Literature Review
—Graphical Tools—

Jimmy Oh

Department of Statistics
University of Auckland

Abstract

This Literature Review examines free Graphical Tools that produce interactive output ideal for web-based viewing, ranging from *Tableau Public* to *Google Chart Tools* and *D3.js*. To compare between the complex tools (defined as *Low-level Languages*) like *Processing* and *Raphaël*, the same graphic is implemented under each *Low-level Language*, which serves as both a method of directly comparing between the tools, and also as an example of how to use the tool. A brief primer on some graphical terms are also provided, such as the difference between *raster* and *vector* graphics, or how the *SVG* image file format works.

Contents

1 Introduction

This Literature Review examines **free** Graphical Tools that produce **interactive output** ideal for **web-based viewing**. We look at three different **Classifications** of tools, *GUI* Tools, *High-level* languages and *Low-level*[1] languages.

Section 5 covers Other Relevant Topics, including how `R` might be used to aid in preparing and exporting data for use with one of the tools, and explanations of some graphical terms such as the difference between *raster* and *vector* graphics, or how the *SVG* image file format works.

Post-Script

This Literature Review is conducted as part of a thesis examining *Presentation Methods for Open Data* with the broad aim of making the data more accessible and usable for a wider audience.

1.1 Free

In keeping with the spirit of Open Data, we are interested in free tools, so that a casual user interested in exploring some Open Data can use such a tool at no cost.

Ideally, not only is the tool free, it is also *Open Source*. The distinction is not very important for the casual user, but for us as we look to develop new tools, existing tools that are *Open Source* provide opportunities to learn and/or extend functionality.

Additionally, if the output is also in an *Open Format* (such as SVG), this opens up opportunities to utilise multiple tools that share this *Open Format*, in conjunction for greater effect.

1.2 Interactive Web-based Output

We focus on interactive output rather than more traditional static images as these have more potential for advancements. Being web-based makes the output substantially easier to share, and also passes off much of the rendering burden to web browser developers, allowing us to focus more on visualisation methods.

We use the following terms in our overview of graphical tools:

Simple Interactivity This includes cases where the user can mouseover a bar on a bar-graph, or a slice of a piechart, and obtain more information about that bar or slice. It is technically interactive and not static, but typically adds very little value.

End-user Data Exploration This is where the final output can be used by the *end-user*[2] to *explore* the data in some way. At the very basic level, this may involve merely resorting the data. At a more advanced level, this can effectively turn the output into a simple *GUI* tool of its own, where the *end-user* can change the variables being examined, the chart type, and so on.

[1] By more conventional definition, these should be *Very High* and *High*, but for better distinguishing power we use more extreme words.

[2] **End-User:** The user of the final output. Distinguished from the user of one of the graphical tools we cover, to produce the output.

1.3 Tool Classification

We classify the tools broadly into three categories: *GUI* (Graphical user interface), *High-Level* language and *Low-Level* language.

Note that the definition of *High-* and *Low-* level programming languages is largely relative. A quick google search suggests a conventional definition for a *Low-level* language might be something like an assembly language. For our purposes, approaching from the perspective of a casual user, we will use the following definitions:

GUI A typical example would be a point and click interface, requiring absolutely no writing of code to accomplish the desired task. Without doubt, to a casual user the *GUI* is the most accessible and desired tool.

High-level A language that can achieve the desired result with a few lines of very simple code, usually involving specific *High-level* functions that perform several tasks. An example might be doing a simple linear regression model in R. The task can be accomplished with a single call to `lm` to fit the model. We may then need a few additional calls to check the assumptions are met and print the output. Excel formulae may also fall under this definition, though in that case, the formulae can be used via a GUI rather than direct 'coding'. Under our rather restrictive definition of *High-level* language, it is feasible that a casual user might be willing to learn and use a *High-level* language tool, though they would be resistant to such a notion.

Low-level A language that requires several lines of code to achieve the desired result, including the use of *primitive* or *Low-level* functions that only accomplish a very narrow, simple and specific task (such as drawing a single rectangle). Under this definition, a language like *JavaScript* would be considered *Low-level*, as it would be difficult to learn and use for the casual user. While out of reach for the casual user, these *Low-level* language tools will be valuable in the creation of new tools more appropriate for the casual user.

We are most interested in the *Low-level* language tools, as these are likely to be used to develop new tools and methods for presenting Open Data. However, a brief overview of existing *GUI* tools and *High-level* language tools are necessary to understand what already exists, what they do well, and what gaps exists where a new tool might be valuable.

2 GUI Tools

2.1 Many Eyes

`http://www-958.ibm.com/software/data/cognos/manyeyes/`

Key Points

Learning Difficulty Very Easy.

Has a library of standard plots Yes.

Output Java or Flash object.

Interactivity Wide variety depending on plot type chosen.

Legal Free to use, but any output using Many Eyes will be published online via the website, making the visual and underlying data public. Data has a size limit of 5 MB.

Main Features

- Tool is entirely web-based requiring no download.

- But output is Java and Flash based, meaning the end-user must download and install the platform.
- Has good community support, such as rating of visualisations and datasets, and commenting by users.
- An 'experiment' by IBM, future of the tool is unknown.
- Purely a GUI with no command-line support (Terms of Use expressly forbid automation: "You agree not to send automated queries of any sort to the Services").
- All data uploaded, and any visualisation created, becomes publicly available on the website. However, data and visuals can be deleted at a later date (This method should not be considered a 'secure' way of ensuring limited publicity).
- Very strict about accepted data format. A lot of effort is often required to get the data in just the right format for the desired plot.

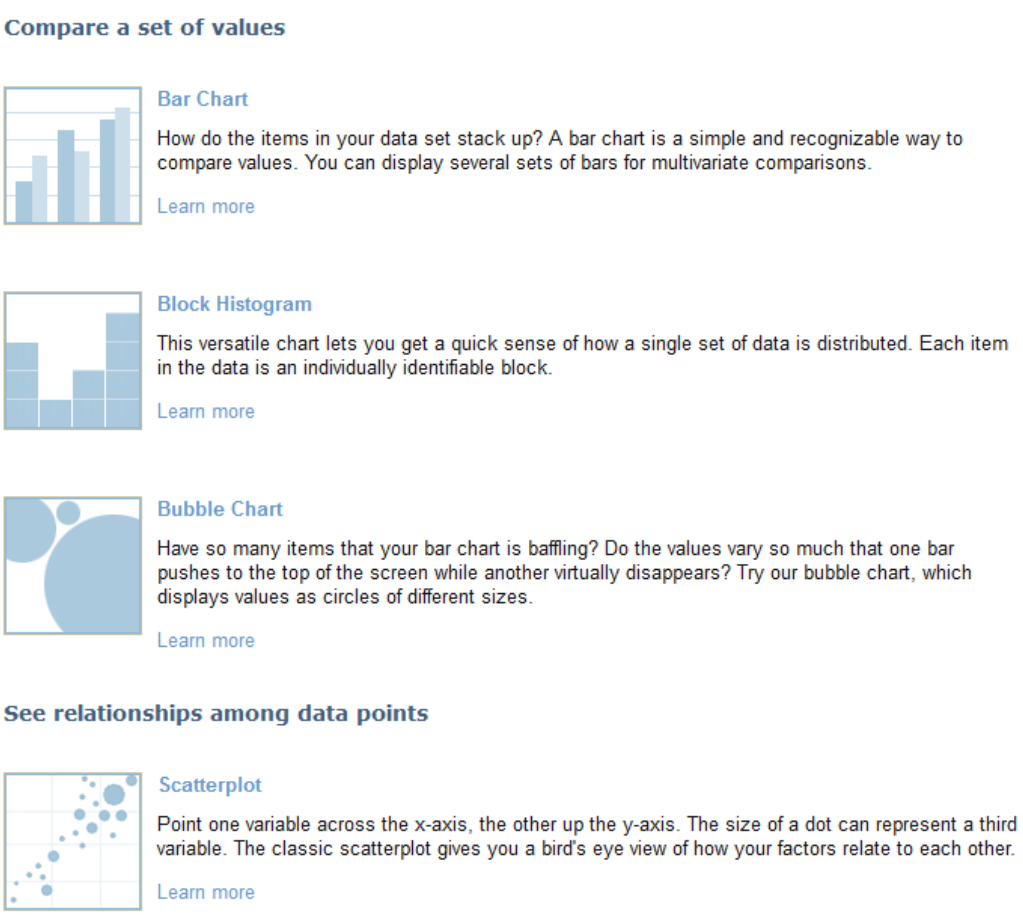


Figure 1: The Many Eyes 'interface'. This is a screenshot of one of the webpages used to create a new plot. Many Eyes is entirely web-based and requires no separate client software.

