University of Auckland

Department of Statistics

Dynamic, Interactive and Reactive Statistical Graphics for the Web

Author:
Simon J. Potter

Supervisor:
Dr. Paul Murrell

A thesis submitted in partial fulfilment of the requirements for the degree of Master of Science in Statistics, The University of Auckland, 2013.

Abstract

This thesis describes a body of work in the area of web-based statistical graphics. This is primarily concerned with the development of gridSVG, a package for the R statistical software system. The result of developing gridSVG is that flexible dynamic, interactive, reactive graphics for web pages can now be produced by R. In addition, two other packages have been developed to assist the creation of graphics with gridSVG: selectr and animaker. The motivation and development of all three packages and their intended use cases are discussed.

Acknowledgements

I would like to thank Dr. Paul Murrell for not only introducing me to grid graphics, but also to gridSVG. Without his prior work, none of this would have been possible. Whenever I needed guidance, he was always there to give clarity to my thoughts or an insightful path for me to pursue. His supervision throughout the duration of my thesis was fantastic and I could not have asked for anything more.

I would also like to thank Prof. Chris Wild for leading me to think of the motivating idea for this thesis. His interest in my work, and the suggestions he put forward were greatly appreciated.

Preface

The content described in this thesis refers to web pages and images that are dynamic, interactive and animated. The nature of this printed document means that it cannot accurately reproduce some of the described content. However, technical reports have been published for most chapters and are viewable as web pages which demonstrate the full dynamic and interactive features of gridSVG images. All technical reports were published on the University of Auckland's Department of Statistics Technical Report Blog (`http://stattech.wordpress.fos.auckland.ac.nz/`). These reports contain examples (and their source code) that are interactive and animated, consequently they may be of interest to readers and are listed below:

- Title: Working with the gridSVG Coordinate System
  URL: `http://stattech.wordpress.fos.auckland.ac.nz/2012-6-working-with-the-gridsvg-coordinate-system/`
  Notes: Relates to the content written in chapter 4.

- Title: A Structured Approach for Generating SVG
  URL: `http://stattech.wordpress.fos.auckland.ac.nz/2012-7-a-structured-approach-for-generating-svg/`
  Notes: Relates to the content written in chapter 3.

- Title: Introducing the selectr Package
  URL: `http://stattech.wordpress.fos.auckland.ac.nz/2012-10-introducing-the-selectr-package/`
  Notes: Relates to the content written in chapter 5.

- Title: Generating Animation Sequence Descriptions
  URL: `http://stattech.wordpress.fos.auckland.ac.nz/2012-11-generating-animation-sequence-descriptions/`
  Notes: Relates to the content written in chapter 6. In particular this relates to section 6.1.

- Title: TimingManager: Animation Sequences in JavaScript
  URL: http://stattech.wordpress.fos.auckland.ac.nz/2012-13-timingmanager-animation-sequences-in-javascript/
  Notes: Relates to the content written in chapter 6. In particular this relates to subsection 6.2.2.

- Title: Generating Unique Names in gridSVG
  URL: http://stattech.wordpress.fos.auckland.ac.nz/2013-3-generating-unique-names-in-gridsvg/
  Notes: Relates to the content written in chapter 2.

- Title: Generating Structured and Labelled SVG
  URL: http://stattech.wordpress.fos.auckland.ac.nz/2013-4-generating-structured-and-labelled-svg/
  Notes: Relates to the content written in chapter 2. It gives an overview on why and how it is particularly useful.

- Title: Advanced SVG Graphics from R
  URL: http://www.stat.auckland.ac.nz/~paul/Reports/leaf/leaf.html
  Notes: Relates to the content written in chapter 7.

Software

The software developed during the course of this thesis are hosted in various locations online. These are listed below:

- gridSVG

  The majority of the content in this thesis describes features of gridSVG that are present in version 1.2.

  This R package is hosted on R's CRAN and is therefore installable in R by running the command `install.packages("gridSVG")`. It is developed on R-Forge, so more up-to-date versions of gridSVG may be available at http://r-forge.r-project.org/projects/gridsvg/.

- selectr

  This R package is hosted on R's CRAN and can be installed by running the command `install.packages("selectr")`. It is developed on GitHub at `https://github.com/sjp/selectr/`, where more recent versions of the package may be available.

- animaker

  At the time of writing, this R package is not available on CRAN. However, the code is freely available online on GitHub `https://github.com/pmur002/animaker/`. This does means that users will have to download, build and install the package in order to use animaker.

- TimingManager

  This is a JavaScript library that is available at `http://sjp.co.nz/projects/timing-manager/`. Development occurs on GitHub at `https://github.com/sjp/TimingManager/` where more recent versions of the library may be available.

Conventions

The following typographical conventions are used throughout this thesis:

softwareName Indicates that the text is a proper noun, referring to the name of a software application or package, or a programming language.

`code` Indicates computer code in a broad sense. This may include: SVG and HTML tags, keywords, attributes, values, variables.

`function()` Indicates computer code that refers to a function.

Contents

List of Figures

List of Tables

1 Introduction

1.1 What are these Graphics?

There are numerous software packages that are capable of creating graphs of some form. Many of these packages are also capable of producing statistical graphics — graphs which communicate statistical information. The R statistical software system (R Development Core Team, 2013) is a popular example of software which generates statistical graphics. The great strength of R is its wealth of statistical functions and graphics, but only for static graphics. The proliferation of dynamic and interactive web-based graphics using JavaScript libraries such as D3 (Bostock, 2013) shows that there is significant interest in creating these graphics in web browsers.

In order to explain the types of graphics we wish to create, we will first examine what R can produce, and how they are used within a web browser.

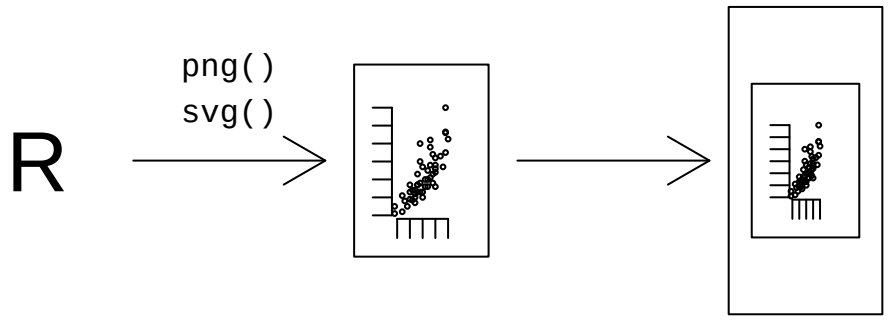


Figure 1.1: An R image being included within an HTML document.

Figure 1.1 demonstrates that R can generate a static image using its built-in graphics devices. In this example the `png()` and `svg()` devices can produce PNG (Portable

Network Graphics) and SVG (Scalable Vector Graphics) images respectively. These images are static, and can be included in an HTML (Hypertext Markup Language) document using the <img> tag.

When we include images onto a web page in this manner, the effect is largely the same as if we were treating the HTML document as a piece of paper. Unlike a piece of paper, a useful feature of web browsers are their ability to view content that is not fixed. A key example of this is when we want to include animated graphics within a web page. There are packages for R that enable animation within a web browser. We will first consider the prominent animation package.

The animation package (Xie, 2013) uses software outside of R to construct its animations, e.g. ImageMagick (ImageMagick Studio LLC, 2013) and ffmpeg (FFmpeg Team, 2013). These are additional pieces of software required to produce animations in various formats. The general idea of animation is to repeatedly draw entire plots, each of which are saved. The saved images are then stitched together into one of the many formats that animation supports. This means that animation can create an animated GIF (Graphics Interchange Format) or a video out of the images that have been saved. The resulting video or animated GIF can then be inserted into a web page using the appropriate HTML tag (<video> and <img> respectively).

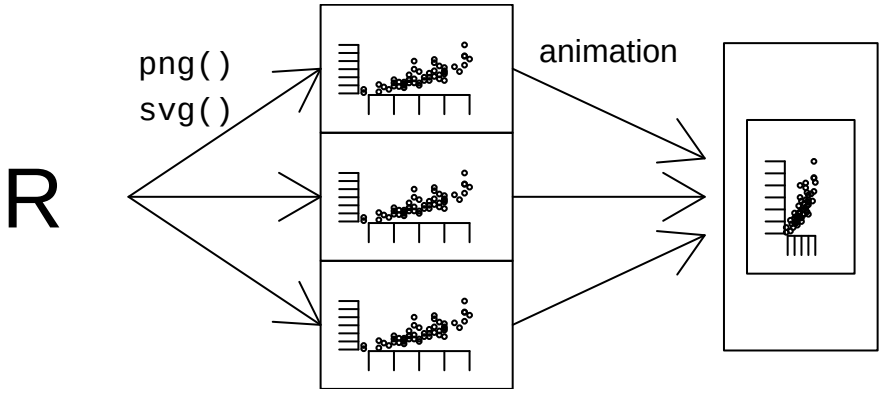


Figure 1.2: Multiple static images generated by the animation package are included in an HTML document. The images are stitched together to create an animation.

The method for using animation is demonstrated in Figure 1.2. The main idea to get across is that several images are generated by R, which are then stitched together into an animation-capable format using animation. The result is then included into a web page.

An alternative to using animation is to write some JavaScript code to be included

within an HTML document. The JavaScript can recreate an animation by referring to generated static images, and then repeatedly update the HTML to reference the next image in the sequence. The described behaviour is in fact already provided by animation through its `saveHTML()` function. However, it is a "black box" solution which may not be suitable for all needs. Furthermore, it does not enable further interactivity that may be possible by writing our own JavaScript and HTML code. This approach is illustrated in Figure 1.3.

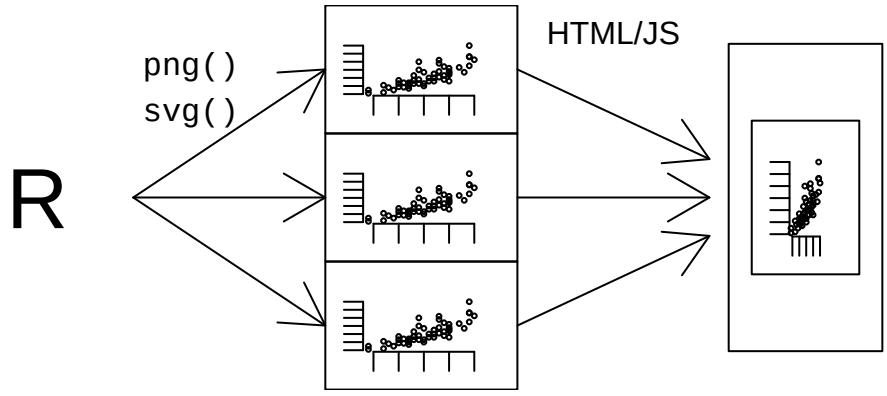


Figure 1.3: Multiple static images can be included within an HTML document with custom JavaScript used to provide interactivity and/or animation.

The JavaScript shown in Figure 1.3 can be any JavaScript code that can interact with static images. In practice the JavaScript code will likely be limited to toggling which image(s) to display at a given point in time.

A limitation in the types of images that are typically used in web pages is that they are *raster* images. This means that the image itself is a description of the colour values encoded in a matrix of pixels. There is no other information available so we cannot identify or manipulate any part of a plot.

Another image format that can be used in a web browser is SVG (Scalable Vector Graphics) (W3C SVG Working Group, 2011). SVG is an XML-based (Extensible Markup Language) (W3C XML Working Group, 2008) vector image format that supports animation and interactivity (via JavaScript). The fact that SVG is a vector image format means that we can scale the image to any size and it will not lose any fidelity. The most encouraging feature is the XML-based nature of SVG, meaning that SVG inherits behaviour from XML.

The most interesting features of XML for creating SVG are that it is a structured

text format and there is a standard interface for manipulating an SVG image once it has been loaded, this interface is called the DOM (Document Object Model) (W3C Web Applications Working Group, 2004). This means that SVG enables (for example) the colour of a piece of text to change after the image has been loaded in a browser.

There are two packages for R which use SVG to add interactivity and animation to plots: SVGAnnotation (Nolan and Temple Lang, 2012) and gridSVG (Murrell and Potter, 2013b). The key difference between the two packages is how they produce animated and interactive SVG images. SVGAnnotation post-processes the SVG produced by R's `svg()` device. It also provides functions for annotating the post-processed SVG with animation, tooltips, and linking of two plots. It is possible to add custom JavaScript to these plots but it is difficult to write the required code because the structure of the SVG is not clear and understandable. gridSVG takes the approach of writing its own SVG output by translating grid primitives into their SVG equivalents. Animation and interactivity is performed by creating or annotating grid graphics objects through high-level functions.

The advantage using the gridSVG approach is that it is easier to manipulate gridSVG output than the output produced by SVGAnnotation as it creates its own SVG and annotates it with grid object names. However, gridSVG is limited to only creating SVG from grid output, while SVGAnnotation is able to do this for both the grid and base graphics systems provided by R.

The approach that both of these packages take to produce their plots is illustrated by Figure 1.4.

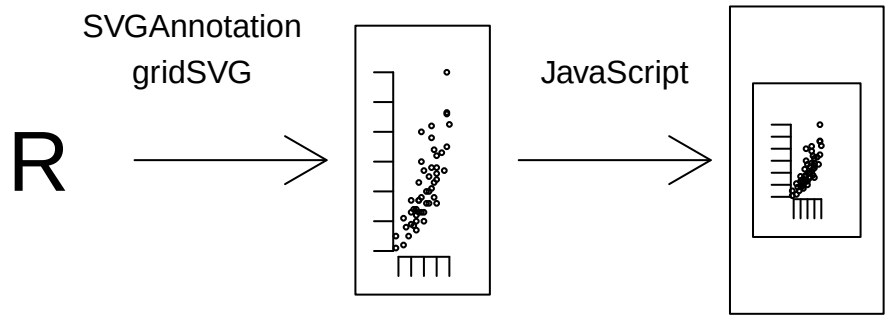


Figure 1.4: A plot generated by packages that can embed animation in an SVG image. The image is included within an HTML document, with interactivity provided by JavaScript.

All of the methods for getting R graphics in the browser have one feature in common —

they are all pre-generated. This means that whenever a plot is created in R, the statistical information encoded in that image (or indeed JavaScript code for SVGAnnotation and gridSVG in particular) becomes fixed. We cannot add more statistical information to the plot without communicating with R, which is not possible with the aforementioned solutions. To produce a plot with different information would require R to create an entirely new plot, regardless of whether the plot is static, dynamic or interactive. What this means is that a web page with embedded statistical graphics cannot update the images without some way of communicating to R.

There are several existing pieces of software that enable a web browser to communicate to R. The way this occurs is that a software package can expose R's functionality by creating a web server that can interface with a web browser using HTTP (Hypertext Transfer Protocol). A web browser can then request resources, such as statistical documents and graphics. In practice this means instead of a loading a standalone HTML file on a local hard disk (or even a remote location), R generates the web page that is loaded within the browser. Because the web page is generated as it is requested, it does not need to be fixed.

There are many R packages that are able to perform the described functionality. Examples of these are shiny (RStudio, 2013), Rook (Horner, 2013) and FastRWeb (Urbanek and Horner, 2013) (which requires Rserve (Urbanek, 2003)). The details of existing server software will be discussed in more detail later in this chapter. The general idea with all of these packages is that a web browser requests a resource — usually a web page or an image — which is generated by R, then delivered to a web browser.

To explain why these packages are more useful than pre-generated web pages and images, consider this example. A web page features a scatter plot with a LOESS Curve where an HTML slider control determines the size of the smoothing window. Each time the value of the slider control changes we want to update the plot by replacing the existing LOESS curve with a new one. This is simple when there is a web server that communicates to R, because we can just request an entirely new plot which replaces the existing plot. This simple example is one where a pre-generated HTML page is infeasible. Attempting a pre-generated solution would require creating every possible plot, which in this case is determined by the possible sizes of the LOESS smoothing window. Because the value of the LOESS spanning window is a continuous variable, this is not possible as it would require an incredibly large amount of plots. Even if were to limit the possible number of plots to 100, that is still a large amount of work to pre-generate, especially if many of the plots will not be seen.

The dynamically generated solutions are able to produce a dynamic HTML document

using the process illustrated in Figure 1.5.

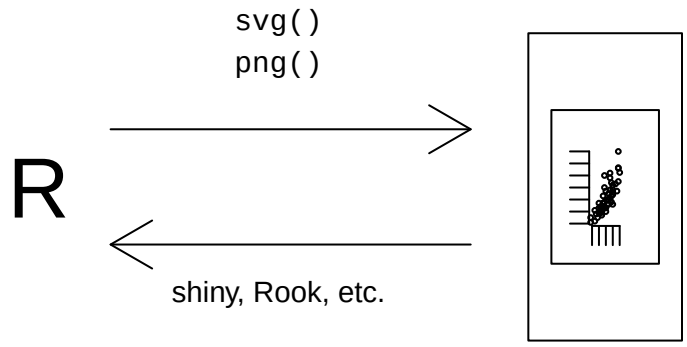


Figure 1.5: A change in the state of a web page allows new static images to be requested from R using one of many R web server packages.

The plots that are dynamically generated can be any one of the types of plots that were discussed earlier. Typically the dynamically generated images are simply a static plot, often the result of drawing on the `png()` device. In the example that was discussed earlier, the simple method for updating the plot that appeared on the web page was removing the old image and inserting the newly generated image. If the goal is to update the LOESS curve then generating an entirely new image is not only wasteful — because most of the plot has not changed — but it does not give us any way of showing *how* the curve changed.

What we aim to achieve is to be able to *modify* an R plot with dynamically generated graphical content within a web browser. What this requires is not only to be able to communicate with R to generate new content, but also to be able to identify and modify *parts* of a plot in a web page. We have already mentioned that some R packages allow us to communicate with R but there are few options available for being able to modify parts of a plot. The only image format that is capable of the desired behaviour is SVG. R's `svg()` device produces SVG that is viewable in a web browser, but unreadable to anyone who wishes to modify it. SVGAnnotation reverse engineers and modifies `svg()`'s output to enhance it with animation and interactivity. Unfortunately, once SVGAnnotation plots are loaded into a web browser, they are still difficult to modify because the resulting SVG is largely unchanged from `svg()`. This leaves gridSVG as the only remaining option as one of its goals is to produce SVG with identifiable output. This means that when gridSVG produces a scatter plot with a LOESS curve, we are able to easily identify the

SVG elements that the LOESS curve corresponds to.

Relating back to the example, we should be able to load a web page with a gridSVG plot. When an HTML slider modifies the value of a LOESS smoothing window, R can give us a new *piece* of a gridSVG plot which represents the new curve. The new piece can then replace or update the existing piece of the plot. A bonus of this approach is that we can (if desired) exploit the powerful functionality of JavaScript libraries such as D3, which can *transition* the curve to its new values. Figure 1.6 exhibits how the pieces of software fit together.

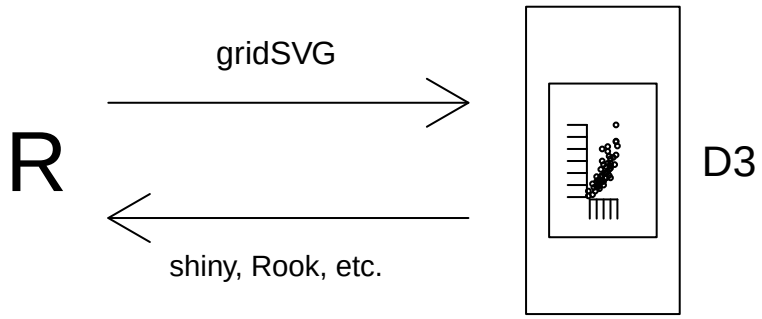


Figure 1.6: A piece of a gridSVG image is sent to a web browser in response to a request made to an R web server instance. D3 handles the process of integrating the piece into the existing image.

Without the ability to identify parts of a plot, it is very difficult to perform any useful modification of a plot. gridSVG's naming of SVG elements allows us to easily identify what we are attempting to modify. The other options, svg and SVGAnnotation are therefore not suitable for the type of dynamically generated graphics that we wish to produce. However, prior to this research gridSVG was not yet a viable solution that is capable of creating these graphics. It was the solution with the greatest potential, and this thesis largely describes how gridSVG has been improved to meet the goal of creating dynamic, interactive and reactive statistical graphics.

1.2 Existing Software

We have discussed the use of R as a tool for generating web pages but have glossed over the details as to how this occurs. For a web browser to be able to communicate to a web

service that exposes functionality provided by R, it needs to communicate over a *protocol*. A protocol is simply a standard set of rules that a web browser and a web server can use to send messages between each other, much like humans use a language to send messages to one another. There are two protocols of interest that are used to construct web pages, HTTP and WebSockets.

HTTP operates in a request-response model. This means that a web browser sends a message to a web server that requests a resource (e.g. a web page or an image). The web server responds with a message containing the resource that the web browser requested, that the browser can then render. The majority of the software packages that provide a web interface to R operate using HTTP.

WebSockets is a protocol which allows for bi-directional communication between a WebSocket server and a web browser. Instead of requesting a resource, a web browser sends a message over the WebSocket which is received by the WebSocket server. The server needs to know how to read the message, and it also needs to know what task the message is asking to be performed. If both of these requirements are fulfilled, then the WebSocket server can send a message along the socket to the web browser. The message could (for example) include new content that should be included in a web page. A potentially useful feature of WebSockets is that the server can also send a message to the browser without the browser having to explicitly request it. An example of this is when the server sends the browser the current time every minute. All the browser needs to do in this instance is to know what to do with the current time information.

The key difference between the two protocols is that HTTP requires the browser to request a resource, while WebSockets is simply a means to send a message without any explicit action being applied. In other words, HTTP is analogous to sending messages in the mail whereas WebSockets is analogous to communicating via telephone.

1.2.1 Server-side Software

There are several options available to us for enabling R to act as a web server. Some of these options directly expose the HTTP or WebSockets protocol from R, while others allow this to occur indirectly by acting as an intermediary between R and a web browser. We will begin by listing the possible options for generating R content for use within a web browser.

Rserve The Rserve package (Urbanek, 2003) allows R to act as a background service (potentially on a remote machine) that is able to respond to requests over the TCP/IP protocols. This means that it is not directly able to be used by a web

browser. However, most programming environments can communicate to this service because of the ubiquity of TCP/IP as a method of networking computers. This means that any programming environment capable of communicating over TCP/IP that is also able to build a web application can expose R to a web browser. Many programming languages have a package or library that provides a convenient framework for this to occur. Unsurprisingly, a web application framework for R exists called FastRWeb (Urbanek and Horner, 2013). Similar implementations are available in the following languages: Python (Heinkel, 2013), JavaScript (via node.js) (Love, 2013), (Santini, 2013), Java (Urbanek, 2013), C++ (Urbanek, 2003), and PHP (Turbelin, 2013).

rApache The Apache web server is a popular HTTP server that is present on many servers. It is able to be extended by adding *modules* which functionality additional to the features provided by the base installation. rApache (Horner, 2012) extends Apache by including R as a module that is capable of responding to HTTP requests that have been received by Apache. To make this easier, convenience functions have been included to make it easier to develop R web applications with rApache.

Rook The Rook package (Horner, 2013) builds upon R's built-in web server to provide a framework for building web applications. This means that installing Rook only requires installing the package from CRAN, unlike Rserve and rApache which are more complicated to install.

websockets An implementation of the WebSockets protocol for R. The websockets package (Lewis and Horner, 2012) provides little more than the means for opening up communication between R and a web browser. Callback functions can be assigned to events such as receiving a message, establishing a connection, and closing a connection. These event handling functions must be created and managed by a user in both R and a web browser.

shiny The shiny package (RStudio, 2013) provides a collection of functions for creating *reactive* web applications. This builds upon the websockets package to create communication between R and a web browser. This avoids the need to write R and JavaScript code that handles behaviour relating to events, assuming that a web application has been constructed using shiny's "reactive" functions. A feature of shiny is that it is *extensible*, allowing a user to augment the existing functionality.

All of the aforementioned examples are capable of generating R graphics in response to requests from a web browser. It must be mentioned that this list is by no means

exhaustive, but does mention the most popular options for generating web content dynamically from R.

1.2.2 Client-side Software

We know that it is possible to generate R graphics from a web server, but there are alternative options for displaying plots in a web browser. We do not necessarily require that statistical graphics are entirely generated in R. The options for statistical graphics in the browser are shown below:

R Static Graphics This is the most common option where a static image, like a PNG or a plain SVG image is sent to a web browser and remains unchanged. All of the aforementioned R web server options provide some way of easily creating these types of graphics. The main reason for this is because they correspond directly with a built-in R graphics device, whose output can can be sent immediately to a web browser.

Animated and Interactive Graphics As discussed earlier in this chapter, there are animated and interactive graphics that can be produced by packages such as animation, SVGAnnotation and gridSVG which can be delivered to a web browser.

JavaScript Graphics There are several new R packages that have emerged recently. These packages have taken an alternative approach where instead of sending a complete image to a web browser for viewing, R is used to export a *description* of a plot which can then be rendered in a web browser by a JavaScript library. Examples of these packages are rHighcharts (Reinholdsson, 2013a), rVega (Reinholdsson, 2013b), rCharts (Vaidyanathan and Reinholdsson, 2013), and rNVD3 (Vaidyanathan, 2013). The advantage of this approach is that several JavaScript libraries give you animated and interactive graphics very easily. However, this means that the drawn graphics are not R graphics, so we lose the facilities present in the R graphics system. Another alternative is to use R to send data to the browser, where JavaScript code describes and draws the desired plot. Performing both the description and rendering of graphics in a browser may be desirable if a user is fluent in JavaScript but the trade-off is that the benefits of the R graphics system are lost.

JavaScript generated graphics is an area of tremendous growth. Each JavaScript library aims to provide something unique, which other libraries lack. However, a common theme among all libraries is that it is difficult to customise the plots as much as we are used to with R's base or grid graphics. This is because most

libraries tend to provide a high-level function like barchart(), but do not easily expose primitives such as line drawing. Unlike grid, the structure of the generated plots is not always clear and there is no concept of a display list. Consequently, it is difficult to modify a plot generated by a JavaScript library.

A library that has generated a lot of interest is D3. It differs from most plotting libraries because it is a high-level interface for modifying and generating HTML and SVG content. Therefore to generate a plot using D3 requires knowing SVG and building up a plot using the existing SVG elements. Many libraries have recognised the power of modifying and generating SVG graphics using D3 and have used it to create high-level plotting functions (e.g. to create a barchart, linechart, etc).

An option that has not yet been discussed is RFirefox (Becker and Temple Lang, 2013). RFirefox embeds R within the Mozilla Firefox web browser as an extension. It allows R code to be executed in a web page in much the same manner as JavaScript, including the ability to draw R plots. We omit RFirefox from any consideration because we are unable to serve the R generated content over a network, nor is it easy to build upon existing tools to extend its behaviour in order to generate new types of graphics.

1.3 Summary

In order to create the types of graphics we desire, we can use any R server option available to us. For the purposes of this thesis it is assumed that the Rook package is used. The primary reason for this is its ease of use and installation. However, it must be made clear that *all* of the web server options are capable of creating gridSVG graphics and indeed pieces of gridSVG graphics.

