# Enhanced Vector Image Import: The grConvert and grImport2 packages

*by Simon Potter and Paul Murrell*

**Abstract** Two new packages, **grConvert** and **grImport2**, have been created to improve the support for importing external vector images and rendering those images as part of an R graphics scene. Where the old **grImport** package can import PostScript images, the new **grImport2** package can import SVG images, which means that more complex images can be imported. The **grConvert** package provides functions for converting images to either PostScript or SVG for importing into R.

## The problem

For some time, the **grImport** package (Murrell, 2009) has allowed PostScript files to be imported into R and drawn as part of an R plot. As long as the original image is a PostScript image and the image consists of only paths and text, the **grImport** package performs quite well. For example, Figure 1 shows a test image consisting of a coloured background rectangle, some text, and a circle. The following code is all that is required to import that image and draw it in R (the result is shown in Figure 1).

```
> library(grImport)

> PostScriptTrace("testimage-simple.ps", "testimage-simple.xml")
> image <- readPicture("testimage-simple.xml")

> grid.picture(image)
```


**Figure 1:** On the left is a simple PostScript test image consisting of only paths and text. On the right is the result of importing and rendering the test image using **grImport**. Because the original image is simple, **grImport** does a good job of reproducing it in R.

However, there are two main weaknesses in the **grImport** package: not all images that we might want to import are in a PostScript format; and **grImport** does not import all possible PostScript images.

An example of the latter weakness is the fact that **grImport** does not import any raster elements from the original PostScript image. To demonstrate this, Figure 2 shows a test image that includes a small checkerboard raster element in the top-right corner, along with the result of importing and rendering that image with **grImport** (to show that the raster element does not make it).


**Figure 2:** On the left is a PostScript test image that contains a raster element in the top-right corner. On the right is the result of importing and rendering the test image using **grImport**. The raster element is lost from the image during the import process.

When the original image is not in a PostScript format, it must be converted to PostScript using third-party software, such as ImageMagick or Inkscape (Still, 2005; Bah, 2007), and this creates two further problems: first, it places a burden on the user to install and learn to use a conversion utility; second, and worse, the conversion to PostScript may result in degradation or loss of features from the original image.

An example of the second problem is shown in Figure 3, where an original image is in an SVG format and it contains a semitransparent fill. Converting this image to PostScript with ImageMagick

leads to rasterisation of the image and we have already seen that **grImport** cannot deal with raster input.


**Figure 3:** On the left is a test image that is in an SVG format and that contains a semitransparent fill. In the middle is the result of converting the original image to a PostScript format with ImageMagick, which results in rasterisation of the image. On the right is the result of importing and rendering the converted image with **grImport**, which does not work very well because **grImport** does not import raster elements.

In some cases, the conversion problem can be worked around simply by using a different conversion tool. For example, Inkscape does a better job than ImageMagick at retaining semitransparent fills during conversion to PostScript. However, for more complex images (or images containing more sophisticated features), there may be no solution. Figure 4 shows another test image (requiring conversion because it is in an SVG format), that contains a semitransparent *gradient* fill. The conversion to PostScript results in either rasterisation, if we use ImageMagick, or simply loss of the fill information, if we use Inkscape.


**Figure 4:** On the left is a test image that is in an SVG format and that contains a semitransparent gradient fill. In the middle is the result of converting the original image to a PostScript format with Inkscape, which results in loss of the gradient fill. On the right is the result of importing and rendering the converted image with **grImport**, which (apart from dropping the raster element) works fine, but the result is imperfect because of loss of information during the conversion to PostScript.

This article describes two new packages that are designed to address the limitations of the **grImport** package. The **grConvert** package aims to improve support for converting original images to a format that can be imported into R and the **grImport2** package aims to improve support for importing more complex images into R.

## The grConvert package

The **grConvert** package provides a simple function, called `convertPicture()`, to convert between different graphics formats. The original image format can be PostScript, PDF, or SVG and any of those can be converted to either PostScript or SVG.

Converting an image to PostScript is useful for importing an image with the **grImport** package. For example, if the simple test image from Figure 1 was originally an SVG image, it could be converted to PostScript with the following code.

```
> library(grConvert)

> convertPicture("testimage-simple.svg", "testimage-simple.ps")
```

Using **grConvert** to convert an image to SVG is useful for importing an image with the new **grImport2** package, as we will see in the next section.

