing context to be retained for a `grob`. It will mean more viewports hanging around in general, but hopefully there will be no major negatives (and hopefully no negative impact on existing **grid**-based code).

A simple demonstration of the change is shown below. First we define a very simple viewport ...

```
> vp <- viewport(name="test")
```

... now, prior to the redesign, if we drew a rectangle with this viewport specified in the `vp` slot, the viewport would be pushed, the rectangle drawn, then the viewport popped ...

```
> grid.newpage()
> grid.rect(vp=vp)
> current.vpTree()
viewport[ROOT]
```

... so the viewport is not retained and cannot be revisited. Interestingly, `grid.ls()`, somewhat misleadingly, reports the viewport (because it is in the `vp` slot of the rectangle `grob`) ...

```
> grid.ls(viewports=TRUE, fullNames=TRUE)
viewport[ROOT]
  viewport[test]
    rect[GRID.rect.2]
    upViewport[1]
```

... even though it cannot be revisited ...

```
> downViewport("test")
Error in grid.Call.graphics(L_downviewport, name$name, strict) :
  Viewport 'test' was not found
```

After the redesign, the viewport is retained and can be revisited (and `grid.ls()` is now telling the truth!) ...

```
> grid.newpage()
> grid.rect(vp=vp)
> current.vpTree()
```

```
viewport[ROOT]->(viewport[test])
```

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

```
viewport[ROOT]
  viewport[test]
    rect[GRID.rect.257]
    upViewport[1]
```

```
> downViewport("test")
```

Backward compatibility

The new hook functions are *added* to the standard behaviour of **grid** `grob`s. This means that the old approach of `drawDetails()`, `preDrawDetails()`, and `postDrawDetails()` methods will still work, so no existing code *should* break.

The idea is that developers of new `grob`s will use the new hook functions going forward. The transition path for developers of existing packages is to *replace* their old `drawDetails()`, `preDrawDetails()`, and `postDrawDetails()` methods with new hook functions.

One exception to the nothing-should-break rule is that there is a possibility of unforseen consequences for existing code with the change from `popViewport()` to `upViewport()` when cleaning up the drawing context based on the `x$vp` slot of a `grob`.