We intend for gridSVG to generate these plots and the pieces of a plot. This is because it is the only package for R that is able to not only create animated and interactive plots but also to produce identifiable pieces of a plot. The second point is crucial because without the ability to identify a piece of the plot it is very difficult to modify a plot. Therefore because we can identify a piece of the plot gridSVG is suitable for creating the type of dynamically generated graphical content that we want.

The only extra piece of software we need is a JavaScript library to manipulate an image to add, modify and remove pieces of a plot. The library that has been chosen is D3. This is because it is not strictly focused on creating plots which is largely unnecessary because gridSVG will perform that task instead. D3's focus on working with SVG directly is exactly what is necessary when SVG content has already been provided by gridSVG.

Additionally D3 also provides a powerful and performant interface for selecting pieces of a plot and transitioning content to new states via CSS transitions (W3C CSS Working Group, 2013).

To summarise this introduction, the aim of this research was to develop gridSVG to be a bridge between the computational and graphical facilities provided by R and the interactivity that D3 provides. gridSVG was not able to accomplish this in the past, but it represented the best opportunity for further development, and the rest of this thesis documents how this goal has been achieved.

2 Generating Unique Names

The key advantage of gridSVG is that we can identify its output. This was previously performed by translating the name of the grid object to an `id` attribute on the generated SVG element. The fact that SVG is an XML-based image format means that if we are to identify SVG output by name, we are required to produce SVG `id` attributes that are unique. The previous implementation was not guaranteed to produce unique `id` attributes. This chapter describes how gridSVG retains the names associated with grobs and viewports, along with the difficulties in doing so.

2.1 Name Translation

When gridSVG exports the grid display list, it attempts to give SVG `id` attributes the same value as the name associated with a grob or viewport. However, the fact we require a unique `id` presents us with problems in maintaining these names for a few reasons, which will be discussed later. For now, we will first look at an image drawn in grid and what gridSVG produces from that grid scene.

A simple image will be drawn where we have two viewports and a circle is then drawn inside those viewports. The code to produce that image and the display list that grid records for the image are shown in Figure 2.1.

```
R> pushViewport(viewport(name = "a"))
R> pushViewport(viewport(name = "b", width = 0.5, height = 0.5))
R> grid.circle(name = "a", gp = gpar(fill = "steelblue"))
R> grid.ls(viewports = TRUE, fullNames = TRUE)
viewport[ROOT]
  viewport[a]
    viewport[b]
      circle[a]
```

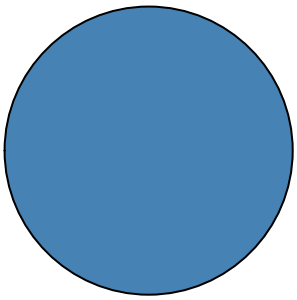


Figure 2.1: A grid image with viewports and a single grob.

What we can see in Figure 2.1 is that there are three viewports, named ROOT, a, and b, one of which shares its name with a circle called a. The ROOT viewport is a viewport that grid creates by default that corresponds to the entire drawing canvas. This explains why ROOT exists in the display list without explicitly being created.

Ideally we would like to see that grid's viewport and grob names are mapped directly to SVG id attributes. However, because we are constrained to having our SVG element id attributes being unique, gridSVG must take action to ensure this is the case. Before explaining how gridSVG does this, let us first consider the simple example we just created by examining the relevant output from gridSVG.

```
<g id="a.1">
  <g id="a::b.1">
    <g id="a.2">
      <circle id="a.2.1" cx="252" cy="252" r="126" fill="rgb(70,13...
    </g>
  </g>
</g>
```

We see here that *none* of the names we have in grid are mapped directly to SVG id attributes. The grid names are still being retained, albeit modified from the original names. The following name translations occurred:

- The viewport called `a` became an SVG `<g>` element whose `id` attribute is `a.1`.
- The viewport called `b` became an SVG `g` element whose `id` attribute is `a::b.1`.
- The circle grob called `a` became an SVG `<g>` element with an `id` attribute of `a.2`. This `g` element has a child `<circle>` element whose `id` attribute is `a.2.1`.

This name translation is clearly evident. How gridSVG performs name translation will now be discussed.

2.2 Paths

In grid, both grobs and viewports can be constructed as a tree of viewports or a tree of grobs. To find a particular viewport or a grob within a tree, we need to use a path. This path is an ordered list of names, specifying parent-child relations. We will be focusing on viewport paths for simplicity, but the same principle applies to trees of graphics objects. An example of a viewport path is shown below:

```
R> vpPath("first", "second", "third")
first::second::third
```

This viewport path describes that we first visit the viewport called `first`, followed by its child, `second`. Once in the `second` viewport, we then traverse to its child viewport `third`. We can see that the resulting path is simply a double-colon separated string of names.

It is possible to create a path where not all names in the path are unique.

```
R> # Creating viewports, note vp1 and vp3 have the same name
R> vp1 <- viewport(name = "a")
R> vp2 <- viewport(name = "b")
R> vp3 <- viewport(name = "a", width = 0.5)
R> # Creating a tree of viewports
R> vpT <- vpTree(vp1, vpList(vpTree(vp2, vpList(vp3))))
R> print(vpT)
viewport[a]->(viewport[b]->(viewport[a]))
R> # Pushing into the tree
R> pushViewport(vpT)
R> # Showing our current viewport path
R> current.vpPath()
```

```
a::b::a
```

In this example we create a viewport tree and push into it. We then observe our current viewport path to be `a::b::a`. Despite there being two `a` viewports in the path, they are each in fact two completely different viewports. As a result, we cannot simply assign the name of each viewport in the path to SVG output because the `id` attribute may not be unique. In our simple example, if we were to do this, we would end up with two SVG elements named `a`. The relevant output showing the result of this is shown below:

```
<g id="a">
  <g id="b">
    <g id="a"/>
  </g>
</g>
```

This demonstrates that names alone are not sufficient for the requirement of unique `id` attributes. A potential solution is to use the *path* as the name of the element. The path would avoid repeating `a` in `id` attributes. This would produce output like the following:

```
<g id="a">
  <g id="a::b">
    <g id="a::b::a"/>
  </g>
</g>
```

This looks like an adequate solution as we have produced unique `id` attributes, despite having viewports with the same names. However, because viewports can be moved in and out of at any point, we cannot guarantee that the viewport tree is fixed while the plot is being drawn. Consider the following:

```
R> pushViewport(vpT)
R> current.vpPath()
a::b::a
R> upViewport()
R> current.vpPath()
a::b
R> pushViewport(viewport(name = "a", height = 0.1))
R> current.vpPath()
```

```
a::b::a
```

What is happening here is that we first push into our tree but then navigate back to the previous viewport path of `a::b`. A new viewport is then created called `a`, and we push into that viewport instead of the viewport that we were previously in. This creates an ambiguity because we have two different viewports that have been pushed into at the same path of `a::b::a`.

```
<g id="a">
  <g id="a::b">
    <g id="a::b::a"/>
    <g id="a::b::a"/>
  </g>
</g>
```

We can see here that despite using paths, they are not sufficient for uniqueness when generating an SVG `id` attribute. This problem is also present when revisiting the same viewports in a viewport path. To overcome this problem, we use an integer suffix that is incremented each time we encounter the same path. To ensure consistency, this integer suffix is applied to every path. The result is shown below:

```
<g id="a.1">
  <g id="a::b.1">
    <g id="a::b::a.1"/>
    <g id="a::b::a.2"/>
  </g>
</g>
```

What we can see here is that we visit the top `a` for the first time. We then traverse to the viewport path `a::b` for the first time. It is important to note however that we can see that we have traversed to `a::b::a` on two separate occasions.

By keeping track of viewport and grob paths we can ensure that their SVG `id` attributes are unique. In addition, their uniqueness allows us to easily retain viewport coordinate information (see chapter 4), because the coordinate information will be paired with the SVG `id` that was generated by gridSVG. This is necessary because each time a viewport path is visited there may be a different coordinate system in use. For example, consider the case where we create two different viewports that share the same name but use different coordinate systems:

```
R> a1 <- viewport(width = 0.5, height = 0.5, name = "a")
R> a2 <- viewport(width = 0.1, height = 0.1, name = "a")
R> pushViewport(a1)
R> current.vpPath()
a
R> popViewport()
R> pushViewport(a2)
R> current.vpPath()
a
R> popViewport()
```

Firstly, two viewports with the name `a` have been created. These two viewports have a different width and height to one another. This is important because each time the viewports are used the viewport path is the same despite different coordinate systems being used. The `id` attributes that gridSVG will generate in this situation should now be familiar:

```
<g id="a.1">
<g id="a.2"/>
```

The unique `id` attributes of `a.1` and `a.2` allow us to look up coordinate information based on these generated names. To see this in action, we only need to look at the relevant subset of coordinate information that has been exported to JSON (Crockford, 2006), a structured data format.

```
{
    "a.1" : {
        "x" : 54,
        "y" : 54,
        "width" : 108,
        "height" : 108,
        "xscale" : [
            0,
            1
        ],
        "yscale" : [
            0,
            1
```

```
        ],
        "inch" : 72
    },
    "a.2" : {
        "x" : 97.2,
        "y" : 97.2,
        "width" : 21.6,
        "height" : 21.6,
        "xscale" : [
            0,
            1
        ],
        "yscale" : [
            0,
            1
        ],
        "inch" : 72
    }
}
```

This clearly illustrates that the viewport coordinate systems for both viewports have been retained. We know this is the case because the `x`, `y`, `width` and `height` attributes for the viewports are indeed different. By generating unique `id` attributes for viewport paths, we can guarantee that coordinate information is not only retained, but is also unambiguous.

2.3 Name Sharing

In grid, both viewports and grobs contain names. Indeed, we have seen they can also be referred to by a path. One problem that gridSVG has that grid isn't concerned with is that viewports and grobs can have the same name. Consider the following example:

```
R> pushViewport(viewport(name = "a"))
R> grid.circle(name = "a")
R> grid.ls(viewports = TRUE, fullNames = TRUE)
viewport[ROOT]
  viewport[a]
```

```
    circle[a]
```

We can see that a viewport has a name that is the same as a circle grob's name. grid is able to draw this scene without any issues but we are presented with a problem when exporting it using gridSVG. We want the SVG `id` attribute to be assigned with the name of the object that we are representing. However, in this example, because this is the first time each grob path and each viewport path is encountered, we can end up with non-unique elements. This is shown below:

```
<g id="a.1">
  <g id="a.1">
    <circle cx="108" cy="108" r="108"/>
  </g>
</g>
```

The reason why both the viewport and circle grob are given the suffix of `.1` is because it is both the first time that the viewport path has been visited and it is the first time that the grob path has been visited. This presents us with a case where `id` attributes between grobs and viewports are shared. To correct this, we not only need to track paths, but we also need to track the names and how often they have been assigned to both viewports *and* grobs. The solution currently used by gridSVG is shown below:

```
<g id="a.1">
  <g id="a.2">
    <circle cx="108" cy="108" r="108"/>
  </g>
</g>
```

Now instead of just tracking how often each viewport or grob path has been used, we track how often each grid name has been used. This is shown in our output because when the circle grob is drawn, it is the second time that the name `a` has been encountered, so we end up with a suffix of `.2`.

In summary, by tracking the names that we attempt to apply to `id` attributes, we ensure that unique `id` attributes are generated by gridSVG by adding an integer suffix.

2.4 Sub-grobs

We have already seen that when a grob is drawn, we create an SVG <g> element. The contents of this grouping element are graphical elements (e.g. rectangles, circles, lines)

that are drawn to an SVG canvas. The reason why grouping output is necessary is because there are cases where gridSVG cannot create a one-to-one mapping between a grid grob and SVG output. For example, while it is possible for a single grid circle to be drawn simply as an SVG `<circle>` element, we cannot assume this to always be true. We can draw multiple circles using a single call to `grid.circle()`. An example of this is shown in Figure 2.2.

```
R> grid.circle(r = 1:3 / 10, name = "a")
```

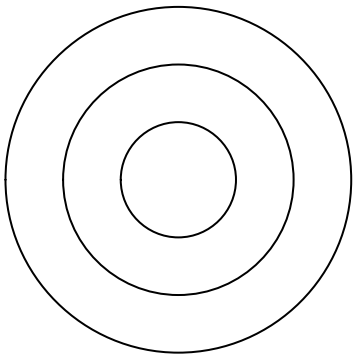


Figure 2.2: A single circle grob that draws three circles.

```
R> grid.ls()
a
```

A *single* circle grob (as listed on the display list) has managed to draw three separate circles. These will be referred to as sub-grobs. It is clear that we cannot apply any name given to the grob to all of its sub-grobs because all sub-grobs would therefore have identical names. The solution gridSVG applies is to use an integer suffix to identify each sub-grob that is drawn. Using the example above, we will take a look at the SVG output that gridSVG produces.

```
<g id="a.1">
  <circle id="a.1.1" cx="108" cy="108" r="21.6"/>
  <circle id="a.1.2" cx="108" cy="108" r="43.2"/>
```

```
  <circle id="a.1.3" cx="108" cy="108" r="64.8"/>
</g>
```

Firstly, we can see that the original grob name has been changed to `a.1` because it is the first time that we use the name `a`. However, its children also have an integer suffix applied. The first circle drawn (i.e. the circle with a radius of 0.1) is given the name `a.1.1`. The second and third circles are assigned the names `a.1.2` and `a.1.3` respectively.

This technique also applies to grobs where there is an `id` parameter. An example of such a grob is a `polylineGrob`.

```
R> grid.polyline(x = c(0:4 / 10, rep(.5, 5), 10:6 / 10, rep(.5, 5)),
R+               y = c(rep(.5, 5), 10:6 / 10, rep(.5, 5), 0:4 / 10),
R+               id = rep(1:5, 4),
R+               gp = gpar(col = 1:5, lwd = 3),
R+               name = "a")
```

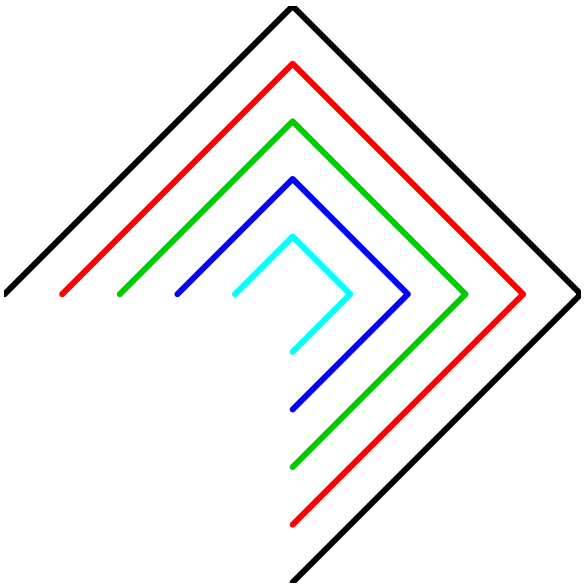


Figure 2.3: A single polyline grob drawing 5 separate lines.

Figure 2.3 shows that a single call to `grid.polyline()` has produced 5 distinct lines. This is because of `grid.polyline()`'s `id` parameter which determines the sub-grob that each line coordinate belongs to. The SVG output is shown below, and demonstrates that the same rule applies to grobs with vectorised parameters and to those with an `id` parameter.

```
<g id="a.1">
  <polyline id="a.1.1" ... />
  <polyline id="a.1.2" ... />
  <polyline id="a.1.3" ... />
  <polyline id="a.1.4" ... />
  <polyline id="a.1.5" ... />
</g>
```

The addition of an integer suffix to sub-grobs allows us to not only generate unique `id`s for SVG elements, but also allows us to identify each sub-grob that is being drawn in a consistent manner.

2.5 Controlling Output

This article has shown why gridSVG needs to modify names to produce unique output. One of the problems in doing this is that the SVG `id` attributes are now much harder to predict. This means that any name that is assigned to a grob or viewport in grid, when exported to SVG by gridSVG, does not map to an easily predictable SVG `id` attribute. However, to aid predictability, gridSVG does offer some options for controlling how it constructs `id` attributes.

2.5.1 Paths

It was discussed earlier why viewport paths are used as part of the exported `id`s. However, there are cases where this unnecessarily complicates the SVG output. Primarily this is the case when the names of each viewport — and therefore every viewport path — are unique. Using viewport paths as part of the generated `id` attributes is therefore not strictly necessary. We add the complication of dealing with paths when our viewport names are sufficiently specific.

The `usePaths` parameter for gridSVG's `grid.export()` function allows us to determine whether paths are used when creating `id`s for grobs and viewports. There are four possible options:

`vpPaths` Use paths in SVG `id`s for viewport paths. This is the default behaviour because it makes coordinate system exporting clearer as there are likely to be fewer name conflicts.

`gPaths` Use paths in SVG `id`s for grob paths.

`none` Do not use paths for either viewport paths or grob paths.

`both` Generate SVG `id`s with paths for both viewport paths and grob paths.

To demonstrate the effect of these options, a simple image will be drawn, then we will examine the relevant SVG output that gridSVG generates from each option.

```
R> # Create viewports and grobs
R> vpa <- viewport(name = "a")
R> vpb <- viewport(name = "b", width = 0.5, height = 0.5)
R> rectc <- rectGrob(name = "c")
R> circd <- circleGrob(name = "d")
R> # Construct trees of the viewports and grobs
R> vpt <- vpTree(vpa, children = vpList(vpb))
R> gt <- gTree(children = gList(rectc, circd), name = "gt")
R> # Draw the image
R> pushViewport(vpt)
R> grid.draw(gt)
```

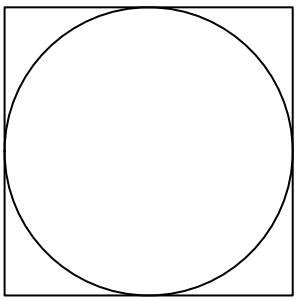


Figure 2.4: A simple grid image featuring a grob tree and a tree of viewports.

```
R> # Examine what grid sees
R> grid.ls(viewports = TRUE, fullNames = TRUE)
viewport[ROOT]
```

```
  viewport[a]
    viewport[b]
      gTree[gt]
        rect[c]
        circle[d]
```

What has been drawn in Figure 2.4 are two grobs, a circle and a rectangle. They are the only children in a single tree of grobs called `gt`. This tree has been drawn inside the viewport path `a::b`. Because we have trees of content, we can easily compare the effect of each option. We will first look at the output when we only want viewport paths to be used.

```
R> grid.export(usePaths = "vpPaths")
<g id="a.1">
  <g id="a::b.1">
    <g id="gt.1">
      <g id="c.1">
        <rect id="c.1.1" x="54" y="54" width="108" height="108"/>
      </g>
      <g id="d.1">
        <circle id="d.1.1" cx="108" cy="108" r="54"/>
      </g>
    </g>
  </g>
</g>
```

Only viewport paths are being used here. As a result each of the grob names are kept unchanged and instead of viewport names we use the viewport path. We know this is the case because there is an `id` that has been exported that can only belong to a viewport path, `a::b.1`. In addition, both the `c` and `d` grobs do not use paths for their names so they are exported as `c.1` and `d.1` respectively.

```
R> grid.export(usePaths = "gPaths")
<g id="a.1">
  <g id="b.1">
    <g id="gt.1">
      <g id="gt.1::c.1">
        <rect id="gt.1::c.1.1" x="54" y="54" width="108" height="...
```

```
      </g>
      <g id="gt.1::d.1">
        <circle id="gt.1::d.1.1" cx="108" cy="108" r="54"/>
      </g>
    </g>
  </g>
</g>
```

The viewports are now being retained as just being the names, while we are now using paths for grobs. The viewport path `a::b` is consequently exported simply to `b.1`, which can only be the case if we ignore the path prefix of `a`. The grob path is being used in particular with the rectangle and circle grobs because they are now being exported with the `id`s of `gt.1::c.1` and `gt.1::d.1` respectively. This clearly indicates that they are children of the `gTree` named `gt`.

```
R> grid.export(usePaths = "none")
<g id="a.1">
  <g id="b.1">
    <g id="gt.1">
      <g id="c.1">
        <rect id="c.1.1" x="54" y="54" width="108" height="108"/>
      </g>
      <g id="d.1">
        <circle id="d.1.1" cx="108" cy="108" r="54"/>
      </g>
    </g>
  </g>
</g>
```

No paths are being used so we are only exporting the names of the viewports and grobs. This is particularly evident because the default path separator of `::` is no longer present in any of our `id` attributes.

```
R> grid.export(usePaths = "both")
<g id="a.1">
  <g id="a::b.1">
    <g id="gt.1">
      <g id="gt.1::c.1">
```

```
        <rect id="gt.1::c.1.1" x="54" y="54" width="108" height="...
      </g>
      <g id="gt.1::d.1">
        <circle id="gt.1::d.1.1" cx="108" cy="108" r="54"/>
      </g>
    </g>
  </g>
</g>
```

Finally, we observe the output created by exporting ids as both viewport paths and grob paths.

2.5.2 Custom Separators

When gridSVG exports paths as SVG ids, the result is that each name in the path is separated by ::. This is the default path separator used by grid. However, there may be situations where a custom path separator may be more appropriate. An example where this is the case is when using ids within CSS selectors (W3C CSS Working Group, 2011). This is because the colon character is a special character in CSS (CSS Working Group, 2011), as it prefixes a pseudo-selector. Therefore, if we were to use the default gridSVG path separator, we would need to *escape* it for use within a CSS selector. This would require modifying each instance of :: and replacing it with \:\:. Ideally we would like to avoid performing any escaping by using a different separator. This section discusses how custom separators can be used by gridSVG when it exports an SVG image.

There are three types of separators that gridSVG uses:

vpPath The separator used between names in a viewport path. The default value is ::.

gPath The separator used between names in a grob path. The default value is ::.

id The separator between the name given to a grob or viewport and an additional integer suffix, used for the purposes of ensuring uniqueness. By default this value is ".". This also applies to sub-grobs and additional SVG output like clipping paths and marker names.

We can change the values of these separators, avoiding the need to escape them for use within CSS selectors. Another possible reason why using custom separators might be useful is if we have grob names containing . characters. By changing the id separator,

we can make it easier to determine the grob or viewport name from the generated SVG `id` attribute.

gridSVG provides three functions that are useful for the purposes of changing the separators used when generating SVG `id` attributes: `setSVGoptions()`, `getSVGoptions()`, and `getSVGoption()`. `setSVGoptions()` allows us to change the separators, while `getSVGoptions()` allows us to query gridSVG for all current separators. `getSVGoption()` is a convenience function that gives us the value of a single separator. Example usage is shown below:

```
R> # See what the current id separator is
R> getSVGoption("id.sep")
[1] "."
R> # Now let's see all of them
R> getSVGoptions()
$id.sep
[1] "."

$gPath.sep
[1] "::"

$vpPath.sep
[1] "::"
R> # Set new separators
R> setSVGoptions(vpPath.sep = "_",
R+               gPath.sep = "_",
R+               id.sep = "-")
```

Now that we have changed the separators, we can examine the effect of these changes by drawing our earlier example again with `usePaths` being set to `both`. The relevant output is shown below:

```
R> pushViewport(vpt)
R> grid.draw(gt)
R> grid.export(usePaths = "both")
<g id="a.1">
  <g id="a::b.1">
    <g id="gt.1">
      <g id="gt.1::c.1">
```

```
        <rect id="gt.1::c.1.1" x="54" y="54" width="108" height="...
      </g>
      <g id="gt.1::d.1">
        <circle id="gt.1::d.1.1" cx="108" cy="108" r="54"/>
      </g>
    </g>
  </g>
</g>
```

Notice how the each of the grob and viewport paths now have underscore characters in them. Additionally, every `id` now has a dash as a separator to the integer suffix.

2.5.3 Unique Names

By default, to ensure valid SVG content, gridSVG adds an integer suffix for the purposes of making the generated `id` attribute unique. A consequence of this is that there is not a one-to-one mapping between grid names and SVG ids. This makes it hard to predict the SVG `id` that is generated for a grob or viewport, presenting challenges when we want to use the SVG output. For example, in JavaScript, if we want to change the colour of a grob as we hover our mouse over it, we first need to know the `id` of the SVG element that we are applying this effect to.

If a grid plot has been drawn that is known to have unique grob and viewport names, this procedure of adding an integer suffix is not required. gridSVG provides an option for enabling this process, `uniqueNames`, which is `TRUE` by default. In the case when this parameter is `FALSE` it is possible to produce valid SVG without the addition of any integer suffixes. This means that we can create a one-to-one mapping between grid grob names and the `id` attributes that gridSVG generates. This parameter only affects grob names because modifying viewport names could affect retention of coordinate information. A simple demonstration of the effect of `uniqueNames` is shown below:

```
R> grid.circle(name = "circle")
R> grid.ls()
circle
R> grid.export(uniqueNames = FALSE)
<g id="circle">
  <circle id="circle.1" cx="108" cy="108" r="108"/>
</g>
```

We can see that the id generated for the grob named circle is still circle. One important thing to note is that gridSVG does not change its behaviour for sub-grobs. This is why the <circle> element has an id of circle.1.

When the uniqueNames argument is set to FALSE, it is possible to generate invalid SVG. This may occur when grobs and/or viewports share names when exported to SVG. gridSVG will generate non-unique names, but it will provide a warning in this case because invalid SVG is being produced. See the following:

```
R> # Giving a rect, and a viewport the same "name" when exported
R> pushViewport(viewport(name = "a"))
R> grid.rect(name = "a.1")
R> grid.ls(viewports = TRUE, fullNames = TRUE)
viewport[ROOT]
  viewport[a]
    rect[a.1]
R> grid.export(uniqueNames = FALSE)
Warning: Not all element IDs are unique.
Consider running 'grid.export' with 'uniqueNames = TRUE'.
<g id="a.1">
  <g id="a.1">
    <rect id="a.1.1" x="0" y="0" width="216" height="216"/>
  </g>
</g>
```

In this example, gridSVG is not checking whether the id a.1 already exists. The viewport is given the expected gridSVG name of a.1 because it is the first time that the a viewport path has been pushed into. Now when we come across a grob called a.1, no checking is occurring to see whether the id already exists. Additionally, because uniqueNames is set to FALSE, no integer suffix is added for the purpose of ensuring uniqueness. Therefore we end up with two id attributes that are the same, creating invalid SVG, for which gridSVG is providing a warning message.

Careful consideration should be made when using this parameter because it is the only parameter that has the potential to produce invalid SVG documents. In fact, the need to change this parameter from the default of TRUE is rarely necessary when mapping information is used.

2.5.4 Prefixes

It has been shown earlier in this chapter that gridSVG is now able to generate documents with unique `id` attributes. This fulfils the requirements for producing valid SVG documents. When each image is embedded in a web page using the `<img>` tag, or viewed standalone in other software, these gridSVG images work well. However, when gridSVG images are inserted inline into HTML, or inserted into other XML or SVG documents, the uniqueness of `id` attributes cannot be guaranteed. This is because the documents that a gridSVG image is inserted into may already have the same `id` attributes as those present in the gridSVG image. The problem will be demonstrated with two simple images (see Figure 2.5 and Figure 2.6). The grid images will be exported to SVG and inserted into an HTML document inline.

```
R> grid.newpage()
R> grid.circle(name = "mygrob")
R> firstImage <- grid.export("")$svg
```

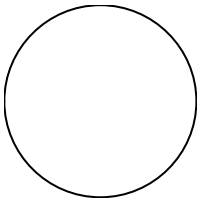


Figure 2.5: A grid image with a circle grob named "mygrob".

```
R> grid.newpage()
R> grid.rect(name = "mygrob")
R> secondImage <- grid.export("")$svg
```

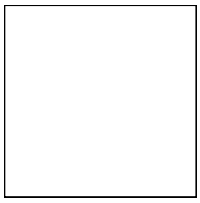


Figure 2.6: A grid image with a rect grob named "mygrob".

