Writing Grid Graphics Code

December 17, 2001

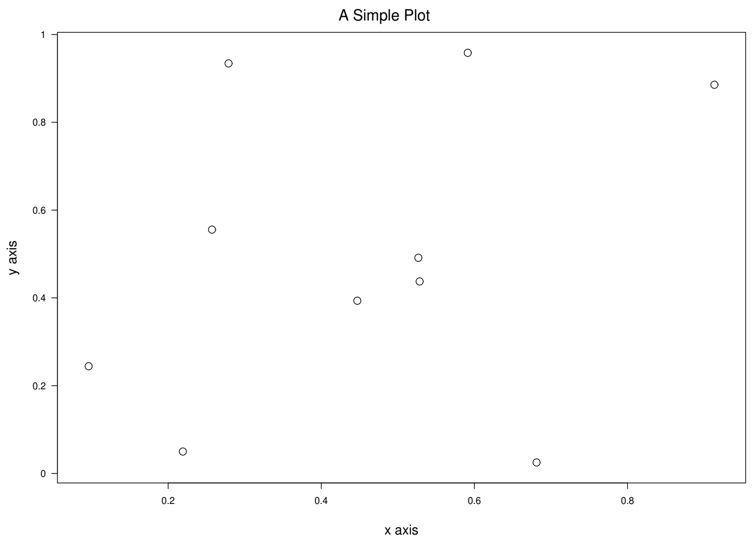
⟨*⟩≡

The Grid Graphics system contains a degree of complexity in order to allow things like interactive graphics. This means that many of the predefined Grid graphics functions are relatively complicated[1].

One design aim of Grid Graphics is to allow users to create simple graphics simply and not to force them to use complicated concepts or write complicated code unless they actually need to. Along similar lines, it is intended that people should be able to prototype even complex graphics very simply and then refine the implementation into a more sophisticated form if necessary.

With the predefined graphics functions being fully-developed and complicated implementations, there is a lack of examples of simple, prototype code. Furthermore, given that the aim is to allow a range of ways to produce the same graphical output, there is a need for examples which demonstrate the various stages, from simple to complex, that a piece of Grid Graphics code can go through.

This document describes the construction of a scatterplot object, like that shown below, going from the simplest, prototype implementation to the most complex and sophisticated. It demonstrates that if you only want simple graphics output then you can do it pretty simply and quickly. It also demonstrates how to write functions that allow your graphics to be used by other people. Finally, it demonstrates how to make your graphics fully interactive (or at least as interactive as Grid will let you make it).



This document should be read *after* the Grid Graphics Users' Guide. Here we are assuming that the reader has an understanding of viewports, layouts, and units.

[1] Although there are exceptions; some functions, such as `grid.show.viewport`, are purely for producing illustrative diagrams and remain simple and procedural.

Procedural Grid Graphics

The simplest way to produce graphical output in Grid is just like producing standard R graphical output. You simply issue a series of graphics commands. For example, `plot(x, y)` followed by `points(x, y2, pch=16)`, followed by `text(x, y3, y3)`, and so on[2]. The purpose of the commands is simply to produce graphics output; in particular, we are not concerned with any values returned by the plotting functions. I will call this *procedural graphics*[3].

In this document, we will not be using predefined high-level plotting functions like `plot()` because our ultimate aim is actually to build such a function. For the current purposes, we are assuming that no convenient high-level plotting functions exist.

In order to draw a simple scatterplot, we can issue a series of commands which draw the various components of the plot.

First of all, we generate some data to plot.

⟨*proceduralGraphics*⟩≡

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

Next we allocate regions for the different parts of the plot to go into. This is the most difficult part of the exercise, but it is possible to do some very complicated things relatively easily using viewports, layouts, and units.

The goal is to end up with a data region in the middle of the plot, where the points will be plotted, and a margin around the outside for the axes to fit in, with a space for a title at the top.

⟨*proceduralGraphics*⟩+≡

```
plot.layout <- grid.layout(ncol=3, nrow=3,
                           widths=unit(c(5, 1, 2),
                                       c("lines", "null", "lines")),
                           heights=unit(c(3, 1, 5),
                                        c("lines", "null", "lines")))
plot.vp <- viewport(layout=plot.layout)
data.vp <- viewport(layout.pos.row=2, layout.pos.col=2,
                    xscale=range(x) + c(-.05, .05)*diff(range(x)),
                    yscale=range(y) + c(-.05, .05)*diff(range(y)))
title.vp <- viewport(layout.pos.row=1)
```

Now we draw the components of the plot: points, axes, labels, and a title.

⟨*proceduralGraphics*⟩+≡

```
push.viewport(plot.vp)
grid.points(x, y, vp=data.vp)
grid.rect(vp=data.vp)
grid.xaxis(vp=data.vp)
grid.yaxis(vp=data.vp)
grid.text("x axis", y=unit(-4, "lines"),
          gp=gpar(fontsize=14), vp=data.vp)
grid.text("y axis", x=unit(-4, "lines"),
          gp=gpar(fontsize=14), rot=90, vp=data.vp)
grid.text("A Simple Plot", gp=gpar(fontsize=16), vp=title.vp)
pop.viewport()
```

[2] You can't actually do this in Grid yet, but it should be possible "soon".

