Working with **grid** Viewports

Paul Murrell

November 9, 2026

This document describes some features of **grid** viewports which make it easy to travel back and forth between multiple regions on a device (without having to recreate those regions), and provides a mechanism for a complex plotting function to provide users with access to all of the regions created during plotting.

The viewport tree

grid maintains a tree of pushed viewports on each device. When the `upViewport()` function is called it works like `popViewport()` except that it does not remove viewports from the viewport tree. For example, the following code pushes a viewport, then navigates back up to the top level viewport and pushes another viewport, without removing the first viewport.

```
> pushViewport(viewport())
> upViewport()
> pushViewport(viewport())
```

There are now two viewports pushed directly beneath the top-level viewport. This immediately creates an ambiguity; if I navigate back up to the top-level and attempt to revisit one of the viewports, how can I specify which one I want? The answer is that viewports have a `name` argument[1]. This name, combined with the `downViewport()` function, makes it possible to navigate back down to viewports in the viewport tree. Consider the following example, which pushes two viewports called `"A"` and `"B"`, then navigates to viewport `"A"` from the top level.

```
> pushViewport(viewport(name = "A"))
> upViewport()
> pushViewport(viewport(name = "B"))
> upViewport()
> downViewport("A")
```

The `downViewport()` function searches down the tree from the current position in the tree. The `seekViewport()` function is similar, but it always starts searching from the top-level viewport. In the previous example we ended up in viewport `"A"`; the following command navigates from `"A"` to `"B"` in a single step.

[1] The print method for viewports shows the viewport name within square brackets. Try typing `current.viewport()`.

```
> seekViewport("B")
```

The function `current.vpTree()` provides a (textual) view of the current viewport tree.

```
> current.vpTree()
```

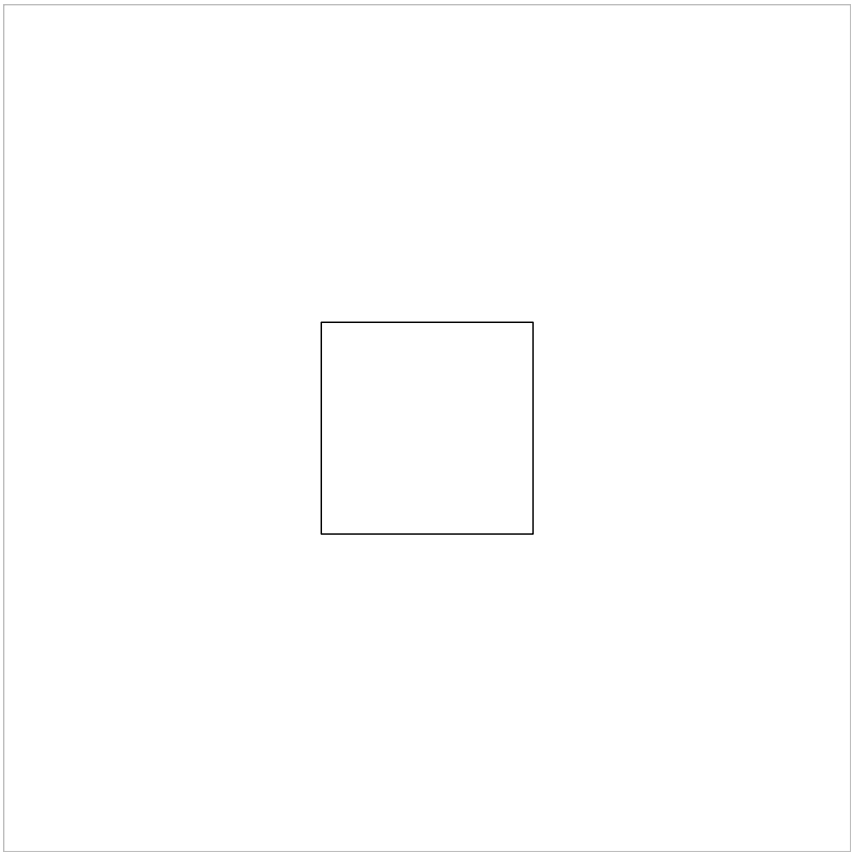
```
viewport[ROOT]->(viewport[A], viewport[B])
```

```
>
```