When we combine the content of these two images into a single HTML document, their `id` attributes will no longer be unique. The relevant subset of the document is shown below:

```
R> cat(paste0("<html><body>",
R+            saveXML(firstImage, file = NULL),
R+            saveXML(secondImage, file = NULL),
R+            "</body></html>"))
<html>
  <body>
    <svg ... >
      ...
      <g id="gridSVG" ... >
        <g id="mygrob.1">
          <circle id="mygrob.1.1" cx="48" cy="48.01" r="48"/>
        </g>
      </g>
      ...
    </svg>
    <svg ... >
      <g id="gridSVG" ... >
        <g id="mygrob.1">
          <rect id="mygrob.1.1" x="0" y="0"
                width="96" height="96.02"/>
        </g>
      </g>
      ...
    </svg>
  </body>
</html>
```

The HTML document clearly shows that *all* of the `id` attributes collide. In other words there are two instances of `gridSVG`, `mygrob.1` and `mygrob.1.1` when they are required to be unique.

gridSVG cannot be expected to know the `id` attributes present outside the SVG images it generates, but what it can do is apply a prefix to all `id` attributes. This prefix provides a way of avoiding any cross-document `id` collisions. For example, if we applied a prefix of "first-" to the first image, then the generated grob `id` would be `first-mygrob.1` instead of `mygrob.1`. The `grid.export()` function has a `prefix` argument that is set to `""` by default, but changing it allows us to prefix all `id` attributes with a character string of

our choice.

To demonstrate the `prefix` parameter, we will revisit the examples shown in Figure 2.5 and Figure 2.6 to produce an HTML document with unique `id` attributes. The images themselves will not be shown but we will add the prefix "first-" to the first image and "second-" to the second image.

```
R> grid.newpage()
R> grid.circle(name = "mygrob")
R> firstImage <- grid.export("", prefix = "first-")$svg
R> grid.newpage()
R> grid.rect(name = "mygrob")
R> secondImage <- grid.export("", prefix = "second-")$svg
```

These images will now be inserted inline into an HTML document, but only a relevant subset of the output will be shown.

```
<html>
  <body>
    <svg id="first-" ... >
      ...
      <g id="first-gridSVG" ... >
        <g id="first-mygrob.1">
          <circle id="first-mygrob.1.1"
                  cx="48" cy="48.01" r="48"/>
        </g>
      </g>
      ...
    </svg>
    <svg id="second-" ... >
      <g id="second-gridSVG" ... >
        <g id="second-mygrob.1">
          <rect id="second-mygrob.1.1" x="0" y="0"
                width="96" height="96.02"/>
        </g>
      </g>
      ...
    </svg>
```

```
  </body>
</html>
```

The output shows us that `all` of the `id` attributes are now unique. Furthermore, an `id` has been given to the `<svg>` elements that holds the value of the associated prefix. This makes it easy to find a particular SVG image in a web page because it is simply the prefix that we used to generate it.

The advantages of using prefixing have been discussed and although they are particularly useful for including gridSVG images inline in HTML, there are other applications for prefixes. Another example would be that we can combine pieces of two gridSVG-generated SVG images in a single SVG image, while ensuring uniqueness amongst the `id` attributes.

2.5.5 Classes

The majority of this chapter is concerned with the `id` attributes applied to SVG elements. This is useful for identifying individual components of a gridSVG plot but it does not assist with identifying *groups* of graphical components unless a reasonable naming scheme has been applied to the grid scene. SVG allows a `class` attribute to be added to any element that determines membership within a whitespace-separated list of groups. Fortunately, all grid grobs and viewports are S3 objects, which means that they possess a character vector of class names. We can use these class names to add a `class` attribute to the SVG elements that gridSVG generates. The classes that an S3 object is a member of can be inspected by calling `class()`. We will demonstrate this by applying `class()` to some grid objects.

```
R> class(rectGrob())
[1] "rect"  "grob"  "gDesc"
R> class(circleGrob())
[1] "circle" "grob"   "gDesc"
R> class(xaxisGrob())
[1] "xaxis" "axis"  "gTree" "grob"  "gDesc"
R> class(yaxisGrob())
[1] "yaxis" "axis"  "gTree" "grob"  "gDesc"
R> class(viewport())
[1] "viewport"
```

The output from the `class()` function shows us that all of the grid grobs are members of the `grob` class. Furthermore, both x and y axis grobs are members of the `axis` class.

This shows us that if we can use these classes in SVG then we have a way of selecting groups of related SVG content. gridSVG caters for this with the `addClasses` parameter for `grid.export()`. When this parameter is `TRUE`, the class of a grid object is transferred to its associated SVG output. We will demonstrate this with a simple image, as shown in Figure 2.7.

```
R> pushViewport(viewport(width = 0.6, height = 0.6,
R+                       name = "wrapper"))
R> grid.rect(name = "border")
R> grid.circle(r = 0.3, name = "circ")
R> grid.xaxis(name = "xaxis")
R> grid.yaxis(name = "yaxis")
R> popViewport()
R> grid.export("image-with-classes.svg", addClasses = TRUE)
```

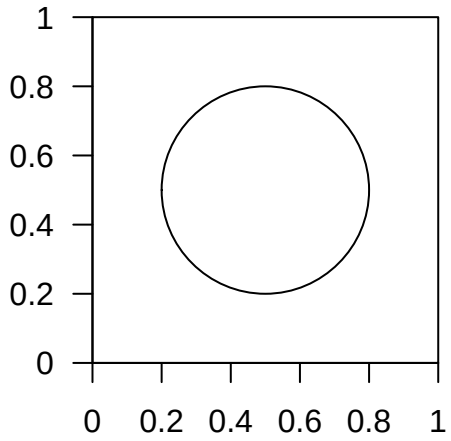


Figure 2.7: A grid image composed of classed objects.

When exporting SVG with `addClasses` set to `TRUE`, the classes of all grid objects in the image are retained as class attributes. A relevant subset of the SVG image is shown below.

```
<g id="wrapper.1" class="pushedvp viewport">
  <g id="border.1" class="rect grob gDesc">
    ...
```

```
  </g>
  <g id="circ.1" class="circle grob gDesc">
    ...
  </g>
  <g id="xaxis.1" class="forcedgrob xaxis axis gTree grob gDesc">
    <g id="major.1" class="lines grob gDesc">
      ...
    </g>
    <g id="ticks.1" class="segments grob gDesc">
      ...
    </g>
    <g id="labels.1" class="text grob gDesc">
      ...
    </g>
  </g>
  <g id="yaxis.1" class="forcedgrob yaxis axis gTree grob gDesc">
    <g id="major.2" class="lines grob gDesc">
      ...
    </g>
    <g id="ticks.2" class="segments grob gDesc">
      ...
    </g>
    <g id="labels.2" class="text grob gDesc">
      ...
    </g>
  </g>
</g>
```

The output clearly shows how the class of a grid grob has been transferred to an SVG `class` attribute. This may be useful for styling classes of SVG content differently. For example, we could insert the following SVG and CSS snippet into the document to change how it appears.

```
<style type="text/css">
  .axis  { stroke: blue;      }
  .yaxis { stroke-width: 5px; }
```

```
</style>
```

This CSS code is designed to make all axes blue, but only the y axes will be thicker with a stroke width of 5 pixels. This is shown in Figure 2.8.

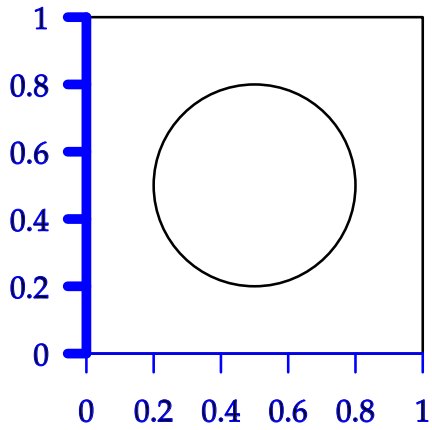


Figure 2.8: A gridSVG image with additional styling performed using CSS classes.

It is not only easy to style content with CSS classes, but also to select and manipulate related SVG nodes in both R and a web browser. This is because we can use CSS selectors to query and collect SVG nodes. These were actually already shown in the CSS snippet described earlier because we were able to make all axes blue by using the selector `.axis`. This same selector can be used to find the SVG nodes in JavaScript using the `querySelector()` and `querySelectorAll()` methods. In R, this can be achieved using the selectr package. More information about CSS selectors will be discussed in chapter 5.

2.5.6 Output Annotation

We have shown how gridSVG can be used to modify its SVG output. These settings can be particularly useful for deriving the structure of a source image. For example, we might like to know whether an `id` attribute is part of a grob path or a viewport path. We can make more sense of an SVG document generated by gridSVG if we know the values of separators used when exporting. This is also particularly useful for debugging a gridSVG image.

Settings used for controlling SVG output are exported as metadata in the SVG that gridSVG creates. To demonstrate this, we will draw a simple grid image (not shown), but show only the SVG metadata that was exported by gridSVG.

```
<metadata xmlns:gridsvg="http://www.stat.auckland.ac.nz/~paul/R/g...
  <gridsvg:generator name="gridSVG" version="1.2-0" time="2013-06...
  <gridsvg:argument name="name" value=""/>
  <gridsvg:argument name="exportCoords" value="none"/>
  <gridsvg:argument name="exportMappings" value="none"/>
  <gridsvg:argument name="exportJS" value="none"/>
  <gridsvg:argument name="res" value="72"/>
  <gridsvg:argument name="prefix" value=""/>
  <gridsvg:argument name="addClasses" value="FALSE"/>
  <gridsvg:argument name="indent" value="TRUE"/>
  <gridsvg:argument name="htmlWrapper" value="FALSE"/>
  <gridsvg:argument name="usePaths" value="vpPaths"/>
  <gridsvg:argument name="uniqueNames" value="TRUE"/>
  <gridsvg:separator name="id.sep" value="."/>
  <gridsvg:separator name="gPath.sep" value="::"/>
  <gridsvg:separator name="vpPath.sep" value="::"/>
</metadata>
```

The metadata shows us exactly how the image was drawn. In particular, the `gridsvg:argument` elements tell us how the `id` attributes were controlled. We can see that as this particular image was exported, it was ensuring that unique names were being used, and the only paths it was generating were for viewport paths. Additionally, we can also see the values of the separators when the image was being exported.

2.6 Mappings

We have discussed the many ways in which gridSVG modifies grob and viewport names, including ways to control how that happens. However, the key issue with this name translation is that it is difficult to predict how to map the names that are used in grid with the output produced by gridSVG. A recent development in gridSVG is the ability to retain mapping information that provides us with information on how to map a grid grob or viewport name to an SVG `id` attribute.

It is useful to have mapping information available both in R, and in JavaScript. In R,

we might want to perform some post-processing on the XML nodes that a grob maps to. If the `id` can be retrieved easily then performing this task is far simpler than writing an XPath expression (W3C XSL and XML Linking Working Groups, 1999). Similarly, if we want to perform some modification on an SVG image in the browser, using tools like D3, then knowing what content we are trying to select is an important problem to solve.

We will first look at the mapping information that gridSVG is exporting. We start with the following image:

```
R> grid.newpage()
R> pushViewport(viewport(name = "a"))
R> grid.rect(name = "b")
R> grid.circle(name = "b")
R> grid.export()
```

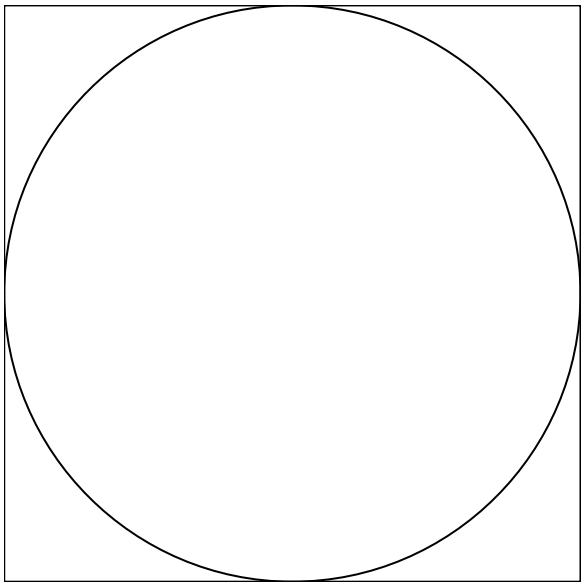


Figure 2.9: A simple grid image that has been exported to SVG.

This exports mapping information as JSON, a structured data format that is convenient for use within a web browser. The mapping information from this plot is shown below:

```
{
    "vps" : {
        "a" : {
            "suffix" : 1,
```

```
            "selector" : "#a\\.1",
            "xpath" : "//*[@id='a.1']"
        }
    },
    "grobs" : {
        "b" : {
            "suffix" : [
                1,
                2
            ],
            "selector" : [
                "#b\\.1",
                "#b\\.2"
            ],
            "xpath" : [
                "//*[@id='b.1']",
                "//*[@id='b.2']"
            ]
        }
    },
    "refs" : null,
    "id.sep" : ".",
    "prefix" : ""
}
```

This is showing that we store both viewport and grob mapping information. Within each category, we store the name of the object, which has three pieces of information associated with it. The first are the integer suffixes that the name has been mapped to. For our example, we used the grob name `b` twice, so there are two suffixes associated with the `b` grob. This can be used to construct an `id` attribute by concatenating the name with the `id.sep` value.

Also included are CSS selectors and XPath expressions which target the same `id`. These are included for convenience, and special characters are already escaped. This means that if we use a JavaScript library like D3 or jQuery (The jQuery Foundation, 2013), we can select the content immediately by just using the exported CSS selector.

In order to make the mapping information easy to use, gridSVG provides conve-

nience functions in both R and JavaScript. The primary function that is used is `getSVGMappings()`, which is named the same in both R and JavaScript. To demonstrate, we will be building upon the mapping information shown earlier.

```
R> mappings <- readMappingsJS("mappings.js")
R> gridSVGMappings(mappings)
R> getSVGMappings("a", "vp") # getting 'a', which is a viewport
[1] "a.1"
R> getSVGMappings("b", "grob") # getting 'b', which are grobs
[1] "b.1" "b.2"
```

The first thing that occurs is that because mapping information is stored in a file as JSON, we need to read it into R. The `readMappingsJS()` function takes the filename containing mapping information, and reads that file into R and parses it as a list. The result can then be given to `gridSVGMappings()`. Once this is done we can apply the mapping information by using `getSVGMappings()`.

It is important to note that when a name is used more than once, instead of getting a single `id` value, we can end up with multiple `id`s. This ambiguity cannot be resolved because of issues discussed earlier, but at least we can reduce the search to only the `id`s that have been returned from the function. Typically there are few instances where multiple results are returned.

The same example above can be performed in a browser using JavaScript, the output is shown below:

```
JS> getSVGMappings("a", "vp");
["a.1"]
JS> getSVGMappings("b", "grob");
["b.1", "b.2"]
```

In this example, it is shown that the function always returns an array of values, even when there is only one matching result. This is for consistency across single and multiple matching results.

To return a CSS selector or XPath expression instead of an `id` we just need to specify that in the optional third parameter. Again, this is the case in both the R and JavaScript implementations of the function. This is shown below:

```
R> getSVGMappings("a", "vp", "selector")
[1] "#a\\.1"
R> getSVGMappings("a", "vp", "xpath")
```

```
[1] "//*[@id='a.1']"
R> getSVGMappings("b", "grob", "selector")
[1] "#b\\.1" "#b\\.2"
R> getSVGMappings("b", "grob", "xpath")
[1] "//*[@id='b.1']" "//*[@id='b.2']"
```

```
JS> getSVGMappings("a", "vp", "selector");
["#a\\.1"]
JS> getSVGMappings("a", "vp", "xpath");
["//*[@id='a.1']"]
JS> getSVGMappings("b", "grob", "selector");
["#b\\.1", "#b\\.2"]
JS> getSVGMappings("a", "vp", "xpath");
["//*[@id='b.1']", "//*[@id='b.2']"]
```

An example where this becomes useful is if you want to use D3 to modify content, perhaps using a transition. All that is required is to get the appropriate selector and D3 can select the appropriate content based on that selector. For example the following shows how this might occur:

```
JS> var sel = getSVGMappings("a", "vp", "selector")[0];
JS> d3.select(sel)
JS+     .transition()
JS+     ...
```

In R, the use of the XML package (Temple Lang and Nolan, 2011) is more familiar, so we can use XPath expressions instead of CSS selectors.

```
R> xp <- getSVGMappings("a", "vp", "xpath")
R> vpa <- getNodeSet(image, xp)[[1]]
R> vpa
<g id="a.1">
  <g id="b.1">
    <rect id="b.1.1" x="0" y="0" width="216" height="216"/>
  </g>
  <g id="b.2">
    <circle id="b.2.1" cx="108" cy="108" r="108"/>
  </g>
```

```
</g>
```

With the development of retaining name mapping information, it is easier to manipulate SVG images that have been exported by gridSVG.

2.7 Conclusion

We have demonstrated that grid grob and viewport names are required to be modified as they are translated to SVG `id` attributes. We have also shown why this is necessary and the process gridSVG takes to ensure valid SVG is generated.

gridSVG also provides two parameters in its `grid.export()` function which affect how modification of grob and viewport names occur. Despite the modification of names, it is straightforward to retrieve possible matching `id`s using convenience functions that access gridSVG's name mapping information.

3 Structured SVG Generation

This chapter describes a new development in gridSVG that changes the mechanism used to convert grid grobs and viewports to an SVG representation. The purpose of this change is to enable the serving of an SVG image in memory, to a web browser, without having to save to a file first. Although this was the primary motivation for the content described in this chapter, further benefits are discussed.

3.1 The Previous Approach

In order to aid our explanation, a simple grid plot will be drawn in Figure 3.1 using the code below.

```
R> library(grid)
R> grid.rect(width = unit(0.5, "npc"),
R+           height = unit(0.5, "npc"),
R+           name = "example-rect")
```

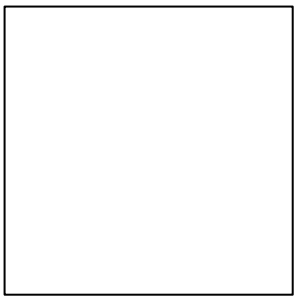


Figure 3.1: A simple grid plot.

The output from `grid.ls()` shows the grid display list. This represents the list of grobs that have been plotted on the current graphics device. The display list shows that the rectangle has been drawn and we can see that it is named `example-rect`. When gridSVG translates `example-rect` into SVG, the rectangle translates into the following markup:

```
<g id="example-rect.1">
  <rect id="example-rect.1.1" x="54" y="54" width="108" height="1...
</g>
```

Prior to the recent development, gridSVG would create this SVG markup by concatenating strings. The first step involved is to create an SVG group (a `<g>` element). This group needs to have all of its appropriate attributes inserted, which always include an `id` attribute, but can also include attributes related to animation, hyperlinking, or custom attributes by "garnishing" attributes. In R, string concatenation is accomplished using the `paste()` function. A fragment of pseudo-code follows, which would generate the SVG group markup.

```
R> paste("<g",
R+       ' id="', groupId, '"',
R+       " ... ",
R+       ">\n", sep = "")
```

```
R> increaseIndent()
```

In this case the ... represents the optional attributes applied to a group, e.g. hyperlinking. We can see already that the code to produce the SVG markup is reasonably complex compared to the markup itself. Note that we have also increased the level of indentation so that children of this group are clearly observed to be children of this particular group.

The next step is to add a child <rect> element to this SVG group. We need to first indent to the correct position on a new line, and then draw the rectangle. The code that would be used to produce the rectangle is shown below.

```
R> paste(indent(),
R+       "<rect",
R+       ' id="', rectId, '"',
R+       ' x="', rectX, '"',
R+       ' y="', rectY, '"',
R+       ' width="', rectWidth, '"',
R+       ' height="', rectHeight, '"',
R+       " ...",
R+       " />\n", sep = "")
<rect id="example-rect.1.1"
      x="126" y="126" width="252" height="252" ... />
```

We can clearly see how attribute values are inserted into the SVG output, in particular with our location and dimension attributes. Again, the ... represents other attributes that may be inserted (though not demonstrated). What is also being shown here is how we are applying the indentation. This is done by calling a function that returns a vector character with the correct number of spaces to indent our <rect> element.

Once all children have been added to the SVG group, we can close the group so that all <rect> elements are contained with it. Because we are closing an element, we need to decrease the level of indentation to preserve the hierarchical structure of the SVG markup. This means when closing any element, we need to do something similar to the following code which closes an SVG group.

```
R> decreaseIndent()
R> paste(indent(),
R+       "</g>\n", sep = "")
</g>
```

We have shown how SVG images are built using a series of concatenated strings. It is important to note that these strings are written directly to a file (specified when calling `grid.export()`). This means each time an SVG fragment is created using `paste()`, it is appended to a specified file.

This approach has a few limitations. For instance, we cannot guarantee that the output that is produced is valid SVG markup. We are also writing directly to a file, which means that we need to read the file to observe its contents; we do not retain the SVG content in resident memory. Finally, but less importantly, performance is a concern when generating output using repeated string concatenation as it is known to be a slow operation (this is less important because the drawing of the original image by grid, before export, is also slow).

To remedy these limitations a rewrite of the markup generating component of gridSVG was undertaken.

3.2 Structured Output with the XML package

The rewrite of part of the gridSVG package was achieved by utilising the XML package. The XML package is an R wrapper for the libxml2 (Veillard, 2012) XML parser and generator. The key feature that the XML package provides us with is a way of representing an SVG image as a collection of SVG nodes (elements), instead of a long character vector. We simply need to define the structure of the document, the XML package will take care of how this will be exported to text.

3.2.1 Image Construction

To define the structure of an SVG image, we need to establish how elements relate to each other. In the case of gridSVG, the only relationship of importance is the parent/child relationship. The earlier example with the rectangle will be recreated using the XML package to demonstrate the differences between the two approaches. The code that creates an SVG group is shown below. Notice that when we print out the node itself, the markup is generated for us by the XML package.

```
R> g <- newXMLNode("g",
R+                 attrs = list(id = "example-rect"))
R> setParentNode(g)
R> g
```

```
<g id="example-rect"/>
```

The group is given the (modified) name of the grob that it is going to be representing. Because we wish to add children to this <g> element, we set it as the current parent node with a call to the setParentNode() function.

The next piece of code creates a <rect> element. It is important to note in this code, that the parent parameter is given the result of the function call getParentNode(). Earlier we set the current parent node to be the <g> element. This means that the <rect> element will be a child of the <g> element.

```
<rect id="example-rect.1.1" x="126" y="126" width="252" height="252"/>
<g id="example-rect">
  <rect id="example-rect.1.1" x="126" y="126" width="252" height="252"/>
</g>
```

We can now see how the document is beginning to build up as the <rect> node is added to the g node.

A complete SVG document must have a "root" <svg> element. This has been left out of the examples so far, but it is worth mentioning here because with the XML approach we include several *namespace* definitions in the <svg> element. This allows the XML package to ensure that we are producing valid SVG output.

```
R> svgRoot <-
R+   newXMLNode("svg", namespaceDefinitions =
R+                list("http://www.w3.org/2000/svg",
R+                     xlink = "http://www.w3.org/1999/xlink"),
R+              attrs = list(width = svgWidth,
R+                           height = svgHeight,
R+                           version = "1.1"))
R> setParentNode(svgRoot)
R> svgRoot
<svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w...
```

This <svg> element has become the parent node so that the <g> element we created earlier can be made a child of the root <svg> element.

```
R> addChildren(svgRoot, g)
```

If we print out the <svg> node now we see the <g> and <rect> elements nested neatly within it.

```
R> svgRoot
<svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w...
  <g id="example-rect">
    <rect id="example-rect.1.1" x="126" y="126" width="252" heigh...
  </g>
</svg>
```

This demonstrates how SVG images can be built up in a more reliable way than with simple string concatenation. It is clear that the way in which we define our SVG image is less prone to error in creating markup, and it also ensures that images are both well-formed (conform to XML syntax) and valid (conform to SVG syntax).

3.2.2 In-Memory Images

The node-based approach to SVG creation offers more advantages than just being a cleaner way of building up an image. We are saving the root node (and thus its descendents) after the image has been created. This means we can keep the image in memory until we want to save to disk, or some other output. An example where this is useful is for embedding plots in an HTML document because plots can be inserted directly as inline SVG (rather than having to create an external file and then link to that file from the HTML document).

3.2.3 XPath

Another advantage is that because we are dealing with XML nodes, we can manipulate those nodes using other powerful XML tools such as XPath (W3C XSL and XML Linking Working Groups, 1999). XPath is a query language for XML. For example, we can retrieve and add subsets within the SVG image.

We will demonstrate this idea using a ggplot2 plot (the ggplot2 package uses grid for rendering so a ggplot2 plot consists of a large number of grid viewports and grobs).

```
R> library(ggplot2)
R> qplot(mpg, wt, data = mtcars, colour = cyl)
```

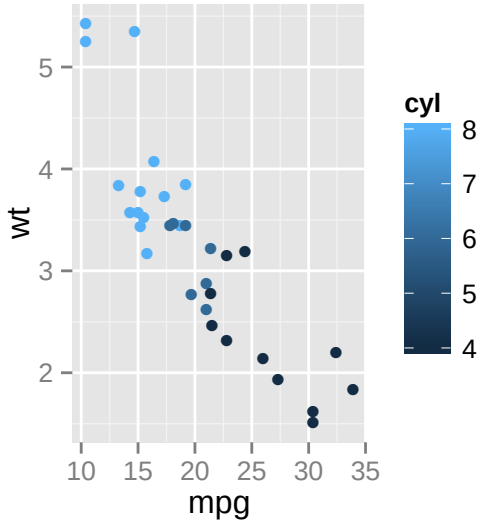


Figure 3.2: A ggplot2 scatter plot.

We can reduce the image shown in Figure 3.2 by removing the legend, so that only the plot is shown (see Figure 3.3). This code relies on standard functionality from the XML package for identifying and removing nodes; all we have to do is provide the XPath that describes the node that we want (in this case, a <g> element that has a specific id attribute).

```
R> svgDoc <- grid.export(NULL)$svg
R> legendNode <-
R+   getNodeSet(svgDoc,
R+              "//svg:g[@id='layout::guide-box.3-5-3-5.1']",
R+              c(svg = "http://www.w3.org/2000/svg"))[[1]]
R> removeChildren(xmlParent(legendNode), legendNode)
R> saveXML(svgDoc, file = "no-legend.svg")
```

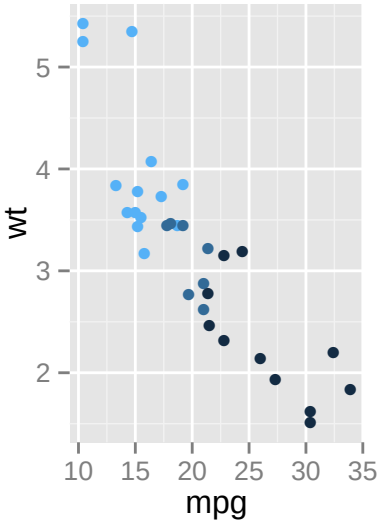


Figure 3.3: A ggplot2 scatter plot with a legend removed using the XML package.