[3] It could also reasonably be called *incremental graphics* because of the way that plots can be built up in a piece-wise fashion.

Writing a Grid Graphics Function

Issuing commands like in the previous section is most useful for adding annotations to an existing graphic or for prototyping initial ideas. When we are producing a graphic over and over it is far more convenient to wrap up the graphics commands in a function.

The code below shows how we might wrap our scatterplot code within a couple of functions. This is nothing more than an exercise in making the code tidier, more convenient, and more reusable. The important point is that the functions are *procedural*; they are designed to be called only for the graphical output that they produce.

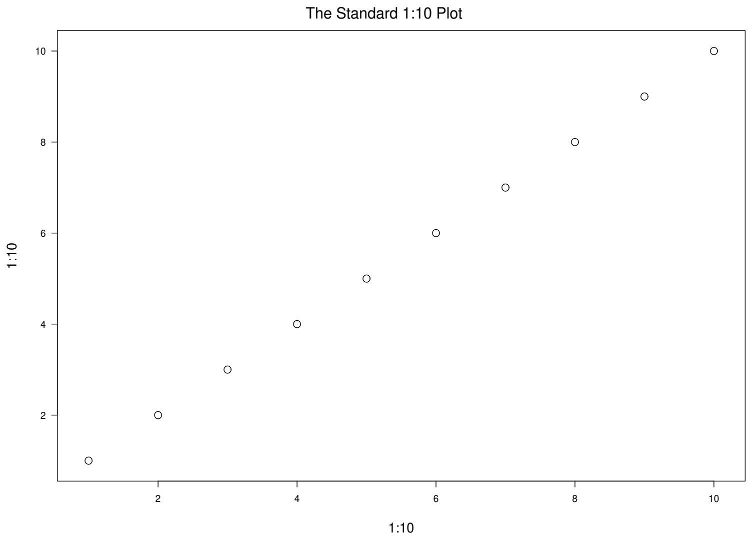
The following points are worth noting:

1. we have provided various arguments to control different aspects of the graph such as the axis labels, the title, and the size of the plot margins.
2. we have made use of the `push.viewport` and `pop.viewport` functions to avoid specifying a viewport for every graphical operation in `splot.draw.data`.
3. we have added a call to `grid.newpage()` so that we draw each plot on a fresh window/page.
4. we have bundled some of the scatterplot actions within their own functions so that the main function is less bloated.

⟨*graphicsFunction*⟩≡
```
splot.layout <- function(margins) {
  grid.layout(ncol=3, nrow=3,
              widths=unit.c(margins[2], unit(1, "null"), margins[4]),
              heights=unit.c(margins[3], unit(1, "null"), margins[1]))
}

splot.draw.data <- function(x, y, xlabel, ylabel, vp) {
  push.viewport(vp)
  grid.points(x, y)
  grid.rect()
  grid.xaxis()
  grid.yaxis()
  grid.text(xlabel, y=unit(-4, "lines"), gp=gpar(fontsize=14))
  grid.text(ylabel, x=unit(-4, "lines"), gp=gpar(fontsize=14), rot=90)
  pop.viewport()
}

splot <- function(x=runif(10), y=runif(10),
                  xlabel="x axis", ylabel="y axis",
                  title="A Simple Plot",
                  margins=unit(c(5, 5, 3, 2), "lines")) {
  grid.newpage()
  plot.layout <- splot.layout(margins)
  plot.vp <- viewport(layout=plot.layout)
  push.viewport(plot.vp)
  data.vp <- viewport(layout.pos.row=2, layout.pos.col=2,
                      xscale=range(x) + c(-.05, .05)*diff(range(x)),
                      yscale=range(y) + c(-.05, .05)*diff(range(y)))
  splot.draw.data(x, y, xlabel, ylabel, data.vp)
  title.vp <- viewport(layout.pos.row=1)
  grid.text(title, gp=gpar(fontsize=16), vp=title.vp)
  pop.viewport()
}
```

We can now do things like the following to produce variations on the simple scatterplot. The output from the second example is shown below the code.

⟨*graphicsFunctionDemo*⟩≡
```
splot()

splot(1:10, 1:10, title="The Standard 1:10 Plot",
      xlabel="1:10", ylabel="1:10")
```



Making a Grid Graphics Function Embeddable

Another design aim of Grid is that any graphic should be able to be drawn within any coordinate system (and there are potentially a lot of coordinate systems available because users can define their own viewports). The idea is to encourage the development of graphical building blocks which can then be assembled by higher-level functions to create complex graphics. For example, we have been using the `grid.text()` function within different coordinate systems to produce axis labels and a plot title[4].

The function `splot()` described in the previous section is useful for producing your own scatterplots with some minor control over the appearance. However, this function is not in a useful state for acting as a building block within other graphics functions. For a start, it always clears the whole device, and secondly, it doesn't clean up the viewport stack after itself.

The function has been rewritten below[5] to make it more like a graphical building block. The following points are of note:

1. we have added a `vp` argument, which allows someone else to specify where the plot should be placed.
2. we simply call `push.viewport()` with the `vp` argument to establish the context specified by `vp`.
3. we call `pop.viewport()` for any viewports that we have pushed onto the viewport stack.
4. we have added a logical `add` argument, which indicates whether the plot should be added to some other graphics, or whether it is a complete graphic on its own; `grid.newpage()` is only called if `add` is `FALSE`.