The main purpose of the **grConvert** package is to provide more convenience for the task of converting an image from its original format to a format that can be imported into R. However, the **grConvert** package depends on the Spectre library for reading and writing PostScript files (freedesktop.org, 2013b), the Poppler library for reading PDF files (freedesktop.org, 2013a), and the rsvg library for reading and writing SVG files (GNOME Project, 2013), so these must be available for the package to install. This means that, in practice, the package is only available for Linux systems at the time of writing.

One reason for making **grConvert** a separate package was to emphasise the separation between the two steps of preparing an image for import, which is a one-off operation, and importing the

prepared image into R. These steps can be carried out by different people on different platforms if necessary.

## The grImport2 package

The **grImport2** package provides two main functions: `readPicture()` for reading an external image into R and `grid.picture()` for drawing the image.

In contrast to the `readPicture()` function from the **grImport** package, which is used to import PostScript files, the `readPicture()` function in **grImport2** imports SVG files.

The **grImport2** package does not understand all of SVG, but rather a subset defined to be SVG code that is produced by the Cairo graphics library (Packard et al., 2013). This subset of SVG is used because it is much easier to develop code to import a subset rather than all of the possible complexity of the SVG language. Furthermore, this Cairo SVG format is exactly what the `grConvert` package produces when it converts an image to SVG. This means that the typical process for importing an image into R using **grImport2** involves a call to the `convertPicture()` function from the **grConvert** package, to produce a Cairo SVG version (even if the original image format was SVG), then a call to `readPicture()` to import the image, and finally a call to `grid.picture()` to render the imported image.

As an example, the following code starts with the simple test image from Figure 1 in an SVG format, converts it to Cairo SVG, and then imports and renders it with the **grImport2** package (see Figure 5).

```
> library(grImport2)

> convertPicture("testimage-simple.svg", "testimage-simple-cairo.svg")
> image <- readPicture("testimage-simple-cairo.svg")

> grid.picture(image)
```


**Figure 5:** On the left is a simple SVG test image consisting of only paths and text. On the right is the result of converting the image to Cairo SVG using **grConvert** then importing and rendering the image using **grImport2**.

The previous example demonstrates that the **grImport2** package can be used in a very similar manner to the **grImport** package and it should be just as good at handling images consisting of just paths and text. The next example begins to demonstrate where the **grImport2** package performs better than the **grImport** package.

In the following code, we start with an SVG version of the test image from Figure 2, which contains a raster element, convert it to Cairo SVG and import and render it with **grImport2**. The result is shown in Figure 6, which illustrates that **grImport2** is able to import all features of the original image, in this case including the raster element in the test image.

```
> convertPicture("testimage-raster.svg", "testimage-raster-cairo.svg")
> image <- readPicture("testimage-raster-cairo.svg")

> grid.picture(image)
```


**Figure 6:** On the left is an SVG test image that contains a raster element. On the right is the result of converting the image to Cairo SVG using **grConvert** then importing and rendering the image using **grImport2**. The important point is that the raster element is retained during the import process.

The next example shows that **grImport2** can also correctly import the test image that contains semitransparency (see Figure 7).

```
> convertPicture("testimage-semitrans.svg", "testimage-semitrans-cairo.svg")
> image <- readPicture("testimage-semitrans-cairo.svg")

> grid.picture(image)
```


**Figure 7:** On the left is an SVG test image that contains a semitransparent fill. On the right is the result of converting the image to Cairo SVG using **grConvert** then importing and rendering the image using **grImport2**. The important point is that the semitransparent fill is correctly reproduced.