Alternatively, we could extract just the legend from the plot and use it to create a new image, as shown in Figure 3.4.

```
R> svgDoc <- grid.export(NULL)$svg
R> legendNode <-
R+   getNodeSet(svgDoc,
R+              "//svg:g[@id='layout::guide-box.3-5-3-5.1']",
R+              c(svg="http://www.w3.org/2000/svg"))[[1]]
R> rootNode <-
R+   getNodeSet(svgDoc,
R+              "/svg:svg/svg:g",
R+              c(svg="http://www.w3.org/2000/svg"))[[1]]
R> removeChildren(rootNode, "g")
R> addChildren(rootNode, legendNode)
R> newSvg <-
R+   newXMLNode("svg", namespaceDefinitions =
R+                list("http://www.w3.org/2000/svg",
R+                     xlink = "http://www.w3.org/1999/xlink"),
R+              attrs = list(width = "50",
R+                           height = "200",
R+                           viewBox = "435 150 50 200",
```

```
R+                              version = "1.1"))
R> addChildren(newSvg, rootNode)
R> saveXML(newSvg, file = "legend-only.svg")
```

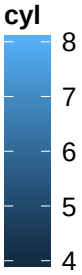


Figure 3.4: A ggplot2 scatter plot reduced to only show its legend.

These simple examples demonstrate the basic idea of extracting and combining arbitrary subsets of an SVG image. More complex applications are possible, such as combining the contents of two or more plots together. It is also important to note that these manipulations are made more convenient because the SVG produced by gridSVG has a clear and labelled structure; these tasks would be considerably more difficult if we had to work with the SVG output from the standard R `svg()` device.

3.2.4 Inserting Nodes

Another advantage of this new approach is that when we create an XML node, it can then be inserted into the SVG document at any location. Previously, with the string concatenation approach we were forced to simply append to the document. We now have the option of inserting nodes at any point in the document.

A case where this is useful is within gridSVG itself. When `grid.export()` is called, there are four parameters of particular interest: the filename, `exportCoords`, `exportMappings` and `exportJS`. The latter three parameters determine how JavaScript code is to be included within an SVG image, if at all. If we are going to be including JavaScript code, the SVG image is first generated. Once the image is created we insert new `<script>` node(s) to the root `<svg>` element. This demonstrates the ability to insert nodes at any location because rather than being forced to append to the document, we are able to add the nodes to become children of the root `<svg>` element.

3.2.5 Tree Simplification

One particular case where the XML package gives us some advantages is when saving an XML document. The function `saveXML()` provides a boolean option, `indent`. This determines whether there is going to be any visual structure in the form of indentation and line breaks or none at all. An example of its effect is shown below.

```
R> a <- newXMLNode("a")
R> b <- newXMLNode("b", parent = a)
R> saveXML(a, indent = TRUE)
<a>
  <b/>
</a>
R> saveXML(a, indent = FALSE)
<a><b/></a>
```

We can see that without indentation present the output is much more compact. In complex SVG images, particularly those with deep hierarchical structure, this could reduce the size of the resulting file greatly, which would improve the delivery speed of gridSVG plots being sent over the web by reducing the amount of data that needs to be transferred.

Another case where removing indentation is useful is when manipulating the SVG image in the browser using JavaScript. When parsing the SVG DOM with indentation present, the whitespace used for indentation is counted as a "node". This makes it difficult to traverse the DOM as it forces us to check whether the node that we have encountered is simply whitespace text or not. When indentation is removed, we no longer have this problem and can be certain that all nodes are either elements, or the actual content within them.

3.3 Conclusion

This chapter describes changes to the mechanism used by the gridSVG package to convert grid viewports and grobs to SVG representations. Instead of pasting strings together to generate SVG code as text within an external file, the gridSVG package now uses the XML package to create XML nodes in resident memory. The advantages of this approach include: guaranteed validity of the SVG representation; greater flexibility in the production of the SVG representation; improved access to the SVG representation;

and greater flexibility in the formatting of the SVG code. There are also possible speed benefits from these changes.

These advantages have been demonstrated through simple examples, but they also have an impact on much more complex scenarios. For example, if R is being used to serve web content to a browser, it is now possible for gridSVG to provide SVG fragments (rather than complete plots) and to supply them directly from resident memory (rather than having to generate an external file as an intermediate step).

4 The gridSVG Coordinate System

This chapter describes new features in gridSVG that allow grid coordinate system information to be exported along with an SVG image. This allows an SVG image to be modified dynamically in a web browser, with full knowledge of coordinate system information, such as the scales on plot axes. As a consequence, it is now possible to create more complex and sophisticated dynamic and interactive R graphics for the web.

4.1 Introduction

Two key features of grid distinguish it from the base graphics system provided by R: graphics objects (grobs) and viewports. Viewports are how grid defines a drawing context and plotting region. All drawing occurs relative to the coordinate system within a viewport. Viewports have a location and dimension and set scales on the horizontal and vertical axes. Crucially, they also have a name so we know how to refer to them.

Graphics objects (grobs) store information necessary to describe how a particular object is to be drawn. For example, a grid `circleGrob` contains the information used to describe a circle, in particular its location and its radius. As with viewports, graphics objects also have names.

The following code provides a simple demonstration of these grid facilities. A viewport is pushed in the centre of the page with specific x- and y-scales, then axes are drawn on the bottom and left edges of this viewport, and a set of data symbols are drawn within the viewport, relative to the scales. The viewport is given the name `panelvp` and the data symbols are given the name `datapoints`.

```
R> library(grid)
R> x <- runif(10, 5, 15)
R> y <- runif(10, 5, 15)
R> panelvp <- viewport(width = 0.7, height = 0.7,
R+                     xscale = c(0, 20),
```

```
R+                     yscale = c(0, 20),
R+                     name = "panelvp")
R>
R> pushViewport(panelvp)
R> grid.xaxis()
R> grid.yaxis()
R> grid.points(x, y, pch = 16, name = "datapoints")
R> upViewport()
```

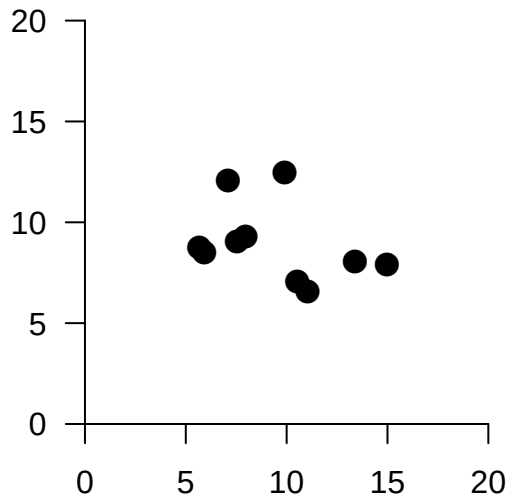


Figure 4.1: A basic plot produced by grid.

One advantage of having named viewports is that it is possible to *revisit* the viewport to add more output later. This works because revisiting a viewport means not only revisiting that region on the page, but also revisiting the coordinate system imposed on that region by the viewport scales. For example, the following code adds an extra (red) data point to the original plot (relative to the original scales).

```
R> downViewport("panelvp")
R> grid.points(3, 14, pch = 16,
R+             size = unit(4, "mm"),
R+             gp = gpar(col = "red"))
```

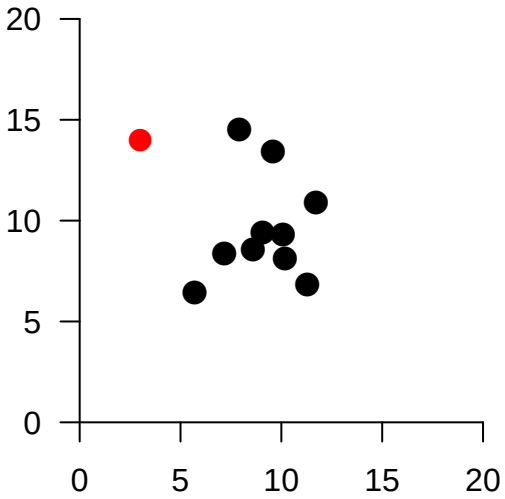


Figure 4.2: The basic plot produced by grid with an extra point added.

The task that gridSVG performs is to translate viewports and graphics objects into SVG equivalents. In particular, the exported SVG image retains the naming information on viewports and graphics objects. The advantage of this is we can still refer to the same information in grid and in SVG. However, prior to the work undertaken as part of this thesis, gridSVG was not able to retain any information about a viewport's coordinate system. The rest of this chapter discusses how this occurs and why it is useful.

4.2 The gridSVG Coordinate System

When exporting grid graphics as SVG, instead of positioning within a viewport, all drawing occurs within a single pixel-based coordinate system. This chapter describes how gridSVG now exports additional information so that the original grid coordinate systems are still available within the SVG document.

To demonstrate this, we will show how to add a new data symbol to a plot that has been drawn with grid and then exported with gridSVG. This is similar to the task performed in the previous section, but the difference is that we will add points to the plot without using grid itself, or even without using R at all. We will just make use of the information that has been exported by gridSVG and stored with the plot itself.

Firstly, consider the following code which draws a simple plot using grid, exactly as before, but this time exports the plot to to SVG in a file called `pointsPlot.svg` using

gridSVG. The file can then be viewed in a web browser.

```
library(gridSVG)
R> panelvp <- viewport(width = 0.7, height = 0.7,
R>                     xscale = c(0, 20),
R>                     yscale = c(0, 20),
R>                     name = "panelvp")
R> pushViewport(panelvp)
R> grid.xaxis()
R> grid.yaxis()
R> grid.points(x, y, pch = 16, name = "datapoints")
R> upViewport()
R> grid.export("pointsPlot.svg")
```

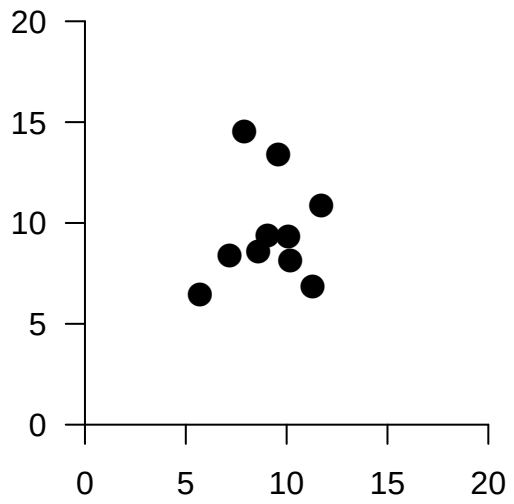


Figure 4.3: A basic plot produced by gridSVG.

It is important to note that `grid.export()` was called with the `exportCoords` argument set to `"file"`. This means that when gridSVG exports the image, it also saves the coordinate system to a file (in this case `pointsPlot.svg.coords.js`).

We will now consider the task of modifying this exported plot so that we can add extra information, such as a new data point. An important point is that we want to add a new data point relative to the axis coordinate systems on the plot and an important difference is that we are going to do this using only the information that was exported

by gridSVG, with no further help from grid.

When the SVG file was exported, all of the locations on the plot were transformed into pixels. This means that in our SVG file, none of the axis scales exist, and the locations of points are no longer native coordinates, but absolutely positioned pixels. See the x attributes in the <use> elements below as a demonstration of this process.

```
<use id="datapoints.1.1" xlink:href="#gridSVG.pch16" x="100.93" y...
<use id="datapoints.1.2" xlink:href="#gridSVG.pch16" x="104.85" y...
<use id="datapoints.1.3" xlink:href="#gridSVG.pch16" x="97.39" y=...
<use id="datapoints.1.4" xlink:href="#gridSVG.pch16" x="109.38" y...
<use id="datapoints.1.5" xlink:href="#gridSVG.pch16" x="75.56" y=...
<use id="datapoints.1.6" xlink:href="#gridSVG.pch16" x="117.74" y...
<use id="datapoints.1.7" xlink:href="#gridSVG.pch16" x="120.99" y...
<use id="datapoints.1.8" xlink:href="#gridSVG.pch16" x="86.59" y=...
<use id="datapoints.1.9" xlink:href="#gridSVG.pch16" x="92.2" y="...
<use id="datapoints.1.10" xlink:href="#gridSVG.pch16" x="108.66" ...
```

Prior to the work performed on gridSVG as part of this thesis, this task was awkward because it was difficult to determine the correct location of a new point in terms of pixels. Recent changes in gridSVG have enabled us to keep viewport information by exporting viewport metadata in the form of JSON, a structured data format. This enables us to be able to retain viewport locations and scales so that we can now transform pixel locations to native coordinates, and vice versa.

The following fragment shows the coordinates file that is exported by gridSVG. It is exported in the form of a JavaScript statement that assigns an object literal to a variable, `gridSVGCoords`.

```
var gridSVGCoords = {
    "ROOT" : {
        "x" : 0,
        "y" : 0,
        "width" : 216,
        "height" : 216,
        "xscale" : [
            0,
            216
        ],
```

```
        "yscale" : [
            0,
            216
        ],
        "inch" : 72
    },
    "panelvp.1" : {
        "x" : 32.4,
        "y" : 32.4,
        "width" : 151.2,
        "height" : 151.2,
        "xscale" : [
            0,
            20
        ],
        "yscale" : [
            0,
            20
        ],
        "inch" : 72
    }
};
```

This shows all of the information available to gridSVG. This JavaScript object contains a list of viewport names, with each viewport name associated with its metadata. The metadata includes the viewport location and dimensions in terms of SVG pixels. Also included are the axis scales, along with the resolution that the viewport was exported at. The resolution simply represents the number of pixels that span an inch.

This coordinate information is important for use with JavaScript functions which are also exported by gridSVG. Examples of such functions are shown in the next section.

4.3 Browser-based Modification

In this section, we will consider the task of modifying the exported SVG image using only a web browser, with no connection to R at all.

We can modify an SVG image within a web browser by executing JavaScript code to

insert SVG elements representing points into the plot. To start off we first load the image into the browser. This loads the SVG image, and executes any JavaScript code that is referenced or included by the image. By default gridSVG does not export coordinate information to a JavaScript file. This can be enabled by calling `grid.export()` with the `exportCoords` parameter set to `"file"`. A utility JavaScript file that contains functions useful for working with gridSVG graphics can also be exported. This can be achieved by setting the `exportJS` parameter in `grid.export()` to `"file"`. The utility code is particularly useful because it includes functions that enable us to do unit conversion in the browser, e.g. from `native` to `npc` or to `inches`.

Because gridSVG must perform some name manipulation to ensure that SVG element `id` attributes are unique, a couple of JavaScript utility functions require introduction. Firstly, although not strictly necessary, if we know the `id` of a grob (see section 2.6), we can find out which viewport path it belonged to by calling `grobViewport()`.

```
JS> var grobID = getSVGMappings("datapoints", "grob")[1];
JS> grobViewport(grobID);
"panelvp.1"
```

We see that the viewport name is not exactly what we chose in R, but suffixed with a numeric index. Now that we can query the viewport name, we know which viewport to draw into and the SVG element that we can add elements to. However, the issue remains that we really want to be able to use `native` units in the browser, rather than SVG pixels. To remedy this, unit conversion functions have been created. These functions are:

- `viewportConvertX()`
- `viewportConvertY()`
- `viewportConvertWidth()`
- `viewportConvertHeight()`

The first two conversion functions take three mandatory parameters, the viewport `id` you want the location of, the size of the unit, and what type of unit it is. Optionally a fourth parameter can be specified to determine what the unit is converted to, by default this is SVG pixels. The value returned from this function is a number representing the location in the new unit.

The second two conversion functions are the same but the fourth parameter, the new type of unit, is mandatory. This means we can convert between `inches`, `native` and `npc`

in the browser without requiring an instance of R available, as long as we stick to our existing viewports.

As an example of how we might use these functions, we can find out where the coordinates $(3, 14)$ are in the viewport panelvp (the main plot region) by running the following code:

```
JS> viewportConvertX("panelvp.1", 3, "native");
110.45
JS> viewportConvertY("panelvp.1", 14, "native");
283.1
```

We now know that the location of $(3, 14)$ in SVG pixels is $(110.45, 283.1)$. Using this information we can insert a new point into our plot at that location. We also want the the radius of this point to be 2mm, so we can work out how big the point is going to be in a similar manner. The following code shows that a radius of 2mm will translate to 5.67 SVG pixels.

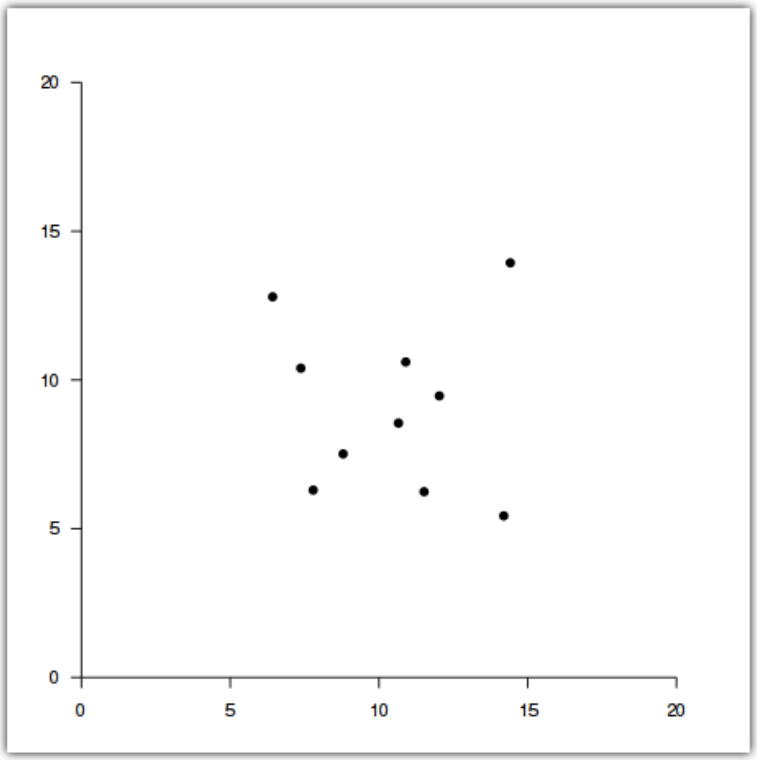
```
JS> viewportConvertWidth("panelvp.1", 2, "mm", "svg");
5.67
```

To insert this new point this requires some knowledge of JavaScript, and knowledge of the SVG DOM (Document Object Model). To demonstrate this, a red SVG circle is going to be inserted into the plot at $(3, 14)$ with a radius of 2mm using JavaScript.

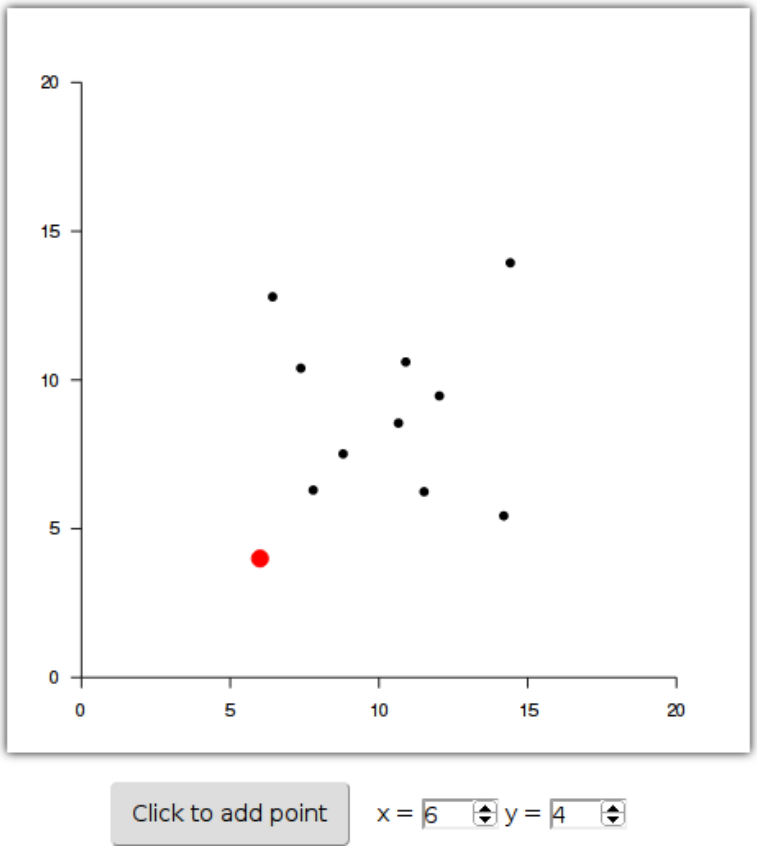
```
JS> // Getting the element that contains all existing points
JS> var panel = document.getElementById("panelvp.1");
JS> // Creating an SVG circle element
JS> var c = document.createElementNS("http://www.w3.org/2000/svg",
JS+                                  "circle");
JS> // Setting some SVG properties relating to the appearance
JS> // of the circle
JS> c.setAttribute("stroke", "rgb(255,0,0)");
JS> c.setAttribute("fill", "rgb(255,0,0)");
JS> c.setAttribute("fill-opacity", 1);
JS> // Setting the location and radius of our points
JS> // via the gridSVG conversion functions
JS> c.setAttribute("cx",
JS+     viewportConvertX("panelvp.1", 3, "native"));
JS> c.setAttribute("cy",
```

```
JS+     viewportConvertY("panelvp.1", 14, "native"));
JS> c.setAttribute("r",
JS+     viewportConvertWidth("panelvp.1", 2, "mm", "svg"));
JS> // Add new point to the same "viewport" as the existing points
JS> panel.appendChild(c);
```

Figure 4.4 shows a live demonstration of this task in action on a web page. When the "Click to add point" button is clicked, the browser generates a new SVG circle object and adds it to the image, relative to the plot scales, using JavaScript code and the information that was exported from R.



(4.4a) The web page before the "Click to add point" button has been clicked.



(4.4b) The x and y coordinates have been set to $(6, 4)$ and the "Click to add point" button has been clicked. The result is that a new red point has been inserted at the appropriate location.

Figure 4.4: A data point being inserted dynamically using JavaScript.

The example in Figure 4.4 provides a very simple demonstration of the idea that an SVG image can be manipulated within a web browser by writing JavaScript code. The significance of the new development in gridSVG is that, if an image was generated by grid and exported to SVG using gridSVG, extra information and JavaScript functions are now exported with the image so that the image can be manipulated with full knowledge of all grid coordinate systems that were used to draw the original image (such as plot axis scales).

Much more complex applications of this idea are possible, including complex animations and interactions, limited only by the amount and complexity of the JavaScript code that is required. This can involve leveraging third-party JavaScript libraries for manipulating the content of a web page, particularly SVG. The D3 library is one very powerful example.

4.4 Modification via the XML Package

In this section, we will consider modifying the exported SVG image using R, but in an entirely different R session, without recourse to grid and without any access to the original R code that was used to generate the original image.

When an SVG image is modified within a web browser using JavaScript, as in the previous section, all changes are lost when the image is reloaded. In this section, we will consider a permanent modification of the image that generates a new SVG file.

We will make use of the gridSVG and XML packages to modify our SVG image. As gridSVG automatically loads the XML package, all of the functionality from the XML package becomes readily available to us.

The first step is to parse the image, so that it is represented as a document within R.

```
R> svgdoc <- xmlParse("pointsPlot.svg")
```

We know that the name of the viewport we are looking for has the exported `id` of `panelvp.1`. An XPath query can be created to collect this viewport.

```
R> # Getting the object representing our viewport that contains
R> # our data points
R> panel <- getNodeSet(svgdoc,
R+                     "//svg:g[contains(@id, 'panelvp')]",
R+                     c(svg="http://www.w3.org/2000/svg"))[[1]]
```

Now we need to read in the JavaScript file that contains the coordinates information. However, some cleanup is needed because the code is designed to be immediately loaded within a browser, and is thus not simply JSON. We need to clean up the data so that it is able to be parsed by the RJSONIO package's `fromJSON()` function. The `readCoordsJS()` function performs this task in addition to parsing the resulting JSON data.

```
R> # Reading in and cleaning up the coordinate system data
R> coordsData <- readCoordsJS("pointsPlot.svg.coords.js")
```

We now have valid JSON in the form of a list. Using this, we can initialise a coordinate system in R by utilising `gridSVGCoords()`. Nothing is returned from `gridSVGCoords()` because we are *setting* coordinate information. If we call `gridSVGCoords()` with no parameters we can get the coordinate information back.

```
R> gridSVGCoords(coordsData)
```

Now that a coordinate system is initialised we are able convert coordinates into SVG pixels. This means we can create a `<circle>` element and correctly position it using `native` units at $(3, 14)$.

```
R> # Creating an SVG circle element to insert into our image
R> # that is red, and at (3, 14), with a radius of 2mm
R> circ <- newXMLNode("circle",
R+   parent = panel,
R+   attrs = list(
R+     cx = viewportConvertX("panelvp.1", 3, "native"),
R+     cy = viewportConvertY("panelvp.1", 14, "native"),
R+     r = viewportConvertWidth("panelvp.1", 2, "mm", "svg"),
R+     stroke = "red",
R+     fill = "red",
R+     "fill-opacity" = 1))
```

Note that we have used the `viewportConvert*()` functions to position the circle at the correct location and with the correct radius. This demonstrates that the same functions that are available in JavaScript are also available in R (via the gridSVG package).

This point has been inserted into the same SVG group as the rest of the points by setting the `parent` parameter to the object representing the viewport group.

The only thing left to do is write out the new XML file with the point added.

```
R> # Saving a new file for the modified image
R> saveXML(svgdoc, file = "newPointsPlot.svg")
```

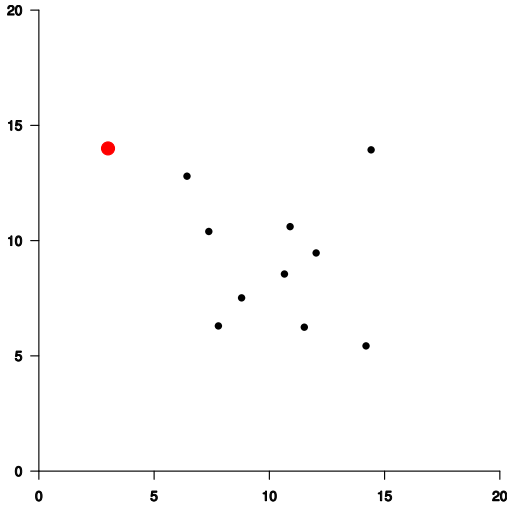


Figure 4.5: A gridSVG image modified by the XML package to insert an additional data point.