> ASIDE: *There is some awkwardness here compared to standard R graphics because Grid graphics does not have the notion of a "current plot" or a "number of plots on page". Such notions could be programmed into a package so that functions could go to the "next plot" and would know when this meant also going to a new page.*

⟨*embeddableGraphics*⟩≡

```
splot <- function(x=runif(10), y=runif(10),
                  xlabel="x axis", ylabel="y axis",
                  title="A Simple Plot",
                  margins=unit(c(5, 5, 3, 2), "lines"),
                  vp=NULL, add=FALSE) {
  if (!add)
    grid.newpage()
  plot.layout <- splot.layout(margins)
  plot.vp <- viewport(layout=plot.layout)
  if (!is.null(vp))
    push.viewport(vp)
  push.viewport(plot.vp)
  data.vp <- viewport(layout.pos.row=2, layout.pos.col=2,
                      xscale=range(x) + c(-.05, .05)*diff(range(x)),
                      yscale=range(y) + c(-.05, .05)*diff(range(y)))
  splot.draw.data(x, y, xlabel, ylabel, data.vp)
  title.vp <- viewport(layout.pos.row=1)
  grid.text(title, gp=gpar(fontsize=16), vp=title.vp)
```

[4] This is in contrast to the R/S setup where there is a `text()` function for text in the data region and an `mtext()` function for text in the margins.

[5] The functions `splot.layout` and `splot.draw.data` remain unchanged

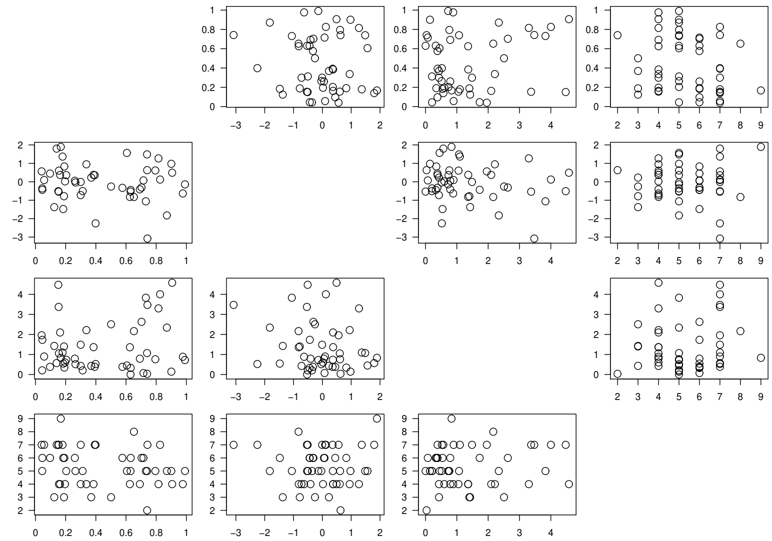
```
    pop.viewport()
    if (!is.null(vp))
      pop.viewport()
  }
```

It is now possible to use the plot in other graphics functions. For example, the following code produces a very crude scatterplot matrix (shown below).

> Aside: *Again this may appear laborious compared to standard R graphics, but this is only because the notions of "par(mfrow)" and such like have not yet been programmed into Grid - not that they necessarily will ever be, but I intend to program them into a package built on top of Grid eventually. As mentioned previously, the idea of these demonstrations is to show the power and flexibility not to compare with the convenience of a predefined high-level function.*

⟨*embeddableGraphicsDemo*⟩≡

```
  w <- runif(50)
  x <- rnorm(50)
  y <- rexp(50)
  z <- rbinom(50, 10, .5)
  data <- data.frame(w, x, y, z)
  top.vp <- viewport(layout=grid.layout(4, 4))
  push.viewport(top.vp)
  for (i in 1:4)
    for (j in 1:4)
      if (i != j)
        splot(data[,j], data[,i],
              title="", xlabel="", ylabel="",
              margins=unit(c(3, 3, 0, 0), "lines"),
              vp=viewport(layout.pos.row=i, layout.pos.col=j),
              add=TRUE)
  pop.viewport()
```



Making a Grid Graphics Function Customisable

In the previous example, our scatterplot was used as a component of a scatterplot matrix. The scatterplot matrix was able to control things such as the margins of the plot, the text used for labels and titles, and where the plot was drawn. However, functions which call `splot()` might want a lot more control than that over the appearance of the plot. For example, we might want to control the colour of different elements of the plot, or we might want to control the size of the labels.

One simple approach to allowing this sort of customisation is to extend the list of arguments to the `splot` function. This is reasonable for simple graphical objects which have few aspects to control. However, the scatterplot object is relatively complex and it would require a very long list of arguments to cover all possible customisations.

Another approach is to provide a single argument for each of the components of the scatterplot: the data, the axes, the labels, and the title. The default values for these arguments are either functions or graphical objects. If you want to override the default components then you can substitute your own functions or graphical objects. Here we see for the first time that Grid Graphics functions produce graphical objects as well as graphical output.