Comments

The primary advantage of Many Eyes is that it is a Visualisation GUI Tool that is entirely online, requiring no download or install beyond the software platforms it requires (Java and Flash). Any data uploaded and any visualisation created is automatically published publicly online, making it extremely easy to share (conversely it also means that it cannot be made private. Less of an issue when dealing with Open Data, but has problems for Closed Data).

The number of datasets being uploaded and visualisations being created would indicate Many Eyes has an active user base. Many Eyes first began in 2007 and is still available, though the last update appears to have been in March 2011. Though it is being developed by a well-known major corporation (IBM), its current classification as an 'experiment' does cast a slight shadow of doubt on its future availability.

For example output, visit the public gallery: `http://www-958.ibm.com/software/data/cognos/manyeyes/visualizations`

2.2 Tableau Public

`http://www.tableausoftware.com/products/public`

Key Points

Learning Difficulty Very Easy.

Has a library of standard plots Yes.

Output Proprietary Format.

Interactivity Extensive potential, including end-user exploration of the data via interaction with the output, but still limited to what the tool provides.

Legal Free to use, but any output using Tableau Public must be published online via their website, making the visual and underlying data public. There is also a data size limitation of 100,000 rows. Paid versions of Tableau are available which relaxes these constraints.

Main Features

- Requires a download and install of a client software, that requires an active internet connection to work.
- Quite powerful relative to ease of use.
- Attempts to automatically detect data type, such as whether it is a 'Dimension' (Qualitative) or a 'Measure' (Quantitative), or identification of names, such as the States of the USA (which greatly simplifies the creation of map-based graphics).
- Suggests appropriate chart types based on data variables selected.
- Powerful interactivity features, including a 'dashboard' feature that can enable data exploration at by the end-user. That is, once a graphic output is made and published, any future user of that graphic may have the capability to rearrange categories, change graph type, etc.
- Purely a GUI with no command-line support.
- All data uploaded, and any visualisation created, becomes publicly available on the website.
- The free Public version has various restrictions.

Comments

Starting out in 2003 as an output of a PhD project called Polaris, "an interface for the exploration of multidimensional databases that extends the Pivot Table interface to directly generate a rich, expressive set of graphical displays" (Stolte et al. 2008), it became commercialised as Tableau later that year. In 2010, the free version, Tableau Public was released.

Tableau Public requires a client to be downloaded and installed, and also requires an internet connection to function. Rather than accepting a specific data structure for a specific plot type, Tableau accepts an entire database, and allows the user to explore the variables in the data via a variety of potential plots.

For example output, visit the public gallery: `http://www.tableausoftware.com/public/gallery`

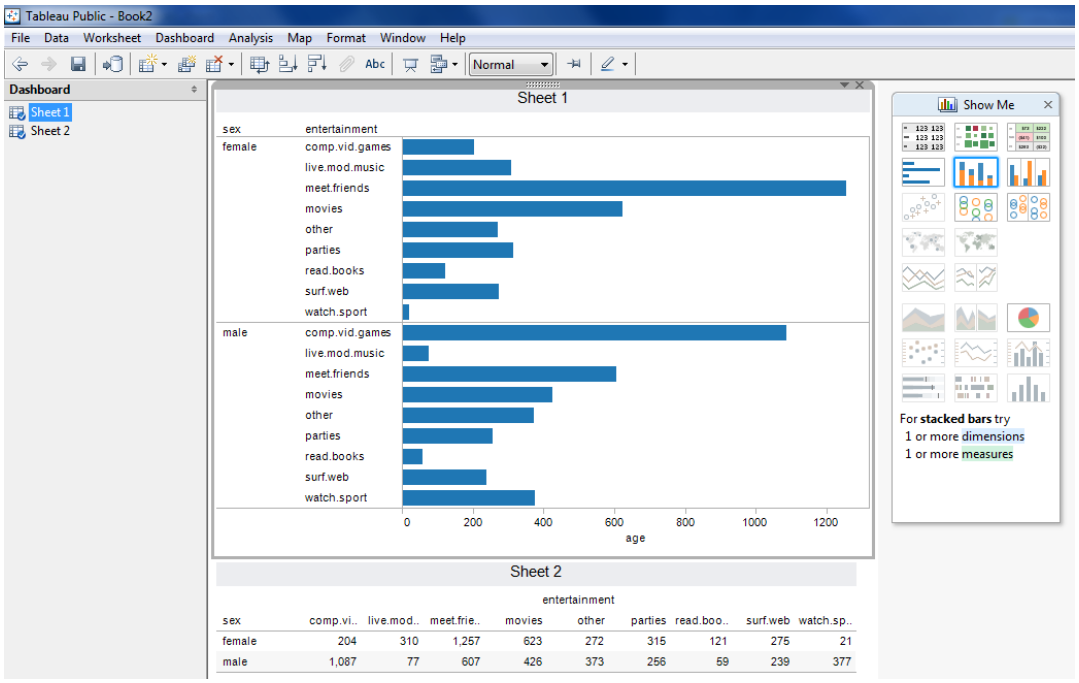


Figure 2: The Tableau Public interface. In this example, we are looking at a 'dashboard', allowing us to place multiple graphics on the same page. On the right, Tableau Public suggests some appropriate graph types for the selected data subset.

2.3 Other GUI Tools

We are concerned with *GUI Tools* that produce **Interactive Web-based Output** in this Literature Review, but we would like to note that there are many offline *GUI Tools* that can be used for visualising and exploring data interactively. For instance *Improvise* (`http://www.cs.ou.edu/~weaver/improvise/`) or *The InfoVis Toolkit* (`http://ivtk.sourceforge.net/`), which may better serve the needs of the reader if they do not require easy facilities for publishing the output online.

3 High-level Languages

3.1 Google Chart Tools

`https://developers.google.com/chart/`

Key Points

Learning Difficulty Easy (Very Easy using something like the *Google Vis* R Package, which greatly reduces the work).

Has a library of standard plots Yes.

Output SVG, with support for VML for compatibility with older versions of Internet Explorer. Some of the older graphs (e.g. the Annotated Time Line or Hans Rosling's Gapminder Motion Chart) use Flash.

Interactivity Varies by plot type, most only have simple interactivity, though some graphs offer more interesting interactivity, such as the Annotated Time Line (similar to the one used for the Google Finance graphs) or Hans Rosling's Gapminder Motion Chart (which was acquired by Google in 2007).

Legal Free to use, some fine print in the Terms of Service.

Main Features

- Very easy to use, especially when using the *Google Vis* R Package (or other similar packages).
- A significant variety of plot types to choose from, including community contributed plots.
- In active development by Google.

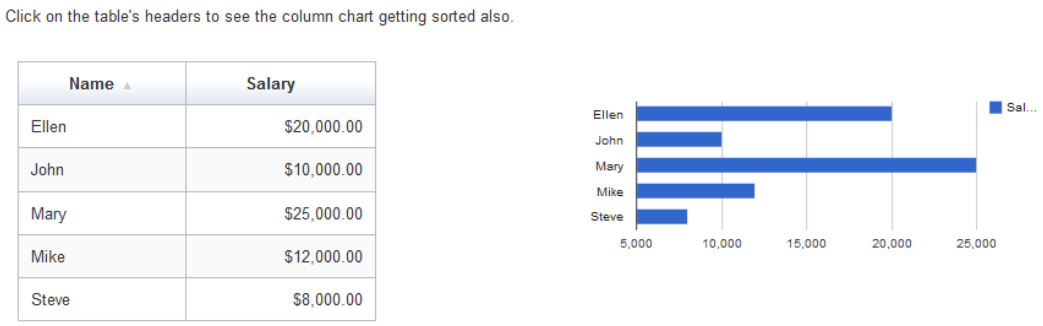


Figure 3: A screenshot from `https://developers.google.com/chart/interactive/docs/examples` demonstrating interactivity between a *Table* and a *BarChart*, sorting by Name by clicking on the appropriate column in the *Table*.

