Importing text

Paul Murrell
The University of Auckland

April 22, 2010

This vignette concentrates on the issue of importing *text* from an external graphics image (which has much more sophisticated support in the **grImport** package as of version 0.6).

Figure 1 shows a very simple image that contains text. This is a PostScript file called `hello.ps`, which just displays the word "hello".

Figure 1: A simple image that contains text.

Importing text as paths

The default approach for importing text is to convert all characters into paths and stroke (draw the outline of) the paths. The following R code traces the simple text image shown in Figure 1, reads the resulting RGML file into R, and draws it. The image drawn by R is shown in Figure 2.

```
R> library(grImport)

R> PostScriptTrace("hello.ps", "hello.xml")
R> hello <- readPicture("hello.xml")
R> grid.picture(hello)
```

A small improvement can be gained by using a thinner line to draw the text outline and it is possible to improve the smoothness of the character outlines via the `setflat` argument when tracing the original image. The following code provides an example, with the result shown in Figure 3.

Figure 2: The simple image from Figure 1 after default tracing, importing, and rendering by **grImport**.

Figure 3: The simple image from Figure 1 after smoother tracing, normal importing, and rendering with a thin line. Compare the smoothness of the 'o' in this image with the rougher 'o' in Figure 2.

```
R> PostScriptTrace("hello.ps", "hello-smooth.xml", setflat=0.5)
R> helloSmooth <- readPicture("hello-smooth.xml")
R> grid.picture(helloSmooth, use.gc=FALSE, gp=gpar(lwd=1))
```

The characters are drawn as outlines by default because the character shapes are complex polygons that R cannot necessarily render correctly. It is possible to instruct **grImport** to use a simple algorithm that attempts to fill the characters (basically fill the first path in the character using the text colour then fill the remaining paths in the character using another "background" colour). This approach is controlled via the `fillText` and `bgText` arguments to `grid.picture` (technically, they are arguments passed to the `grobify` function). The `bgText` argument is either a single colour, or a named vector of colours; in the latter case, a different background colour can be specified for different characters (e.g., `"white"` to fill the hole in an 'o', but `"black"` to fill the dot in an 'i'). The following code draws the smoothly-traced simple text image by filling the character paths (see Figure 4).

```
R> grid.picture(helloSmooth, fillText=TRUE)
```

This simple filling algorithm will not work with all characters and all fonts and it will not work if the background for the rendered image is supposed to be transparent. Nevertheless, the result in Figure 4 has the nice property that it is very close to the original text. If there is a small amount of large text in an image, this approach may produce the best result.

However, it is important to note that filling the paths of characters is not the same thing as drawing text using fonts (e.g., fonts contain "hinting" information for drawing at small sizes), so for an image that contains lots of small text, this

Figure 4: The simple image from Figure 1 after smoother tracing, normal importing, and rendering by filling the character paths.

Figure 5: The simple image from Figure 1 after tracing *as text* and normal importing and rendering. The dotted boxes indicate the bounding boxes for the characters in the original text.

approach is probably not the best idea. If case space is an issue, tracing text as character paths will also produce a large RGML file.

Importing text as text

The alternative to converting text to a set of character paths is to simply record the text as a character value, plus its location and size. This is achieved via the `charpath` argument to `PostScriptTrace` and the result is rendered as text by **grImport**. The following code does this for the simple text image and the result is shown in Figure 5.

```
R> PostScriptTrace("hello.ps", "helloText.xml", charpath=FALSE)
R> helloText <- readPicture("helloText.xml")
R> grid.picture(helloText)
```

This result is not as nice as the result from tracing the text as paths. However, it does have the benefit that it is drawing the text using fonts. This means that, although it does not look exactly like the original text, it will look cleaner and clearer than the filling-paths approach from the previous section, especially at small sizes, plus we do not have to worry about things like getting the central hole in an 'o' the right colour. The rendering will also probably be faster if that is an issue.

The major difference between Figure 5 and the original image is the font. The default text font in R graphics is a sans-serif font (Helvetica in this document), while the font in the original image is (serif) Times Roman. One thing that we can do is at least make the font a lot closer to the original by selecting a

Figure 6: The simple image from Figure 1 after tracing *as text*, normal importing, and rendering with a Times Roman font. The dotted boxes indicate the bounding boxes for the characters in the original text.