We will use the following set of scatterplot components:

xaxis, yaxis These will just be graphical objects. `splot()` will set up the appropriate viewport and then draw the given object in that viewport. The default values are the objects returned by `grid.xaxis()` and `grid.yaxis` respectively.

xlabel, ylabel, title These will also be graphical objects. We will, however, allow the caller to specify either a string or a graphical object. If a string is given, we will turn it into a graphical object ourselves. A small helper function for this is given below.

data This will be a function. `splot()` will set up a viewport then call this function to draw the data points and the bounding box. The default function is shown below.

> ASIDE: *Of course, this is only one possible break down of the scatterplot object. There are infinitely many alternatives; for example, you could just have an* `axes` *function rather than separate* `xaxis` *and* `yaxis` *grahical objects.*

⟨*customGraphics*⟩≡

```
draw.str.or.obj <- function(text, ...) {
  if (is.character(text))
    grid.text(text, ...)
  else
    grid.draw(text)
}

splot.data <- function(x, y) {
  grid.points(x, y)
  grid.rect()
}
```

Another issue with customisation is being able to add extra graphics to the scatterplot; in other words annotating the plot. The main assistance that you can provide for this purpose is to make the viewports created by the scatterplot available to others by returning them as the value of the function.

With all this in mind, it is convenient to package the layout creation within its own function, as shown below.

⟨*customGraphics*⟩+≡

```
splot.viewports <- function(x, y, margins) {
  plot.layout <-
    grid.layout(ncol=3, nrow=3,
                widths=unit.c(margins[2], unit(1, "null"), margins[4]),
                heights=unit.c(margins[3], unit(1, "null"), margins[1]))
  plot.vp <- viewport(layout=plot.layout)
  data.vp <- viewport(layout.pos.row=2, layout.pos.col=2,
                      xscale=range(x) + c(-.05, .05)*diff(range(x)),
                      yscale=range(y) + c(-.05, .05)*diff(range(y)))
  title.vp <- viewport(layout.pos.row=1)
  list(plot.vp=plot.vp, data.vp=data.vp, title.vp=title.vp)
}
```

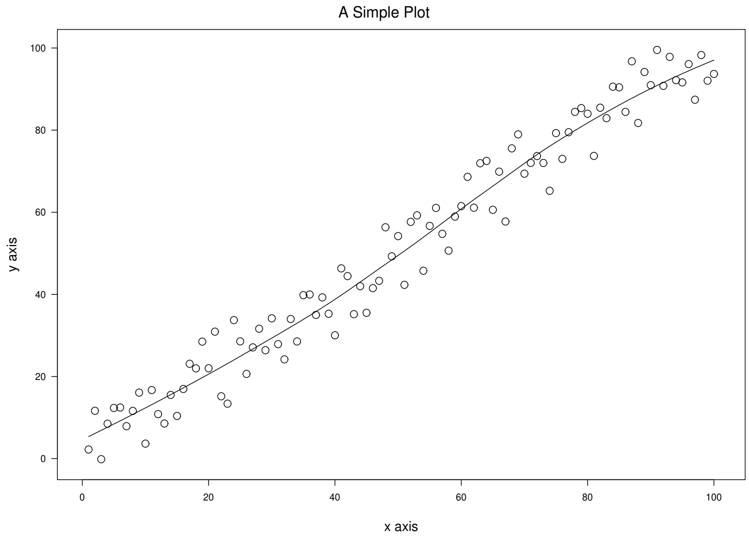
With all of these support functions in place we can now rewrite the `splot` function with customisation in mind. Note that we specify `draw=FALSE` for `grid.xaxis()` and `grid.yaxis()`. This is because we are only using these functions to produce graphical objects as parameters to `splot()`; we want to control when we produce graphical output from these objects.