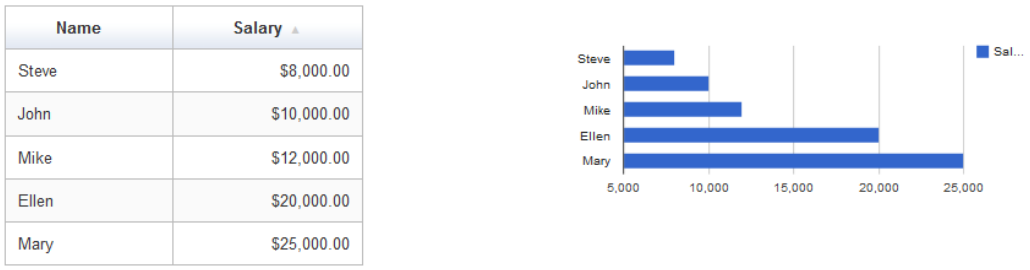


Figure 4: A continuation of Figure 3, where the *Table* has been resorted by Salary, with the *BarChart* following suit.

Comments

For users already familiar with R, *Google Chart Tools* is the easiest to use of the two *High-level* language tools covered, because of the fantastic *Google Vis* R Package. The package allows everything to be done within R, removing the need to learn and fiddle with HTML and JavaScript code, generally reducing the coding required and enabling easy access to the many data processing and manipulation tools of R. This makes *Google Chart Tools* as easy to use as a *GUI Tool* if not easier due to the ease of scripting.

One possible problem with *Google Chart Tools* is that it is still in active development, and may not be suited towards use that requires long-term reliability and stability.

A bargraph in Google Chart Tools

Examples are available on the official website for directly writing the HTML and JavaScript code. We present here the method to do this in R, which is considerably shorter than writing directly.

```
library("googleVis")
## Generate some data
dat = data.frame(val = runif(10, 0, 100), label = LETTERS[1:10])

## Make the plot
googBar = gvisBarChart(data = dat, xvar = "label", yvar = "val",
                       options = list(width = 1280, height = 720))

## Save the plot as an html file
## Package takes care of all the required html code including
##   exporting of the dataset into an appropriate JSON format.
print(googBar, file = "googBar.html")
```

There is an extensive list of additional arguments that can be passed to `options` to fine tune the chart to your liking.

3.2 Highcharts

`http://www.highcharts.com/`

Key Points

Learning Difficulty Easy.

Has a library of standard plots Yes.

Output SVG, with support for VML for compatibility with older versions of Internet Explorer.

Interactivity Varies by plot type, most only have simple interactivity, but API allows for more interesting interactivity.

Legal Open Source (CC By-NC). Has provisions for commercial use (can buy a commercial licence).

Main Features

- Download comes with an impressive set of examples in the form of complete `.html` files, making it very easy to tweak for actual use.
- Export capability built-in to the graphs, allowing for export of the charts to several formats, including static PNG and JPG images for offline use.
- Commercial side likely to provide incentives for the active development and continuing support of the tool.

Comments

Though harder to use than *Google Chart Tools* (even if you're not familiar with R), *Highcharts* has a bit more flexibility and is technically Open Source (as long as you have no commercial interest).

A potentially large barrier to use is getting the data into the right format. For instance, the official *How To Use* (`http://www.highcharts.com/documentation/how-to-use`) goes into some detail on loading in data, including writing your own parser for CSV files. We would instead recommend using R to prepare and export the data to JSON (see Section 5.1.1), which makes the process substantially less work.

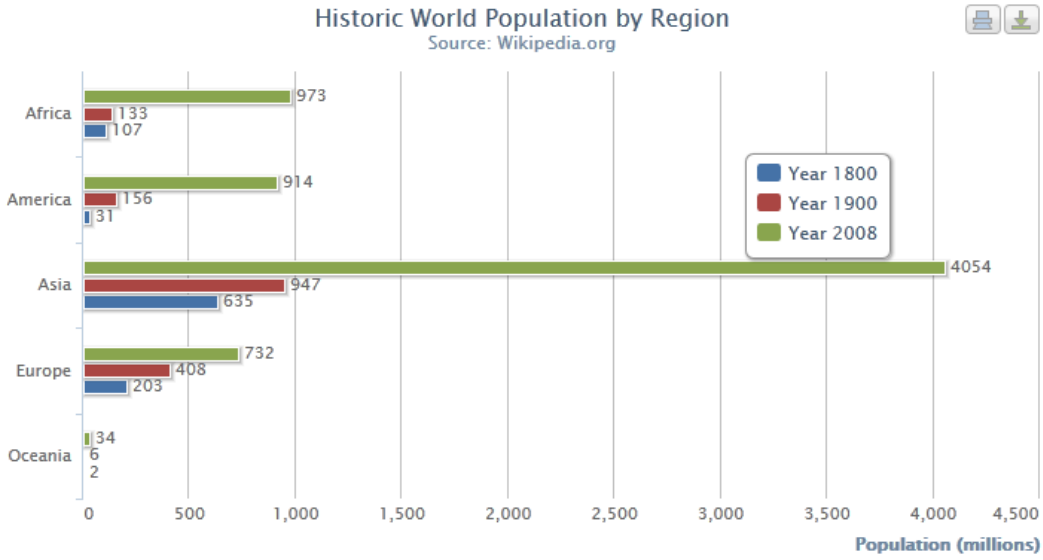


Figure 5: A screenshot from `http://www.highcharts.com/demo/bar-basic` demonstrating a basic bar chart.

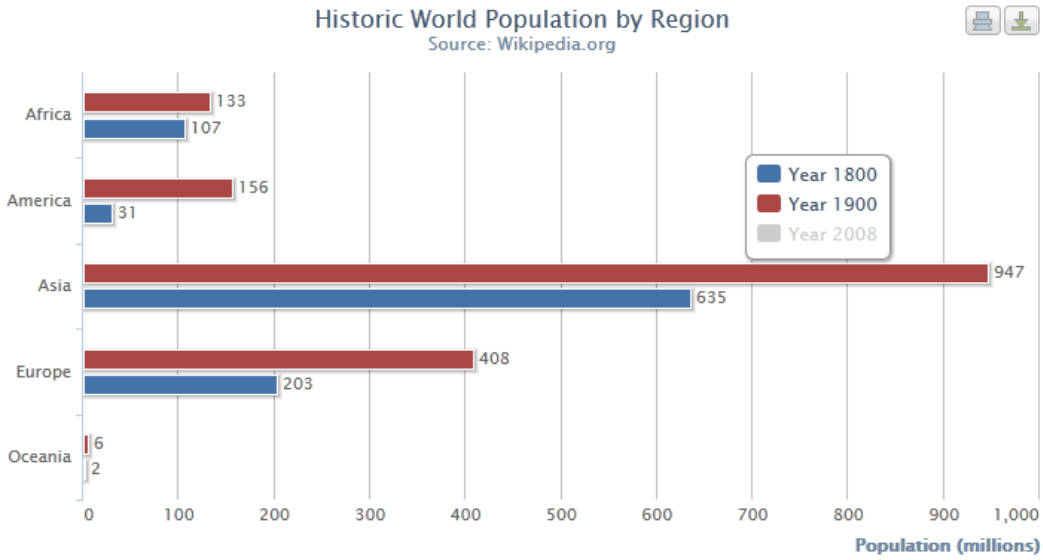


Figure 6: A continuation of Figure 5, where clicking on **Year 2008** in the legend has filtered this data and the bar chart (including axis) has updated accordingly.

A bargraph in Highcharts

Refer to the examples that come with the download.

For reference and comparison purposes, the `bar-basic` example JavaScript code (excluding HTML code, data and blank spaces) is 60 lines and 738 characters.

4 Low-level Languages

As Low-level Languages can be used to create custom visualisations from scratch, we will use each language to create roughly the same visualisation. We will call this the **Test Bargraph**.