The new SVG image is located at `newPointsPlot.svg` and when loaded into a web browser shows the new point. The appearance of the plot should be identical to the modifications we made using JavaScript, except these modifications are permanent and are able to be distributed to others.

This example provides a very simple demonstration of the idea that an SVG image can be manipulated in R using the XML package. The significance of the new development in gridSVG is that, if an image was generated by grid and exported to SVG using gridSVG, extra information is now exported with the image, and new functions are available from gridSVG to work with that exported information, so that the image can be manipulated with full knowledge of all grid coordinate systems that were used to draw the original image (such as plot axis scales).

Much more complex modifications of an image are possible, limited only by the amount and complexity of the R code that is required.

4.5 Conclusion

This chapter describes several new features of the gridSVG package. The main idea is that grid coordinate system information can now be exported, in a JSON format, along with the image (in an SVG format). In addition, JavaScript functions can be

exported to support the manipulation of the SVG image within a web browser, using this coordinate system information. Furthermore, new R functions are provided so that the SVG image, and its associated coordinate information, can be loaded back into a different R session to manipulate the SVG image within R (using the XML package). These features significantly enhance the framework that gridSVG provides for developing dynamic and interactive R graphics for the web.

5 The selectr Package

The selectr package (Potter, 2012) translates a CSS selector into an equivalent XPath expression. This allows the use of CSS selectors — which are usually easier, more terse and more declarative than equivalent XPath expressions — to query XML documents using the XML package. Convenience functions are also provided to mimic functionality present in modern web browsers. These selectors allow any part of a gridSVG image to be easily identified and manipulated, using a more convenient query language than XPath.

5.1 Introduction

When working with XML documents, a common task is to be able to search for parts of a document that match a search query. For example, if we have a document representing a collection of books, we might want to search through for a book matching a certain title or author. A language exists that constructs search queries on XML documents, called XPath (W3C XSL and XML Linking Working Groups, 1999). XPath is capable of constructing complex search queries but this is often at the cost of readability and terseness of the resulting expression.

An alternative way of searching for parts of a document is using CSS selectors (W3C CSS Working Group, 2011). These are most commonly used in web browsers to apply styling information to components of a web page. We can use the same language that selects which nodes to style in a web page to select nodes in an XML document. This often produces more concise and readable queries than the equivalent XPath expression. It must be noted however that XPath expressions are more flexible than CSS selectors, so although all CSS selectors have an equivalent XPath expression, the reverse is not true.

An advantage of using CSS selectors is that most people working with web documents such as HTML and SVG also know CSS. XPath is not employed anywhere beyond querying XML documents so is not a commonly known query language. Another important reason why CSS selectors are widely known is due to the common use of them in popular

JavaScript libraries. jQuery and D3 are two examples where they select elements of a page to perform operations on using CSS selectors, rather than XPath. This is mostly due to the complexity of performing an XPath query in the browser in addition to the more verbose expressions. An example of how one would use CSS selectors to retrieve content using popular JavaScript libraries is with the following code:

```
JS> // jQuery
JS> $("#listing code");
JS> // D3
JS> d3.selectAll("#listing code");
JS> // The equivalent XPath expression
JS> "descendant-or-self::*[@id='listing']/descendant-or-self::*/code"
```

Selecting all <code> elements that are descendents of the element with the `id` "listing".

The XML package for R is able to parse XML documents, which can later be queried using XPath. No facility exists for using CSS selectors on XML documents in the XML package. This limitation is due to the XML package's dependence on the libxml2 library which can only search using XPath. For reasons mentioned above, it would be ideal if we had the option of using CSS selectors to query such documents as well as XPath. If we can translate a CSS selector to XPath, then the restriction to only using XPath no longer applies and we can therefore use a CSS selector whenever an XPath expression is required.

A mature Python package exists that performs translation of CSS selectors to XPath expressions. Unfortunately this package, cssselect (Sapin and Bicking, 2013), cannot be used in R because it would require Python to be present on a user's system which cannot be guaranteed, particularly on Windows. The selectr package is a translation of cssselect to R so that we have the same functionality within R as we would have using Python.

The rest of this article describes the process that selectr takes to translate CSS selectors to XPath expressions along with example usage.

5.2 Parsing

The first step in translating any language to another is to first *tokenise* an expression into individual words, numbers, whitespace and symbols that represent the core structure of an expression. These pieces are called tokens. The following code shows character representation of tokens that have been created from tokenising a CSS selector expression:

```
R> tokenize("body > p")
<IDENT 'body' at 1>
<S ' ' at 5>
<DELIM '>' at 6>
<S ' ' at 7>
<IDENT 'p' at 8>
<EOF at 9>
```

The selector `body > p` is a query that looks for all "p" elements within the document that are also direct descendants of a "body" element. We can see that the selector has been tokenised into 6 tokens. Each token has the following structure: type, value, position. The type is the kind of token it is, an identifier, whitespace, a number or a delimiter. The value is the actual text that a token represents, while the position is simply the position along the string at which the character was found.

Once we have the required tokens, it is necessary to *parse* these tokens into a form that applies meaning to the tokens. For example, in CSS a # preceding an identifier means that we are looking for an element with an ID matching that identifier. After parsing our tokens, we have an understanding of what the CSS selector means and therefore have the correct internal representation prior to translation to XPath. The following code shows what our example selector is understood to mean:

```
parse("body > p")
CombinedSelector[Element[body] > Element[p]]
```

This shows that the selector is understood to be a combined selector that matches when a "p" element is a direct descendant of a "body" element. Once the parsing step is complete, it is necessary to translate this internal representation of a selector into its equivalent XPath expression.

5.3 Translating

XPath is a superset of the functionality of CSS selectors, so we can ensure that there is a mapping from CSS to XPath. Given that we already know the parsed structure of the selector, we work from the outer-most selector inwards. This means with the parsed selector `body > p` we look at the `CombinedSelector` first, then the remaining `Element` components. In this case we know that the `CombinedSelector` is going to map to `Element[body]/Element[p]`, which in turn produces `body/p`.

Some of these mappings are straightforward as was the case in the given example, but others are more complex. Table 5.1 shows a sample of the translations that occur.

CSS Selector	Parsed Structure	XPath Expression
`#test`	`Hash[Element[*]#test]`	`descendant-or-self::*[@id = 'test']`
`.test`	`Class[Element[*].test]`	`descendant-or-self::*[class and contains(concat(' ', normalize-space(class), ' '), ' test ')]`
`body p`	`CombinedSelector[Element[body] <followed> Element[p]]`	`descendant-or-self::body/descendant-or-self::*/p`
`a[title]`	`Attrib[Element[a][title]]`	`descendant-or-self::a[title]`
`div[class^='btn']`	`Attrib[Element[div][class ^= 'btn']]`	`descendant-or-self::div[class and starts-with(class, 'btn')]`
`li:nth-child(even)`	`Function[Element[li]:nth-child(['even'])]`	`descendant-or-self::*/*[name() = 'li' and ((position() +0) mod 2 = 0 and position() >= 0)]`
`#outer-div :first-child`	`CombinedSelector[Hash[Element[*] #outer-div] <followed> Pseudo[Element[*]:first-child]]`	`descendant-or-self::*[@id = 'outer-div']/ descendant-or-self::*/* [position() = 1]`

Table 5.1: Examples of translations from CSS selectors to XPath expressions, including intermediary parsed structures.

The examples shown in Table 5.1 only touch on the possible translations, but it demonstrates that a mapping from CSS selectors to XPath expressions exists.

5.4 Usage

The selectr package becomes most useful when working with the XML package. Most commonly selectr is used to simplify the task of searching for a set of nodes. In the browser, there are two JavaScript functions that perform this task using CSS selectors, `querySelector()` and `querySelectorAll()`. These functions are methods on a document or element object. Typical usage in the browser using JavaScript might be the following:

```
JS> document.querySelector("ul li.active");
JS> document.querySelectorAll("p > a.info");
```

Retrieving content from a document using CSS selectors in JavaScript.

Because these are so commonly used in popular JavaScript libraries, the behaviour has been mimicked in selectr. The selectr package also provides these functions but instead of being methods on document or element objects, they are functions. These functions typically take two parameters, the XML object to be searched on, and the CSS selector to query with, respectively.

The difference between the two functions is that `querySelector()` will attempt to return the *first* matching node or `NULL` in the case that no matches were found. `querySelectorAll()` will always return a list of matching nodes, this list will be empty when there are no matches. To demonstrate the usage of these functions, the following XML document will be used:

```
R> library(XML)
R> exdoc <- xmlRoot(xmlParse('
R+ <a>
R+     <b class="aclass"/>
R+     <c id="anid"/>
R+ </a>
R+ '))
R> exdoc
<a>
  <b class="aclass"/>
  <c id="anid"/>
</a>
```

We will first see how `querySelector()` is used.

```
R> library(selectr)
R> querySelector(exdoc, "#anid")   # Returns the matching node
<c id="anid"/>
R> querySelector(exdoc, ".aclass") # Returns the matching node
<b class="aclass"/>
R> querySelector(exdoc, "b, c") # First match from grouped selection
<b class="aclass"/>
R> querySelector(exdoc, "d")       # No match
NULL
```

Now compare this to the results returned by `querySelectorAll()`:

```
R> querySelectorAll(exdoc, "b, c") # Grouped selection
[[1]]
<b class="aclass"/>

[[2]]
<c id="anid"/>

attr(,"class")
[1] "XMLNodeSet"
R> querySelectorAll(exdoc, "b")    # A list of length one
[[1]]
```

```
<b class="aclass"/>

attr(,"class")
[1] "XMLNodeSet"
R> querySelectorAll(exdoc, "d")    # No match
list()
attr(,"class")
[1] "XMLNodeSet"
```

The main point to get across is that `querySelector()` returns a *node*, and `querySelectorAll()` returns a *list* of nodes.

Both `querySelector()` and `querySelectorAll()` are paired with namespaced equivalents, `querySelectorNS()` and `querySelectorAllNS()` respectively. These functions will be demonstrated in more detail later in this chapter (see subsection 5.5.1).

While the aforementioned functions are certainly useful, they do not cover all possible use cases. For other uses of CSS selectors, the `css_to_xpath()` function can be used where an XPath expression would normally be expected. The `css_to_xpath()` function has three parameters. The first parameter is simply the CSS selector, then a prefix on the resulting XPath expression. This prefix is useful in the circumstance when you already know some XPath and know where the selector should be scoped to. The final parameter determines the translator to use when translating selectors to XPath expressions. The generic translator is sufficient in most cases except when (X)HTML is used; in those cases a translator can be used which is aware of (X)HTML pseudo-selectors. A case where `css_to_xpath()` may be used is when using XML's `*apply()` functions, as shown below.

```
R> # Let's see all tag names present in the doc
R> xpathSApply(exdoc, css_to_xpath("*"), xmlName)
[1] "a" "b" "c"
R> # What is the value of the class attribute on all "b" elements?
R> xpathSApply(exdoc, css_to_xpath("b"),
R+             function(x) xmlGetAttr(x, "class"))
[1] "aclass"
```

Rather than returning nodes, we are processing each node using a given function from the XML package, but specifying paths using CSS selectors instead.

5.5 Examples

While the example usage of the selectr package has been demonstrated earlier, the real-world usage may not be clear, or indeed the benefits over just using the XML package. To show how succinct it can be, we will try to create a data frame in R that lists the titles and URLs of technical reports hosted on the Department of Statistics Technical Report Blog, along with their publishing dates. First, lets examine part of the HTML that comprises the page to see how we're going to be selecting content.

```
...
<article>
<header>
<h1 class="entry-title">
<a href="http://stattech.wordpress.fos.auckland.ac.nz/2012-9-writ...
   title="Permalink to 2012-9 Writing grid Extensions"
   rel="bookmark">2012-9 Writing grid Extensions</a>
</h1>
<div class="entry-meta">
<span class="sep">Posted on </span>
<a href="http://stattech.wordpress.fos.auckland.ac.nz/2012-9-writ...
   title="9:48 pm"
   rel="bookmark">
<time class="entry-date"
      datetime="2012-11-06T21:48:17+00:00" pubdate>
    November 6, 2012
</time>
</a>
...
```

This fragment shows us that we have the information available to us, we just need to know how to query it. For example, we can see that the URL to each technical report is in the `href` attribute within an `<a>` element. In particular, this `<a>` element has an `<h1>` parent with a class of `entry-title`. The `a` element also contains the title of the technical report. Similarly we can see a `<time>` element that tells us via the `datetime` attribute when the post was published. We first start by loading the required packages and retrieving the data so that we can work with it.

```
R> library(XML)
R> library(selectr)
R> page <- htmlParse("http://stattech.wordpress.fos.auckland.ac.nz/")
```

Now that the page has been parsed into a queryable form, we can write the required CSS selectors to retrieve this information using `querySelectorAll()`.

```
R> # CSS selector to get titles and URLs: "h1.entry-title > a"
R> links <- querySelectorAll(page, "h1.entry-title > a")
R> # Now lets get all of the publishing times
R> timeEls <- querySelectorAll(page, "time")
```

Now that we have gathered the correct elements, it is reasonably simple to manipulate them using the XML package. We want to extract the correct attributes and values from the selected nodes. The code below shows how we would do this.

```
R> # Collect all URLs
R> urls <- sapply(links, function(x) xmlGetAttr(x, "href"))
R> # Collect all titles
R> titles <- sapply(links, xmlValue)
R> # Collect all datetime attributes
R> dates <- sapply(timeEls, function(x) xmlGetAttr(x, "datetime"))
R> # To play nice with R, lets parse it as a Date
R> dates <- as.Date(dates)
R> # Create a data frame of the results
R> technicalReports <- data.frame(title = titles,
R+                                url = urls,
R+                                date = dates,
R+                                stringsAsFactors = FALSE)
R> # and show one column at a time
R> technicalReports$title
[1] "2013-5 Open Data in New Zealand"
[2] "2013-4 Generating Structured and Labelled SVG"
[3] "2013-3 Generating unique names in gridSVG"
[4] "2013-2 Modelling competitive exclusion and limited dispersal...
[5] "2013-1 TimingManager: Animation Sequences in JavaScript"
[6] "2012-12 Post-Processing grid Graphics"
[7] "2012-11 Generating Animation Sequence Descriptions"
```

```
[8] "2012-10 Introducing the 'selectr' Package"
[9] "2012-9 Writing grid Extensions"
[10] "2012-8 Meta-analysis of a rare-variant association test"
R> technicalReports$url
[1] "http://stattech.wordpress.fos.auckland.ac.nz/2013-5-open-dat...
[2] "http://stattech.wordpress.fos.auckland.ac.nz/2013-4-generati...
[3] "http://stattech.wordpress.fos.auckland.ac.nz/2013-3-generati...
[4] "http://stattech.wordpress.fos.auckland.ac.nz/2012-14-modelli...
[5] "http://stattech.wordpress.fos.auckland.ac.nz/2012-13-timingm...
[6] "http://stattech.wordpress.fos.auckland.ac.nz/2012-12-post-pr...
[7] "http://stattech.wordpress.fos.auckland.ac.nz/2012-11-generat...
[8] "http://stattech.wordpress.fos.auckland.ac.nz/2012-10-introdu...
[9] "http://stattech.wordpress.fos.auckland.ac.nz/2012-9-writing-...
[10] "http://stattech.wordpress.fos.auckland.ac.nz/2012-8-meta-an...
R> technicalReports$date
 [1] "2013-05-31" "2013-04-10" "2013-04-03" "2013-03-01"
 [5] "2013-01-09" "2012-12-11" "2012-11-30" "2012-11-27"
 [9] "2012-11-06" "2012-11-04"
```

5.5.1 A Complex Example

An example written for the gridSVG package will be revisited (shown previously in subsection 3.2.3). The example first shows a ggplot2 plot that has been exported to SVG using gridSVG. The aim is to then remove the legend from the plot by removing the node containing all legend information. Once the node has been removed, the resulting document can be saved to produce an image with a legend removed.

What is of particular interest with this example is that it uses SVG, which is a namespaced XML document. This provides some challenges that require consideration, but the selectr package can handle this case.

```
R> library(ggplot2)
R> library(gridSVG)
R> qplot(mpg, wt, data = mtcars, colour = cyl)
R> svgdoc <- grid.export("")$svg
```

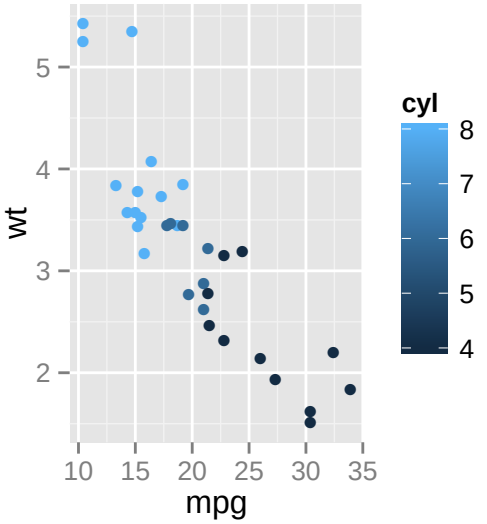


Figure 5.1: A ggplot2 scatter plot that has been exported in-memory to SVG.

So far we have simply reproduced the original plot and stored the resulting XML in a node tree called `svgdoc`. In order to remove the legend from the plot we first need to select the legend node from the SVG document. We will compare the XML-only approach with one enhanced with `selectr`. The comparison is shown below:

```
R> # XPath
R> legend <- getNodeSet(svgdoc,
R+                      "//svg:g[@id='layout::guide-box.3-5-3-5.1']",
R+                      c(svg = "http://www.w3.org/2000/svg"))[[1]]
R> # CSS
R> legend <- querySelector(svgdoc,
R+                         "#layout\\:\\:guide-box\\.3-5-3-5\\.1",
R+                         c(svg = "http://www.w3.org/2000/svg"),
R+                         prefix = "//svg:*/descendant-or-self::")
```

This particular example demonstrates a case where the XPath approach is more concise. This is because the `id` attribute that we're searching for needs to have its CSS selector escaped (due to `:` and `.` being special characters in CSS), while the XPath expression remains unchanged. Additionally, we also need to specify a namespace-aware prefix for the XPath that is generated. To use CSS selectors in this case required knowledge of XPath that would rather be avoided.

To work around this issue, a namespace-aware function should be used instead to abstract away the XPath dependent code. The following code demonstrates the use of selectr's namespace-aware function `querySelectorNS()`:

```
R> legend <- querySelectorNS(svgdoc,
R+                           "#layout\\:\\:guide-box\\.3-5-3-5\\.1",
R+                           c(svg = "http://www.w3.org/2000/svg"))
```

The resulting use of CSS selection is now as concise as the XPath version, with the only special consideration being the requirement of escaping the CSS selector.

Now that the legend has been selected, we can remove it from the SVG document to produce an image with a legend omitted.

```
R> removeChildren(xmlParent(legend), legend)
R> saveXML(svgdoc, file = "qplot-no-legend.svg")
```

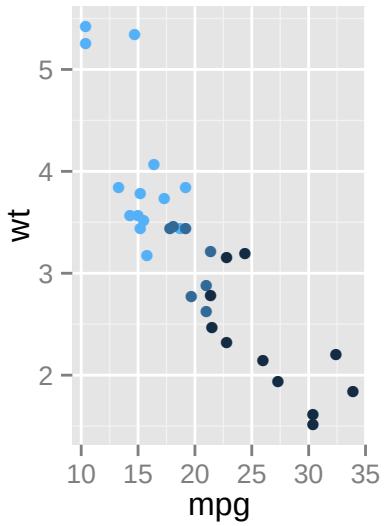


Figure 5.2: A ggplot2 scatter plot with a legend removed.

5.6 Conclusion

This chapter describes the new selectr package. Its main purpose is to allow the use of CSS selectors in a domain which previously only allowed XPath. In addition, convenience functions have also been described; allowing easy use of CSS selectors for the purpose of

retrieving parts of an XML document e.g. an SVG image. It has been demonstrated that the selectr package augments the XML package with the ability to use a more concise language for selecting content from an XML document.

6 Animation Sequencing

This chapter is divided into two sections. The first section describes the animaker package for generating descriptions of animation sequences. An animation sequence is composed by combining atomic animations in series to create sequence animations or in parallel to create track animations. Functions are provided for manipulating animation sequences, generating timing schemes from animation sequences, and producing diagrams to visualise animation sequences.

The second section in this chapter describes how animation sequences can be applied in a web browser using the TimingManager library in JavaScript. It is not concerned with creation or modification of animation timing information, and delegates those tasks to R via the animaker package. Primarily TimingManager is focused on assigning actions to existing animations, then playing animation sequences using either a declarative or frame-based approach.

6.1 Describing Animation Sequences

6.1.1 Introduction

Several R packages exist for generating animations in R. For example, the animation package provides convenience functions for generating an animation, in a wide variety of formats, from a series of static frames, where the frames are drawn in a loop using normal R graphics functions. Several other packages provide a more declarative approach, allowing individual animation actions to be specified in terms of a start value and end value (plus a start time and end time). For example, the animatoR package (Blejec, 2011) provides functions for generating static frames from a declarative animation and the gridSVG package and the SVGAnnotation package both provide functions for adding declarative animation information to an R plot in an SVG format (for viewing on the web).

One of the difficulties that arises when attempting to generate an animation sequence

with these packages is the coordination of multiple animation actions. For example, something as simple as ensuring that action A follows action B can become complicated to specify and difficult to maintain if those two actions are part of a larger collection of animation actions that also need to be run either in sequence or in parallel.

This section describes an R package called animaker (Murrell and Potter, 2012) that can be used to generate a description of an animation sequence. The description is completely abstract and only focuses on a specification of when animation actions should occur. Animation actions are only represented by a label and no actions are actually carried out, though there are functions to generate a timing information for the animation. There are also functions for drawing diagrams (including animated diagrams) to visualise the animation sequence.

6.1.2 Atomic animations

The fundamental component of an animation sequence is an *atomic animation*, which simply consists of a `start` value and a `durn` (duration). It is also possible to specify a `label`, but this will be automatically generated by default. An atomic animation is generated using the `atomic()` function; both the `start` and `durn` default to zero.

```
R> atomic()
Alpha:0:0
```

The following code shows a more realistic example, where an atomic animation is generated with a duration of 2 seconds. There is a `plot()` method for atomic animations which draws a diagram of the animation sequence represented by the atomic animation (see Figure 6.1).

```
R> atmc <- atomic(durn = 2)
R> atmc
Bravo:0:2
```

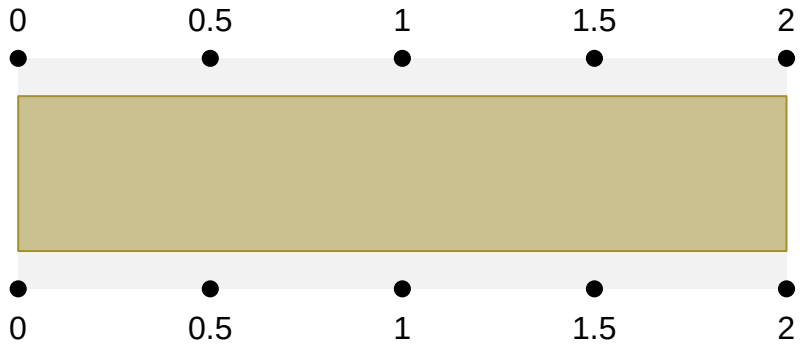


Figure 6.1: A diagram of an atomic animation with duration of 2 seconds.

The `start` argument can be used to delay the start of the animation, as shown in the code below and in Figure 6.2.

```
R> delay <- atomic(start = 1, durn = 2)
R> delay
Charlie:1:2
```

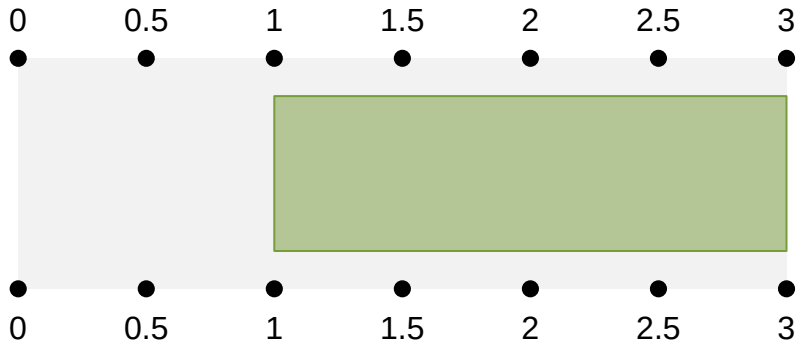


Figure 6.2: A diagram of an atomic animation with a delay of 1 second (and a duration of 2 seconds).

Note that this system only uses the start and duration times to determine when an animation action will be *started*. There is no requirement that the animation action actually plays for exactly the prescribed duration. It is expected that the duration will be passed to the animation action and that some actions will take note of the duration and respond accordingly, but it is also acknowledged that the animation action may completely ignore the duration information.

6.1.3 Container animations

A *container animation* is a collection of animations. In the simplest case, a container is a collection of atomic animations, but it is also possible to have a collection of container animations (or a mixture of both).

There are two sorts of containers, *sequence animations* and *track animations*. A sequence animation plays a collection of animations one after the other. A track animation plays a collection of animations simultaneously.

To demonstrate these ideas, we first create three atomic animations: one lasting 1 second, one lasting 2 seconds, and one lasting 3 seconds.

```
R> a <- atomic(durn = 1)
R> b <- atomic(durn = 2)
R> c <- atomic(durn = 3)
```

The `vec()` function can be used to create a sequence animation by running these animations sequentially (see Figure 6.3).

```
R> v <- vec(a, b, c)
```

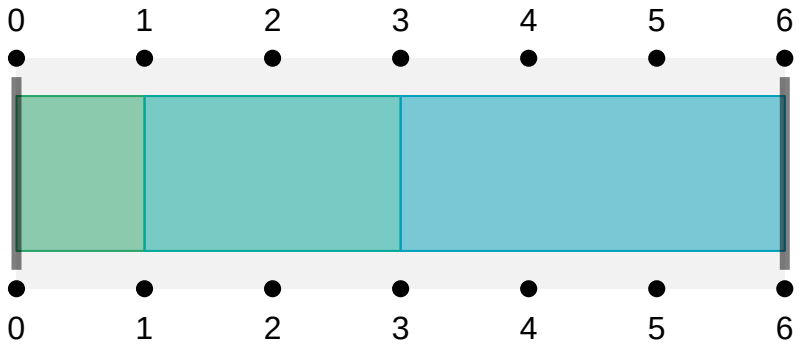


Figure 6.3: A sequence animation consisting of three atomic animations.

The `trac()` function can be used to create a track animation by running these animations in parallel (see Figure 6.4).

```
R> t <- trac(a, b, c)
```

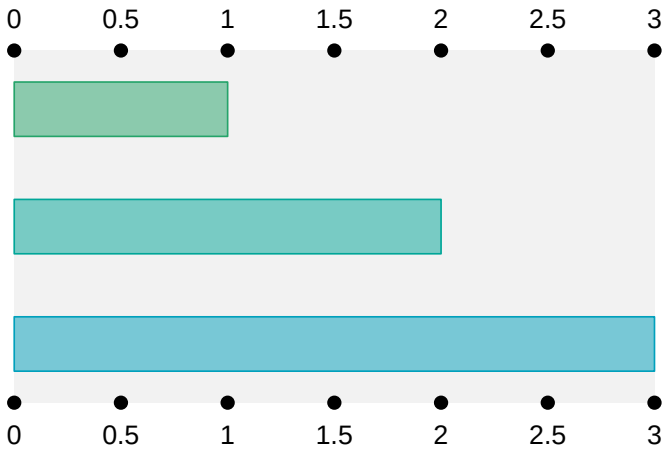


Figure 6.4: A track animation consisting of three atomic animations.