Figure 7: The simple image from Figure 1 after tracing *as text*, normal importing, and rendering using the original text height to choose the font size. The dotted boxes indicate the bounding boxes for the characters in the original text.

serif font for drawing text. This is what the following code does and the result should match up much better (it matches up *very* well in this PDF document because the default serif font for the PDF device is Times Roman; see Figure 6).

```
R> grid.picture(helloText, gp=gpar(fontfamily="serif"))
```

Note that, although Figure 6 looks very much like Figure 4, they are actually quite different; the former is drawing the text "hello" using a Times Roman font, while the latter is drawing a set of character paths.

Drawing with the same font as the original text is not always going to be possible. It may be difficult to determine what the original font is (though see later) and the original font may not be available (it may not be installed on the computer where R is doing the drawing). In such cases, we have to make do with the fonts at our disposal and the main problem that we face is determining the font size to use for drawing text.

Going back to Figure 5 (i.e., the original text drawn with the wrong font), this shows the default behaviour for drawing text with **grImport**, which is to choose a font size so that the text will end up the same *width* as the original text.

An alternative is to choose a font size based on the font size used in the original text. This is done via the `sizeByWidth` argument, as shown in the following code and Figure 7.

```
R> grid.picture(helloText, sizeByWidth=FALSE)
```

Figure 8: The simple image from Figure 1 after tracing *as text*, normal importing, and rendering by positioning each character based on the individual character locations in the original text (and using the original text height to choose the font size). The dotted boxes indicate the bounding boxes for the characters in the original text.

It is important to note that font size (usually expressed as a number of "points", e.g., 10pt) is actually only an indication of the size of the characters in a font. As Figure 7 shows, the characters from a 10pt Helvetica font are actually larger than the same characters from a 10pt Times Roman font (the font that was used in the original text).

In this particular example, the result of choosing font size based on the font size of the original text is worse than choosing font size based on the width of the original text. However, when an image contains several pieces of text at the same font size, this approach will at least reproduce those text elements at the same size as each other (even if neither their heights nor their widths are completely faithful to the original image).

A further piece of fine tuning can be applied when tracing the original image. The `charpos` argument to `PostScriptTrace()` can be used to specify that text is recorded as individual characters, each with its own location. When combined with choosing font size based on the original font size, this may produce a better result than drawing the entire piece of text. The following code shows how to perform this sort of tracing and Figure 8 shows the result. The overall width of the result is much closer to the original text width, at the cost of some unusual spacing between characters.

```
R> PostScriptTrace("hello.ps", "helloChar.xml",
+                  charpath=FALSE, charpos=TRUE)
R> helloChar <- readPicture("helloChar.xml")
R> grid.picture(helloChar, sizeByWidth=FALSE)
```

It is also possible to trace the original text as individual characters, each with their own locations, but to size by the *width* of the original characters. However, this is unlikely to produce a very useful result because the font size will probably vary for individual characters within a single piece of text. For completeness, the following code and Figure 9 demonstrate this approach.

```
R> grid.picture(helloChar)
```

Figure 9: The simple image from Figure 1 after tracing *as text*, normal importing, and rendering by positioning each character based on the individual character locations in the original text (and using the width of the original characters to choose the font size). The dotted boxes indicate the bounding boxes for the characters in the original text.

Implementation Details

The following information is recorded for each piece of text:

string: the text itself (even if the text is being traced as character paths);

x, y: (bottom-left) location of the text;

width and height: width and height of the text, though the latter is based on the font size (in points) not the phyical height of the text;

angle: angle of rotation (degrees anti-clockwise from the positive x-axis);

bbox: tight bounding box for the text, which will depend on the characters in the text, not just on the font;

fontName: the name of the font (and depending on the font there may also be one or both of **fontFamilyName** and **fontFullName**).

The font name information is not read into R, but it is recorded in the RGML file, so could be parsed to attempt to extract. for example, the font face (italic or bold) for text elements of an image.