We require the following basic features:

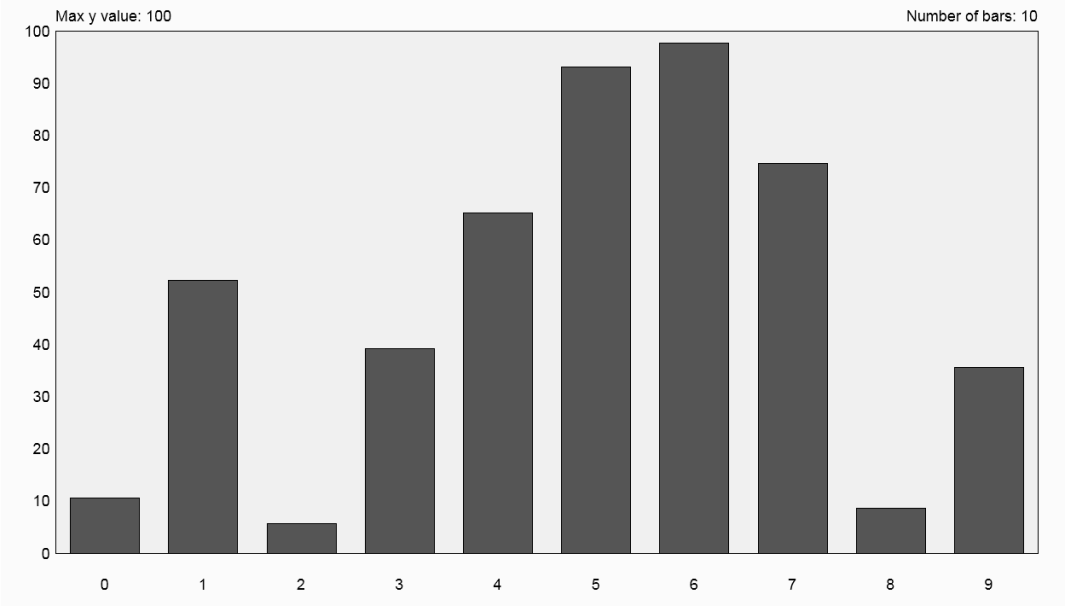


Figure 7: The **Test Bargraph** we will use to explore each Low-level Language. We will attempt to recreate this same bargraph as closely as possible with every Low-level Language we cover.

Size Any reasonable arbitrary size of the Canvas or SVG image can be specified, including aspect ratio.

Number of Bars Can handle any positive integer.

Max Value We will be generating a random number between 0 and an arbitrary maximum value. The graph, including the y axis label, should be able to handle this.

Margins The width of the margins around the central plot can be specified. This is also where the axis labels will go, so there is a minimum constraint. For simplicity all our examples will simply use margin sizes that are proportionate to the total Size, but it should be possible to specify a fixed number (say to perfectly fit the axis labels).

To test how difficult it is to add interactivity, we will also add in a feature to enable the user to alter the heights of the bars. This will involve the following elements:

Ghost Bar A 'ghost bar' tracks the position of the mouse and displays the potential height of the updated bar.

Bar Updates Upon a mouse click, the height of the bar is updated to the new value based on the position of the mouse.

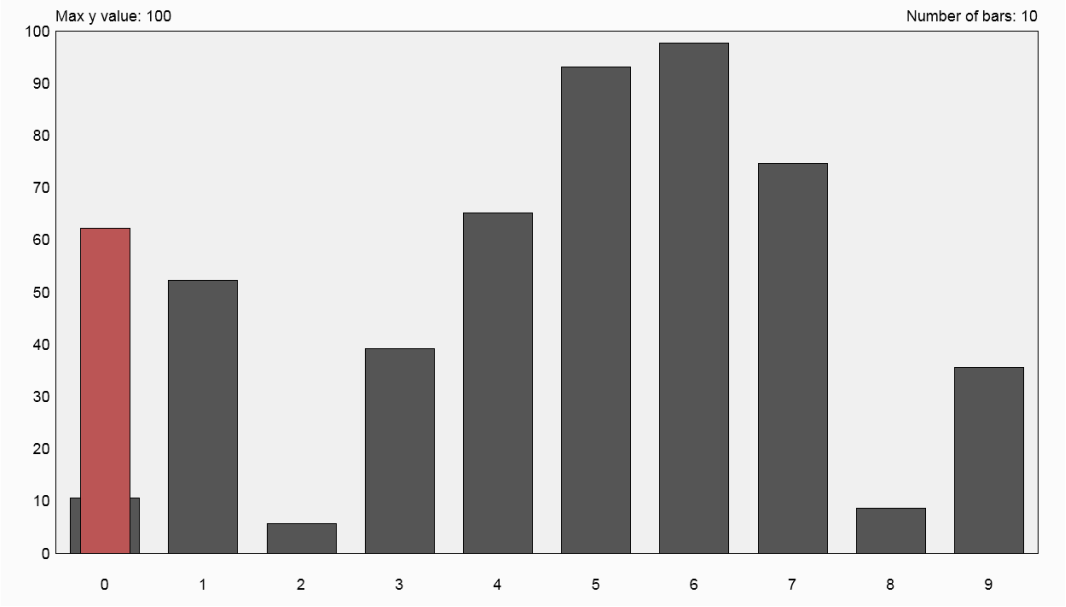


Figure 8: The **Test Bargraph** showing a 'ghost bar' which displays the potential height of the updated bar if the user clicks. Note that this screenshot does not capture the mouse pointer.

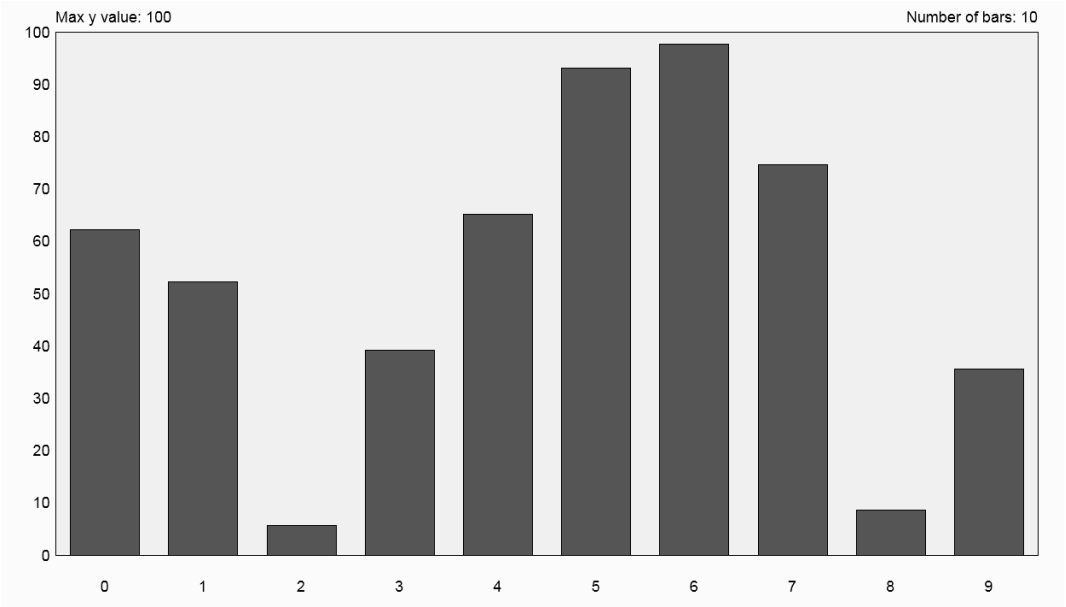


Figure 9: The **Test Bargraph** with the bar height updated.

4.1 Processing.js

`http://processing.org/`
`http://processingjs.org/`

Key Points

Learning Difficulty Hard (Moderate if experienced in Java, JavaScript or Java-like languages).

Has a library of standard plots No.

Output HTML5 Canvas.

Interactivity Broad potential, limited only by what the user can code. No native object support.

Legal Processing: Open Source (GPL, version unknown).
Processing.js: Open Source via Seneca's *Centre for Development of Open Technology*.

Main Features

- Despite being based on Java, can generally be considered a language on its own, meaning almost everything needed to learn and write Processing code can be found in one place (the Processing API Reference).
- Extending functionality to include interactivity easy.
- Have to build graphics from scratch using low-level functions.
- Can be used in two ways, either writing *Processing* code directly, or using *Processing.js* as a JavaScript library.