The contents of a container can be another container. For example, the following code produces an animation sequence consisting of a track animation followed by a sequence animation (see Figure 6.5).

```
R> container <- vec(t, v)
```

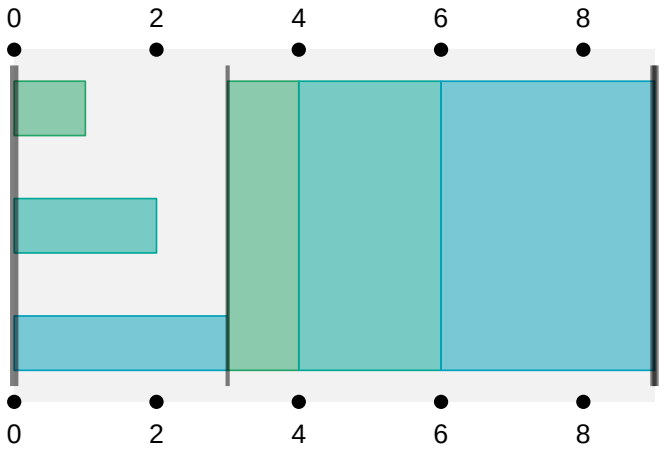


Figure 6.5: A sequence animation consisting of a track animation followed by a sequence animation.

6.1.4 Controlling the start and duration of containers

A container animation may have its own `start`, which delays the entire collection of animations (see Figure 6.6).

```
R> vDelay <- vec(a, b, c, start = 1)
```

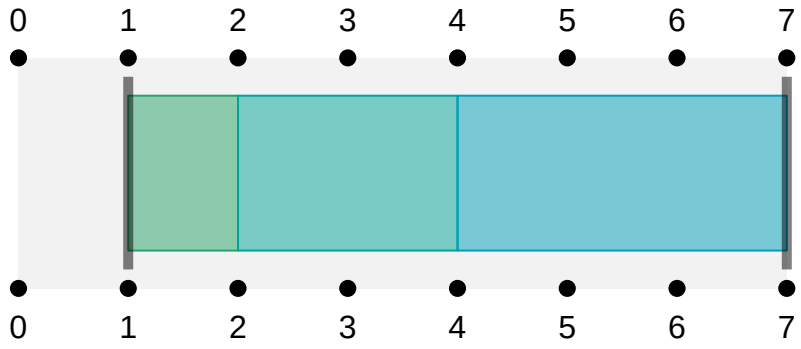


Figure 6.6: A sequence animation (consisting of three atomic animations) with a delayed start.

6.1.5 Controlling the start and duration of container content

By default, the `durn` of a container animation is calculated from the animations within the container, but this may be overridden (see Figure 6.7).

```
R> vDurn <- vec(a, b, c, start = 1, durn = 10)
```

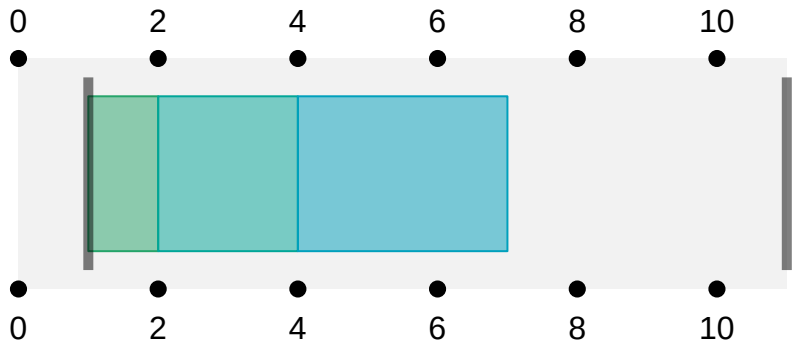


Figure 6.7: A sequence animation (consisting of three atomic animations) with a delayed start and an explicit duration.

For the *contents* of a sequence animation, the `start` value represents a delay that is relative to the end of the previous animation in the sequence. For example, the following code generates an atomic animation that starts after a delay of 0.5 seconds and lasts for 2 seconds. If this animation is used as part of an animation sequence, the delay on the atomic animation creates a pause within the sequence (see Figure 6.8).

```
R> d <- atomic(start = 0.5, durn = 2)
R> vContentDelay <- vec(a, b, c, d)
```

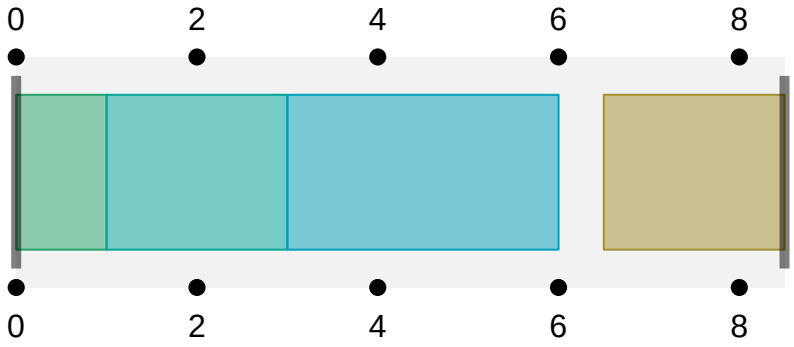


Figure 6.8: A sequence animation (consisting of four atomic animations) where the fourth atomic animation has a delay.

For the contents of track animations, a delay is just relative to the start of the track animation. For example, the following code generates a track animation from two atomic animations where the second atomic animation has a delay of 0.5 seconds so it starts after the first atomic animation (see Figure 6.9).

```
R> tContentDelay <- trac(a, d)
```

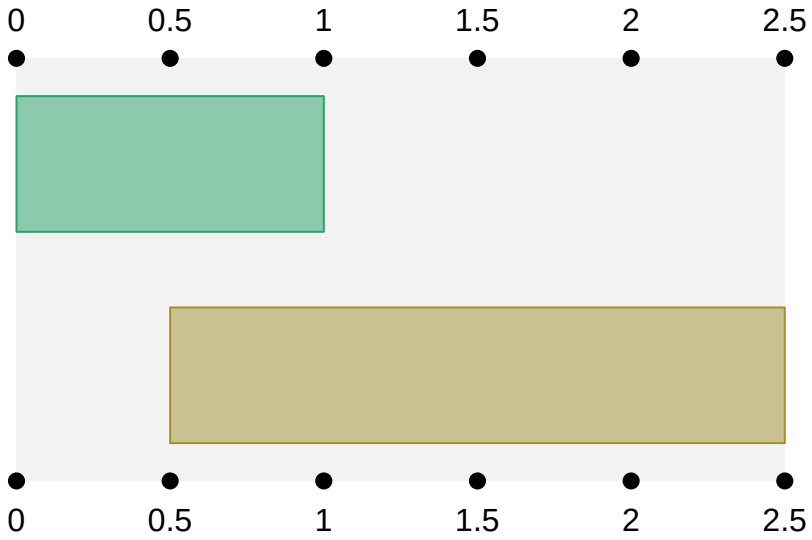


Figure 6.9: A track animation (consisting of two atomic animations) where the second atomic animation has a delay.

It is also possible for the container to override the start and duration values for its contents. This is achieved by specifying a *vector* of values for either `start` or `durn`. The following code shows this feature being used to insert a delay between animations within a sequence. There are three atomic animations in the sequence, each of which has a start of zero, but the sequence animation specifies `start` values of 0, 0.5, and 0.5, which override the atomic animation start values (see Figure 6.10).

```
R> vParentDelay <- vec(a, b, c, start = c(0, 0.5, 0.5))
```

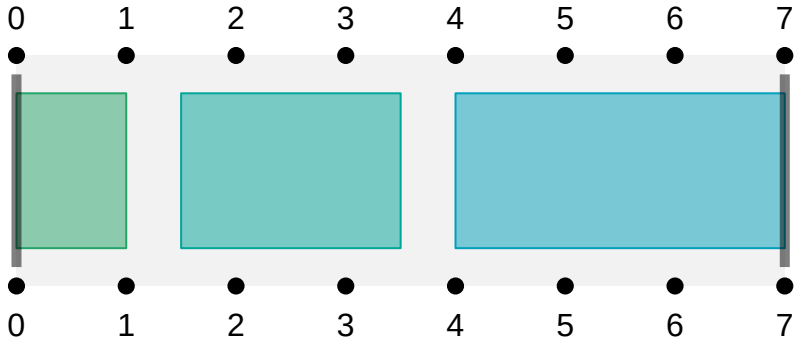


Figure 6.10: A sequence animation (consisting of three atomic animations) where the parent sequence has inserted a delay between the atomic animations.

The following code combines the three atomic animations into a track animation where the track animation specifies start values for the atomic animations in order to stagger the start of the animations within the track (see Figure 6.11).

```
R> tParentDelay <- trac(a, b, c, start = c(0, 0.5, 1))
```

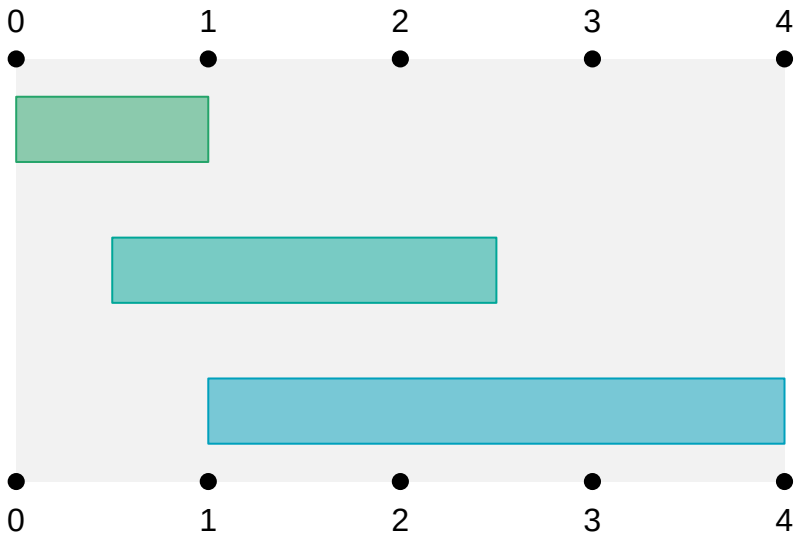


Figure 6.11: A track animation (consisting of three atomic animations) where the parent track has staggered the start of the atomic animations.

An example of the container explicitly controlling the duration of its contents will be shown later in subsection 6.1.9.

Another possibility is that an atomic animation may abdicate responsibility for its duration to its parent container. This is achieved by specifying `NA` for the `durn` of an atomic animation. This only works if the container animation is not also relying on calculating its duration from its contents.

In the case of a sequence animation, if the container has an explicit duration then any atomic content of the container can have `NA` duration, in which case the container duration is equally shared out amongst such atomic content.

In the case of a track animation, if either the container has an explicit duration or at least one animation in the contents of the container has an explicit duration (from which the container can calculate its duration), then any atomic content with `NA` duration is given the container duration.

For example, the following code generates a sequence animation from five atomic animations, where the second and fourth atomic animations have NA duration. The sequence animation has an explicit duration of 10 seconds, three of the atomic animations have a combined duration of 6 seconds, so the two atomic animations with NA duration get 2 seconds each (4 seconds divided by 2).

```
R> f <- atomic(durn = NA)
R> navec <- vec(a, f, b, f, c, durn = 10)
```

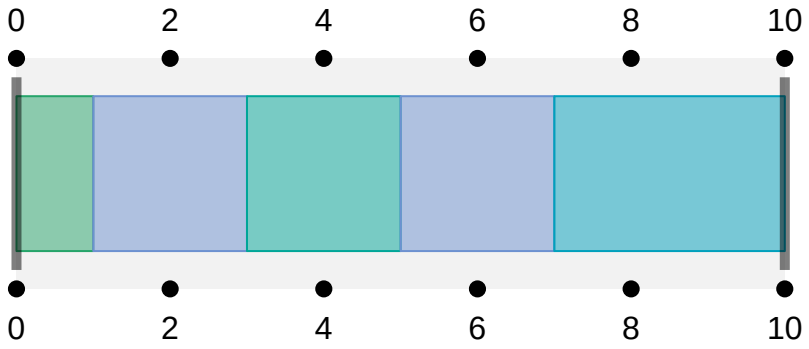


Figure 6.12: A sequence animation consisting of five atomic animations, where the second and fourth atomic animations have a duration of NA, so their duration is calculated from the parent sequence.

In the following code, a track animation is generated from four atomic animations, with the fourth atomic animation having NA duration. The fourth atomic animation gets the same duration as the longest of the atomic animations with a known duration.

```
R> natrac <- trac(a, b, c, f)
```

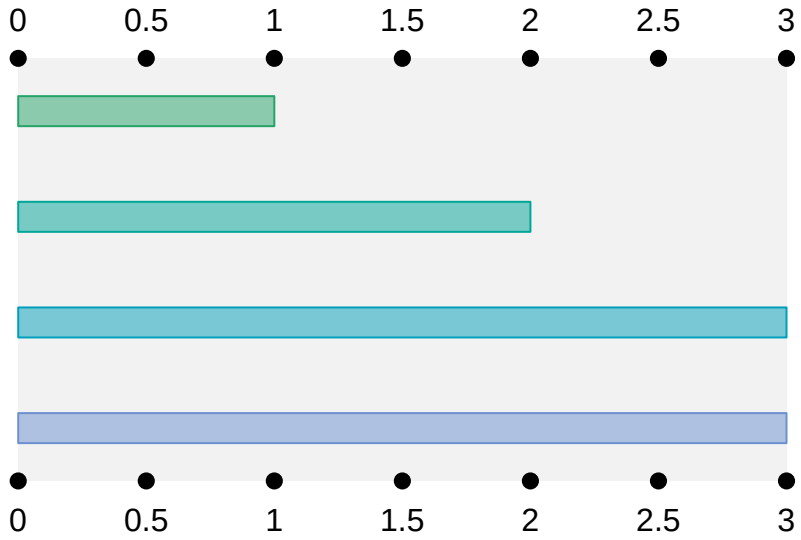


Figure 6.13: A track animation consisting of four atomic animations, where the fourth atomic animation has a duration of `NA`, so its duration is calculated from the longest of the atomic animations with a known duration.

6.1.6 Operations on containers

Several useful manipulations are possible on container animations. For example, there is a `rep()` method to generate regular patterns of animation actions. The following code shows a sequence animation being repeated three times (see Figure 6.14).

```
R> vRep <- rep(v, 3)
```

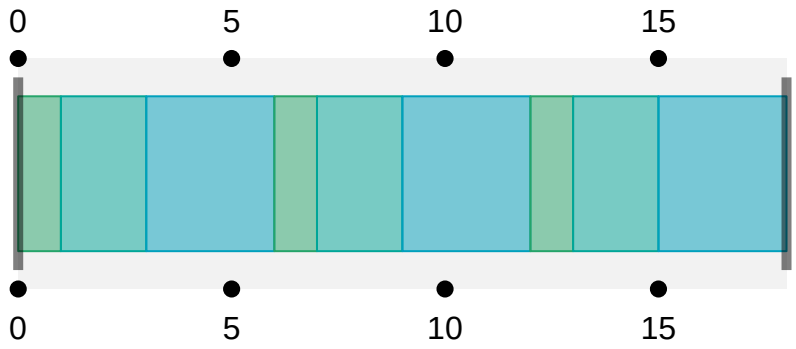


Figure 6.14: A sequence animation (consisting of three atomic animations) repeated three times.

Notice that the result of repeating a container is just a container with the original *contents* repeated (not a container with the original container repeated). The result of repeating an atomic animation is special and it produces a container animation (by default a sequence, but a boolean *vec* argument allows the result to be a track animation instead).

It is also possible to subset a container. Single square brackets produce a smaller version of the original container, consisting of a subset of the original contents, and double square brackets produce a single item from the original contents. The following code shows various subsets being generated from a sequence animation.

```
R> vRep
Delta:0:1|Echo:0:2|Foxtrot:0:3|Delta:0:1|Echo:0:2|Foxtrot:0:3|Del...
R> vRep[2]
Echo:0:2
R> class(vRep[2])
[1] "vecAnim"       "containerAnim" "anim"
R> vRep[[2]]
Echo:0:2
R> class(vRep[[2]])
[1] "atomicAnim" "anim"
R> vRep[3:5]
Foxtrot:0:3|Delta:0:1|Echo:0:2
R> vRep[c(TRUE, FALSE)]
Delta:0:1|Foxtrot:0:3|Echo:0:2|Delta:0:1|Foxtrot:0:3
R> vRep[rep(2:3, 2)]
Echo:0:2|Foxtrot:0:3|Echo:0:2|Foxtrot:0:3
```

Assignment to a subset of a container animation is also possible, using both single and double square brackets. The value being assigned must be compatible with the container so, for example, only a sequence animation can be assigned to a subset of a sequence animation, and an atomic animation may only be assigned using double square brackets.

Finally, there is a `splice()` function for inserting new content into a container animation. Both atomic animations and container animations may be inserted into a container. For example, the following code appends an atomic animation onto the end of a sequence animation of three atomic animations to make a sequence of four atomic animations (see Figure 6.15) and then splices in a track animation after position 2 in the sequence (see Figure 6.16).

```
R> e <- atomic(durn = 1)
R> vAppend <- splice(v, e)
```

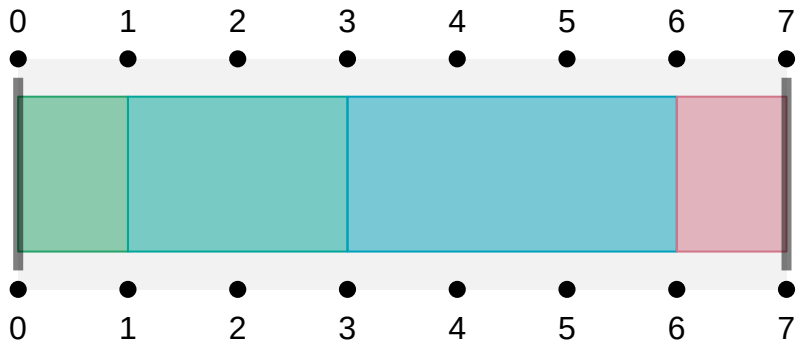


Figure 6.15: A sequence animation (consisting of three atomic animations) with a fourth atomic animation appended.

```
R> vSplice <- splice(vAppend, t, after = 2)
```

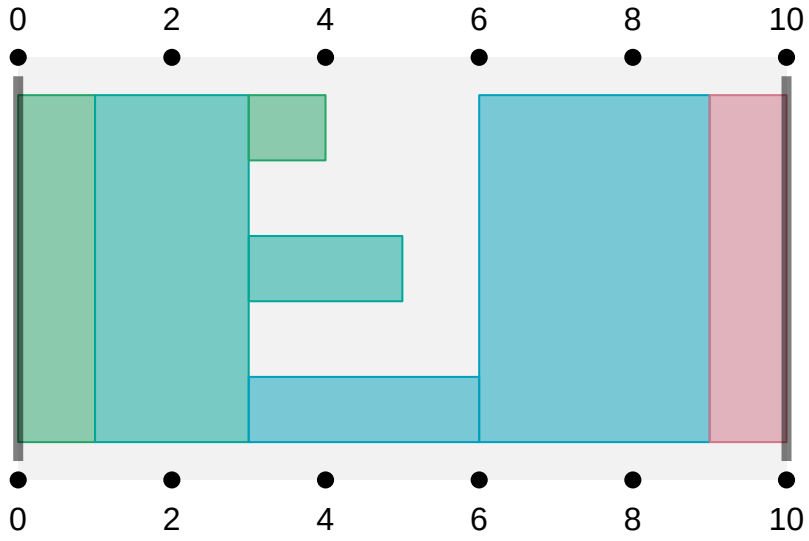


Figure 6.16: A sequence animation (consisting of four atomic animations) with a track animation spliced in after the second atomic animation.

In addition to the `after` argument to `splice()`, which allows animations to be inserted in between existing animations in a sequence or track, there is also an `at` argument to

`splice()`, which allows animations to be inserted *alongside* existing animations in a sequence or track. For example, the following code splices an atomic animation into a sequence animation at position 2. The result is a sequence that splits into two tracks at position 2; one track has the original remainder of the sequence and the other track has the spliced in atomic animation (see Figure 6.17).

```
R> sequence <- vec(a, b, c)
R> newSeq <- splice(sequence, e, at = 2)
```

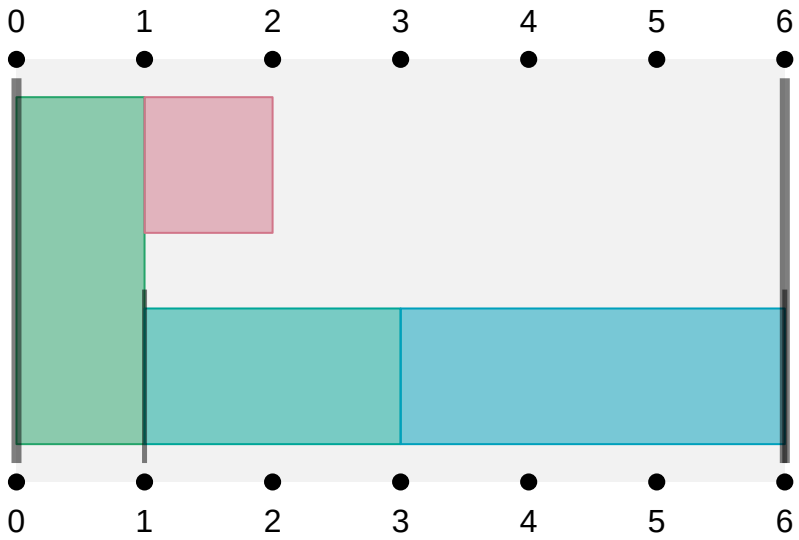


Figure 6.17: A sequence animation (consisting of three atomic animations) with a new atomic animation spliced in alongside the second atomic animation in the original sequence.

6.1.7 Timing schemes

The ultimate purpose of constructing an animation description is to be able to coordinate the timing of multiple animation actions. The `timing()` function is provided to turn an animation description into a timing scheme. The following code shows the simplest case for an atomic animation. This shows the basic information that is produced for each atomic animation: the label of the animation, when it starts and its duration (the animation diagram is also shown in Figure 6.18 for comparison).

```
R> timing(a)
  label start durn vec vecNum trac tracNum
1 Delta 0     1
```

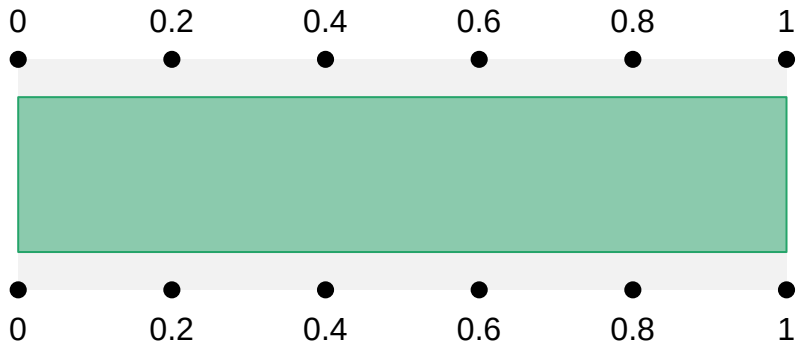


Figure 6.18: A simple atomic animation that starts at after 0 seconds and lasts for 1 second.

The next example shows the timing information from a sequence animation (consisting of three atomic animations; see Figure 6.19). The first thing to note is that there is a line of timing information for each atomic animation. Secondly, in addition to the basic start and duration information, there is information about the *context* in which the atomic animation occurs. In particular, the context tells us the sequence animation and the track animation in which each atomic animation occurs, plus which iteration within the sequence animation and which track within the track animation the atomic animation represents. In this case, the atomic animations are the first, second, and third iterations within a single sequence animation (and there are no tracks).

```
R> timing(v)
  label   start durn vec  vecNum trac tracNum
1 Delta   0     1    Golf 1
2 Echo    1     2    Golf 2
3 Foxtrot 3     3    Golf 3
```

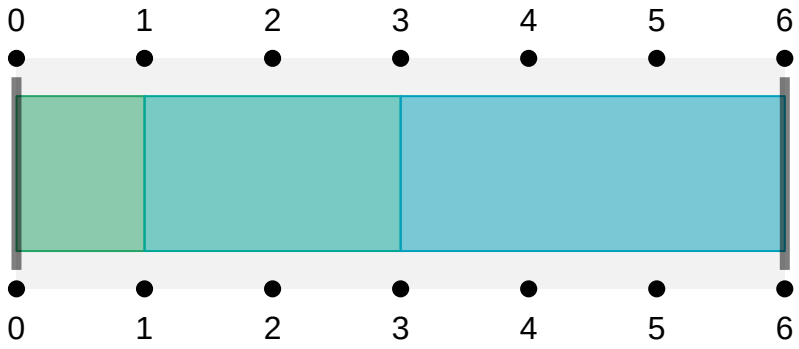


Figure 6.19: A simple sequence animation consisting of three atomic animations.

The next example shows a more complex animation, consisting of a sequence animation made up from a sequence animation followed by a track animation (see Figure 6.20). One complication is that there are now many atomic animations to give timing information for. There is also a lot more context information to convey for each atomic animation. For example, the third atomic animation starts after 0 seconds and lasts for 3 seconds. This atomic animation is part of the first iteration within a sequence *and* it is on the third track of a track animation. The fourth atomic animation starts after 3 seconds and lasts for 1 second. This atomic animation is the first iteration within a sequence, which is itself the second iteration within a parent sequence.

```
R> timing(container)
  label   start durn vec        vecNum trac  tracNum
1 Delta   0     1    India      1      Hotel 1
2 Echo    0     2    India      1      Hotel 2
3 Foxtrot 0     3    India      1      Hotel 3
4 Delta   3     1    India:Golf 2:1
5 Echo    4     2    India:Golf 2:2
6 Foxtrot 6     3    India:Golf 2:3
```

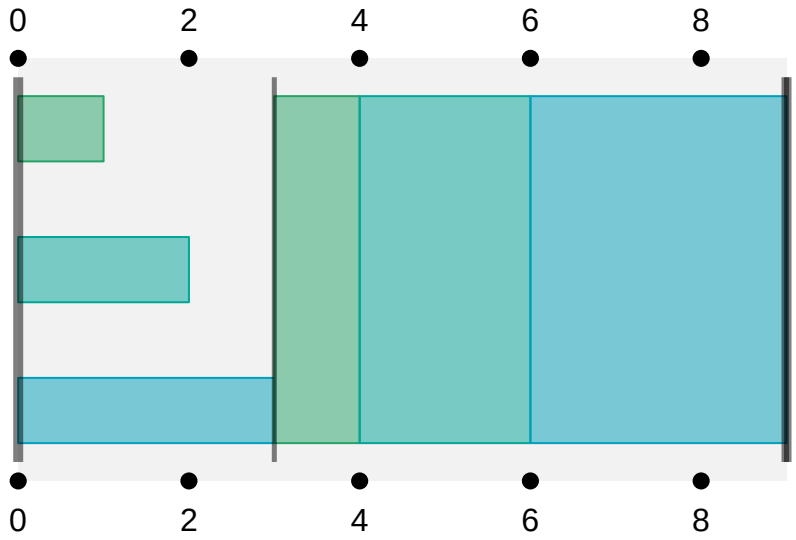


Figure 6.20: A sequence animation that consists of a track animation followed by a sequence animation.

