A Guide to Implementing an R Graphics Device with the RGraphicsDevice package

Table of Contents

Introduction

The idea is that we provide this package so that others can write new graphics devices entirely in R. An R programmer might create an SVG or Flash graphics device by writing R functions that implement the different graphical primitive operations needed by an R graphics device.

To create a running graphics device with our own functions, we call the `graphicsDevice`() function. While there are several methods for this, essentially we give it a list of named functions that specify the implementation of some or all of the 21 graphical primitive operations. We might give this as a list or as an instance of `RDevDescMethods` or of a sub-class that we define for a particular type of device. So we focus on writing these functions.

The names of the functions can be found with

```
[1] "activate"        "circle"          "clip"            "close"
 [5] "deactivate"      "locator"         "line"            "metricInfo"
 [9] "mode"            "newPage"         "polygon"         "polyline"
[13] "rect"            "size"            "strWidth"        "text"
[17] "onExit"          "getEvent"        "newFrameConfirm" "textUTF8"
[21] "strWidthUTF8"
```

Almost every graphics device will need to implement circle, line, rect, polygon, polyline, text and strWidth. For non-interactive graphics devices, e.g. those creating files that are displayed in a separate step, we don't need to implement locator, activate, deactivate, mode, getEvent. onExit is also probably not necessary but can be of value for recovering from errors in evaluating code that produces graphics. If we can handle UTF8 encoded strings, we can implement textUTF8 and strWidthUTF8, but this is not imperative. metricInfo allows the device to return information about the size of a character. A device does not have to provide this information.

So now we must implement each of the important functions needed by our new device. To do this, we need the signatures, i.e. the number and the type of the arguments they will be passed when the R graphics engine invokes these. These are given in the following with the name of the method and the comma-separate list of the R types of the parameters, and the number of parameters on the right:

Table 1. Signatures of the device operations

signature	**# parameters**
activate (DevDescPtr)	1
circle (numeric, numeric, numeric, R_GE_gcontextPtr, DevDescPtr)	5
clip (numeric, numeric, numeric, numeric, DevDescPtr)	5
close (DevDescPtr)	1
deactivate (DevDescPtr)	1
getEvent (ANY, character)	2
line (numeric, numeric, numeric, numeric, R_GE_gcontextPtr, DevDescPtr)	6
locator (numeric, numeric, DevDescPtr)	3
metricInfo (integer, R_GE_gcontextPtr, numeric, numeric, numeric, DevDescPtr)	6
mode (integer, DevDescPtr)	2
newFrameConfirm (DevDescPtr)	1
newPage (R_GE_gcontextPtr, DevDescPtr)	2
onExit (DevDescPtr)	1
polygon (integer, doublePtr, doublePtr, R_GE_gcontextPtr, DevDescPtr)	5
polyline (integer, doublePtr, doublePtr, R_GE_gcontextPtr, DevDescPtr)	5
rect (numeric, numeric, numeric, numeric, R_GE_gcontextPtr, DevDescPtr)	6
size (doublePtr, doublePtr, doublePtr, doublePtr, DevDescPtr)	5
strWidth (character, R_GE_gcontextPtr, DevDescPtr)	3
strWidthUTF8 (character, R_GE_gcontextPtr, DevDescPtr)	3
text (numeric, numeric, character, numeric, numeric, R_GE_gcontextPtr, DevDescPtr)	7
textUTF8 (numeric, numeric, character, numeric, numeric, R_GE_gcontextPtr, DevDescPtr)	7

The choice of parameter names is entirely up to you.

The next topic we need to discuss is the set of classes that are new to R programmers and provided as part of this package. These are *`DevDescPtr`*, *`R_GE_gcontextPtr`*, *`doublePtr`*. The class *`ANY`* used in `getEvent`() means what it says, i.e. any R object.