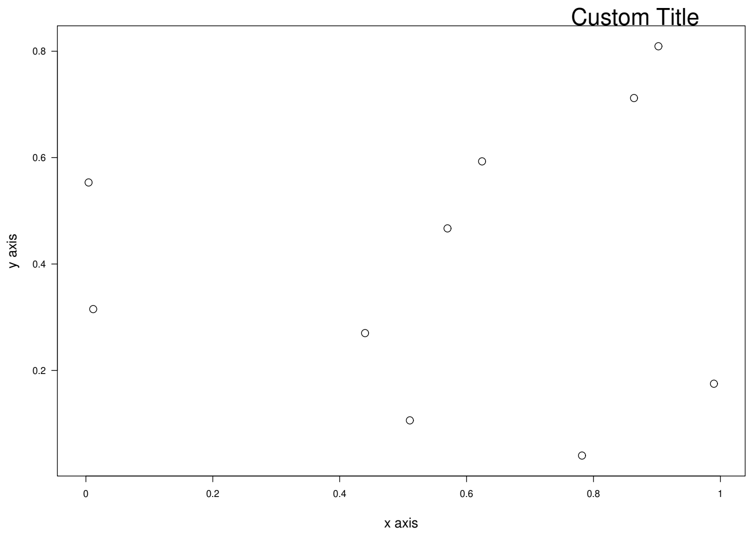
⟨*customGraphics*⟩+≡

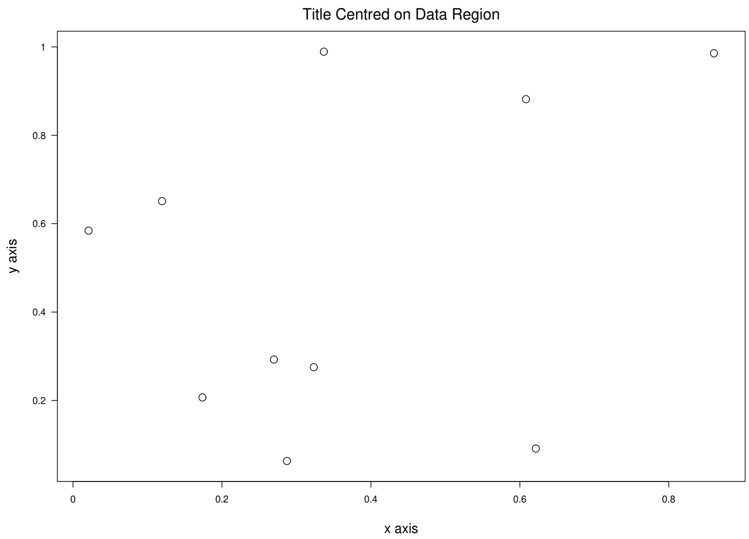
```
splot <- function(x=runif(10), y=runif(10),
                  xlabel="x axis", ylabel="y axis", title="A Simple Plot",
                  margins=unit(c(5, 5, 3, 2), "lines"),
                  xaxis=grid.xaxis(draw=FALSE), yaxis=grid.yaxis(draw=FALSE),
                  data=splot.data,
                  vp=NULL, add=FALSE) {
  if (!add)
    grid.newpage()
  vps <- splot.viewports(x, y, margins)
  if (!is.null(vp))
    push.viewport(vp)
  push.viewport(vps$plot.vp, vps$title.vp)
  draw.str.or.obj(title, gp=gpar(fontsize=16))
  pop.viewport()
  push.viewport(vps$data.vp)
  data(x, y)
  grid.draw(xaxis)
  grid.draw(yaxis)
  draw.str.or.obj(xlabel, y=unit(-4, "lines"), gp=gpar(fontsize=14))
  draw.str.or.obj(ylabel, x=unit(-4, "lines"), gp=gpar(fontsize=14), rot=90)
  pop.viewport(2)
  if (!is.null(vp))
    pop.viewport()
  invisible(vps)
}
```

The code below demonstrates how this new function can be used. The first example just shows that the function still works :) The second example shows a simple customisation of the `data` argument using our own function to add a smooth line through the points in the scatterplot. Note that the it will often be convenient to call the default function from your custom function (e.g., `my.data` calls `splot.data`). The third example shows a customisation of the `title` argument by specifying a new text object. This makes the title bigger and right-justifies it 1'' in from the right-hand edge of the plot. The fourth example demonstrates the use of the viewports produced by `splot()`. The default title is horizontally centred within the entire plot region; we turn off the default title and annotate the plot with our own title which is horizontally centred relative to the data region of the plot.

⟨*customGraphicsDemo*⟩≡
```
splot()

library(modreg)
my.data <- function(x, y) {
  splot.data(x, y)
  lo <- (loess(y ~ x))
  grid.lines(lo$x, lo$fitted, default.units="native")
}
splot(1:100, 1:100+runif(100, -10, 10), data=my.data)

splot(title=grid.text("Custom Title", just="right", gp=gpar(fontsize=24),
                      x=unit(1, "npc") - unit(1, "inches"), draw=FALSE))

svps <- splot(title="")
push.viewport(svps$plot.vp)
grid.text("Title Centred on Data Region", gp=gpar(fontsize=16),
          vp=viewport(layout.pos.row=1, layout.pos.col=2))
```




Custom Title
y axis
x axis
0.8
0.6
0.4
0.2
0
0.2
0.4
0.6
0.8
1


Title Centred on Data Region
y axis
x axis
1
0.8
0.6
0.4
0.2
0
0.2
0.4
0.6
0.8

Making a Grid Graphics Object

The function `splot` is now a highly customisable, embeddable graphical building block. However, it is still only useful for the production of graphics output; it is still only for *procedural* use.

In the previous section we made use of `grid.text()`, `grid.xaxis()`, and `grid.yaxis()` for their ability to produce graphical objects, which we used as default argument values to `splot()`. In this section we look at how we need to change `splot` so that it produces a graphical object too.

The main thing to do is to split `splot()` into a function which does the drawing and a function which creates an object. The object which we create is going to be derived from the class `"grob"`. This is the base class of all Grid graphical objects. The purpose of deriving it from `"grob"` is that we then inherit lots of useful functionality which is shared by all Grid graphical objects. We will give it its own class, `"splot"`, so that we can write special methods that will allow an `"splot"` object to behave differently to other `"grob"`s[6].

The drawing function is one such method. It is a method for the generic Grid function `draw.details()` and it allows us to control exactly what gets drawn by an `"splot"` object. It basically consists of all of the drawing operations in the old `splot()` function[7].