Although the timing information for this example is relatively complex, it is still straightforward to extract the basic information for any atomic animation. The information for the overall animation is structured as a list, with one component for each atomic animation. The information for each atomic animation is itself just a list of seven components. For example, the following code shows how we can easily extract just the start time and duration for the fourth atomic animation.

```
R> tim <- timing(container)
R> tim[[4]]$start
[1] 3
R> tim[[4]]$durn
[1] 1
```

It is also important to note that the context information is stored as a *vector* of information for each atomic animation. For example, the following code shows the sequence context for the fourth atomic animation.

```
R> tim[[4]]$vec
[1] "India" "Golf"
R> tim[[4]]$vecNum
[1] 2 1
```

It is therefore possible to use this contextual information programmatically to determine useful information about when within a larger animation a particular atomic animation occurs.

Another way to access timing information is to ask which atomic animations are *active* at a particular time point. This sort of information is more useful for generating an animation as a series of static frames because it provides information about what to draw within each frame.

The `frameTiming()` function returns timing information for atomic animations that are active at a specific time point. For example, the following code tells us that, after 1.5 seconds of the `container` sequence animation, atomic animations `b` and `c` are active (within track 2 and track 3 of the first step in the parent sequence, respectively; see Figure 6.20) and after 5 seconds only `b` is active (this time as step 2 of the sequence within step 2 of the parent sequence see Figure 6.20).

```
R> frameTiming(container, 1.5)
  label   start durn vec   vecNum trac  tracNum
1 Echo    0     2    India 1      Hotel 2
2 Foxtrot 0     3    India 1      Hotel 3
R> frameTiming(container, 5)
  label start durn vec        vecNum trac tracNum
1 Echo  4     2    India:Golf 2:2
```

A `frameApply()` is also provided to call a user-specified function on the timing information for an animation at a set of time points (frames) spanning the entire animation. The default is just to print the timing information for each frame, but this can be used to draw the content for each frame (an example is given in subsection 6.1.9). The following code shows the default output from `frameApply()` for the simple sequence animation `v` with one frame per second.

```
R> capture.output(frameApply(v, pause = FALSE))
 [1] "  label start durn vec  vecNum trac tracNum"
 [2] "1 Delta 0     1    Golf 1                  "
 [3] "  label start durn vec  vecNum trac tracNum"
 [4] "1 Echo  1     2    Golf 2                  "
 [5] "  label start durn vec  vecNum trac tracNum"
 [6] "1 Echo  1     2    Golf 2                  "
 [7] "  label   start durn vec  vecNum trac tracNum"
```

```
 [8] "1 Foxtrot 3     3    Golf 3                   "
 [9] "  label   start durn vec  vecNum trac tracNum"
[10] "1 Foxtrot 3     3    Golf 3                   "
[11] "  label   start durn vec  vecNum trac tracNum"
[12] "1 Foxtrot 3     3    Golf 3                   "
```

6.1.8 Drawing animation diagrams

A `plot()` method is defined for animations and this was used to produce all of the figures in this report. There is also a `dynPlot()` function, which draws a dynamic (SVG) version of the diagram in which each rectangle is animated appropriately.

6.1.9 Examples

This subsection provides some demonstrations of how these functions could be used to construct animations.

The first example shows a sequence being constructed from many repetitions of an atomic animation. The idea here is that there is an atomic animation that is demonstrated a few times slowly, then again a bit faster, and a third time very fast. The atomic animation is identical each time and the container varies the duration (see Figure 6.21).

```
R> r <- rep(a, 15)
R> durn(r) <- rep(c(10, 5, 1), each = 5)
```

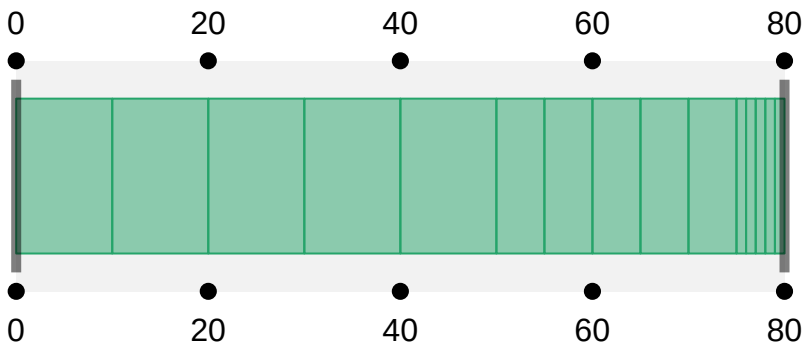


Figure 6.21: A sequence animation (consisting of four atomic animations) with a track animation spliced in after the second atomic animation.

A gridSVG example

The gridSVG package includes the `grid.animate()` function, which enables us to declaratively embed animations within an SVG image. In order to demonstrate how the animaker can be used in conjunction with gridSVG, we will perform the following animation:

1. Begin with a black rectangle called `mainrect` at the left of the page.
2. Move the rectangle from the left of the page to the right of the page in two seconds.
3. Change the fill colour to white and move from the right of the page to the centre of the page in 1 second.
4. Change the fill color to red and end the animation after 1 second.

This can be described in animaker with the following sequence:

```
R> moveRight <- atomic(durn = 2, label = "moveRight")
R> changeAndMove <- atomic(durn = 1, label = "changeAndMove")
R> fillRed <- atomic(durn = 1, label = "fillRed")
R> completeAnim <- vec(moveRight, changeAndMove, fillRed,
R+                     label = "completeAnim")
```

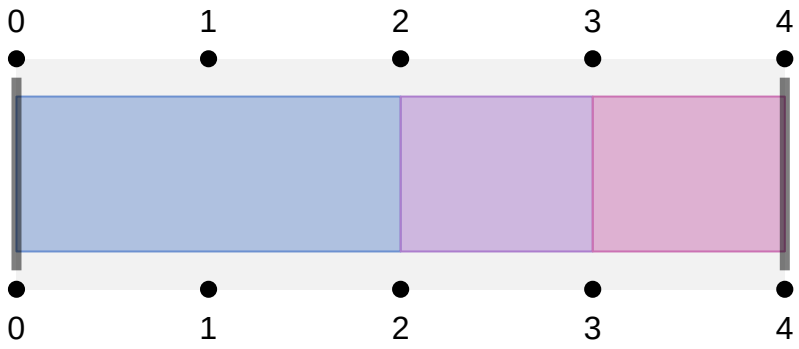


Figure 6.22: Describing an example animation that is a sequence animation composed of three atomic animations.

The animation sequence has now been described, but we have yet to write any actions for this sequence. Each action function should take a single parameter, which is a timing object (i.e. the result of calling `timing()` on an animation). This object contains all the information that an animation function needs to know to apply its action. It therefore requires no hard-coding of any timing information.

```
R> # Action functions that apply each step of the animation
R> moveRect <- function(info) {
R+     grid.animate("mainrect",
R+                  x = c(0.2, 0.8),
R+                  begin = info$start,
R+                  duration = info$durn)
R+ }
R> moveAndFillWhite <- function(info) {
R+     grid.animate("mainrect",
R+                  x = c(0.8, 0.5),
R+                  fill = "white",
R+                  begin = info$start,
R+                  duration = info$durn)
R+ }
R> fillRed <- function(info) {
R+     grid.animate("mainrect",
R+                  fill = "red",
R+                  begin = info$start,
R+                  duration = info$durn)
R+ }
R> # Wrapper function that selects which action function to call
R> animate <- function(info) {
R+     if (info$label == "moveRight")
R+         moveRect(info)
R+     if (info$label == "changeAndMove")
R+         moveAndFillWhite(info)
R+     if (info$label == "fillRed")
R+         fillRed(info)
R+ }
```

We can now apply these actions and then export to SVG to view the animation sequence.

```
R> # Drawing our starting rectangle
R> grid.rect(x = 0.2, name = "mainrect",
R+           width = unit(1, "inches"), height = unit(1, "inches"),
```

```
R+            gp = gpar(fill = "black"))
```

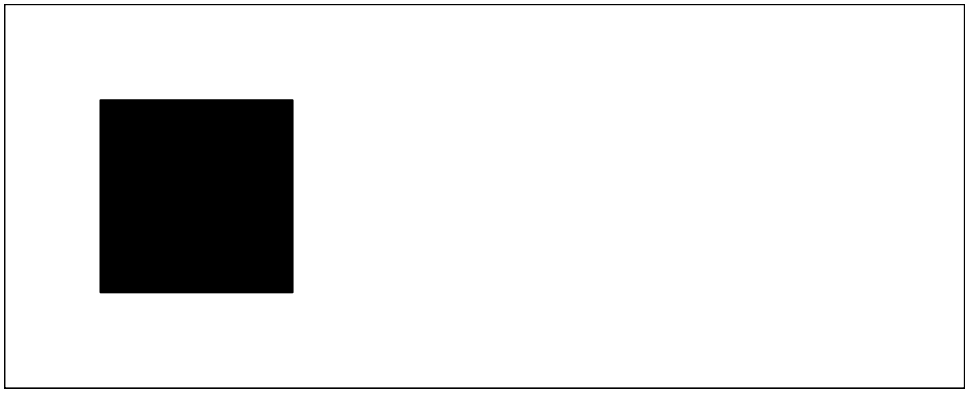
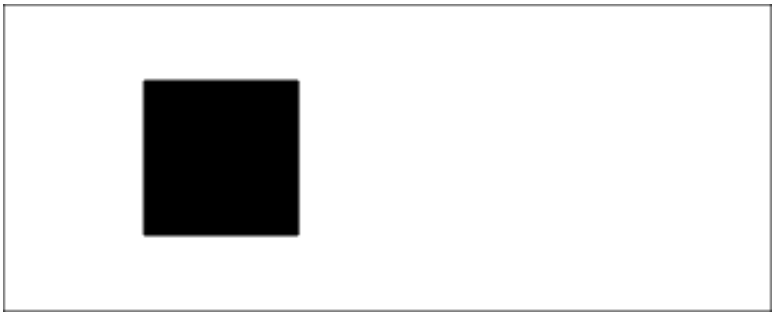


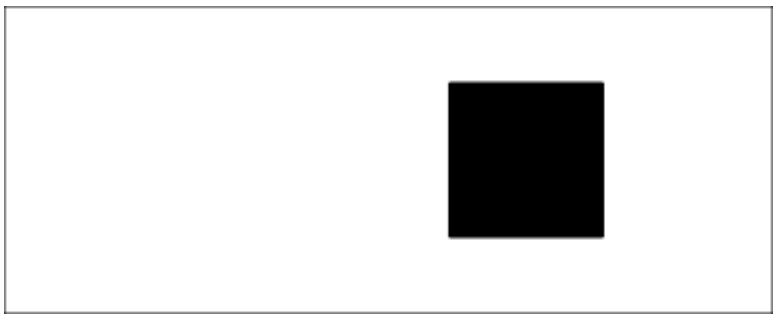
Figure 6.23: The starting point for our example animation sequence.

```
R> # Applying animation functions using timing information
R> tim <- timing(completeAnim)
R> invisible(lapply(tim, animate))
R> # Exporting
R> grid.export("animaker-gridsvg-example.svg")
```

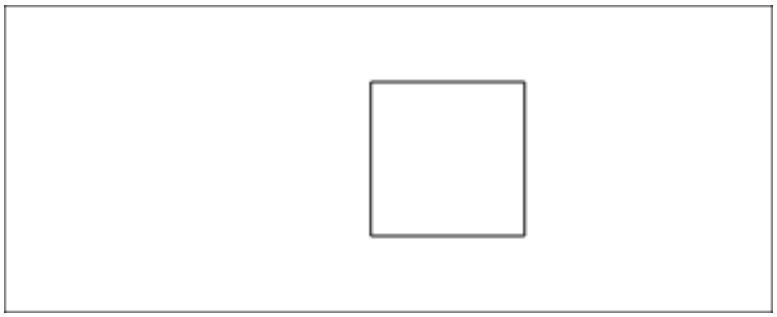
Screenshots of the resulting animation can be seen in Figure 6.24.



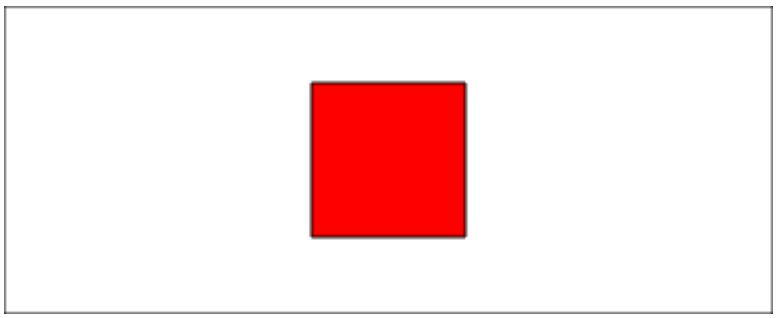
(6.24a) The start of the animation where the rectangle has just started moving to the right.



(6.24b) The rectangle has almost reached the right-hand side of the page.



(6.24c) The rectangle has reached the right-hand side of the page, where it was filled with white colour. It is now is moving left.



(6.24d) The rectangle has moved to the centre and has now been filled red.

Figure 6.24: Screenshots of a simple animation described by animaker and applied by gridSVG.

The overall timing of this animation example is not rocket science, but the animaker package makes the task simpler, encourages separation between the timing of the animation steps and the actions that occur in each step, and encourages the animation "director" to break an animation into smaller, simpler steps.

An animation example

We will revisit the example shown in Figure 6.1.9 by creating a frame-based implementation using the animation package. Instead of relying on a timing scheme created by `timing()`, we use the information returned from the `frameTiming()` function.

The existing animation description enables us to generate timing information for individual frames. For example, the following code produces timing information for frames after 0 seconds, 2 seconds, and 3 seconds, which shows how different animation actions appear in different frames.

```
R> frameTiming(completeAnim)
  label     start durn vec          vecNum trac tracNum
1 moveRight 0     2    completeAnim 1
R> frameTiming(completeAnim, 2)
  label         start durn vec          vecNum trac tracNum
1 changeAndMove 2     1    completeAnim 2
R> frameTiming(completeAnim, 3)
  label   start durn vec          vecNum trac tracNum
1 fillRed 3     1    completeAnim 3
```

We now need functions that can make use of the timing information to draw a frame. These functions contain calls to the `grid.rect()` function to draw the animated rectangle for each frame.

```
R> # global variables that each frame might like
R> # to know about
R> fps <- 10
R> frameNumber <- 0
R> # Action functions
R> drawRect <- function(info) {
R+     rectx <- seq(0.2, 0.8, length.out = fps * info$durn)
R+     rectx <- rectx[frameNumber - (fps * info$start)]
R+     grid.rect(x = rectx,
R+               width = unit(1, "inches"),
R+               height = unit(1, "inches"),
R+               gp = gpar(fill = "black"))
R+ }
R> drawRectThenFillWhite <- function(info) {
```

```
R+     rectx <- seq(0.8, 0.5, length.out = fps * info$durn)
R+     rectx <- rectx[frameNumber - (fps * info$start)]
R+     grid.rect(x = rectx,
R+               width = unit(1, "inches"),
R+               height = unit(1, "inches"),
R+               gp = gpar(fill = "white"))
R+ }
R> fillRectRed <- function(info) {
R+     grid.rect(x = 0.5,
R+               width = unit(1, "inches"),
R+               height = unit(1, "inches"),
R+               gp = gpar(fill = "red"))
R+ }
```

These functions (e.g. `drawRect()` are only ever drawing a single rectangle each time they are called. They are just a bit more complicated because they need to select which rectangle to draw, which are based on starting times and durations (provided by animaker), and the current frame number and number of frames per second.

With these functions defined, an overall function can be created to draw just the components that are active in a frame. This function makes use of the fact that we chose labels for the atomic actions to match the names of the functions that draw the different components of a frame.

```
R> # Delegates frame drawing at each time step
R> drawFrame <- function(info) {
R+     frameNumber <<- frameNumber + 1
R+     grid.newpage()
R+     lapply(info, function(x) {
R+         if (x$label == "moveRight")
R+             drawRect(x)
R+         if (x$label == "changeAndMove")
R+             drawRectThenFillWhite(x)
R+         if (x$label == "fillRed")
R+             fillRectRed(x)
R+     })
R+ }
```

We can now play the animation by calling `frameApply()`. This calls our `drawFrame()` function for each of a series of animation frames. The `fps` argument specifies that there should be ten frames drawn per second.

```
R> frameApply(completeAnim, drawFrame, fps = 10)
```

Functions from the animation package can be used to generate an animated movie from this sequence of frames. For producing a movie we use more frames per second to get a smoother result and we do not bother to pause between generating frames.

```
R> # Reset the frame counter
R> frameNumber <- 0
R> # Now save via the animation package
R> library(animation)
R> saveGIF(frameApply(completeAnim, drawFrame,
R+                    fps = fps, pause = FALSE),
R+         interval = 1 / fps, ani.width = 400, ani.height = 200)
```

The result of this code is that we get an animated GIF image that shows our animation sequence. This is not shown because the images would appear the same as in Figure 6.24. The only difference being that we had to procedurally draw each frame, so the rectangle's movement across the page is not fluid.

The timing of this animation makes the contribution of the animaker package more significant in this case. This is because at no point did any of the frames need to hard-code their start or duration times. It is also worth noting that the separation between the overall timing of the animation and the individual animation actions is again useful, particularly when experimenting with different timings. All that was required was to change the animation descriptions that were shown earlier, everything else flowed from that.

6.1.10 Conclusion

This section describes an R package that provides a convenient and flexible way to build descriptions of animation sequences. An animation description can be used to generate a timing scheme that contains information about the start and duration of every action in the animation (or for every frame in the animation).

The main functions in the package are:

- `atomic()` to describe a single step in an animation.

- `vec()` to combine animation steps in series.
- `trac()` to combine animation steps in parallel.
- `timing()` to generate a timing scheme for an animation.

Further functions are provided to manipulate animation descriptions and there are functions for drawing diagrams to help visualise animations descriptions.

Acknowledgements

The concepts of sequence tracks and sequence animations derive their intellectual ancestry from various systems that can be used to coordinate multimedia resources. These include Microsoft Movie Maker, Microsoft PowerPoint, Adobe Flash, and several JavaScript libraries: Timeline.js (Ignac, 2001), burst-core (Waldron, 2010), and Mozilla's Popcorn.js (Mozilla Foundation, 2013).

6.2 Applying Animation Sequences in JavaScript

The animaker package can be used to describe an animation sequence, which allows us to focus on when animation actions should occur. It does not express any description of an animation action, only representing them by labels. The key result from describing an animation sequence is the ability to generate timing schemes from it. These timing schemes tell us when an animation is to occur, for how long, and what its context is. For example, the context can tell us whether an animation action is the first, second, or third action within a sequence of actions.

The timing information that is generated by animaker can be exported for use within a web browser and the TimingManager library provides us with an easy way of using this timing information.

This does not allow us to modify exported animations but it enables us to use the animations created by the animaker package. animaker tells us when an animation occurs, and its duration, but it does not assign actions to animations. The TimingManager library not only assigns actions to animation descriptions, it also allows us to play back animations in a simple manner.

We demonstrate how TimingManager can be used for both declarative animation and iterative animation by frames. The declarative animation is demonstrated via D3 and its use of SVG and CSS transitions. The framed animation is performed by the HTML

<canvas> element, which uses a painter-model approach, rather than SVG's object-based approach to image construction.

6.2.1 Exporting an Animation Sequence

An example showing how animaker and TimingManager can be used together will be demonstrated by creating a simple plot where we have 3 squares that move around and change colour. The aim is to perform the following animation sequence:

1. Move a red square from left to right. This starts immediately and lasts for 1 second. The square will turn black once it has completed moving.

2. Move a green square from right to left. This starts 1 second after the previous animation and lasts for 1 second. The square will turn black once it has completed moving.

3. Move a blue square from the left to the middle of the plot. This starts 1 second after the previous animation and lasts for 1 second. The square will turn black once it has completed moving.

4. All squares will then move into the centre of the plot, also transitioning to white. This starts 1 second after the previous animation and lasts for 3 seconds.

Now that we have described this animation sequence, we first need to describe the timings of the animation sequence using animaker within R.

```
R> library(animaker)
R> redSq <- atomic(label = "red", durn = 1)
R> greenSq <- atomic(label = "green", start = 1, durn = 1)
R> blueSq <- atomic(label = "blue", start = 1, durn = 1)
R> final <- atomic(label = "final", start = 1, durn = 3)
R> # Combine animations into a sequence
R> completeAnim <- vec(redSq, greenSq, blueSq, final,
R+                     label = "complete")
```

A description of the animation has now been created. This enables us to create a timing scheme that tells us when each action is to be called, along with the context in which it is being applied.

```
R> timing(completeAnim)
  label start durn vec      vecNum trac tracNum
1 red   0     1    complete 1
2 green 2     1    complete 2
3 blue  4     1    complete 3
4 final 6     3    complete 4
```

We can see what this timing information looks like by calling `plot()` on our animation description. This is shown in Figure 6.25.

```
R> plot(completeAnim)
```

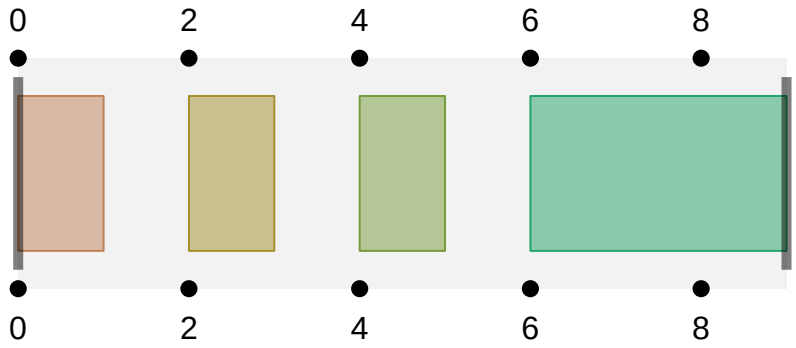


Figure 6.25: A sequence animation composed of four atomic animations.

Now that the description of the animation sequence has been created, we need to be able to export it into a form that a web browser can use. In order to perform this task, animaker uses the RJSONIO package (Temple Lang, 2013). This package translates R data structures into JSON equivalents. This means we can access the data in JavaScript that has been exported from R in a natural manner. animaker can export animation timings to a JavaScript variable that is assigned timing information. We do this by calling the `export()` function, as shown below.

```
R> export(completeAnim,
R+        jsVar = "simpleAnim",
R+        file = "simple-anim-timing.js")
```

What is happening here is that the timing information is going to be represented as a JavaScript object that is assigned to a variable called `simpleAnim`. The resulting code is saved to a file called `simple-anim-timing.js`. The first 20 lines of the file

`simple-anim-timing.js` are the following:

```
var simpleAnim = [
    {
        "start" : 0,
        "durn" : 1,
        "label" : "red",
        "vec" : "complete",
        "vecNum" : 1,
        "trac" : null,
        "tracNum" : null
    },
    {
        "start" : 2,
        "durn" : 1,
        "label" : "green",
        "vec" : "complete",
        "vecNum" : 2,
        "trac" : null,
        "tracNum" : null
    },
...
```

What we can see is that there is a close mapping between the timing scheme we printed out earlier and the JSON that was exported from it. The headers in our timing scheme have now become keys in JavaScript objects, each object representing a row in our timing scheme.

We can import this information into the browser using the HTML `<script>` tag. This allows us to make use of TimingManager because we exposed a JavaScript variable, `simpleAnim`, which contains all of the necessary timing information.

6.2.2 Using Timing Information in the Browser

In order to use the exported timing information, we will first begin by instantiating a TimingManager object in JavaScript. There are two parameters of interest when constructing this object; the first of which is simply the timing information that we have previously stored in a JavaScript variable. The second parameter is simply noting that we are using seconds as the base unit of time.

```
JS> var tm = new TimingManager(simpleAnim, "s");
```

Next we create JavaScript functions that we will assign as actions to our atomic animations. For clarity, the code used to generate the initial scene has been omitted. Note that each of the following functions takes a single parameter which contains the information about the animation as it is being called. In this case, the timing information is used to set the duration of each animation.

```
JS> var redAction = function(info) {
JS+     d3.select("#redsq")
JS+         .transition()
JS+         .duration(info.durn * 1000)
JS+         .attr("x", 400)
JS+         .transition()
JS+         .attr("fill", "black");
JS+ };
JS> var greenAction = function(info) {
JS+     d3.select("#greensq")
JS+         .transition()
JS+         .duration(info.durn * 1000)
JS+         .attr("x", 100)
JS+         .transition()
JS+         .attr("fill", "black");
JS+ };
JS> var blueAction = function(info) {
JS+     d3.select("#bluesq")
JS+         .transition()
JS+         .duration(info.durn * 1000)
JS+         .attr("x", 250)
JS+         .transition()
JS+         .attr("fill", "black");
JS+ };
JS> var finalAction = function(info) {
JS+     d3.select("#redsq, #greensq, #bluesq")
JS+         .transition()
JS+         .duration(info.durn * 1000)
```

```
JS+         .attr({
JS+             x: 250,
JS+             y: 250,
JS+             fill: "white"
JS+         });
JS+ };
```

The only timing calculation we needed to make is to convert seconds to milliseconds. This is because D3, like almost all time related JavaScript code, uses milliseconds as its primary unit of time. Although the D3 code shown above is what we're going to be using to perform animation, it is not strictly necessary to understand it, only the resulting actions it performs.