Viewport stacks, lists, and trees

It is possible to create multiple viewport descriptions *and their relationships*. The functions `vpStack()`, `vpList()`, and `vpTree()` can be used to create a stack, a list, or a tree of viewport descriptions, respectively. It is then possible to push these multiple descriptions at once; viewports in a stack are pushed in series, viewports in a list are pushed in parallel, and for a tree of viewports, the parent is pushed then the children are pushed in parallel. The following simple example demonstrates one usage of this feature; a **grid** rectangle is drawn *two* viewports below the current level by specifying a stack of viewports in its `vp` argument.

```
> vp <- viewport(width = 0.5, height = 0.5)
> grid.rect(vp = vpStack(vp, vp))
```



Viewport paths

The previous example demonstrates a subtle feature of **grid**'s viewport tree. The same viewport, *with the same name*, was pushed twice (in series). This demonstrates that viewport names only have to be unique for viewports which share the same parent. This makes it possible, especially in repetitive plot arrangements, to reuse convenient viewport names. For example, in a **lattice** style plot, each panel could have a viewport called `"strip"` to represent the strip region.

This design creates a further ambiguity because there may be more than one viewport with the same name within the viewport tree[2]. This ambiguity can be resolved by using the `vpPath()` function to generate a specification of a stack of viewports that must be matched by name. This path can be passed to either `downViewport()` or `seekViewport()` as in the following example; notice that we are calling `current.vpTree(FALSE)` in order to see the current viewport tree *only from the current viewport down*.

```
> pushViewport(viewport(name = "A"))
> pushViewport(viewport(name = "B"))
> pushViewport(viewport(name = "A"))
```

When we do a seek on just `"A"`, we find the first `"A"` (just below the top-level viewport).

```
> seekViewport("A")
> current.vpTree(FALSE)
```

```
viewport[A]->(viewport[B]->(viewport[A]))
```

By specifying a `vpPath`, we can get the `"A"` directly below viewport `"B"`.

```
> seekViewport(vpPath("B", "A"))
> current.vpTree(FALSE)
```

```
viewport[A]
```

A viewport path is conceptually just a concatenation of several names using a path separator (currently `::`).

```
> vpPath("A", "B")
```

```
A::B
```

For interactive use, it is possible to specify the path as a simple string, but this is not recommended otherwise in case the path separator changes in future versions of **grid**. As an example, the following two commands are currently equivalent.

```
> seekViewport(vpPath("A", "B"))
> seekViewport("A::B")
```

[2] `downViewport()` and `seekViewport()` will stop at the first match they find (the search is currently depth-first).

An example

In this section, we consider a simple example to demonstrate how these new viewport features might be used together. The goal is to produce a simple scatterplot. We will work with some random data.

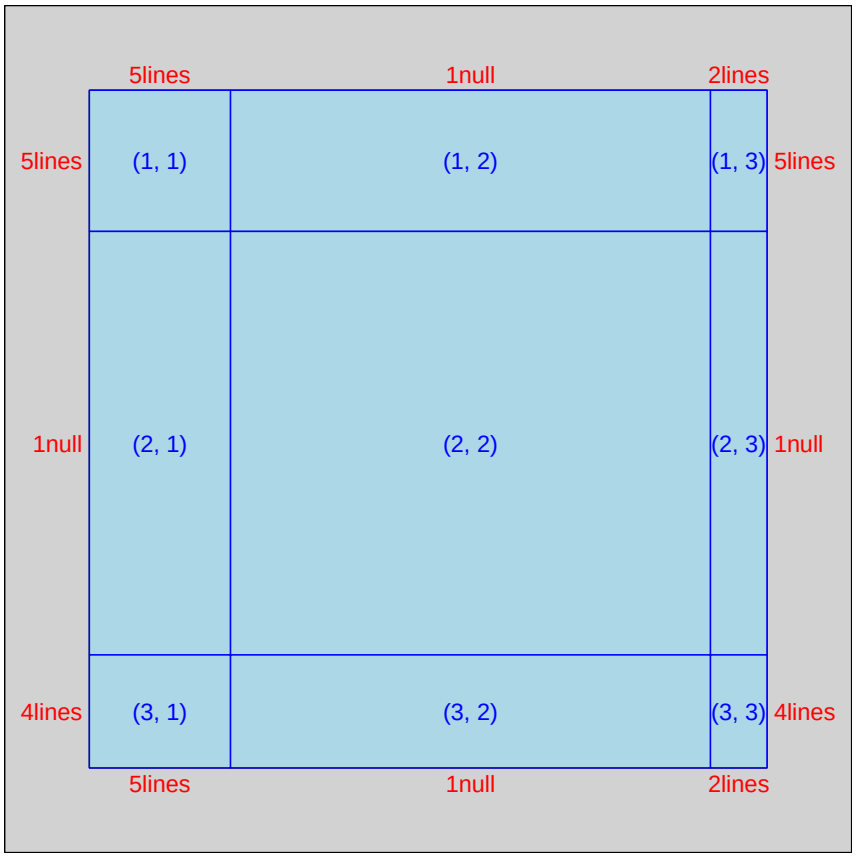
```
> x <- runif(10)
> y <- runif(10)
```