⟨*objectGraphics*⟩≡

```
draw.details.splot <- function(sp, grob, recording=TRUE) {
  if (!sp$add)
    grid.newpage(recording=FALSE)
  push.viewport(sp$plot.vp, sp$title.vp, recording=FALSE)
  grid.draw(sp$title, recording=FALSE)
  pop.viewport(recording=FALSE)
  push.viewport(sp$data.vp, recording=FALSE)
  grid.draw(sp$data, recording=FALSE)
  grid.draw(sp$xaxis, recording=FALSE)
  grid.draw(sp$yaxis, recording=FALSE)
  grid.draw(sp$xlabel, recording=FALSE)
  grid.draw(sp$ylabel, recording=FALSE)
  pop.viewport(2, recording=FALSE)
}
```

[6] Technically, the `"splot"` object is actually wrapped within a `"grob"` object, but this detail is hidden from the user.

[7] Note that we do not have to push the `vp` slot of our object; Grid treats a slot of that name specially and automatically pushes and pops it as part of the default drawing process

The `splot()` function is now responsible for creating an object of class `"splot"`. This consists of creating a list of all of the components of the object and then calling the function `grid.grob()` to turn this into an object.

Because we are now creating a list of objects rather than simply calling functions to produce graphics output, we require a couple of changes to the helper functions so that they return objects and do not do any drawing.

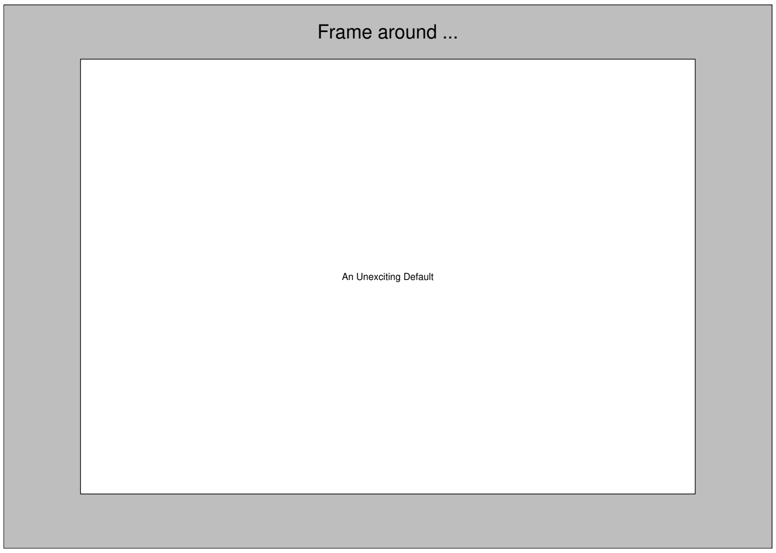
Notice that `splot.data` creates a `grid.collection`. This is just the simplest sort of graphical object. Its `draw.details` method just draws all of its elements. It is just a convenient way of dealing with several graphical objects all at once.

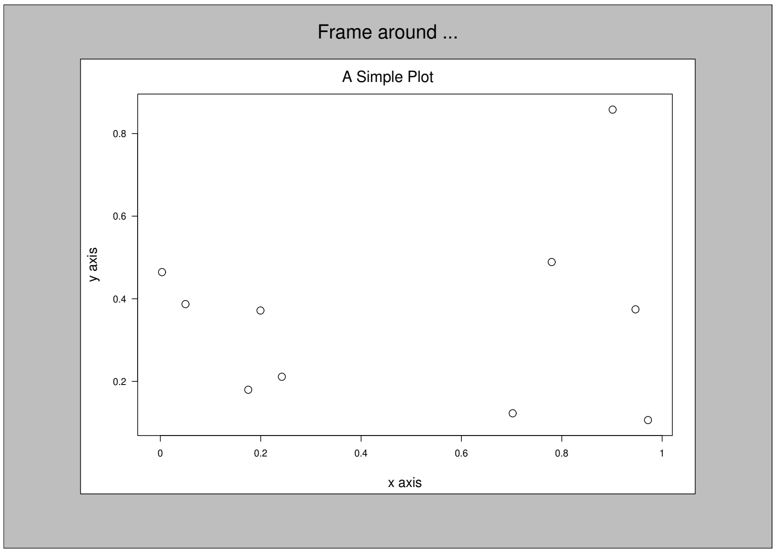
⟨*objectGraphics*⟩+≡
```
make.str.or.obj <- function(text, ...) {
  if (is.character(text))
    grid.text(text, ..., draw=FALSE)
  else
    text
}

splot.data <- function(x, y) {
  grid.collection(points=grid.points(x, y, draw=FALSE),
                  box=grid.rect(draw=FALSE),
                  draw=FALSE)
}
```

⟨*objectGraphics*⟩+≡
```
splot <- function(x=runif(10), y=runif(10),
                  xlabel="x axis", ylabel="y axis", title="A Simple Plot",
                  margins=unit(c(5, 5, 3, 2), "lines"),
                  xaxis=grid.xaxis(draw=FALSE), yaxis=grid.yaxis(draw=FALSE),
                  data=splot.data, draw=TRUE, add=FALSE,
                  vp=NULL) {
  vps <- splot.viewports(x, y, margins)
  title <- make.str.or.obj(title, gp=gpar(fontsize=16))
  xlabel <- make.str.or.obj(xlabel, y=unit(-4, "lines"), gp=gpar(fontsize=14))
  ylabel <- make.str.or.obj(ylabel, x=unit(-4, "lines"),
                            gp=gpar(fontsize=14), rot=90)
  sp <- list(x=x, y=y, title=title, xlabel=xlabel, ylabel=ylabel,
             data=data(x, y), data.func=data, xaxis=xaxis, yaxis=yaxis,
             plot.vp=vps$plot.vp, data.vp=vps$data.vp,
             title.vp=vps$title.vp, add=add, vp=vp)
  grid.grob(sp, "splot", draw)
}
```