Each of the methods is passed an object of class *`DevDescPtr`*. This is also the type of the value returned by the top-level function `graphicsDevice`() . This is a reference the C-level data structure that represents the graphics device. We can use this to query the settings of the graphics device. Some of these fields in the device are used when initializing the device rather than within the functions (e.g. those whose names are prefixed with "start"). Other fields are structural information about the rendering of different aspects of the device. For example, we can find the dimensions of the drawing area, The `DevDescPtr` class is essentially an opaque data type in R (containing an external pointer to the C-level data structure) and is intended to be used as if it were an R-level list. We can use the `$`() operator to access individual fields and we can find the names of these fields with `names`() . These are

```
[1] "left"                        "right"
 [3] "bottom"                      "top"
 [5] "clipLeft"                    "clipRight"
 [7] "clipBottom"                  "clipTop"
 [9] "xCharOffset"                 "yCharOffset"
[11] "yLineBias"                   "ipr"
[13] "cra"                         "gamma"
[15] "canClip"                     "canChangeGamma"
[17] "canHAdj"                     "startps"
[19] "startcol"                    "startfill"
[21] "startlty"                    "startfont"
[23] "startgamma"                  "deviceSpecific"
[25] "displayListOn"               "canGenMouseDown"
[27] "canGenMouseMove"             "canGenMouseUp"
[29] "canGenKeybd"                 "gettingEvent"
[31] "hasTextUTF8"                 "wantSymbolUTF8"
[33] "useRotatedTextInContour"
```

Under some rare circumstances, it is convenient to convert the reference to an R object. We can do this by coercing it to the corresponding R class named `DevDesc` (i.e. with the "Ptr" remove), i.e. `as(dev, "DevDesc")`. This copies each of the fields in the C-level structure to the corresponding slot in the R class.

The second of these classes is `R_GE_gcontextPtr`. This is another reference to an instance of a C-level data type. This is the information about the "current" settings of the device. This gives us information about the current pen/foreground color, the background color, the setting for the gamma level, the line width, style, join, the character point size and expansion/magnification, and the font information. The available fields are

```
names(new("R_GE_gcontextPtr"))

 [1] "col"        "fill"       "gamma"      "lwd"        "lty"
 [6] "lend"       "ljoin"      "lmitre"     "cex"        "ps"
[11] "lineheight" "fontface"   "fontfamily"
```

These are the values that your graphics device must reflect when it renders the display. These control the colors, line characteristics and fonts.

Many of these fields are scalar values. **`lend`** and **`ljoin`** are special types that are enumeration constants. These identify particular types of line endings and line joins. **`fontfamily`** is a character vector with 201 individual characters.

The one other class of parameter is `doublePtr`. This is a simple reference to an R numeric vector. The only thing that is needed to convert this to a numeric vector is the number of elements. In the two methods (`polyline`() and `polygon`()), we are given the length of the vector in the first parameter. This allows us to convert these references to R numeric vectors as `x[ 1 : n ]` where **`n`** is the length of the vector.

So now that we know about the types in the functions, we can start to define the methods for a particular type of graphics device. We'll focus on what these functions might do later. But first we'll talk about how we gather them together to form a device. We can collect these functions in an instance of the `RDevDescMethods` class. We first create an instance and then set the slots, e.g.

```
funs = new("RDevDescMethods")

funs@activate = function(dev) { cat("Activating the device\n") }
funs@line =
 function(x1, y1, x1, y1, gcontext, dev) {
      # do something to render the line
 }
```

Alternatively, we can collect the functions into a list and coerce this to an `RDevDescMethods`. And when we are exploring an implementation, we might want to use functions that print the name of the method each time the are called. *`dummyDevice`*() creates such an instance.

