OK, no stats today, just fidgeting about with graphics devices and type faces. Caveat – the details of graphics formats and typefaces is not my area of real expertise, this blog post is me noting down things that I found useful and others might too. As always, corrections and comments are very welcome.

There are two main types of computer graphic formats:

* *vector* graphics, where the file is basically a set of instructions (start at x, draw a line at 30 degrees angle for y units, draw a circle of radius w centred at z, etc). Like something you generated with Inkscape or Adobe Illustrator.
* *raster* or *bitmap* graphics which are basically an array of numbers telling the computer which how to colour each pixel. Like a digital photograph.

Generally vector formats are far superior if you have the choice but large complex graphics (eg with millions of points, or maps with many fiddly bits) can be slow to render (and large to store). And until relatively recently many users’ web browsers could only show bitmap graphics. [Internet Explorer 8](https://en.wikipedia.org/wiki/Internet_Explorer_8) was the last high-use browser of this sort we had to worry about.

For my blog I work almost purely with Scalable Vector Graphic (SVG) format by choice, and PNG (a relatively efficient and high quality bitmap format) when I have to.

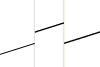
My graphics workflow for this blog is generally:

1. Write the graphic to a Scalable Vector Graphic (SVG)
2. Convert the SVGs to PNGs in batches using [ImageMagick](https://www.imagemagick.org/)

I use the SVGs on the web pages that make up this blog; I need the PNG copies for posting on Twitter, Facebook and LinkedIn which don’t take SVGs, and also for some complex objects that would be too large as SVGs.

**A quick aside on anti-aliasing**

To get in the groove of thinking about graphics I’m going to start with the concept of [anti-aliasing](https://www.webopedia.com/TERM/A/antialiasing.html). Here are three closeups of a straight line in a computer graphic, generated in R by three different graphics devices. From left to right these were generated by CairoSVG(), CairoPNG() and png().



Aliasing (in graphics) refers to the jagged staircase-like bits of an image from approximating an ideal shape with crude collections of pixels. Anti-aliasing is basically the smoothing or blurring process to try to make the problem less obvious.

On a Windows machine (where I spend nearly all my working time), the png() device, no matter high a resolution you specify, will not use anti-aliasing and close ups of the image will reveal jagged bits like that in the right of the three options above. At the opposite end, a graphic that has been saved as SVG won’t have any aliasing at all coming from the file format itself; any jaggedness comes from the final rendering of the information in the file onto the screen (in the end, everything has to become pixels to get on the screen), not from how you generated the file.

The Cairo option in the middle has anti-aliasing taking place when the PNG file itself was generated. Files generated with CairoPNG() look much better than those from png(), but still can’t compete with the vector format even when the PNG is high resolution (eg 720 or more dots per inch).

Here’s the code that generated these examples

library(Cairo)

library(svglite)

library(frs)

library(clipr)

png("0150-antialias-none.png", 8 \* 600, 4 \* 600, res = 600)

plot(1:2, 1:2, type = "l", bty = "n", main = "No anti-alias png() device")

dev.off()

CairoPNG("0150-antialias-cairo.png", 8 \* 600, 4 \* 600, dpi = 600)

plot(1:2, 1:2, type = "l", bty = "n", main = "Anti-aliased CairoPNG device")

dev.off()

CairoSVG("0150-antialias-cairo.svg", 8, 4)

plot(1:2, 1:2, type = "l", bty = "n", main = "CairoSVG device")

dev.off()

Potentially useful whimsical tip: once your awareness is raised about the need for anti-aliasing, you might become painfully aware of those unprofessional looking jagged bits in graphics. When working interactively in RStudio on Windows, all the graphics in your plot pane will look this way. If this hurts you like it hurts me, you can bring up a anti-aliased window with CairoWin() and plots will be rendered nicely on that window, and look much better than the crude default renditions. If you’re using two screens it’s also convenient for placing on your second screen.

**Different flavours of SVG**

Until recently I’ve been generating my SVG files with CairoSVG(). However, I’ve gotten very disatisfied with how text is rendered by that format. It’s hard to put my finger on what is wrong, but it looks blurry, particularly when it’s about 10 point in size which of course is nearly all the time in statistical graphics. Sometimes the problem goes away when you zoom in, but that isn’t really the point.

I eventually solved this problem by moving to the excellent svglite package by Hadley Wickham, Lionel Henry, T Jake Luciani, Matthieu Decorde and Vaudor Lis. svglite creates smaller and faster SVGs and has a much better treatment of text in particular. But svglite alone wasn’t enough for the way I needed typefaces treated – I’ll get to that later.

To go into this further, here are three different SVG files, rendered in this web page as images, created by Cairo::CairoSVG(), grDevices::svg() and svglite::svglite().