Once again, the first example below is just a check that things still work, although it does also show that the new, object-oriented `splot()` can still be used procedurally, basically ignoring the fact that it returns an object. The second example demonstrates that `"splot"` objects can be used as arguments to graphics functions; this allows other people to write highly customisable graphics which includes `"splot"`s as components. The third example shows `"splot"` grobs being used as components of another graphical *object*. The output of the second example is shown below the code.

⟨*objectGraphicsDemo*⟩≡
```
splot()

framer <- function(any.old.grob=grid.text("An Unexciting Default", draw=FALSE)) {
  grid.newpage()
  grid.rect(gp=gpar(border=NULL, fill="grey"))
  vp <- viewport(width=0.8, height=0.8)
  push.viewport(vp)
  grid.rect(gp=gpar(fill="white"))
  grid.text("Frame around ...", y=unit(1, "npc") + unit(1, "cm"),
            gp=gpar(fontsize=20))
  grid.draw(any.old.grob)
  pop.viewport()
}
framer()
framer(splot(add=TRUE, draw=FALSE))

draw.details.simple <- function(simple, grob, recording=TRUE) {
  grid.draw(simple$splot, recording=FALSE)
}
simple <- function() {
  splot <- splot(draw=FALSE)
  grid.grob(list(splot=splot), "simple")
}
simple()
```


Frame around ...
An Unexciting Default


Frame around ...
A Simple Plot
0.8
0.6
0.4
0.2
y axis
0
0.2
0.4
0.6
0.8
1
x axis

Making an Interactive Grid Graphics Object

We do not only produce graphical objects for providing default argument values or providing components for other graphical objects. It is also useful to be able to keep a record of what you have drawn and to be able to edit that record and possibly even have the graphical output automatically updated[8].

The only thing we have to add to make this possible with our `"splot"` object is an `edit.details` method. Grid provides an `edit.grob` function which allows us to specify changes to any `"grob"` object.

It is actually already possible to edit the *components* of an `"splot"` object, because the components already have methods written. For example,

```
grid.edit(splot.obj, "title", gp=gpar(col="red"))
grid.edit(splot.obj, "axis", at=0.5)
grid.edit(grid.display.list{\Tt{}2}, "xlabel", label="My Axis Label")
```

However, it is not possible to edit the description of the scatterplot itself.

For example, suppose that we want to change the values that are being plotted. This will require altering the plot viewport for the new x- and/or y-range and redrawing all of the points (and anything else that may have been specified in the `"splot"`'s `data` argument).

The code below shows a breakdown of the function to create splot viewports, just so that we don't have to recreate all of the viewports unnecessarily.

⟨*interactiveGraphics*⟩≡

```
splot.plot.vp <- function(margins) {
  plot.layout <-
    grid.layout(ncol=3, nrow=3,
                widths=unit.c(margins[2], unit(1, "null"), margins[4]),
                heights=unit.c(margins[3], unit(1, "null"), margins[1]))
  plot.vp <- viewport(layout=plot.layout)
}

splot.data.vp <- function(x, y) {
  data.vp <- viewport(layout.pos.row=2, layout.pos.col=2,
                      xscale=range(x) + c(-.05, .05)*diff(range(x)),
                      yscale=range(y) + c(-.05, .05)*diff(range(y)))
}

splot.title.vp <- function() {
  title.vp <- viewport(layout.pos.row=1)
}

splot.viewports <- function(x, y, margins) {
  list(plot.vp=splot.plot.vp(margins),
       data.vp=splot.data.vp(x, y),
       title.vp=splot.title.vp())
}
```

[8] This is a requirement for implementing a GUI interface for graphics with, for example, dialog boxes for editing components of a graph.

Now we can write the `edit.details` method itself. There are several important features to note:

1. The arguments to the method are the `splot` list to be modified, and a list of named arguments specifying the changes to be made (e.g., `gp=gpar(col="red")`, `at=0.5`, ...).
2. The return value of the method is the altered `splot` list. This is vital for the changes we have made to become permanent.