Comments

Processing is a *Low-level* language for visualisations in general, though often of an interactive nature. Because of this, even simple plots like a bar plot requires a fair level of coding to build from scratch using low-level functions.

No generalised plotting library was found, but it is quite possible to write a library of low-level plotting aid functions that would make creation of new graphs almost as easy as R's default graphics system.

The *Processing* language is based on Java, and *Processing.js* is a JavaScript implementation that essentially compiles *Processing* code into JavaScript to run. Due to the way *Processing.js* works, a user has two choices when coding. One can either write *Processing* code, or one can choose to use *Processing.js* as a JavaScript library, in effect gaining access to the *Processing* API while writing JavaScript.

Choosing to write in *Processing* code leaves open the possibility of compiling this using the original *Processing* software, which may lead to performance gains. However, *Processing.js* is generally easier to use than Processing, as the required file is small (403 KB) and requires no installation beyond a modern web browser, which should already be installed on any modern operating system.

The use of the HTML5 Canvas element as the output attracts with it certain properties (see Section 5.4), perhaps the biggest being that there is no native support for objects. This can make certain kinds of interactivity more difficult than it would be if objects were recognised.

The redraw framerate can be easily specified using the `frameRate()` function, and the live framerate can be checked via the `frameRate` variable (e.g. by inserting `println(frameRate);` somewhere in the `draw()` function).

A bargraph in Processing

A working example of the **Test Bargraph** in *Processing* can be found at `http://www.stat.auckland.ac.nz/~joh024/Research/Processingjs/test_bargraph.html` (HTML) and `http://www.stat.auckland.ac.nz/~joh024/Research/Processingjs/test_bargraph_fullcomments.pde` (Code).

Getting the desired interactivity to work requires matching the coordinates to the 'objects'. We must first check if the mouse is inside the graph region (and not in the margins), then match the `x` coordinate of the mouse to the appropriate bar. Once this is accomplished, it is trivial to then draw the *Ghost Bar* based on the mouse position, and also to have the matching bar update on mouse click.

For reference and comparison purposes, the *Processing* code (excluding data and blank spaces) is 84 lines and 2071 characters. Due to the excessive features (low-level plotting aid functions for generality, demonstrative interactivity), this is not directly comparable to the *High-level* languages, but can be loosely compared to other *Low-level* languages.

As we chose to write *Processing* code rather than using *Processing.js* as a JavaScript library, the inclusion of this code is slightly different to the usual procedure for JavaScript libraries. As usual, we must link to the JavaScript library (in this case, *processing.js*), then we must use include a `canvas` element specifying the `data-processing-sources` attribute to be the *Processing* file.

```
<html>
  <head>
    <meta lang="en" charset="utf-8">
    <script src="processing.js"></script>
    <canvas data-processing-sources="processingjs_bar.pde"></canvas>
  </head>
</html>
```

Alternatively, one could use *Processing* (rather than *Processing.js*) to compile the `pde`. Several outputs are possible, including an HTML output or a stand-alone executable (which requires Java). A caveat, *Processing* is slightly stricter in grammar than *Processing.js*. For instance, in the example code there are cases where a `float` is implicitly coerced into an `int`. This is fine in *Processing.js*, but will cause errors in *Processing* (the coercion must be explicit using `int()`).

4.2 Paper.js

`http://paperjs.org/`

Key Points

Learning Difficulty Hard (Moderate if experienced in JavaScript).

Has a library of standard plots No.

Output HTML5 Canvas.

Interactivity Some potential, but not all object properties will cause the object to update in real-time restricting possibilities. Some internal object support.

Legal Open Source (MIT License).

Main Features

- JavaScript library for visualisations in general. Thus will require at least passing familiarity with JavaScript.
- Extending functionality to include interactivity cumbersome due to a lack of documentation on which specific properties can be altered dynamically. Thus potential is limited.
- Internal tracking of individual objects, allowing for object-based interactivity, despite using Canvas.
- Have to build graphics from scratch using low-level functions.

Comments

Paper.js is a JavaScript library for visualisations in general. Because of this, even simple plots like a bar plot requires a fair level of coding to build from scratch using low-level functions.

No generalised plotting library was found, but it is quite possible to write a library of low-level plotting aid functions that would make creation of new graphs almost as easy as R's default graphics system.

It can use a so-called *PaperScript*, this is merely JavaScript extended slightly. Thus using *Paper.js* does require some familiarity with JavaScript.

Though *Paper.js* outputs to a HTML5 Canvas element it internally tracks objects, allowing for object based interaction. However, this object system is not quite as robust as something like SVG.

A bargraph in Paper

A partially working example of the **Test Bargraph** in *Paper.js* can be found at `http://www.stat.auckland.ac.nz/~joh024/Research/Paperjs/test_bargraph.html`. Of all the Low-level Languages covered, *Paper.js* performed the worst.

- We found no way to control vertical justification of text. If it does exist (and we assume some method exists) it is very difficult to find.
- Though the internally tracked objects have several properties, such as `width`, `height`, `left`, `topLeft`, etc. most of these do not work dynamically. That is, one cannot adjust an object's size or position by changing these properties, and only a certain few properties work (no documentation was found clearly stating which ones work). This frustrating limitation has meant the numerous attempts to make the **Test Bargraph** interactivity work has failed. The only possible way we can see that should work requires scaling of the bars by computing the relative size of the new bar height with the previous height. This was considered to be too cumbersome to be bothered with, thus the example is only partially working, lacking the interactive features.

For reference and comparison purposes, the *Paper.js* code (excluding data and blank spaces) is 71 lines and 2277 characters. Due to the excessive features (low-level plotting aid functions for generality, but lacking interactivity), this is not directly comparable to the *High-level* languages, and is also not very comparable to other *Low-level* languages due to the incomplete nature of the example.

4.3 Raphaël.js

`http://raphaeljs.com/`

Key Points

Learning Difficulty Very Hard (Hard if experienced in JavaScript, Moderate if also familiar with the SVG specification).

Has a library of standard plots Yes but limited, see `http://g.raphaeljs.com/`.

Output SVG, with support for VML for compatibility with older versions of Internet Explorer.

Interactivity Broad object-based interactivity.

Legal Open Source (MIT License).

Main Features

- JavaScript library for visualisations in general. Thus will require at least passing familiarity with JavaScript.

- Extending functionality to include interactivity easy.
- Have to build graphics from scratch using low-level functions.
- As the output is SVG, specifying stylistic features, such as an object's colour, line width, etc., requires knowledge of the relevant SVG object specification.
- Does not give finer control over the specification of the SVG.

Comments

Raphaël is a JavaScript library for visualisations in general. Because of this, even simple plots like a bar plot requires a fair level of coding to build from scratch using low-level functions. As it is a JavaScript library, some familiarity with JavaScript is required.

Raphaël essentially treats SVG as another type of drawing canvas, handling many of the underlying SVG specifications via wrapper functions like `Paper.rect`. Thus, an in-depth understanding of the SVG specifications is not required, however, if changes to specific attributes are desired (e.g. colour, stroke width, etc.), then one must still refer to the relevant SVG specification. As *Raphaël* treats SVG as another type of canvas, this does limit the degree of control over the exact SVG specification. For instance, SVG's grouping feature (`g`) cannot be used with *Raphaël*, and certain optional attributes may not be available.

No generalised plotting library was found, but there is a very basic plotting library called *gRaphaël*. Usage is similar to a *High-level* language, but functionality is limited. It does however provide a basis that can be extended in more interesting directions. Additionally, it is quite possible to write a library of low-level plotting aid functions that would make creation of new graphs almost as easy as R's default graphics system.

A bargraph in Raphaël

A working example of the **Test Bargraph** in *Raphaël* can be found at `http://www.stat.auckland.ac.nz/~joh024/Research/Raphaeljs/test_bargraph.html`.

Due to the interactivity we desire, the object-based interactivity does not help us. In fact, it poses a barrier as the various bar objects will interfere with capturing mouse movement. One workaround is to draw a transparent rectangle over the entire graph region and working off this surface. The rest works similarly to a raster graphic (see *Processing*).

