Duncan Temple Lang, University of California at Davis

Table of Contents

Developing R Graphics Device using R Code

Over the last two decades, R and S-Plus have had graphics devices for common formats such as Postscript, PDF, JPEG, TIFF and PNG. More recently, there is a graphics device that produces TeX code that renders the graphical display when it is processed by TeX itself. As Web 2.0 grows and there is a multitude of ways to display graphics on the Web, it is becoming increasingly interesting to develop R graphics devices which can display R graphics in different formats, venues and media. For example, we might want to create a plot in R to be displayed within a Flash "document". Similarly, we might want to display two R plots in separate JavaScript canvas objects within an HTML page. Or we might want to overlay contour lines or smooth surfaces computed and produced in R but displayed on Google Earth.

Not every new graphics format will stand the test of time, but it is attractive to be able to rapidly generate R graphics targeted for that format. Fortunately, the R graphics device model is extensible so that one can implement new devices as R packages and they will function in exactly the same manner as those that come with R and might be considered "built-in" to R. Unfortunately, to develop a graphics device typically involves writing C-code so that the R graphic's engine can communicate directly and generically with that device. C is a low-level language and is not as convenient for rapid prototyping as a high-level language such as R. Furthermore, many people who might want to target a new format may not know C. Even if they do, they may not want to invest the time needed to write the C code for an experimental graphics device. And a further issue is that often a graphics device will perform a good deal of string manipulation which is cumbersome in C.

These constraints and impediments are our motivation for developing an approach by which R programmers can implement a full-fledged graphics device using R code and no C code. The idea is very straightforward. We implement a single R graphics device via C code. Our graphics device by providing C routines that implement each of the required graphical primitives (i.e. circle, rectangle, text, new page, ...). Each of these routines looks up the corresponding R function registered with this instance of the graphics device to implement the graphical primitive. If it is not registered, the routine does nothing. If it is available, the C routine calls that R function, passing the parameters it was called with, e.g. the x and y coordinates and character string in a call to add text, a collection of x and y coordinates for drawing lines. The R function can do anything it likes and will typically generate content based on the inputs in order to render the graphical element.

In order to create a new graphics device, the R programmer need only supply R functions to implement the graphical primitives in which she is interested. She creates this collection by creating an instance of the S4 class *`RDevDescMethods`* and setting the relevant slots, e.g. the **`line`**, **`rect`**, **`circle`** corresponding to the graphical primitives. This structured collection is then passed to *`graphicsDevice`*() to create the actual graphics device instance and this becomes the active device. While this is still active, any subsequent plotting commands in R are then routed through this device and the specified R functions.

Most R functions have no side effect. In other words, they take inputs and return a result, but do not change any other variables outside of their working environment, i.e. the call frame. This is often termed functional programming and is a very useful paradigm. However, our R functions are supposed to add instructions to create a graphical element and store this somewhere. In this respect, we must update/modify some variables that persist across calls to our graphical primitive functions so that we can collect all the graphical elements in our display.