We will be establishing some scales appropriate for these data, so we calculate sensible ranges now.

```
> xscale <- extendrange(x)
> yscale <- extendrange(y)
```

We now produce a set of viewports that will be useful in creating the plot. The first viewport contains a layout to divide the drawing region into several rows and columns. The left and right columns and top and bottom rows provide room for axes and labels, while the central cell provides a region for plotting the data. The diagram below the code shows the layout that we create.

```
> top.vp <-
+     viewport(layout=grid.layout(3, 3,
+              widths=unit(c(5, 1, 2), c("lines", "null", "lines")),
+              heights=unit(c(5, 1, 4), c("lines", "null", "lines"))))
>
```



Next we create a set of viewports which will occupy different areas within the layout, corresponding to the margins for axes and labels, and the plotting region.

```
> margin1 <- viewport(layout.pos.col = 2, layout.pos.row = 3,
+                     name = "margin1")
> margin2 <- viewport(layout.pos.col = 1, layout.pos.row = 2,
+                     name = "margin2")
> margin3 <- viewport(layout.pos.col = 2, layout.pos.row = 1,
+                     name = "margin3")
> margin4 <- viewport(layout.pos.col = 3, layout.pos.row = 2,
+                     name = "margin4")
> plot <- viewport(layout.pos.col = 2, layout.pos.row = 2,
+                  name = "plot", xscale = xscale, yscale = yscale)
```

Notice that we have not pushed any of these viewports yet so no regions exist on the output device. We first of all arrange the viewports into a tree structure, with the `top.vp` as the parent node and all of the other viewports as its children.

```
> splot <- vpTree(top.vp, vpList(margin1, margin2, margin3, margin4, plot))
```

Now we can push this entire tree of viewports in order to create all of the different areas within the drawing region that we need to draw the scatterplot. The result of this push is that we are left in the `plot` viewport.

```
> pushViewport(splot)
```

Now we can navigate to whichever viewport we require and draw the different elements of the plot[3].

The data symbols and axes are drawn relative to the plot region ...

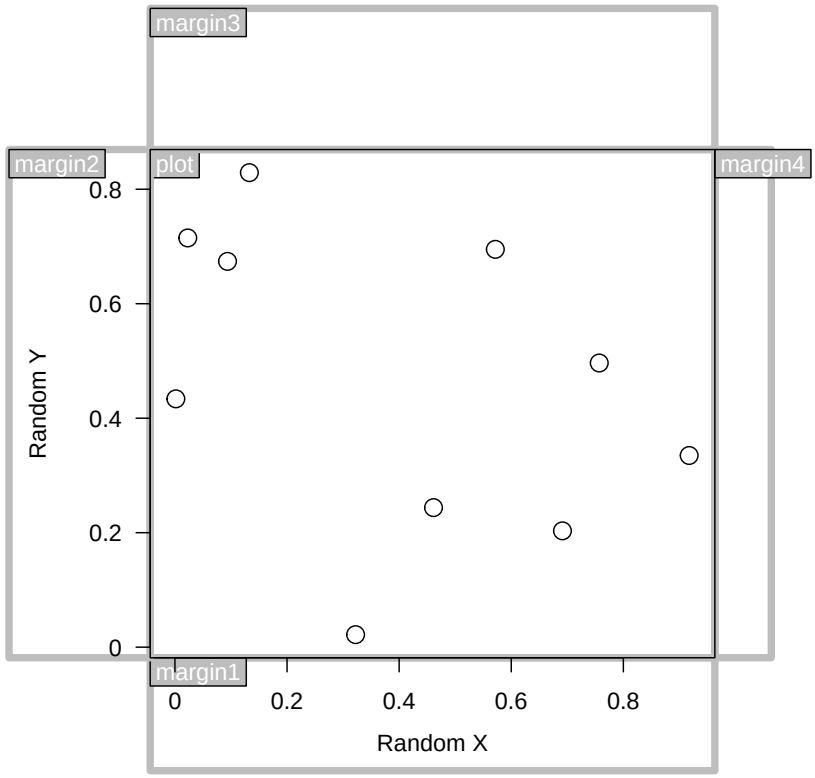
```
> seekViewport("plot")
> grid.points(x, y)
> grid.xaxis()
> grid.yaxis()
> grid.rect()
```