For reference and comparison purposes, the *Raphaël* code (excluding data and blank spaces) is 78 lines and 2382 characters. Due to the excessive features (low-level plotting aid functions for generality, demonstrative interactivity), this is not directly comparable to the *High-level* languages, but can be loosely compared to other *Low-level* languages.

4.4 D3.js (Data-Driven Documents)

`http://d3js.org/`

Key Points

Learning Difficulty Extremely Hard (Very Hard if experienced in JavaScript, Hard if also familiar with the SVG specification).

Has a library of standard plots Examples available, but not as high-level functions.

Output SVG, but the nature of D3 allows for any document-based output (e.g. a simple HTML document containing only text).

Interactivity Broad object-based interactivity, generally limited only by what the user can code.

Legal Open Source (BSD 3-Clause License).

Main Features

- JavaScript library for "manipulating documents based on data" to create data-based visualisations. It requires passing familiarity with JavaScript, HTML and anything else desired in that document, which in the case of using it for graphics will mean SVG and CSS.
- The requirement of having to understand several almost independent standards (HTML, JavaScript, SVG and CSS), then bring them together by using the D3 API, results in a very high barrier to entry.
- Extending functionality to include interactivity easy, assuming user familiarity with all the required components.
- Have to build graphics from scratch using low-level functions.
- As the output is SVG, specifying stylistic features, such as an object's colour, line width, etc., requires knowledge of the relevant SVG object specification.
- Gives a high level of control and flexibility over the specification of the document, and in turn the SVG image(s).

Comments

D3.js is a JavaScript library intended to be *Transformative* rather than *Representative*. That is, unlike the other Low-level Languages where the user uses the library's API to draw a rectangle here, or a circle there, *D3.js* is instead used to create a structured document, and in the case of the SVG that document describes an image. So the user might define a *circle* element with certain attributes, or a *rect* element, etc. This 'document' can then be rendered into an image with a modern browser.

In practice what this means is the user must have some understanding of all the components involved, which with visualisation would include HTML, JavaScript, *D3.js*, SVG and CSS. Learning all these components, even to the basic level to begin to use *D3.js*, poses a significant challenge, and that is the biggest disadvantage in attempting to use *D3.js*. However, once these components are learnt *D3.js* gives incredible control over the output, as the user is in effect writing the HTML and SVG from scratch, but with the help of the *D3.js* API to make things easier.

Convenience features *D3.js* add include:

Selections A shorter and easy way to select objects (including elements in a structured document like XML or SVG), based on the W3C Selectors API (the same API used by CSS).

Attaching Data Allows data to be attached to a group of elements, for instance attaching 10 numbers to 10 `rect` elements to be used as heights for a bargraph.

Scales Easily create conversion functions to convert to and back from a particular coordinate space, such as in a graph region. Also provides an automatic way to generate ticks appropriate for the scale (including consideration of numbers appropriate for human-reading).

Transitions Automatically makes the necessary calculations and adjustments to display an animated transition from one set of attributes to another. This can be used for instance to smoothly transition (over a specified period of time) the height of a bar from one value to another.

No generalised plotting library was found though there are examples available that draw some standard statistical plots, which can help in writing new code. Additionally, it is quite possible to write a library of low-level plotting aid functions that would make creation of new graphs almost as easy as R's default graphics system.

A bargraph in D3

A working example of the **Test Bargraph** in *D3.js* can be found at `http://www.stat.auckland.ac.nz/~joh024/Research/D3js/test_bargraph.html`.

Due to the interactivity we desire, the object-based interactivity does not help us. In fact, it poses a barrier as the various bar objects will interfere with capturing mouse movement. Due to the level of control *D3.js* grants us, it is possible to utilise the `pointer-events` property of SVG elements to effectively ignore the bars from registering on mouse events. The rest works similarly to a raster graphic (see *Processing*).

For reference and comparison purposes, the *D3.js* code (excluding data and blank spaces) is 94 lines and 2925 characters for the main code plus the CSS with 11 elements and 492 characters. Due to the excessive features (low-level plotting aid functions for generality, demonstrative interactivity), this is not directly comparable to the *High-level* languages, but can be loosely compared to other *Low-level* languages.

4.5 Other Low-level Languages

The tools covered Section 4 are by no means comprehensive. They were chosen as they were found to be popular and comparatively easy to use and implement (as they are simply JavaScript libraries), compared to tools which might use Java, which often requires the Java platform to be installed by both the developer (jdk - Java Developer Kit) and the user (jre - Java Runtime Environment), making implementation and distribution of output considerably more cumbersome. Here we briefly cover *some* other tools we came across but did not cover in depth.

Prefuse `http://prefuse.org/` Uses Java to produce Java-based graphics.

Flare `http://flare.prefuse.org/` By the same developers as Prefuse. Uses ActionScript to produce SWF graphics. This has been shown to have some performance benefits compared to a tool like *D3.js* (Bostock et al. 2011) which may be worth the added hassle of using ActionScript and SWF.

Piccolo2D `http://www.piccolo2d.org/` Uses Java and C# to produce Java-based or .NET-based graphics.

Protovis `http://mbostock.github.com/protovis/` JavaScript Library for producing SVG output. Can be considered obsolete, developers moved on to *D3.js*.

5 Other Relevant Topics

5.1 R

`http://www.r-project.org/`
R is a free software environment for statistical computing and graphics. (R Core Team 2012)

5.1.1 Using R to Prepare and Export the Data

All the *High-* and *Low-level* language tools we cover are JavaScript libraries. This means a data format that's very easy to use with these tools is the JSON (JavaScript Object Notation). Though JavaScript functions can be found to read some other data formats (such as CSV), we have found these to be comparatively cumbersome, and if any data cleaning or other preparation is required, working with JavaScript is not recommended.

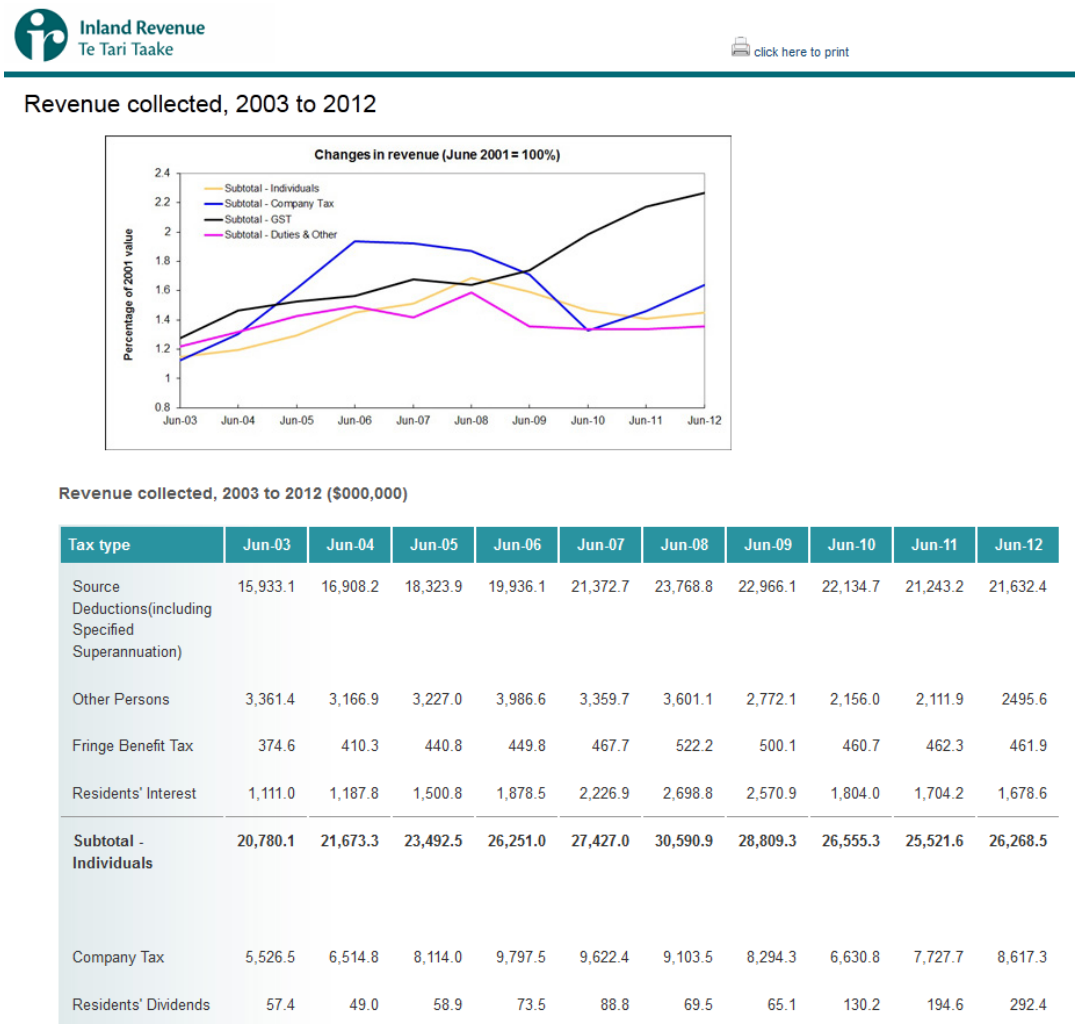
One much better alternative is to use `R`. The base installation gives capabilities to easily read many simple data formats, including CSV (and variants thereof) and fixed width. More valuably, many packages extend functionality, such as the `XLConnect` (Mirai Solutions GmbH 2012) package which enables the reading of Excel files, even from Linux or Mac machines, or the `XML` (Lang 2012b) package which enables extraction of HTML Tables, potentially