The final test image, which contains a semitransparent gradient (see Figure 4), demonstrates a slightly more advanced use of the `grid.picture()` function. This image can be imported by **grImport2**, but rendering is hampered by the fact that the R graphics engine does not support gradient fills. The **grImport2** package provides support for complex image features like these by integrating with the **gridSVG** package, which does have support for gradient fills (among other things). The following code demonstrates the use of the `ext` argument to the `grid.picture()` function that is required to produce a correct rendering of this test image (see Figure 8). Also notice that the rendering on a standard R graphics device does not include the gradient fill, but the exported SVG image (rendered with **gridSVG**) does include the gradient fill.

```
> convertPicture("testimage-gradient.svg", "testimage-gradient-cairo.svg")
> image <- readPicture("testimage-gradient-cairo.svg")

> grid.picture(image, ext="gridSVG")

> library(gridSVG)

> grid.export("testimage-gradient-grimport2-gridsvg.svg")
```


**Figure 8:** On the left is an SVG test image that contains a semitransparent gradient fill. In the middle is the result of converting the image to Cairo SVG using **grConvert** then importing and rendering the image using **grImport2** on a standard R graphics device; although the gradient fill is imported, the standard graphics device cannot render it. On the right is the result of rendering the imported image as an SVG file using **gridSVG**, which is capable of rendering the gradient fill.

## An image test suite

The original motivation for the new packages arose from a problem posed by Toby Dylan Hocking, which involved importing the entire set of flags of the U.S. states. These images are available from Wikipedia in an SVG format.[1] We will use several of the flags of the U.S. states to demonstrate some further examples of the improved performance of **grImport2** over **grImport**.

First up is the state flag of Oklahoma, just to show that, if an image consists of only paths and text, even if there are lots of complicated paths, both the old **grImport** package and the new **grImport2** package produce a good result (see Figure 9).

```
> convertPicture("Flag_of_Oklahoma.svg", "Flag_of_Oklahoma.ps")
> PostScriptTrace("Flag_of_Oklahoma.ps", "Flag_of_Oklahoma.xml")
> oklahomaPS <- grImport::readPicture("Flag_of_Oklahoma.xml")
```

[1] http://en.wikipedia.org/wiki/State_flags

```
> grImport::grid.picture(oklahomaPS)

> convertPicture("Flag_of_Oklahoma.svg", "Flag_of_Oklahoma_cairo.svg")
> oklahoma <- readPicture("Flag_of_Oklahoma_cairo.svg")

> grid.picture(oklahoma)
```

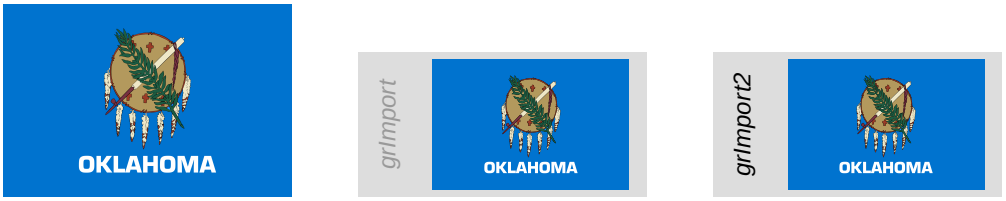


**Figure 9:** At left, the state flag of Oklahoma as an SVG image (from Wikipedia). In the middle, the flag after importing and rendering with **grImport**. At right, the flag after importing and rendering with **grImport2**. Although the image is complex, it only contains paths and text, so both packages reproduce the image faithfully.

The next example, the state flag of Kansas, provides a more dramatic demonstration of the ability to import gradient fills. On the state seal at the centre of the flag, the sky, the sea, and the land all contain linear gradient fills. As with the test image that contained a gradient fill (Figure 4), a correct *rendering* of the Kansas flag requires the involvement of the **gridSVG** package (see Figure 10).

```
> convertPicture("Flag_of_Kansas.svg", "Flag_of_Kansas_cairo.svg")
> kansas <- readPicture("Flag_of_Kansas_cairo.svg")
> grid.picture(kansas, ext="gridSVG")

> grid.export("kansas-grimport2-gridsvg.svg")
```


**Figure 10:** At left, the state flag of Kansas as an SVG image (from Wikipedia). In the middle, the flag after importing and rendering with **grImport2** on a standard R graphics device; the standard device cannot render the gradient fills, which results in black regions on the flag. At right, the flag after importing with **grImport2** and rendering with **gridSVG**.

