# A GUI-Builder Approach to **grid** Graphics

Paul Murrell

October 30, 2026

Grid contains a lot of support for locating, sizing, and arranging graphical components on a device and with respect to each other. However, most of this support relies on *either* the parent object dictating both location and size (layouts) *or* the child dictating both location and size.

Some sorts of arrangements are more conveniently handled by having the parent dictate the location, but letting the child dictate the size. This is the situation for GUI-builders (software which forms arrangements of GUI components or widgets). The approach taken by (many ?) GUI-builders is to allow the user to create a parent *frame* and then *pack* widgets into this frame. The frame locates and arranges the children with the help of hints such as "place this widget at the bottom of the frame", and the children dictate what size they would like to be.

This document describes a first attempt at such an interface for arranging Grid graphical objects.

## The `"frame"` grob

You can create a `"frame"` graphical object using the function `frameGrob()`. You must assign the result to a variable in order to pack grobs into it.

```
> gf <- frameGrob()
```

## The `packGrob()` function

Having created a frame, you can then pack other graphical objects into it using the `packGrob()` function. This function has a complex interface which allows for a variety of methods of packing graphical objects. The required arguments are:

`frame` is a `"frame"` object created by `grid.frame`.

`grob` is the grob to pack into the frame.

The remaining arguments specify where the grob is located in the frame and possibly how much space the grob should occupy. The frame is effectively just a layout; you can add grobs

to existing rows and columns or you can append the grob in a new row and/or column. If the grob is added to an existing row then the height of that row becomes the maximum of the new height and the previous height. If the row is new then it just gets the specified height. Similar rules apply for column widths.

`col` is the column to put the grob in. This can be 1 greater than the existing number of columns (in which case a new column is added).

`row` is like `col` but for rows.

`col.after` specifies that the grob should be put in a new column inserted between `col.after` and `col.after + 1`.

`col.before` specifies that the grob should be put in a new column inserted between `col.before` and `col.before + 1`.

`row.after` **and** `row.before` do what you would expect.

`side` specifies which side to append the new grob to. The valid values are `"left"`, `"right"`, `"bottom"`, and `"top"`.

`width` is the width of the row that the grob is being packed into. If this is not given then the grob supplies the width.

`height` is like `width` but for rows.

It is possible to modify this default behaviour. For example, it is possible to add a grob to a row and force that row to have the specified height by setting `force.height=TRUE` (and similarly for column widths). It is also possible to pack a graphical object into several rows or columns at once (although you cannot simultaneously affect the heights or widths of those rows and columns).

The result of this function is the modified frame, so you must assign the result to a variable.

```
> gf <- packGrob(gf, textGrob("Hello frame!"))
```

## "grobwidth" and "grobheight" units

A `"frame"` object allows a grob to specify its size by making use of `"grobwidth"` and `"grobheight"` units. These units may, of course, be used outside of frames too so their use is described here.

Consider a simple example where I want to draw a rectangle around a piece of text. I can get the size of the piece of text from the `"text"` grob as follows:

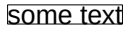
```
> st <- grid.text("some text")
> grid.rect(width = unit(1, "grobwidth", st),
+           height = unit(1, "grobheight", st))
```

You could do the same thing with simple "strwidth" and "strheight" units, but "grobwidth" and "grobheight" give you a lot more power. The biggest gain is that you can get the size of other objects besides pieces of text (more on that soon). Another thing you can do is provide a "reference" to a grob rather than the grob itself; you do this by giving the name of a grob. What this does is make the unit "dynamic" so that changes in the grob affect the unit. The following is a dynamic version of the previous example.

```
> grid.text("some text", name = "st")
> grid.rect(width = unit(1, "grobwidth", "st"),
+           height = unit(1, "grobheight", "st"))
```

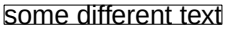
Now watch what happens if I modify the text grob named "st":

```
> grid.edit("st", gp = gpar(fontsize = 20))
>
```




Similarly, I can change the text itself:

```
> grid.edit("st", label="some different text")
>
```




## The widthDetails and heightDetails generic functions

The calculation of "grobwidth" and "grobheight" units is a bit complicated, but fortunately most of it is automated. The simple part is that a grob provides a normal "unit"

object to express its width or height. The complication comes because that "unit" object has to be evaluated in the correct context; in particular, if the grob has a non-NULL vp argument then those viewports have to be pushed so that the size of the grob is the size it would be when it is drawn. This is achieved by calling the preDrawDetails() function for the grob and in most cases what happens by default will be correct. The thing to avoid is having any viewport operations in a drawDetails() method for your grob; they should go in a preDrawDetails() method.