Initialization via *`initDevice`*()

In addition to the twenty one graphical primitive methods in `RDevDescMethods`, there is also *`initDevice`*() that is not called by the R graphics engine. If this is not **`NULL`**, we call this just after creating the `DevDescPtr` object in C but before initializing and registering the device with the R graphics engine. This is an opportunity to set different fields in `DevDescPtr` such as the **`col`** and **`fill`** that are propagated through the graphics system when the device is initialized. Some of these parameters (*`col`*, *`fill`*, *`ps`*) can be specified directly in the call to *`graphicsDevice`*() and if we are just setting these, we do not need a function for *`initDevice`*() . However, if we need to set additional fields, we can do so before the device is initialized by the graphics engine with *`initDevice`*() . Alternatively, if we can set the fields of interest after the device is registered, we can do so directly with the return value of the call to *`graphicsDevice`*() .

We should note that calling R functions from C is more expensive than calling C routines from C. Also, some of the operations that graphical primitive functions can be time consuming and implementing them in R can be very slow. For example, drawing many, m

Examples

The package contains two example devices. They do not produce polished graphics, but serve as prototypes for other to hopefully take and complete. These are in `examples/JavaScriptCanvas`/ and `examples/SVG`/ 1

Drawing on the JavaScript Canvas

Several browsers provide a `<canvas>` element for HTML documents and we can use this and JavaScript to draw within the canvas' area within the HTML document. Information about the API for the JavaScript canvas is available at https://developer.mozilla.org/en/drawing_graphics_with_canvas

The file Rjs.R in `examples/JavaScriptCavas`/ contains the code that implements the device *`jsCanvas`*() and also a related derived device that creates HTML documents, *`htmlCanvas`*() . We'll look at the *`jsCanvas`*() first. It is defined as

```
jsCanvas =
function(file, dim = c(1000, 800), col = "black", ps = 10, wrapup = writeCode, ...
{
}
```

The *file* specifies the connection or file name to which the generated JavaScript code should be written when the device is closed. This can also be a quoted expression (created by *quote*() or *expression*()) giving the R graphics commands. In this case, the plot will be created and the generated code will be returned directly rather than being written to a file.

The *dim*, *col* and *ps* parameters are passed on directly to *graphicsDevice*() to initialize its state. We'll return to *wrapup* later, but suffice to say that it is called to post-process the generated JavaScript code and write it to the connection, if appropriate.

The body of the *jsCanvas*() starts by creating a list (**pages**) in which the generated code for the different plots will be stored, with a character vector for each separate plot. The code for the current plot being created is stored in **commands**.

```
pages = list()
  commands = character()
```

Each of the graphical primitive functions for the device add their code to the **commands** vector as they are called. When the plot is completed, either when the device is closed or when we start a new plot, the approriate method calls *endPage*() which moves the code for the current plot from **commands** into **pages**.

```
add = function(x)
    commands <<- c(commands, x)

  endPage = function() {
      if(length(commands)) {
         pages[[ length(pages) + 1 ]] <<- commands
         commands <<- character()
      }
  }
```

These are helper functions that organize the generated code. We now move on to the functions implementing the graphical operations.

We start by creating a dummy device

```
funs = as(dummyDevice(), "RJavaScriptCanvasMethods")
```

We don't implement several of the functions, so we assign them the value **NULL**:

```
funs@mode = NULL
  funs@metricInfo = NULL
  funs@activate = NULL
  funs@deactivate = NULL
  funs@deactivate = NULL
  funs@locator = NULL
  funs@onExit = NULL
```

We can specify the important device settings such as initial color and point size directly via *graphicsDevice*() and in our *jsCanvas*() . However, we might also want to specify addition device settings. We can do this after the call to *graphicsDevice*() using the *DevDescPtr* object it returns. Alternatively, we can have R call an *initDevice*() function we provide and this is called after we create the device and set its methdods, but before the device is passed back to the R graphics engine to initialize the state, e.g. the *par*() settings. So we might have a function of the form:

```
funs@initDevice = function(dev) {
      # The all important parameter to set ipr to get the plot region with adequa
    dev$ipr = rep(1/72.27, 2)
    dev$cra = rep(c(6, 13)/12) * 10
    dev$canClip = TRUE
    dev$canChangeGamma = TRUE
    dev$startgamma = 1
    dev$startps = 10
    dev$startcol = as("black", "RGBInt")
  }
```