The state flag of Hawaii contains a feature that we have not previously encountered: *clipping*. The Union Jack at the top-left of the flag is drawn as a collection of simple shapes and lines, with clipping used to limit the visible output to just the top-left corner of the flag. This presents a problem for the **grImport** package because that package ignores clipping information (see Figure 11).

The **grImport2** package imports the clipping information for an image, but, similar to gradient fills, the rendering of clipping information is hampered by the limitations of the R graphics engine, which can only clip to rectangular regions. The `ext` argument to the `grid.picture()` function can be given the value `"clipbbox"` to force all clipping information in an image to be converted to bounding boxes (i.e., rectangular clipping regions) for rendering. In the case of the state flag of Hawaii, this works well because the clipping regions in the original image are already rectangles (see Figure 11).

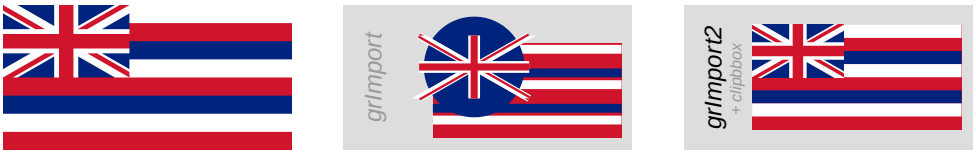


**Figure 11:** At left, the state flag of Hawaii as an SVG image (from Wikipedia). In the middle, the flag after importing and rendering with **grImport**, which does not import clipping information. At right, the flag after importing and rendering with **grImport2**.

The state flag of Ohio presents the more complex clipping case, where the clipping regions are not rectangular (see Figure 12). In this situation, there is no hope of producing the correct result on a standard R graphics device, but, as for gradient fills, we can use `ext="gridSVG"` and export to SVG to obtain the correct rendering (see Figure 12).

```
> convertPicture("Flag_of_Ohio.svg", "Flag_of_Ohio_cairo.svg")
> ohio <- readPicture("Flag_of_Ohio_cairo.svg")
> grid.picture(ohio, ext="gridSVG")

> grid.export("Flag_of_Ohio_gridsvg.svg")
```

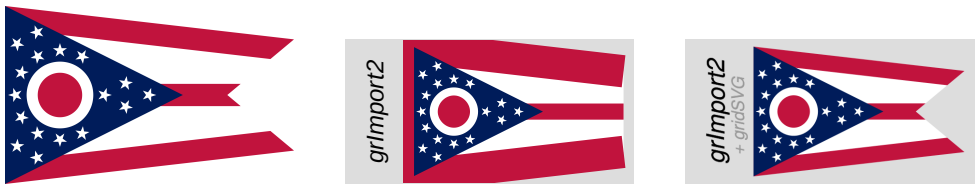


**Figure 12:** At left, the state flag of Ohio as an SVG image (from Wikipedia). In the middle, the flag after importing and rendering with **grImport2** on a standard R graphics device; the non-rectangular clipping information in this flag cannot be rendered correctly on a standard graphics device. At right, the flag after importing with **grImport2** and rendering with **gridSVG**.

In addition to gradient fills and clipping paths, **grImport2** can also import images that make use of masks and pattern fills, although correct rendering of these features will again require the help of the **gridSVG** package.

## A plot example

The main point of importing images into R is not simply to replicate the image by itself, but to incorporate the image as part of an R graphics scene that contains other drawing. This section presents an example of the latter, incorporating an imported image within a **lattice** plot (Sarkar, 2008).

The external image is a balloon icon (an SVG image) by Yuko Iwai, from thenounproject.com collection[2] (see Figure 13). We will use this image within a plot to show the popularity of the top 5 male baby names (in New Zealand in 2000[3]).

```
> convertPicture("noun_project_3663.svg", "balloon.svg")
> balloon <- readPicture("balloon.svg")
> grid.picture(balloon)
```

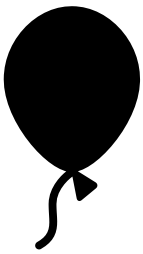

**Figure 13:** A balloon icon by Yuko Iwai, from thenounproject.com collection.

The following code draws a **lattice** plot of the number of male babies with the given name (for the top 5 names), with a custom panel function defined in the `xyplot()` call to add the ballon icon as a plotting symbol. This code demonstrates another function in the **grImport2** package, `grid.symbols()`, which is useful for drawing multiple copies of an imported image at once.

```
> library(lattice)
> library(grid)
```