All that needs to be written (usually) are the functions that provide the "unit" objects. These functions need to be widthDetails and heightDetails methods.

The default methods return unit(1, "null") so your grob will be of this size unless you write your own methods.

The classic example methods are those for "text" grobs; these return unit(1, "mystrwidth", <text grob label>) and unit(1, "mystrheight", <text grob label>) respectively.

The other very important examples of these methods are those for "frame" grobs. These return the sum of the widths (heights) of the columns (rows) of the layout that has been built up by packing grobs into the frame. This means that when a "frame" grob is packed within another "frame" grob the parent automatically leaves enough room for the child.

Another useful pair of examples are those for "rect" grobs. These methods make use of the absolute.size function. When a grob is asked to specify its size, it makes sense to respond with the grob's width and height if the grob has an "absolute" size (e.g., "inches", "cm", "lines", etc; i.e., the grob knows exactly how big itself is). On the other hand, it does not make sense to respond with the grob's width and height if the grob has a "relative" size (e.g., "npc" or "native"; i.e., the grob needs to know about it's parent's size before it can figure out its own). The absolute.size function leaves absolute units alone, but converts relative units to "null" units (i.e., the child says to the parent, "you decide how big I should be"), so you can return something like absolute.size(width(<grob>)) in order to always give a sensible answer.

# Examples

The original motivating example for this GUI-builder approach was to be able to produce a quite general-purpose legend grob.

A legend consists of data symbols and associated textual descriptions. In order to be quite general, it would be nice to allow, for example, multiple lines of text per data symbol. Rather than having to look at the text supplied for the legend in order to determine the arrangement of the legend, it would be nice to be able to simply specify the composition of the legend and let it figure out the arrangement for us. The code below defines just such a legend grob, using the new "frame" grob and packGrob() function. Some points to note are:

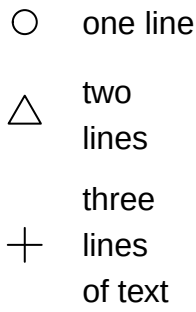
- The use of borders to create space around the legend components.
- The size of the data symbol component is specified, but the size of the text components are taken from the "text" grobs.

- The heights of the rows in the legend will be the maximum of `vgap`+`unit(1, "lines")` and `vgap` + `unit(1, "grobheight", <text grob>)`.
- We have two functions, one for generating a grob and one for producing output.

```
> legendGrob <- function(pch, labels, frame = TRUE,
+                        hgap = unit(1, "lines"), vgap = unit(1, "lines"),
+                        default.units = "lines",
+                        vp = NULL) {
+   nkeys <- length(labels)
+   gf <- frameGrob(vp = vp)
+   for (i in 1:nkeys) {
+     if (i == 1) {
+         symbol.border <- unit.c(vgap, hgap, vgap, hgap)
+         text.border <- unit.c(vgap, unit(0, "npc"), vgap,
+             hgap)
+     } else {
+         symbol.border <- unit.c(vgap, hgap, unit(0, "npc"), hgap)
+         text.border <- unit.c(vgap, unit(0, "npc"), unit(0, "npc"), hgap)
+     }
+     gf <- packGrob(gf, pointsGrob(0.5, 0.5, pch = pch[i]),
+                    col = 1, row = i, border = symbol.border,
+                    width = unit(1, "lines"),
+                    height = unit(1, "lines"), force.width = TRUE)
+     gf <- packGrob(gf, textGrob(labels[i], x = 0, y = 0.5,
+                                 just = c("left", "centre")),
+                    col = 2, row = i, border = text.border)
+   }
+   gf
+ }
> grid.legend <- function(pch, labels, frame = TRUE,
+                         hgap = unit(1, "lines"), vgap = unit(1, "lines"),
+                         default.units = "lines", draw = TRUE,
+                         vp = NULL) {
+   gf <- legendGrob(pch, labels, frame, hgap, vgap, default.units, vp)
+   if (draw) grid.draw(gf)
+   gf
+ }
```

The next piece of code shows the `grid.legend()` function being used procedurally; the output is shown below the code.

```
> grid.legend(1:3, c("one line", "two\nlines", "three\nlines\nof text"))
>
```



