The gridSVG package

Paul Murrell

October 24, 2019

Introduction

This package is an experiment in writing a graphics device, purely in R code, for the grid graphics system. The specific device implemented is for the W3C SVG (Scalable Vector Graphics) format, but there is also some effort at a general device interface that would support other specific devices.

User Interface

There are five functions of interest to the user:

`grid.hyperlink` takes a grid `grob` and turns it into an object of class `linked.grob`, with an associated `href`. This allows the association of hyperlinks with elements of a grid graphic. See `gridSVG/tests/testlink.R` for examples.

`grid.animate` allows the user to associate a duration (plus some other things) with certain aspects of a grid `grob`. This allows a grid graphic element to be animated. See `gridSVG/tests/testanimate.R` `testpendulum.R` and `testball.R` for examples.

`grid.garnish` allows the user to associate arbitrary SVG attributes with a grid `grob`. This provides a way to associate with a grob things that have no corresponding grid concept, like an `onclick` attribute. See `gridSVG/tests/testattrib.R` for a simple example.

`grid.script` allows the user to create a grid `grob` that contains an SVG script (e.g., some ECMAscript code). This provides a way to produce a complete SVG document (complete with scripts) entirely using R grid code (i.e., without having to hand edit the SVG file that `gridSVG` creates. Again, see `gridSVG/tests/testattrib.R` for a simple example.

`gridToSVG()` saves the current grid graphic to an SVG file. See the `gridSVG/tests` directory for examples of what can be done. See the section "Known Problems" below for things that are not yet supported.

In addition to these functions, `gridSVG` supports alpha-transparency by respecting the `alpha` graphical parameter which can be specified in a `grid` `gpar` object. For example, the following code produces overlapping transparent circles[1]:

```
> pushViewport(viewport(gp=gpar(col="black", fill=NA)))
> grid.circle(x=0.33, r=unit(2, "inches"), gp=gpar(alpha=0.3, fill="red"))
> grid.circle(x=0.67, r=unit(2, "inches"), gp=gpar(alpha=0.3, fill="green"))
> popViewport()
> gridToSVG()
>
```

Internal Structure

There are nine `.R` files in the `gridSVG/R` directory, corresponding to the nine different things that gridSVG aims to provide:

dev.R This contain (S4 methods) code defining a generic R-level graphics device interface. In other words, generic functions that may be called by a graphics system (such as grid), and that a graphics device (such as an SVG device) should provide methods for.

griddev.R Code for running through the grid display list and calling generic device functions.

devsvg.R Code implementing SVG methods for the generic device interface.

svg.R A set of R-level functions for producing SVG output. Callable directly (see, e.g., `gridSVG/tests/testsvg.R`), but mostly just called by code in `devsvg.R`.

gridsvg.R The function `gridToSVG()`.

hyper.R Code implementing the `linked.grob` class – i.e., an extension of the standard grid `grob` that supports hyperlinks. Includes the function `grid.hyperlink()`.

animate.R Code implementing the `animated.grob` class – i.e., an extension of the standard grid `grob` that supports animation. Includes the function `grid.animate()`.

script.R Code implementing the `script` class – i.e., an extension of the standard grid `grob` that supports SVG scripts. Includes the function `grid.script()`.

attrib.R Code implementing the `svg.grob` class – i.e., an extension of the standard grid `grob` that supports arbitrary SVG attributes. Includes the function `grid.garnish()`.

[1] The `pushViewport()` call is currently necessary to set some default values. It may be possible to remove this in future versions.

Known Problems

This package is a partial implementation of several ideas. This section describes some of the known holes in and issues with the implementation.

Overall Design

The package is ass-backwards in its design. Normal devices receive calls from grid to perform operations; gridSVG works off grid's display list so only has the information stored there to figure out what to do. This means that it has to replicate some of the work that grid does when grid draws (e.g., in order to enforce vp slots in grobs). If/when normal devices are implemented as R-level objects, so that grid includes a `dev` argument in all its calls to devices, it may be possible to make gridSVG behave more like a normal device and this may lead to some simplifications.