The first two approaches are very similar, but the svglite philosophy is quite different. CairoSVG and svg take text and turn it into many tiny shapes to draw; whereas svglite keeps the text as text in the SVG file and leaves it to the end user’s computer to render it. Zoom in close on any of those images above and the text will magnify nicely; but something about the relatively complicated approach of the first two makes them look a little blurred and complex when they’re looked at from a distance. This is the process of turning the text into individual shapes, interacting with the final rendering on the screen in some way that I don’t want to have to understand.

SVG files are just text files (in XML format) so we can look at the actual code (a huge advantage of SVG over other formats). Here’s the full text of the SVG generated by CairoSVG:

xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink" width="432pt" height="432pt" viewBox="0 0 432 432" version="1.1">

overflow="visible" id="glyph0-0">

style="stroke:none;" d="M 2.921875 0 L 2.921875 -26.128906 L 6.378906 -26.128906 L 6.378906 -15.398438 L 19.960938 -15.398438 L 19.960938 -26.128906 L 23.417969 -26.128906 L 23.417969 0 L 19.960938 0 L 19.960938 -12.316406 L 6.378906 -12.316406 L 6.378906 0 Z M 2.921875 0 "/>

overflow="visible" id="glyph0-1">

style="stroke:none;" d="M 15.363281 -6.09375 L 18.679688 -5.683594 C 18.15625 -3.746094 17.1875 -2.242188 15.773438 -1.175781 C 14.359375 -0.105469 12.550781 0.425781 10.355469 0.429688 C 7.582031 0.425781 5.386719 -0.421875 3.769531 -2.128906 C 2.144531 -3.832031 1.335938 -6.226563 1.335938 -9.304688 C 1.335938 -12.484375 2.152344 -14.953125 3.792969 -16.714844 C 5.429688 -18.472656 7.558594 -19.351563 10.175781 -19.355469 C 12.703125 -19.351563 14.769531 -18.492188 16.378906 -16.769531 C 17.980469 -15.046875 18.785156 -12.621094 18.785156 -9.5 C 18.785156 -9.304688 18.777344 -9.019531 18.765625 -8.644531 L 4.652344 -8.644531 C 4.769531 -6.558594 5.355469 -4.96875 6.414063 -3.867188 C 7.472656 -2.761719 8.792969 -2.207031 10.371094 -2.210938 C 11.546875 -2.207031 12.550781 -2.515625 13.382813 -3.136719 C 14.214844 -3.75 14.875 -4.738281 15.363281 -6.09375 Z M 4.828125 -11.28125 L 15.398438 -11.28125 C 15.253906 -12.867188 14.847656 -14.0625 14.1875 -14.863281 C 13.160156 -16.097656 11.835938 -16.714844 10.210938 -16.71875 C 8.738281 -16.714844 7.5 -16.222656 6.496094 -15.238281 C 5.492188 -14.25 4.9375 -12.929688 4.828125 -11.28125 Z M 4.828125 -11.28125 "/>

overflow="visible" id="glyph0-2">

style="stroke:none;" d="M 2.335938 0 L 2.335938 -26.128906 L 5.542969 -26.128906 L 5.542969 0 Z M 2.335938 0 "/>

overflow="visible" id="glyph0-3">

style="stroke:none;" d="M 1.210938 -9.464844 C 1.207031 -12.964844 2.179688 -15.5625 4.132813 -17.25 C 5.757813 -18.652344 7.742188 -19.351563 10.085938 -19.355469 C 12.6875 -19.351563 14.816406 -18.5 16.46875 -16.796875 C 18.117188 -15.089844 18.941406 -12.734375 18.945313 -9.730469 C 18.941406 -7.292969 18.578125 -5.378906 17.847656 -3.984375 C 17.117188 -2.585938 16.054688 -1.5 14.660156 -0.730469 C 13.261719 0.0429688 11.734375 0.425781 10.085938 0.429688 C 7.433594 0.425781 5.292969 -0.417969 3.660156 -2.117188 C 2.023438 -3.8125 1.207031 -6.261719 1.210938 -9.464844 Z M 4.507813 -9.464844 C 4.503906 -7.035156 5.035156 -5.222656 6.09375 -4.019531 C 7.148438 -2.8125 8.476563 -2.207031 10.085938 -2.210938 C 11.675781 -2.207031 13.003906 -2.8125 14.0625 -4.027344 C 15.117188 -5.234375 15.644531 -7.082031 15.648438 -9.570313 C 15.644531 -11.90625 15.113281 -13.679688 14.050781 -14.890625 C 12.988281 -16.09375 11.664063 -16.699219 10.085938 -16.699219 C 8.476563 -16.699219 7.148438 -16.097656 6.09375 -14.898438 C 5.035156 -13.695313 4.503906 -11.882813 4.507813 -9.464844 Z M 4.507813 -9.464844 "/>