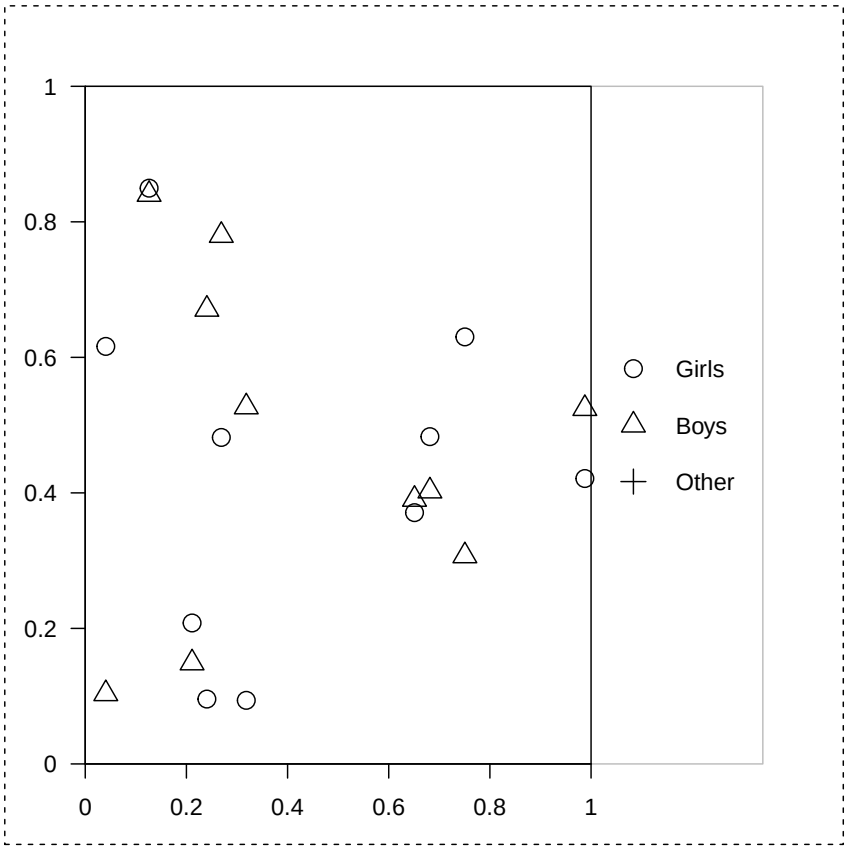

The legend example might not seem too difficult to do by hand rather than using frames and packing, but the next example shows how useful it can be.

Suppose you want to arrange a legend next to a plot. This requires leaving enough space for the legend and then filling the remaining space with the plot. This requires figuring out how much space the legend needs, and that is a task that is neither trivial nor easy to cater for in the general case. Ideally, we want to know as little as possible about the legend.

With the GUI-builder approach this becomes extremely simple. The code below shows how the construction of such a scene might be performed; the output from the code is again shown below.

The following points are noteworthy:

- We don't need to know anything about how the legend was constructed; it could be any sort of grob.
- We specify the height of the legend to be `unit(1, "null")` so that it will occupy the full height of the plot. If we did not do this then the plot would be forced to be the height of the legend (because of the way that `"null"` units interact with other units).
- The width of the legend is calculated from the contents of the legend because the legend is a `"frame"` grob.
- The dimensions of the "plot" default to `unit(1, "null")` because `"collection"` grobs have no width or height methods, which means that the plot fills up whatever space remains once the legend has been accommodated.




# Notes

1. There are `grid.frame()` and `grid.pack()` equivalents for these functions, but these are really only useful to see the changes in the frame as each packing operation takes place.

2. This frame-and-packing stuff is easier to use, *but* (in almost all cases) it is less efficient than specifying the arrangement by hand. There is consequently a penalty to pay in terms of memory (inconsequential I think) and in terms of speed (noticeably slower).

   Perhaps one sensible use of these functions is to build an image interactively using the simple arguments, which will be slow, then attempt to speed up the drawing by exploring some of the more advanced arguments.

   One way to speed things up a bit is to specify the layout when the frame is initially created and then use the `placeGrob()` function to put grobs into existing rows and columns.

   The speed penalty in the cases I have seen are mostly due to the time taken to generate the (sometimes very) complicated unit objects that express the heights and widths of the rows and columns of the frame layout. Future effort will be put into speeding up the creation of unit objects.