Now we move on to the functions that actually generate content. We'll start with drawing a line. We are passed the coordinates of the two end points, the current graphical context and the device. We generate JavaScript code to set the JavaScript drawing parameters/context and then draw the line. We do this by creating path, moving to the starting point and drawing a line to the end point.

```
funs@line = function(x1, y1, x2, y2, context, dev) {
    add(c("// line",
          "ctx.beginPath();",
          setContext(context),
          sprintf("ctx.moveTo(%s, %s);", as.integer(x1), as.integer(y1)),
          sprintf("ctx.lineTo(%s, %s);", as.integer(x2), as.integer(y2)),
          "ctx.stroke();"))
}
```

The code is appended to **`commands`** via the *`add`*() . Since this is a shared vector (as is **`pages`**), we define the *`add`*() function in the scope of our call to *`jsCanvas`*() , and we define our graphics operator functions in that same lexical scope to be able to access *`add`*() .

The function *`setContext`*() takes the R graphics context (of class *`R_GE_gcontextPtr`*) and generates JavaScript code to set the JavaScript graphics context accordingly. Since this just returns the generated code and it is then passed to *`add`*() , the *`setContext`*() function does not need access to **`commands`** and so is defined outside of *`jsCanvas`*() . We'll return to it later as it illustrates some additional facilities of RGraphicsDevice that are useful.

To draw a rectangle, we can use the built-in JavaScript functions strokeRect or fillRect. The former draws just the border and the latter fills in the entire area. Which we use depends on whether the **`fill`** graphics parameter in R is set. We determine this by checking whether the **`fill`** field corresponds to the "color" transparent. The *`isTransparent`*() function hides the details.

```
funs@rect = function(x1, y1, x2, y2, context, dev) {
    op = if(!isTransparent(context$fill))  "fillRect" else "strokeRect"

    add(c("// rect",
          setContext(context),
          sprintf("ctx.%s(%d, %d, %d, %d);",
                       op,
                       as.integer(min(x1, x2)), as.integer(min(y1, y2)),
                       abs(as.integer(x2 - x1)), abs(as.integer(y2 - y1)))))
  }
```

The remainder of the graphical operations are similar, except for those related to text. These are often quite tricky as we have to deal with different fonts and computing the dimensions of the rendered string and even rotating the text and working with that. We have not taken the time to make this pretty for our prototype. Indeed, this is an area where the JavaScript canvas is quite weak. We use Jim Studt's drawing of Hershey fonts for the moment. You can find the code and more information at http://www.federated.com/~jim/canvastext/.

To handle text, we implement both the *`text`*() and *`strWidth`*() functions. We do the simplest thing for computing the width of the string which is to multiply the number of characters by the current font size and the character expansion setting. This is done via the following function:

```
funs@strWidth = function(str, gcontext, dev) {
     nchar(str) * max(10, gcontext$ps) * gcontext$cex
  }
```

Here we see that we are accessing the current point size and character expansion from the *`gcontext`* parameter. Rendering the text is similar to drawing a line or circle. For the JavaScript device, we ignore the rotation and horizontal adjustment for now.

There are two other functions that are important to implement. The first of these is *`newPage`*() and this is called when the R graphics engine is starting a new plot. For our device, this is when we move any generated code in **`commands`** into the **`pages`** list. So we call *`endPage`*() :

```
funs@newPage = function(gcontext, dev) {
     endPage()
  }
```

The second function is *`close`*() which is invoked when we close the R graphics device. This too much take care of moving any generated code into **`pages`**. But it must also output all the generated. Here we call the function that was specified via the *`wrapup`* parameter. By default, this is an external helper function *`writeCode`*() and it is passed the list of generated plot commands (**`pages`**), the *`file`* argument and any additional arguments provided via the **...** mechanism. *`writeCode`*() turns the code for each plot into a separate JavaScript function, adding some initialization JavaScript commands to retrieve the JavaScript graphics context from the associated HTML canvas. Then it writes these JavaScript function definitions to the specified connection.