overflow="visible" id="glyph0-4">

style="stroke:none;" d=""/>

overflow="visible" id="glyph0-5">

style="stroke:none;" d="M 5.898438 0 L 0.105469 -18.925781 L 3.421875 -18.925781 L 6.433594 -8.003906 L 7.554688 -3.9375 C 7.601563 -4.140625 7.929688 -5.441406 8.535156 -7.84375 L 11.546875 -18.925781 L 14.847656 -18.925781 L 17.679688 -7.949219 L 18.625 -4.332031 L 19.710938 -7.984375 L 22.953125 -18.925781 L 26.074219 -18.925781 L 20.15625 0 L 16.824219 0 L 13.8125 -11.335938 L 13.082031 -14.5625 L 9.25 0 Z M 5.898438 0 "/>

overflow="visible" id="glyph0-6">

style="stroke:none;" d="M 2.371094 0 L 2.371094 -18.925781 L 5.257813 -18.925781 L 5.257813 -16.058594 C 5.992188 -17.398438 6.671875 -18.28125 7.296875 -18.710938 C 7.917969 -19.136719 8.605469 -19.351563 9.355469 -19.355469 C 10.433594 -19.351563 11.53125 -19.007813 12.652344 -18.320313 L 11.546875 -15.34375 C 10.761719 -15.804688 9.976563 -16.035156 9.195313 -16.039063 C 8.492188 -16.035156 7.863281 -15.824219 7.304688 -15.40625 C 6.746094 -14.980469 6.347656 -14.398438 6.113281 -13.652344 C 5.753906 -12.511719 5.578125 -11.261719 5.578125 -9.910156 L 5.578125 0 Z M 2.371094 0 "/>

overflow="visible" id="glyph0-7">

style="stroke:none;" d="M 14.6875 0 L 14.6875 -2.386719 C 13.484375 -0.507813 11.71875 0.425781 9.390625 0.429688 C 7.878906 0.425781 6.492188 0.015625 5.230469 -0.816406 C 3.960938 -1.644531 2.980469 -2.808594 2.289063 -4.300781 C 1.589844 -5.792969 1.242188 -7.507813 1.246094 -9.445313 C 1.242188 -11.335938 1.558594 -13.046875 2.191406 -14.585938 C 2.820313 -16.121094 3.761719 -17.300781 5.023438 -18.125 C 6.28125 -18.941406 7.691406 -19.351563 9.25 -19.355469 C 10.386719 -19.351563 11.402344 -19.113281 12.296875 -18.632813 C 13.183594 -18.148438 13.910156 -17.519531 14.472656 -16.753906 L 14.472656 -26.128906 L 17.660156 -26.128906 L 17.660156 0 Z M 4.542969 -9.445313 C 4.539063 -7.019531 5.050781 -5.207031 6.074219 -4.007813 C 7.09375 -2.808594 8.300781 -2.207031 9.695313 -2.210938 C 11.09375 -2.207031 12.285156 -2.78125 13.265625 -3.929688 C 14.246094 -5.074219 14.734375 -6.824219 14.738281 -9.179688 C 14.734375 -11.765625 14.238281 -13.664063 13.242188 -14.878906 C 12.242188 -16.089844 11.011719 -16.699219 9.554688 -16.699219 C 8.125 -16.699219 6.933594 -16.117188 5.976563 -14.953125 C 5.019531 -13.789063 4.539063 -11.953125 4.542969 -9.445313 Z M 4.542969 -9.445313 "/>

id="clip1">

d="M 177.121094 177.121094 L 342.28125 177.121094 L 342.28125 212.679688 L 177.121094 212.679688 Z M 177.121094 177.121094 "/>

id="surface16">

clip-path="url(#clip1)" clip-rule="nonzero">

style="fill:rgb(0%,0%,0%);fill-opacity:1;">

xlink:href="#glyph0-0" x="168.929688" y="207.25"/>

xlink:href="#glyph0-1" x="195.288818" y="207.25"/>

xlink:href="#glyph0-2" x="215.588379" y="207.25"/>

xlink:href="#glyph0-2" x="223.69751" y="207.25"/>

xlink:href="#glyph0-3" x="231.806641" y="207.25"/>

xlink:href="#glyph0-4" x="252.106201" y="207.25"/>

xlink:href="#glyph0-5" x="262.24707" y="207.25"/>

xlink:href="#glyph0-3" x="288.606201" y="207.25"/>

xlink:href="#glyph0-6" x="308.905762" y="207.25"/>

xlink:href="#glyph0-2" x="321.060547" y="207.25"/>

xlink:href="#glyph0-7" x="329.169678" y="207.25"/>

… and here it is for the SVG generated by svglite:

xmlns='http://www.w3.org/2000/svg' xmlns:xlink='http://www.w3.org/1999/xlink' viewBox='0 0 720.00 576.00'>

type='text/css'>

width='100%' height='100%' style='stroke: none; fill: #FFFFFF;'/>

id='cpMTc3LjEyfDYyOS4yOHwzNTUuNjh8MTc3LjEy'>

x='177.12' y='177.12' width='452.16' height='178.56' />

clip-path='url(#cpMTc3LjEyfDYyOS4yOHwzNTUuNjh8MTc3LjEy)'> x='318.96' y='278.15' style='font-size: 60.00px; font-family: Indie Flower;' textLength='168.48px' lengthAdjust='spacingAndGlyphs'>Hello world

The svglite version is *much* shorter because instead of describing exactly how each character looks, it just says “write ‘Hello world’ in the middle of the graphic, at 60 points in size, using the Indie Flower font family”.

That last point – “using the Indie Flower font family” is important too. It means that this font family needs to be available at the point the SVG file is rendered on-screen ie the end user’s machine. Whereas in CairoSVG and svg, no font family is recorded in the SVG file itself, the font family is taken into account at the generation of the SVG and has to be available on the developer’s machine.

As it happens, the “Indie Flower” type face isn’t installed on my computer and probably isn’t on yours either. When R encountered an instruction to use a typeface it didn’t know, it fell back on Arial and that’s how the two images on the left were generated. When a web browser encounters such an instruction, it falls back (at least if it’s Chrome or Edge) on Times New Roman or equivalent. This is why the image on the right looks different to the other two – it’s a question of a web browsers fall-back type face, compared to R’s.

Here’s the code that produced those three images and got the code from the SVG files to put in this blog post:

CairoSVG("0150-cairo-svg.svg")

par(family = "Indie Flower", cex = 3)

plot.new()

text(0.5, 0.5, "Hello world")

dev.off()

svg("0150-svg.svg")

par(family = "Indie Flower", cex = 3)

plot.new()

text(0.5, 0.5, "Hello world")

dev.off()

svglite("0150-svglite.svg")

par(family = "Indie Flower", cex = 3)

plot.new()

text(0.5, 0.5, "Hello world")

dev.off()

print\_to\_screen <- function(fn){

txt <- readChar(fn, file.info(fn)$size)

write\_clip(txt)

}

print\_to\_screen("..http://freerangestats.info/img/0150-cairo-svg.svg")

print\_to\_screen("..http://freerangestats.info/img/0150-svglite.svg")

**Solving the web font problem for svglite**

So, I much preferred the smaller size, faster creation and download, and better look for text from svglite – but I had a new problem, of how to get custom fonts rendering reliably. I use web fonts from Google for my web page to get my HTML/CSS consistent with the graphic images, so in practice I need a way to render Google web fonts in SVGs, regardless of whether the user has them on their end computer or not.

svglite has a user\_fonts argument that is meant to embed fonts that are on the developer’s machine inside the SVG if desired. So I forced myself to look a bit into how the SVG format works, and after wading through a number of out of date bits of info on the web eventually realised that I just needed the line @import url('https://fonts.googleapis.com/css?family=Indie Flower:400,400i,700,700i'); added (for the Indie Flower case). So the svglite file above just needs to become as follows:

xmlns='http://www.w3.org/2000/svg' xmlns:xlink='http://www.w3.org/1999/xlink' viewBox='0 0 720.00 576.00'>

type='text/css'>

width='100%' height='100%' style='stroke: none; fill: #FFFFFF;'/>

id='cpMTc3LjEyfDYyOS4yOHwzNTUuNjh8MTc3LjEy'>

x='177.12' y='177.12' width='452.16' height='178.56' />

clip-path='url(#cpMTc3LjEyfDYyOS4yOHwzNTUuNjh8MTc3LjEy)'> x='318.96' y='278.15' style='font-size: 60.00px; font-family: Indie Flower;' textLength='168.48px' lengthAdjust='spacingAndGlyphs'>Hello world

This renders nicely, with the correct typeface and all:

… albeit with one last change necessary to how I organise my blog. I used to include my SVG files in the blog using the tag, such as:

src='http://freerangestats.info/img/0150-svglite-with-fonts.svg' width='50%'>

An SVG file that is called into a webpage by is not allowed to get information from external sites. It has to instead be included with an

This has the disadvantage that a user can no longer right-click on the image and save it. They can still view the “frame source”, copy it into a text editor and save it from there, but that’s going to put off a lot of users who aren’t used to thinking of an image as being fully represented by a bunch of code that can be pasted into Notepad! I imagine I’ll think of something to get around this.