Sizing of and units in the SVG image

Software that tries to render SVG on a device has the same problem that `R` graphics devices have when trying to render `grid` output: Locations and sizes can be in a variety of units (cm, inches, percentages, ...) *some of which are physical units* with real-world meaning. The renderer has to figure out how big something like `1"` is in the native device units. This problem is worst on computer screens where it is not necessarily easy (or possible) to find out how many pixels there are in a physical inch on the screen. What `R` does is try its best and it seems that SVG renderers must do the same[2].

gridSVG works off the grid display list. This means that the image must first be drawn on some other device (e.g., X11 or PostScript) then copied (via the `gridSVG()` function) to an SVG format.

It is not possible to use the SVG notion of transformations to mirror `grid`'s viewport transformations because the SVG transformations work on ALL graphical elements, including text. In particular, any scaling transformations scale the size of text. Furthermore, `grid` actions such as `upViewport()` and `downViewport()` are difficult to replicate as SVG transformations. So the copying of `grid` output to SVG involves converting all locations and sizes to a single SVG coordinate system.

There are two (serious) possibilities for this coordinate system:

1. specify everything (including the size of the SVG image) in pixels. In this case, what we do is work off the original device's concept of a pixel. The SVG image may be rendered quite a different size compared to the original if the size of pixels on the rendering device is different from the

[2] According to Section 7.1 Introduction of the W3C Scalable Vector Graphics (SVG) 1.0 Specification, "a real number value that indicates the size in real world units, such as millimeters, of a "pixel"" is "highly desirable but not required".

size of pixels on the original device. Things should be pretty consistent – something that is supposed to be half the size of the image should be rendered half the size of the image – though this will *not* be the case if the change in pixel size is different for x- and y-axes. An image drawn first in a screen window then copied to SVG and viewed *on the same screen* should hopefully be the same size.

If the original device is a screen device, there is no guarantee that physical sizes will be respected; this will depend on how accurately the screen device can determine the physical size of its pixels. If the original device is a file device (e.g. PostScript) then physical sizes will be accurate on the original device, BUT the correspondence between "pixels" on the file device and pixels when the SVG is rendered on screen is very unlikely to be good (e.g., "pixels" on PostScript are $\frac{1}{72}$", which is highly unlikely to correspond to pixel size on a modern screen).

2. specify everything (including the size of the SVG image) in inches. In this case, there is no guarantee that the SVG image will end up the right physical size (it will depend on whether the rendering software can find out enough about pixels-to-inches, BUT everything should be in proportion (if the image is overall a little smaller than it should be, at least something that should be half the size of the image will be half the size of the image. The final rendered size of the image will totally depend on where it gets rendered[3]. This appears to have fewer problems; unfortunately it is **totally killed** by the fact that the locations for drawing polylines MUST be in pixels! (technically, that should be "user coordinates", but since I have a single, flat coordinate system structure, it equates to pixels.)

Plotting Symbols

Only `pch=1` and `pch=3` are currently supported. This is just a matter of filling in the other options.

Mathematical Annotation

The use of things like `grid.text(expression(x[i]))` is supported in the main R graphics engine. gridSVG bypasses that and so does not support mathematical annotation (and probably never will!).

Time unit arithmetic

Animation is no longer achieved via "time units" so old problems with arithmetic on time units in previous versions of `gridSVG` disappear (i.e., the user does not need to be as careful when doing animation; just specify some unit values and/or expressions and it should go.) Having said that, there is still only support for

[3]This is not to say that it should be riciculously off; it should be pretty close to the right physical size if it's not exactly the right size.

animating a small set of aspects of grid `grob`s (basically the locations and sizes of `grob`s.

Acknowledgements

Many thanks to Nathan Whitehouse and colleagues who contributed ideas and code for including arbitrary SVG attributes and SVG scripts.