Displaying the JavaScript

This graphics device merely generates the JavaScript code that can be run to display the plot(s). We do this in a Web browser and we do it by having the JavaScript code be included in an HTML document. The file template.html provides, as the name suggests, a template HTML document that you can use. We can generate the JavaScript code for the plot with a command something like

```
jsCanvas("myPlot.js", c(500, 500))
 library(maps)
 map('usa')
dev.off()
```

Then we can edit the template.html file. We add a `<script>` element to reference the newly generated myPlot.js file. We also set the identifier for the canvas to correspond to the value used in the JavaScript function we generated. Finally, we add a call to that function in the `onload` attribute of the `<body>` element.

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html> <head>
```

```
<script type="application/x-javascript" src="canvastext.js"></script>
<script type="application/x-javascript" src="myPlot.js"></script>
<title>Example of the Canvas</title>
</head>

<body onload="rdraw()">

<canvas id="canvas" width="1000" height="800" >
  No support for JavaScript canvas
</canvas>

</body>
</html>
```

This isn't very arduous, but it is tedious. We can have R do this for us and that is what the derived graphics device function *htmlCanvas*() does for us. This is easily defined as

```
htmlCanvas =
function(file, dim = c(1000, 800), template = "template.html")
{
  jsCanvas(file, dim, wrapup = htmlWrapup, template = template, dim)
}
```

It is essentially just a call to *jsCanvas*() , but the key is that we provide our own function to do the *wrapup* and emit the generated code. This *htmlWrapup*() function will create an HTML document with the code inside it. This function will read the template HTML file and will use the existing `<canvas>` element and its `id` attribute value when generating the JavaScript code. It will add the `<script>` element and insert the code directly into that element rather than keeping it in a separate file. Then it updates the `<body>`'s `onload` attribute. It will also set the width and height attributes of the canvas element to match the dimensions of the graphics device.

So we can now use this in calls of the form

```
library(maps)
htmlCanvas("usa.html")
 map('usa')
dev.off()
```

If we generate more than one plot, the *htmlWrapup*() handles this by adding new canvas elements, giving the different identifiers, using these when generating the JavaScript functions and adding calls to all of them in the `onload` attribute.

The *R_GE_gcontext* classes

We mentioned the *setContext*() function above. This accesses the values from R's graphics context and sets the corresponding fields in the JavaScript graphics context. Most of the values are pretty straightforward. The point size is an integer; similarly, line width and line type are integers.

The graphics context contains the drawing color and the filling color. These are represented as integers and interpreted in a specific manner by R. See the file GraphicsDevice.h for more information about the meanings of the different bytes. To make things simpler (we hope), we have provided two classes in the

RGraphicsDevice. These allow us to convert the integer to a string when interpreting the integer and using it in another system, and to convert from a string to an integer when specifying a color for R to use. These classes are named *RGB* and *RGBInt*. We can use them via regular coercion. For example,

```
as(gcontext$col, "RGB")
```

converts the color in R's graphics context to a string. This might be a named color such as "red" or "yellow" or an hexadecimal RGB string, e.g. "#FF882300". The extra digits give the alpha level for the degree of transparency. If we want to set a color in R, we can use RGB strings or named colors and convert them to an integer with, e.g.