[2] http://thenounproject.com/term/balloon/3663/

[3] http://www.stats.govt.nz/~/media/Statistics/browse-categories/population/births/tables/babies-names.xls

```
> xyplot(counts ~ names, pch=16,
+        type="h", lty="dotted", lwd=3, col="grey",
+        ylab="", ylim=c(300, 750),
+        xlab="Most Popular Male Baby Names (NZ 2000)",
+        scales=list(x=list(draw=FALSE)),
+        panel=function(x, y, ...) {
+            panel.xyplot(x, y, ...)
+            grid.symbols(balloon, x, y, size=unit(3.5, "cm"))
+        })
```

Figure 14 shows a slightly more complex result than that produced by the code above, with a few extra embellishments added. In particular, it demonstrates that the imported image can be modified by altering the graphical parameter settings, such as colour, when drawing the image (full code for this figure is available from the second author's web site).

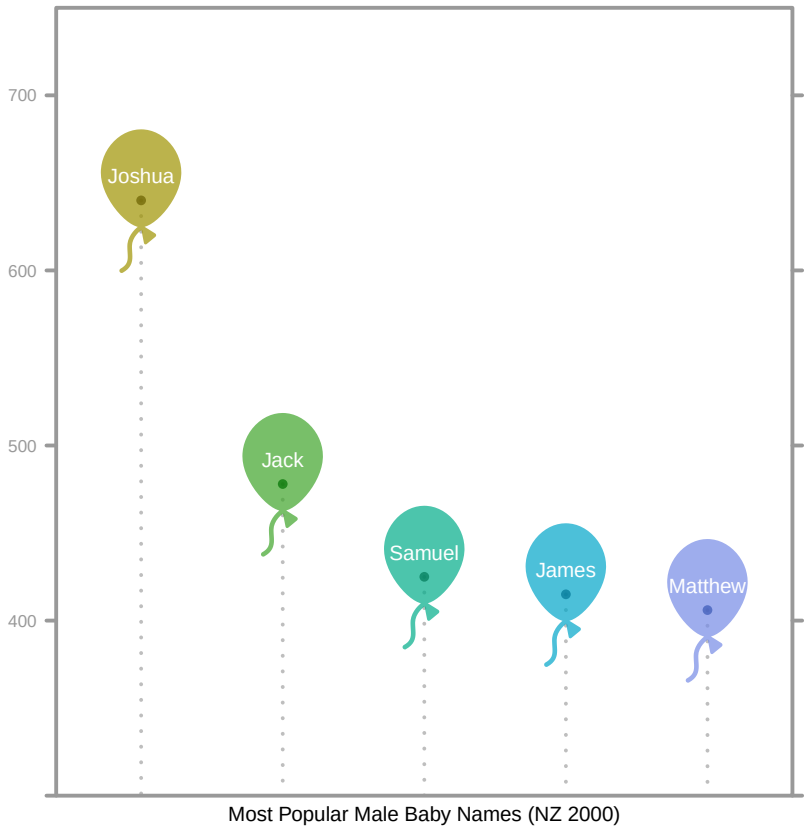


**Figure 14:** The balloon icon from Figure 13 used as a plotting symbol in a **lattice** plot.

## Summary

The following list summarises the main scenarios that arise when we want to import an external image and use it within an R graphic:

- If the original image is in PostScript format and consists only of text and paths (e.g., no raster elements and no clipping), the **grImport** package should work well.
- If the original image is not in PostScript format, but consists only of text and paths, then it should convert well to PostScript for import via **grImport**.
- If the original image is not in PostScript format and/or includes complex content such as raster elements, gradient fills, or clipping paths, it should be possible to convert the image to SVG via **grConvert** and then import with **grImport2**.
- Rendering of some complex images will only be possible in an SVG format with the support of the **gridSVG** package.

## Discussion

The main body of this article has demonstrated some of the limitations of the old **grImport** package and some of the benefits of the new **grImport2** package. This section expands the discussion to include some weaknesses in the new **grImport2** package and some redeeming features of the old **grImport** package.