... the x-axis label is drawn in margin 1 ...

```
> seekViewport("margin1")
> grid.text("Random X", y = unit(1, "lines"))
```

... and the y-axis label is drawn in margin 2 (the final output is shown on the next page).

```
> seekViewport("margin2")
> grid.text("Random Y", x = unit(1, "lines"), rot = 90)
>
```



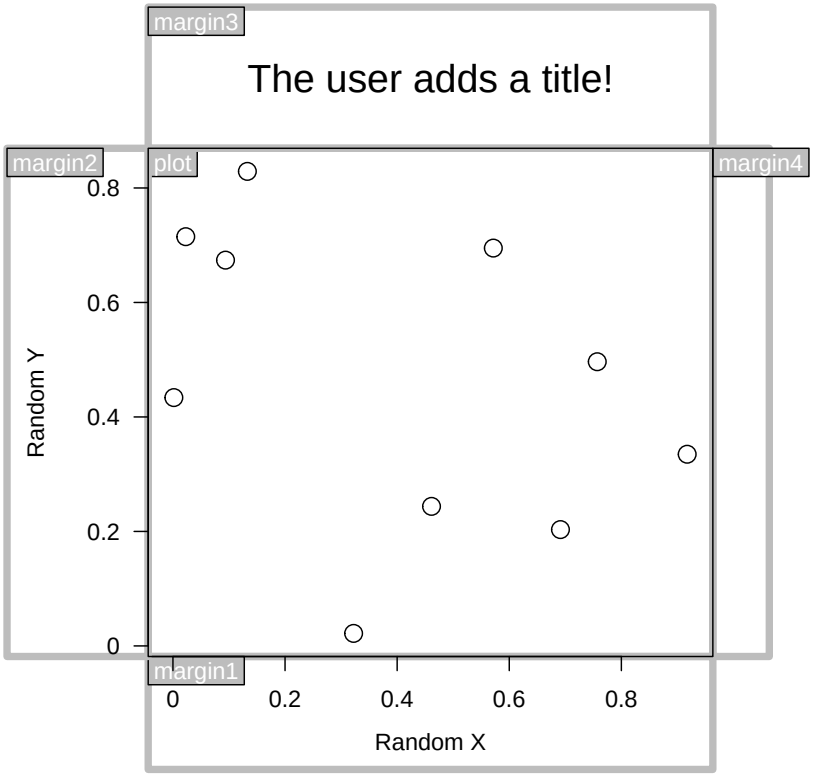
[3]The named viewports that we created are drawn as grey rectangles as a guide.

As a final step, we navigate back to the top-level viewport (i.e., back to the viewport we started in)[4].

```
> upViewport(0)
```

So far this example has just shown an alternative way of constructing this sort of plot. The output we have generated so far could have been done using `pushViewport()` and `popViewport()`. The difference is that we still have all of the viewports in the **grid** viewport tree (and they are addressable by name). This means that a user can seek any of the viewports we used to construct the plot (by name) and add annotations or use `grid.locator()` or whatever. For example, a user could use the following commands to add a title.

```
> seekViewport("margin3")
> grid.text("The user adds a title!", gp = gpar(fontsize = 20))
>
```



[4]Here we have used `0` to indicate "navigate to the top-level viewport". When writing code that could be used by others (i.e., a graphical component that could be embedded within something else), it would be necessary to specify a precise number of viewports to navigate back up. In this case, the number would be 2. The `downViewport()` function returns the number of viewports it went down, so a general solution is of the form: `depth <- downViewport("avp"); upViewport(depth)`.