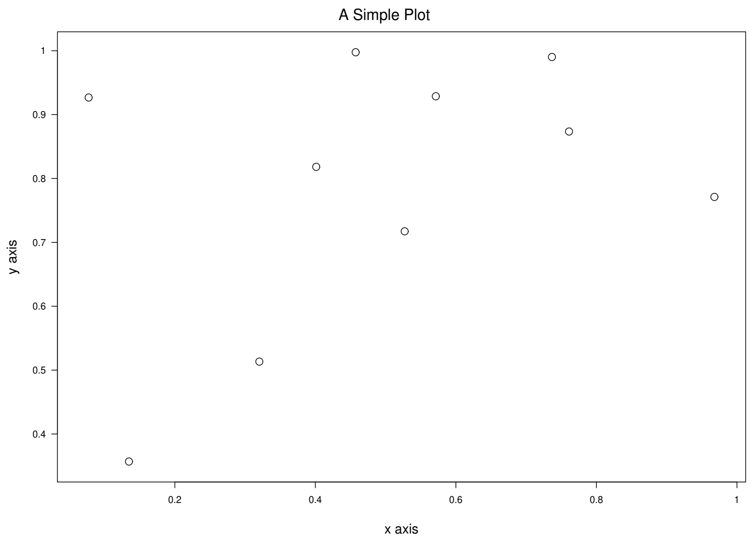
⟨*interactiveGraphics*⟩+≡

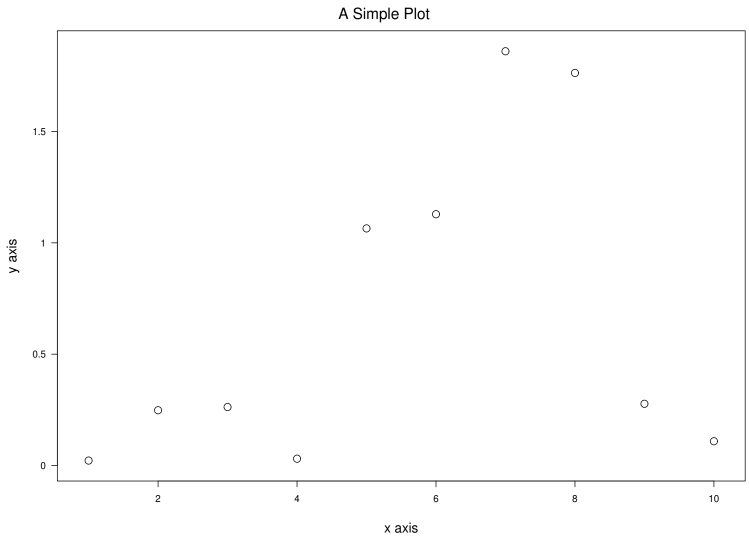
```
edit.details.splot <- function(splot, new.values) {
  slot.names <- names(new.values)
  x.index <- match("x", slot.names, nomatch=0)
  y.index <- match("y", slot.names, nomatch=0)
  if (x.index != 0 || y.index != 0) {
    x <- if (x.index) new.values[[x.index]] else splot$x
    y <- if (y.index) new.values[[y.index]] else splot$y
    splot$data.vp <- splot.data.vp(x, y)
    splot$data <- splot$data.func(x, y)
    grid.edit(splot$xaxis, at=NA, redraw=FALSE)
    grid.edit(splot$yaxis, at=NA, redraw=FALSE)
    if (x.index) x.index <- -x.index else x.index <- NA
    if (y.index) y.index <- -y.index else y.index <- NA
    new.values <- new.values[c(x.index, y.index)]
  }
  splot
}
```

The example below produces a standard scatterplot, then edits the x- and y-values for the scatterplot. The output is shown below the code.

⟨*interactiveGraphicsDemo*⟩≡

```
sp <- splot()
grid.edit(sp, grid.prop.list(x=1:10, y=rexp(10)))
```

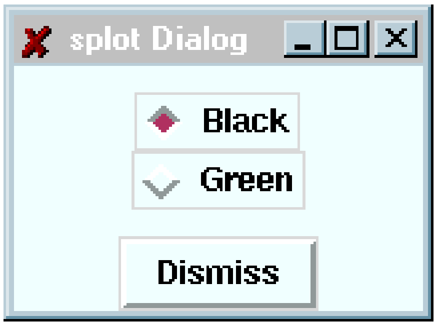




The slightly longer example below gives a small taste of how a GUI interface could be built for the "splot" object. This function produces a scatterplot and a dialog box (shown below the code). When you click on the radio button labelled “Green”, the plot turns green and when you click on “Black” it turns back to black.

⟨*guiGraphics*⟩≡

```
library(tcltk)
gui.splot <- function() {
  sp <- splot()
  recolour <- function(...) {
    if (tclvar$colour == 1)
      grid.edit(sp, col="black")
    else
      grid.edit(sp, col="green")
  }
  top <- tktoplevel()
  tktitle(top) <- "splot Dialog"
  colours <- tkframe(top, borderwidth=10)
  tkpack(colours, side="top")
  tkpack(tkradiobutton(colours, command=recolour, text="Black",
                       value=1, variable="colour"))
  tkpack(tkradiobutton(colours, command=recolour, text="Green",
                       value=2, variable="colour"))
  dismiss <- tkframe(top)
  tkpack(dismiss, side="bottom")
  tkpack(tkbutton(dismiss, text="Dismiss",
                  command=function()tkdestroy(top)))
  tclvar$colour <- 1
}
```



Caveats

- This is ONE way to implement a scatterplot. There may be things you don't like about it. One of the aims of Grid is to make it so that you don't have to think to hard about all the possible things people might want to do with your graphics functions. I consider that if I write something as flexible as the example given here and people still want to do something fancier then (i) they are at a high enough level to consider writing their own graphics code (ii) there is sufficient support in Grid to make writing your own graphics code a feasible proposition.

- I have deliberately left out lots of type-checking and bullet-proofing to avoid cluttering up the examples. Obviously, a function that is going to be used by lots of people should have these things included.