First of all, despite being able to import a wider range of images, there are still some images that **grImport2** cannot import and render correctly. Figure 15 provides an example from the state flag of Arizona. In this case, the problem is that the yellow "rays" are drawn with a very thick dashed line. The **grImport2** package cannot import this image correctly because the dash pattern cannot be properly represented in R graphics.

```
> convertPicture("Flag_of_Arizona.svg", "Flag_of_Arizona_cairo.svg")
> arizona <- readPicture("Flag_of_Arizona_cairo.svg")
> grid.picture(arizona)
```


**Figure 15:** At left, the state flag of Arizona as an SVG image (from Wikipedia). At right, the flag after importing and rendering with **grImport2** on a standard R graphics device.

Another limitation of the **grImport2** package is that it cannot import text content from the original image as text. This is because the Cairo SVG subset does not support text; it represents all text as paths that draw the individual characters. This means that the R version of an imported image will not contain any text; for example, it will not be possible to search for text values within a PDF file produced by R from an imported image.

This leads us to one of the advantages of the old **grImport** package because that package is capable of importing text from an external image and rendering it as text. Another advantage of **grImport** (compared to **grImport2**) is the relative simplicity of the data structure that is used to represent an imported image. The practical result of this difference is that it is a simple matter to extract a subset from the components an imported image with **grImport**, if we only want to render a small part of an imported image rather than the entire image. This subsetting of an imported image is much harder to do with **grImport2**. The **grImport2** package also currently has no support for rendering an imported image as part of a scene based on the standard **graphics** package.

A final reason for retaining the old **grImport** package is the fact that the conversion of an image from a format other than SVG to Cairo SVG can introduce problems, such as loss of features from the original image. This means that, if the original image is in a PostScript format, the best result may be obtained by importing and rendering with **grImport** rather than converting the image to SVG with **grConvert** and importing and rendering with **grImport2**.

## Links to other information

The **grConvert** and **grImport2** packages are currently available from R-Forge.[4] A more detailed discussion of **grConvert** and **grImport2** packages is provided in Potter and Murrell (2013) and the complete set of state flags and how well they are rendered by **grImport** and **grImport2** is available from the following link:
https://dl.dropboxusercontent.com/u/54315147/import/state-table.html.

## Acknowledgements

Simon Potter's work on the **grConvert** and **grImport2** packages was funded by a Google Summer of Code scholarship. The Summer of Code project was originally proposed by Toby Dylan Hocking.

[4] https://r-forge.r-project.org/projects/grimport/

## Bibliography

T. Bah. *Inkscape: Guide to a Vector Drawing Program*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007. ISBN 9780131357945. [p1]

freedesktop.org. *Poppler*, 2013a. Version 0.22.2. [p2]

freedesktop.org. *libspectre*, 2013b. URL http://libspectre.freedesktop.org/. Version 0.2.7. [p2]

GNOME Project. *librsvg*, 2013. URL http://live.gnome.org/LibRsvg. Version 2.37. [p2]

P. Murrell. Importing vector graphics: The grImport package for R. *Journal of Statistical Software*, 30(4): 1–37, 2009. URL http://www.jstatsoft.org/v30/i04/. [p1]

K. Packard, C. Worth, and B. Esfahbod. *Cairo*, 2013. URL http://cairographics.org/. Version 1.12.14. [p3]

S. Potter and P. Murrell. Improved importing of vector graphics in R. Technical Report 2013-9, Department of Statistics, The University of Auckland, 2013. URL http://stattech.wordpress.fos.auckland.ac.nz/2013-9-improved-importing-of-vector-graphics-in-r/. [p8]

D. Sarkar. *Lattice: Multivariate Data Visualization with R*. Springer-Verlag, New York, 2008. URL http://lmdvr.r-forge.r-project.org. ISBN 978-0-387-75968-5. [p6]

M. Still. *The Definitive Guide to ImageMagick*. Apress, 2005. ISBN 9781590595909. [p1]

*Simon Potter*
*The University of Auckland*
*Auckland*
*New Zealand* simon@sjp.co.nz

*Paul Murrell*
*The University of Auckland*
*Auckland*
*New Zealand* paul@stat.auckland.ac.nz