straight from websites (with the `RCurl` (Lang 2012a) package which allows `R` to be used as a text-web-browser).

Once the data has been read in, `R` provides extensive capabilities to manipulate the data, and if any statistical analysis is desired, this can also be conducted in `R`.

When the data is ready to be exported for use in a visualisation tool, a package such as `rjson` (Couture-Beil 2012) makes it very easy to convert any of `R`'s accepted data formats to JSON. This result can then be saved to a `.json` format and read in, or alternatively it can be copied directly into your JavaScript as a vector or array.

As a practical example of using `R`, the following code demonstrates how to download a HTML Table directly from the IRD website (`http://www.ird.govt.nz/aboutir/external-stats/revenue-refunds/tax-revenue/printable/`), convert to JSON, then write to a `.js` file for easy access by any JavaScript code.



Revenue collected, 2003 to 2012 ($000,000)

Tax type	Jun-03	Jun-04	Jun-05	Jun-06	Jun-07	Jun-08	Jun-09	Jun-10	Jun-11	Jun-12
Source Deductions(including Specified Superannuation)	15,933.1	16,908.2	18,323.9	19,936.1	21,372.7	23,768.8	22,966.1	22,134.7	21,243.2	21,632.4
Other Persons	3,361.4	3,166.9	3,227.0	3,986.6	3,359.7	3,601.1	2,772.1	2,156.0	2,111.9	2495.6
Fringe Benefit Tax	374.6	410.3	440.8	449.8	467.7	522.2	500.1	460.7	462.3	461.9
Residents' Interest	1,111.0	1,187.8	1,500.8	1,878.5	2,226.9	2,698.8	2,570.9	1,804.0	1,704.2	1,678.6
Subtotal - Individuals	**20,780.1**	**21,673.3**	**23,492.5**	**26,251.0**	**27,427.0**	**30,590.9**	**28,809.3**	**26,555.3**	**25,521.6**	**26,268.5**
Company Tax	5,526.5	6,514.8	8,114.0	9,797.5	9,622.4	9,103.5	8,294.3	6,630.8	7,727.7	8,617.3
Residents' Dividends	57.4	49.0	58.9	73.5	88.8	69.5	65.1	130.2	194.6	292.4

Figure 10: A screenshot of the webpage containing the HTML Table from which we will obtain our data. **Source:** Inland Revenue and licensed by Inland Revenue for re-use under the Creative Commons Attribution 3.0 New Zealand Licence.

```
## Load required libraries
library(RCurl)
library(XML)
library(rjson)
```

```
## Load webpage and get the data table
## Splitting the URL is only done for display purposes
URL = getURL(paste0("http://www.ird.govt.nz/aboutir/external-stats/",
                    "revenue-refunds/tax-revenue/printable/"))
htmlpage = htmlParse(URL, asText = TRUE)
datatable = readHTMLTable(htmlpage, stringsAsFactors = FALSE)[[1]]
> datatable[1:9,1:5]
                          V1       V2       V3       V4       V5
1                  Tax type   Jun-03   Jun-04   Jun-05   Jun-06
2         Source Deductions 15,933.1 16,908.2 18,323.9 19,936.1
3             Other Persons  3,361.4  3,166.9  3,227.0  3,986.6
4         Fringe Benefit Tax    374.6    410.3    440.8    449.8
5       Residents' Interest  1,111.0  1,187.8  1,500.8  1,878.5
6    Subtotal - Individuals 20,780.1 21,673.3 23,492.5 26,251.0
7
8               Company Tax  5,526.5  6,514.8  8,114.0  9,797.5
9      Residents' Dividends     57.4     49.0     58.9     73.5

## Clean data table
colnames(datatable) = datatable[1,]
datatable = datatable[-c(1, 6:7, 12:13, 15:16, 21:23),]

> datatable[1:9,1:5]
                    Tax type   Jun-03   Jun-04   Jun-05   Jun-06
2         Source Deductions 15,933.1 16,908.2 18,323.9 19,936.1
3             Other Persons  3,361.4  3,166.9  3,227.0  3,986.6
4         Fringe Benefit Tax    374.6    410.3    440.8    449.8
5       Residents' Interest  1,111.0  1,187.8  1,500.8  1,878.5
8               Company Tax  5,526.5  6,514.8  8,114.0  9,797.5
9      Residents' Dividends     57.4     49.0     58.9     73.5
10 Foreign Source Dividends    153.6    138.6    188.4    160.1
11    Non-residents' Income    731.7    799.9    926.7  1,095.6
14                  IRD GST  7,393.9  8,467.9  8,838.6  9,054.4

## Convert to JSON and write to a .js that can be
##   sourced directly for use as a JavaScript array
writeLines(paste("var IRDTable =", toJSON(datatable)), "IRDTable.js")
```

A few notes:

- We have trimmed the row name for Source Deductions for space.
- We would normally prefer to process the numbers to remove the commas before exporting which we have skipped for clarity. For completeness the code required for this particular example is:

  ```
  cbind(datatable[1], apply(datatable[,-1], 2, function(x)
    as.numeric(gsub("-$", "0", gsub(",| ", "", x)))))
  ```

  This handles commas in numbers, erroneous spaces and 0 being represented as - (problems not shown in the screenshot).

5.1.2 Visualisations with R

R has extensive visualisation capabilities mostly focused on traditional offline static graphics, however certain packages extend functionality in ways worth mentioning.

RGL `http://rgl.neoscientists.org/about.shtml` Has the potential to produces 3D output using OpenGL, which utilises the GPU for a very high level of performance.

RGGobi `http://www.ggobi.org/rggobi/` Access GGobi from with `R`, a powerful tool enabling interactive dynamic graphics, though only offline and not so easily shared. Learning curve for the tool is moderate to high.

iPlots `http://www.rosuda.org/iplots/` Interactive Java-based graphics, though only offline and not so easily shared.

playwith `https://code.google.com/p/playwith/` Adds a GUI for editing and interacting with `R` plots.

Of greater relevance however is the `gridSVG` (Murrell 2012) package. This enables graphics drawn in `grid` (R Core Team 2012) (which include plots drawn by `lattice` (Sarkar 2008) and `ggplot2` (Wickham 2009)) to be exported to SVG, while still retaining much of the structured information. This output can then be manipulated from within `R` through the `XML` package, or even extended using a tool such as `D3.js`, opening the possibility of extending existing `R` graphics to become interactive web-based (SVG) output. Of course, doing this would require familiarity with all the tools involved, including `R`, `grid`, `gridSVG`, the `grid`-based plotting library (such as `lattice`), `SVG` and `D3.js`.