```
as("red", "RGBInt")

An object of class "RGBInt"
[1] 4278190335
```

The **lend** and **ljoin** fields are like integers, but have a small set of possible values. They correspond to enumerated constants in C and we map these into *EnumValue*s in R. (For now, you should coerce them yourself as the C code does not explicitly do this at present but will in the future.) If we look at this value, we might see something like

```
as(gcontext$lend, "R_GE_lineend")

             GE_ROUND_CAP
R_GE_lineend            1
```

The name is the value is the important thing to note. This is the human-readable name and the value we should use. If we want to set the value for the line ending, we should use this name. We can do this in either of the following ways:

```
gcontext$lend = "GE_ROUND_CAP"
 gcontext$lend = GE_ROUND_CAP
```

The difference is simply that in the first case, we are coercing the name to an instance of the *R_GE_lineend* class, and in the second, we already have an actual R variable with the corresponding name that is an instance of that class. So we typically use the second approach when setting the value. For accessing the value and using in, for example, JavaScript code, we need to map the name to the corresponding value in Java. JavaScript uses "round", "butt" and "square". We can map our values of "GE_ROUND_CAP", "GE_BUTT_CAP" and "GE_SQUARE_CAP" to these in whatever way we think best. In our JavaScript device, we use string manipulation in the *jsLineCap*() function. We do the same for line joins.

The final part of the R graphics context object that needs some explanation is the font information. The **fontface** is an integer. 0 corresponds to plain, 1 to italic and 3 to bold. The **fontfamily** is a character vector of length 201. Each element is a single character. Look at the C code in *grDevices* package for more information on how this is interpreted.

We have used the approach of creating a sequence of JavaScript commands that render the plot. It is valuable to instead create objects that correspond to the graphical elements and render those. While the visual result is the same, the objects can be programmatically manipulated after they have been created. We can hide objects, change their appearance, move them in animations and allow the viewer to modify them with GUI controls. To be able to do this, we need to be able to a) create objects, and b) associate the objects in the plot with the different elements of the plot, e.g. axes, tick marks, title, data points. One of the benefits of doing this in the R language is that we can examine the call stack via *sys.calls*() and this allows us, albeit in an ad hoc manner, to determine from where the drawing operations were called and to which part of the plot they correspond.

SVG Device

R provides a rich Cairo-based graphics device and there are two C-level graphics devices that generate SVG. Also, we can annotate SVG generated from the Cairo-based device using SVGAnnotation. However, as an illustration and also because we can do more things within R than at the C-level, there is a very simple implementation of an SVG graphics device in the package in `examples/SVG/` .

The *size*() operation

A graphics device has a *size*() method/operation and it is expected to return information about its location and dimensions. In C, it is passed references to 4 numbers and is expected to fill these in. In this interface, we are also passed references to 4 C-level number data types and expected to fill them in. We do this by treating the objects as if they are regular numeric vectors and setting the first value of each. So our default size function might be

```
size =
function(left, right, bottom, top, dev) {
  left[1] = 0
  right[1] = dev$right
  bottom[1] = dev$bottom
  top[1] = dev$right
}
```

We might allow the function to return a numeric vector of length 4 and have the C code insert the values into the corresponding C references.

Future Directions

We have provided a relatively straightforward one-to-one mapping of the internal code to R functions. There are additional features we could add and different idioms and interfaces we could implement. We will at some stage make additional internal R graphics functionality available so that these can be used by an R programmer implementing an R graphics device.

We will also make it possible to dynamically modify the C routines that implement an internal graphics device, e.g. the C routine that is called to draw a line. While we have provided routines that call the corresponding R function in the device, it is useful to be able to implement some of these primitives with R functions and others with C routines and mix code across the languages. This is quite easy and amounts to not removing the function pointers from the code we generate in the tu.R script.

We can avoid lexical scoping by maintaining a state object. We can create an object which represents the current state of the device and store it in the device's state slot. Each of our graphics engine primitive functions have access to this and can both query and set it. This can be done now.

We have specified the functions to use to implement the graphical primitives. This is the most direct way of doing things. Another approach is to use generic functions for these graphical primitives. Then the generic graphics device would invoke the appropriate method based on the class of the graphics device. To define a device, we would define a sub-class of *`DevDescPtr`* and arrange for the C code that invokes our existing proxy C routines to convert the instances to that type of object. (This object could be stored in the device specific state field.) We can implement this with the existing direct framework and the benefit is that this is a more common and familiar object-oriented programming approach for R programmers. Other than that, it is not necessary.