5.2 Raster graphics

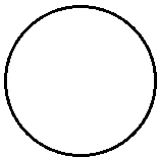


Figure 11: An example raster graphic of a circle. The original image is a 150 by 150 pixel image saved as a png.

The most familiar type of graphic, files like jpeg, png, gif and bmp, are *Raster graphics*. The image is defined by specifying each pixel. An easy way to tell is to zoom in on the image: the individual pixels will be revealed and any curves will become jagged.

Raster graphics has no concept of objects. There is simply the picture, as defined by what each pixel is. Thus in terms of computing power required to render the image, what is most important is how many pixels there are (the *resolution* or *dimensions* of the image).

A simple demonstration of this can be found at `http://www.stat.auckland.ac.nz/~joh024/Research/Processingjs/stress_testing.html`. Every second 1000 circles are drawn on a canvas with dimensions 1280 by 720, however the computing resources required to render the image does not increase. Zooming in or out of the image also poses little computational burden, though zooming in too far will reveal the individual pixels.

5.3 Vector graphics

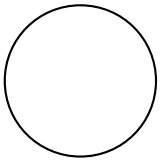


Figure 12: An example vector graphic of a circle. The circle is drawn in LaTeX using the `tikzpicture` package.

Perhaps more familiar than people may realise, text on a pdf file can be considered to be *Vector graphics*. The 'image' (in this case the letters of the text) is defined by code, and is drawn by the rendering software (in this case your pdf viewer). As you zoom in or out, the rendering software redraws, meaning the image always looks good. However this also means the drawn image can look subtly different depending on the rendering software used.

As *Vector graphics* are defined by code, it is possible to identify each individual object. However, this also means that each individual object must be rendered separately, which can become a computational burden when the number of individual objects becomes very large (e.g. a million).

A simple demonstration of this can be found at `http://www.stat.auckland.ac.nz/~joh024/Research/D3js/stress_testing.html`. Every second 1000 circles are drawn on a SVG image with dimensions 1280 by 720, resulting in an increase in CPU and RAM usage. Running the graphic for an extended period of time will cause severe performance loss including affecting the responsiveness of the renderer (web browser) itself. Zooming in or out of the image also results in a high computational burden as each individual circle must be redrawn, though this means the circles always look good at any zoom level.

5.4 HTML5 Canvas Element

Without going into too much technical detail, the HTML5 *Canvas Element* can be understood to be a drawing surface for *Raster graphics*, where the drawing can be specified by scripts.

As the *Canvas* is a *Raster graphic*, objects cannot be separately identified or modified once drawn; dynamic graphics are achieved by redrawing the *Canvas* from scratch. However, web browsers are amazingly good at drawing and redrawing the *Canvas* quickly and high framerates are easily achieved.

This also means object-based interactivity is no easier than any other interactivity, and is accomplished via *coordinate matching* - that is, the coordinate of the mouse is checked to see if it 'hits' any 'object'. This hit detection is trivial for rectangular objects but becomes more difficult for increasingly complex shapes.

Also because of the *Raster* property the image will scale poorly (much like zooming too closely in a jpeg image). This can however be overcome by forcing a redraw of the image dynamically to match the zoom level, or to dynamically resize the Canvas resolution and redraw the image to match this resolution (allowing for a greater level of zoom before defects can be found).

See *Processing* (Section 4.1) or *Paper.js* (Section 4.2) for tools that use Canvas.

5.5 SVG

`http://www.w3.org/Graphics/SVG/`

The *Scalable Vector Graphics* (*SVG*) is an image that is specified via a XML file (but with extension `.svg`). As the name implies, it is a *Vector graphic*, that can be rendered by modern web browsers (like Firefox, Internet Explorer, etc.).

The standards are extensive. The pdf version of SVG 1.1 Second Edition[3] is 826 A4 pages. These standards can be considered to be a 'low-level graphics language' that can be used to draw (via a renderer like a web browser) anything from a rectangle to professionally typeset text. The following code specifies a SVG image of 150 by 150, drawing a rectangle with upper-left corner at x = 25 and y = 25, with width and height of 100. The units are pixels at 100% zoom, but of course being a *Vector graphic* it can be scaled at will.

```
<svg width = "150" height = "150">
  <rect x = "25" y = "25" width = "100" height = "100"></rect>
</svg>
```

Being a *Vector graphic* it also understands objects. For instance that rectangle can be assigned a specific `id`, and further manipulation (via something like JavaScript) can be conducted on an object basis. Thus object-based interactivity (like clicking a specific object)

[3] `http://www.w3.org/TR/SVG/`

is very easy. However, any interactivity that encompasses multiple objects may still require coordinate matching to work.

See *Google Chart Tools* (Section 3.1), *Highcharts* (Section 3.2), *Raphaël* (Section 4.3) or *D3.js* (Section 4.4) for tools that use SVG.

6 Conclusion

One great way to make sense of data is by way of visualisations; by using web-based output these visualisations become very easy to share and capable of some interesting and advanced interactivity. To aid in the creation of such web-based interactive visualisations there are a variety of Graphical Tools already available.

GUI tools and *High-level* languages are easy to use and enable the use of predefined visualisations. Though these tools offer some level of customisation, these are minor and never fundamentally add or remove features from the visual. To gain the freedom to extend existing visualisations to include new features (such as new interactive or dynamic features) or to create a wholly new type of visualisation, requires the use of serious graphical tools, broad graphical tools not limited to just statistical plots but capable of any visualisation - provided the user can code it in. Indeed, that is the major barrier to these *Low-level* languages: they require the learning and use of programming languages (typically JavaScript plus the API of the graphical tool plus any other applicable web standards, such as HTML, SVG and CSS), a barrier that can be considered insurmountable to a significant portion of the general public.

Thus there are opportunities to fill the gap between the very easy and the very difficult, a tool that can enable users who may have creative and innovative ideas for new visualisation methods, but do not have the programming expertise to actually make it.

References

Bostock, M., Ogievetsky, V., Heer, J., 2011. D3: Data-driven documents. IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis) .
URL `http://vis.stanford.edu/papers/d3`

Couture-Beil, A., 2012. rjson: JSON for R. R package version 0.2.10.
URL `http://CRAN.R-project.org/package=rjson`

D3.js, 2012.
URL `http://d3js.org/`

Google Chart Tools, 2012.
URL `https://developers.google.com/chart/`

Highcharts, 2012.
URL `http://www.highcharts.com/`

Lang, D. T., 2012a. RCurl: General network (HTTP/FTP/...) client interface for R. R package version 1.91-1.1.
URL `http://CRAN.R-project.org/package=RCurl`

Lang, D. T., 2012b. XML: Tools for parsing and generating XML within R and S-Plus. R package version 3.9-4.1.
URL `http://CRAN.R-project.org/package=XML`

Many Eyes, 2012.
URL `http://www-958.ibm.com/software/data/cognos/manyeyes/`

Mirai Solutions GmbH, 2012. XLConnect: Excel Connector for R. R package version 0.2-0.
URL `http://CRAN.R-project.org/package=XLConnect`

Murrell, P., 2012. gridSVG: Export grid graphics as SVG. R package version 0.9-1.
URL http://CRAN.R-project.org/package=gridSVG

Paper.js, 2012.
URL http://paperjs.org/

Processing, 2012.
URL http://processing.org/

Processing.js, 2012.
URL http://processingjs.org/

R Core Team, 2012. R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria, ISBN 3-900051-07-0.
URL http://www.R-project.org/

Raphaël.js, 2012.
URL http://raphaeljs.com/

Reas, C., Fry, B., 2006. Processing: programming for the media arts. AI & Society 20 (4), 526 – 538.

Sarkar, D., 2008. Lattice: Multivariate Data Visualization with R. Springer, New York.
URL http://lmdvr.r-forge.r-project.org

Stolte, C., Tang, D., Hanrahan, P., Nov. 2008. Polaris: a system for query, analysis, and visualization of multidimensional databases. Commun. ACM 51 (11), 75–84.
URL http://doi.acm.org/10.1145/1400214.1400234

Tableau, 2012.
URL http://www.tableausoftware.com/products/public

Wickham, H., 2009. ggplot2: elegant graphics for data analysis. Springer, New York.
URL http://had.co.nz/ggplot2/book

World Wide Web Consortium, 2012.
URL http://www.w3.org/