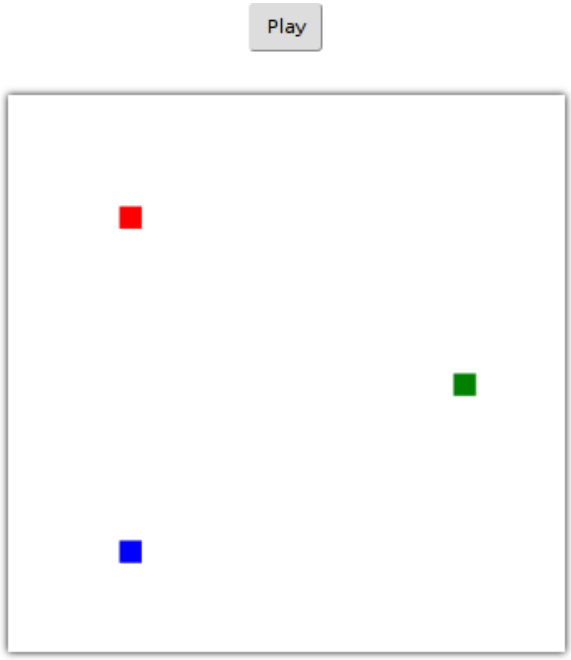
In order to bind actions to animations, we need to *register* the actions within TimingManager. To do this, we simply build a JavaScript object that has animation labels as its keys, and the associated actions as its values. This is shown below:

```
JS> tm.register({
JS+     red: redAction,
JS+     green: greenAction,
JS+     blue: blueAction,
JS+     final: finalAction
JS+ });
```

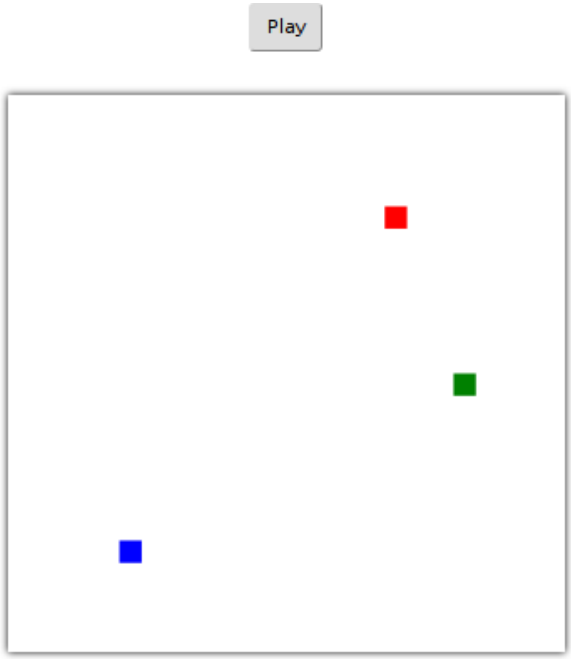
Now that the actions have been *registered* to atomic animations, we can *play* them.

```
JS> tm.play();
```

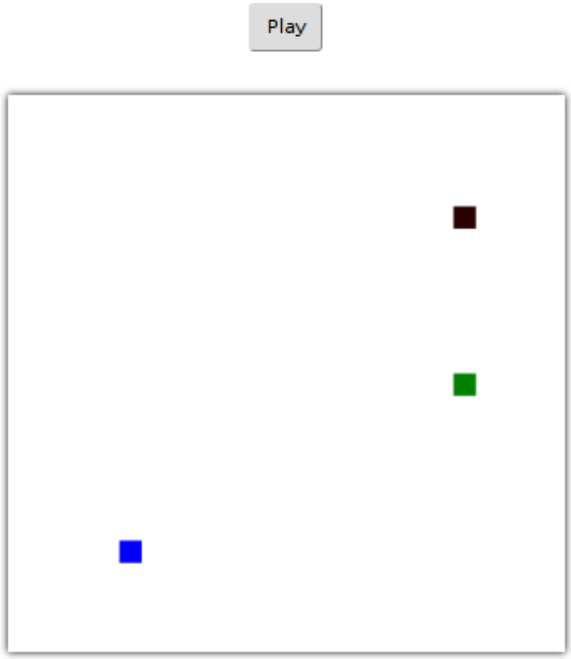
When `play()` is called, TimingManager begins to call the animation actions at the correct times. The final implementation of the animation sequence is shown in Figure 6.26. Because the animations and indeed the JavaScript code runs in a web browser, screenshots have been taken. Consequently, it must be assumed that the images capture the described functionality. In a web browser, the animation sequence begins by clicking the "Play" button.



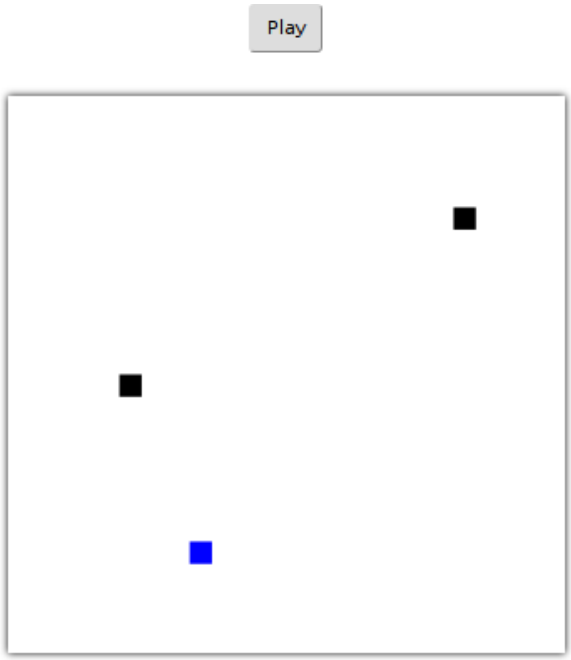
(6.26a) The web page prior to clicking the “Play” button.



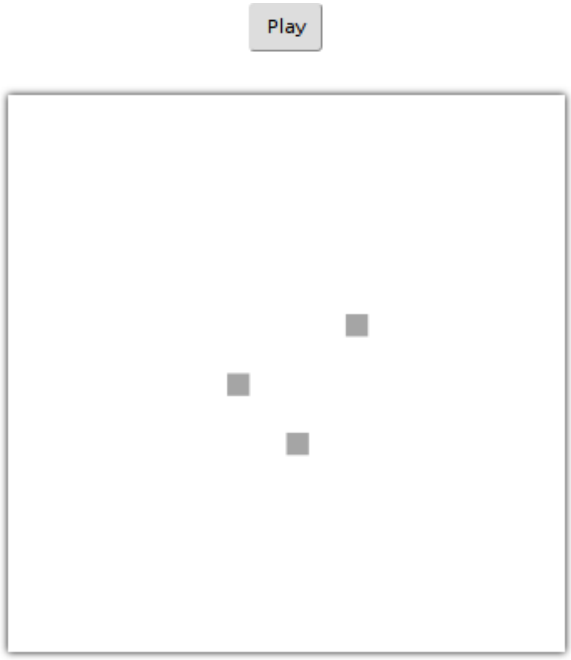
(6.26b) The red square moving to the right.



(6.26c) The red square has finished moving and has now turned black.



(6.26d) The green square has moved to the right and turned black. The blue square is currently moving to the right.



(6.26e) All three squares moving towards the centre of the screen and fading to white.

Figure 6.26: An animation sequenced with animaker and applied in JavaScript with TimingManager and D3.

6.2.3 Complex Examples

More complex examples will be shown that use R to describe animation sequences with animaker, but do not require R to produce any graphics. The first of these examples is an equivalent of animaker's `dynPlot()` function. `dynPlot()` produces an animated plot showing visually when each atomic animation is due to be called. The second is a quick demonstration of frame-based animation in the browser.

Animated Timing Plot

To begin, we will first create an animation and export it from R. We first need to load the animaker package.

```
R> library(animaker)
```

By taking some of the code from section 6.1, we can reconstruct an animation.

```
R> a <- atomic(label = "Alpha", durn = 1)
R> b <- atomic(label = "Bravo", durn = 2)
```

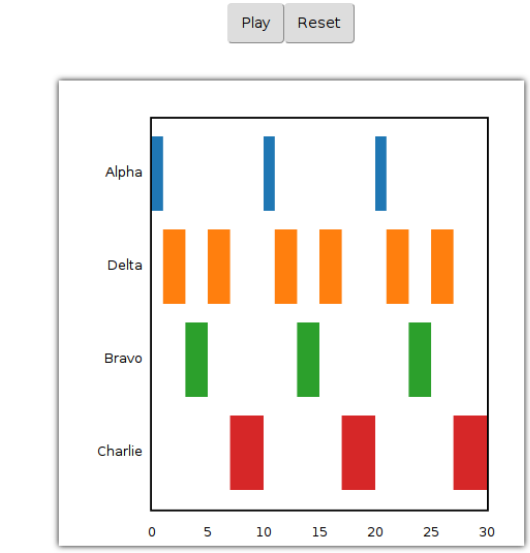
```
R> c <- atomic(label = "Charlie", durn = 3)
R> f <- atomic(label = "Delta", durn = NA)
R> navec <- vec(a, f, b, f, c, durn = 10)
R> # We'll have this repeated 3 times
R> navec <- rep(navec, 3)
```

We can export the timing information from this animation sequence into a form that can be used in the browser.

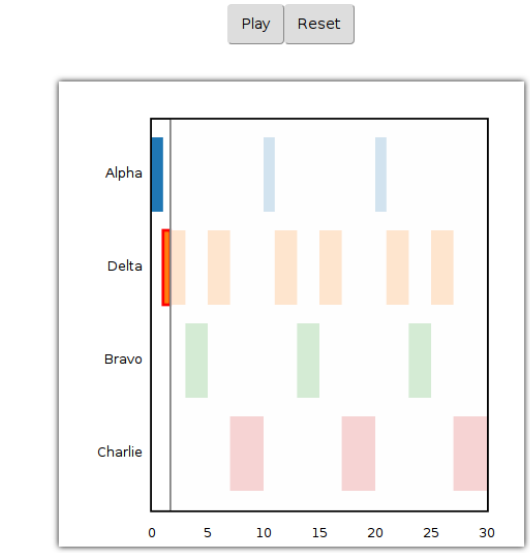
```
R> # Store timing information
R> timingData <- timing(navec)
R> # Turn it into a JS variable and save
R> export(timingData,
R+        jsVar = "timingData",
R+        file = "timing-data.js")
```

The file `timing-data.js` can now be used within an HTML document so that we can use this timing information within a web browser. Once we have the data exported from R, we can begin to use the timing information in the browser.

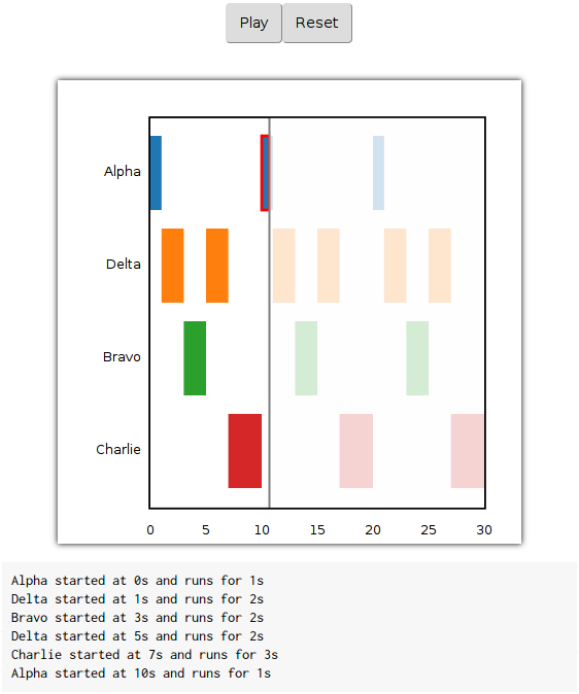
The implementation details have been omitted for brevity but we are able to make use of D3 to create a plot. The fact that this is designed to be viewed in a web page means we can only show screenshots of the timing plot (see Figure 6.27). Note that the *actions* associated with each atomic animation are going to be appending text below the timing plot. By clicking the "Play" button in a web browser we can visualise how the playback is going to occur, along with triggering associated actions.



(6.27a) The timing plot prior to clicking the “Play” button.



(6.27b) Two actions have been performed that append text showing their name, when they started, and how long they run for.



(6.27c) The timing plot as it appears after running for approximately 10 seconds.

Figure 6.27: A timing plot generated from exported timing information.

We can see that actions for each animation are simply to print out their labels, along with the starting times and their durations. This is a simple diagram demonstrating how timing information can be used, but instead of simply appending some text, we could be performing any task that is possible within JavaScript.

Framed Animation

In the case of framed animation, the intention is to provide the illusion of animation by producing several frames close together. This effect is employed in films where 24 individual frames are shown per second, but because they are shown close together, the illusion of a moving picture is produced. In the browser, a natural way of performing this task is by using the <canvas> element. This is mostly useful if we do not have the ability to employ declarative animation, as is the case with R graphics devices.

A frame-based animation has been created, using the same animation sequence description from the previous section. The key difference is that we need to overwrite the actions that are registered with animations, using new action functions. In order to do this, we simply register the actions again, but with an additional parameter, which

determines whether we can overwrite definitions. The following code demonstrates this:

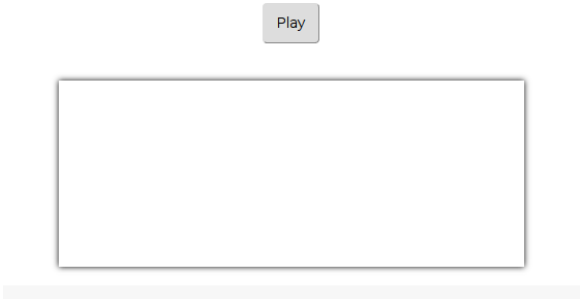
```
JS> tm.register({
JS+     Alpha: alphaFrameAction
JS+     Bravo: bravoFrameAction
JS+     Charlie: charlieFrameAction
JS+     Delta: deltaFrameAction
JS+ }, true);
```

For the sake of brevity, we will not show the definitions of each of the action functions. However, their definitions are such that they aim to simply draw a filled rectangle when they are called. Additionally, they append text to a field below the plot, so we can see when the "frames" are being drawn.

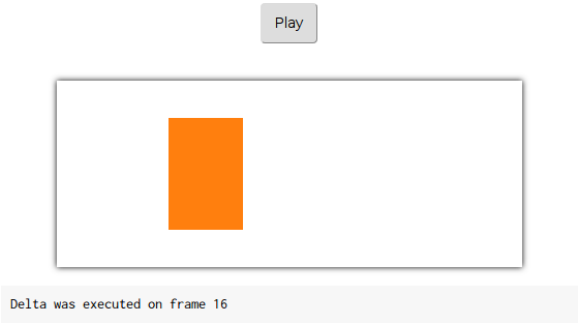
Now that we have our timing information and our actions registered we can play our frame-based animation. To do this, we call the `frameApply()` method. It requires a single parameter, which represents the number of frames to draw per second.

```
JS> tm.frameApply(10);
```

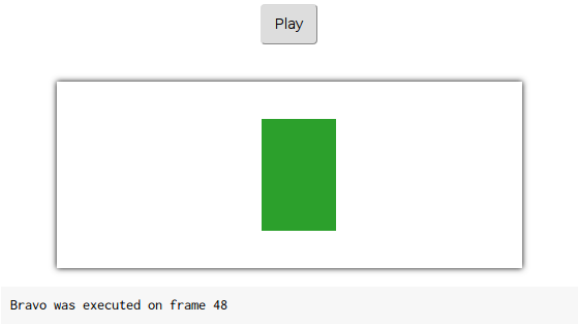
The result of this function call is that animation actions are repeatedly called for the duration of their actions, rather than just once at the starting point of the animation. The resulting animation that has been constructed is shown in Figure 6.28.



(6.28a) The animation prior to clicking the "Play" button.



(6.28b) The “Delta” action is being executed on frame 16.



(6.28c) The “Bravo” action being executed on frame 48.

Figure 6.28: An example of a <canvas> based animation.

Instead of each action describing how to perform an animation, the action simply draws to a canvas. We can also see that a frame counter is being shown to make it clear that we really are drawing multiple times, even if it doesn’t appear to be the case.

An example beyond this simple demonstration for frame-based animations has not been created. The main reason for this is because in a web browser we usually have the capacity to apply declarative animations. Declarative animations should be preferred to frame-based animations primarily because they are much simpler to write. Another reason is because web browsers are able to hardware accelerate declarative animations, whereas frame-based animations do not gain this benefit. As a result, declarative animations are almost always going to look more pleasant than an equivalent frame-based approach.

6.2.4 Conclusion

This has been a quick demonstration to show how we can use both framed animation and declarative animation within a web browser. The use of R’s animaker package and TimingManager provide a convenient interface to describe and apply animations

respectively. It allows us to separate the task of describing *when* things are happening from the task of describing *what* the animations are aiming to show.

7 Advanced SVG Features

Much of the development of gridSVG has focused on translating grid objects to their SVG equivalents. In order to take advantage of SVG features, grid objects can be annotated with additional information. This is how animation and hyperlinking can be generated from a grid image. These are certainly useful features to have but there are many features of SVG that are not possible to implement in grid due to limitations in the R graphics engine. This requires gridSVG to take a different approach to SVG translation than usual; rather than translating grid concepts to SVG, gridSVG attempts to translate SVG concepts to grid.

The ability to use features of SVG that are not possible with the R graphics engine allows a user to create more sophisticated statistical graphics for use within a web browser. gridSVG provides an R interface to SVG features that many other graphics systems either do not provide, or require knowledge of SVG to use.

7.1 Patterns

In grid, a grob is only able to be filled with a solid colour, optionally with some transparency. SVG allows any graphical element to be filled not only with a solid colour, but also with a pattern. This means that, for example, a rectangle can be filled with a pattern "tile" that is repeated horizontally and vertically to fill the entire rectangle. An example of this is shown in Figure 7.1.

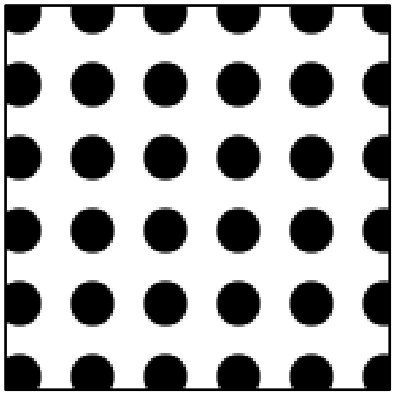


Figure 7.1: A repeating pattern applied to a rectangle.

In the example shown in Figure 7.1 we can see that a single rectangle has been filled with a repeating pattern, that pattern being a single black circle. In order to draw this image, a three step process needed to be followed: the drawing of a graphics object, pattern definition, and pattern application. First, we'll draw a rectangle with the name `myrect`.

```
R> grid.rect(width = 2/3, height = 2/3, name = "myrect")
```

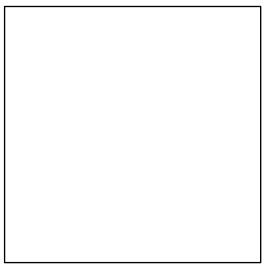


Figure 7.2: A plain rectangle named "myrect".

```
R> grid.ls()
myrect
```

The rectangle that has been drawn in Figure 7.2 can now be seen on the display list. This means that we can now apply a pattern fill to the rectangle simply by referring to its name, `myrect`. To apply a pattern fill we first need to define a pattern, which requires three components to be specified. These components are the following: a grob, which describes what the pattern looks like, dimensions for the size of the pattern tile as it is being used, and finally the dimensions for the tile as it is being defined.

To better explain the difference between the dimensions of a pattern tile as it is being used (the pattern size) and the dimensions of the tile as it is being defined (the device size), an illustration has been drawn in Figure 7.3.

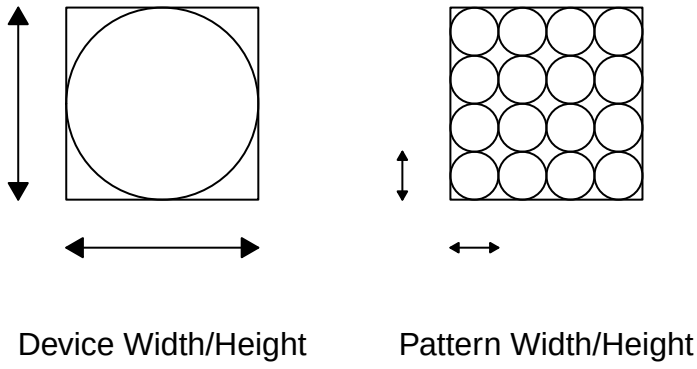


Figure 7.3: The device dimensions determine how the pattern tile is drawn. The width and height describe the size of the pattern tile when it is used.

Figure 7.3 shows how a simple circle grob has been drawn onto a new device as the definition of a pattern. How the pattern definition appears may depend on the size of the device if absolute units are in use, e.g. centimetres, inches. Once the pattern tile has been defined, it can then be applied as a pattern fill to a grob. When the pattern is applied to a grob, the pattern height and width determines the size of the tile as it is being used. We will revisit the earlier example to show how the pattern was defined.

```
R> pat <-
R+     pattern(grob = circleGrob(r = 0.3, gp = gpar(fill="black")),
R+             # Define *pattern* dimensions
R+             width = unit(0.1, "npc"), height = unit(0.1, "npc"),
R+             # Define *device* dimensions (in inches)
R+             dev.width = 7, dev.height = 7)
```

Here we see that the circle was drawn onto a 7 inch by 7 inch canvas as the pattern definition. When the pattern is used, each tile will be one tenth the height and width of the current viewport (in this case that means our entire image).

Now that we have a pattern object, `pat`, we can use this to apply it to our rectangle, `myrect` using gridSVG's `grid.patternFill()` function.

```
grid.patternFill("myrect", pattern = pat)
```

We cannot see that the pattern has been applied until we export the image using `grid.export()`. The result of this operation is shown in Figure 7.4.

```
R> grid.export("pattern-ex.svg")
```

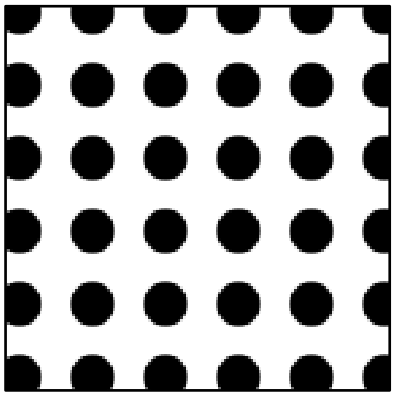


Figure 7.4: A rectangle with a simple pattern applied to it.

The previous example was just a simple demonstration to show how patterns can be constructed and used. However, there are cases where patterns are useful such as showing group classification when colour alone is not sufficient. They may also be useful

for ensuring that a plot can safely be printed by a monochrome printer or when making considerations for colourblindness. Consider the plot displayed in Figure 7.5 which could benefit from the use of patterns.

```
R> # Create means for each group
R> plotdat <- aggregate(hp ~ gear + vs, data = mtcars, mean)
R> plotdat$gear <- factor(plotdat$gear)
R> plotdat$vs <- factor(plotdat$vs)
R> # Let's have a look at the data
R> plotdat
  gear vs    hp
1    3  0 194.2
2    4  0 110.0
3    5  0 216.2
4    3  1 104.0
5    4  1  85.4
6    5  1 113.0
R> # Plot a barchart
R> library(lattice)
R> barchart(hp ~ gear, groups = vs, data = plotdat,
R+          xlab = "Number of Gears", ylab = "Mean Horsepower",
R+          auto.key = list(space = "right"), horizontal = FALSE)
```

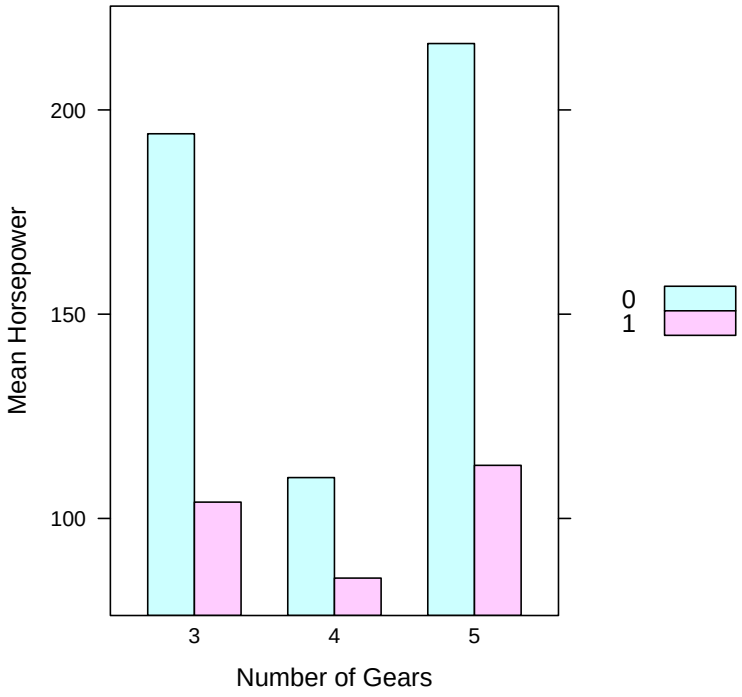


Figure 7.5: A simple lattice bar chart.

The plot in Figure 7.5 uses light green and light red as the differentiating colours for group membership in the "V/S" group. Instead of using colours, patterns can be used to show group membership. This bar chart was created using the lattice package (Sarkar, 2008), which uses the grid graphics system, meaning that we can modify it using gridSVG.

A key advantage of lattice is that it has a clear and well-defined naming scheme (Murrell, 2012b) that allows us to easily identify a component of a plot. To demonstrate this, we will show a relevant subset of the names of objects on the grid display list.

```
R> grid.ls()
...
plot_01.barchart.x.1.rect.panel.1.1
plot_01.barchart.x.2.rect.panel.1.1
plot_01.barchart.x.3.rect.panel.1.1
plot_01.border.panel.1.1
plot_01.key.frame
  GRID.cellGrob.35
```

```
    plot_01.key.background
  GRID.cellGrob.36
    plot_01.key.text.1.1
  GRID.cellGrob.37
    plot_01.key.text.1.2
  GRID.cellGrob.38
    plot_01.key.rect.2.1
  GRID.cellGrob.39
    plot_01.key.rect.2.2
```

The output from `grid.ls()` shows that the bars in the bar chart are drawn as three rectangle grobs, named `plot_01.barchart.x.1.rect.panel.1.1`, `plot_01.barchart.x.2.rect.panel.1.1` and `plot_01.barchart.x.3.rect.panel.1.1`. In the legend, each of the keys for the values of "V/S" are named `plot_01.key.rect.2.1` and `plot_01.key.rect.2.2`.

With the naming information in hand, we can now begin to define patterns that will be applied to each of these drawn grobs. We will define two patterns to show group membership: for the value of "0" we will draw a pattern that will appear as repeated diagonal lines from bottom-left to top-right, for the value of "1" we will draw polka-dot pattern. Additionally, these patterns will be sized to be quite small at 2mm wide by 2mm high so that they repeat often.

```
R> val0pat <- pattern(linesGrob(),
R+                    width = unit(2, "mm"), height = unit(2, "mm"),
R+                    dev.width = 1, dev.height = 1)
R> val1pat <- pattern(circleGrob(r = 0.25,
R+                               gp = gpar(fill = "black")),
R+                    width = unit(2, "mm"), height = unit(2, "mm"),
R+                    dev.width = 1, dev.height = 1)
```

Now that they have been defined, they need to be registered so that we apply them by label. This will be explained in more detail in section 7.5. The registered patterns will then be applied, and we know how to apply the patterns to grobs because we can reference them by name.

```
R> # Registering patterns
R> registerPatternFill("val0pat", val0pat)
R> registerPatternFill("val1pat", val1pat)
```

```
R> # Applying pattern fills
R> barNames <- paste0("plot_01.barchart.x.", 1:3,
R+                    ".rect.panel.1.1")
R> for (i in 1:3) {
R+     grid.patternFill(barNames[i],
R+                      label = c("val0pat", "val1pat"),
R+                      group = FALSE)
R+ }
R> # Applying legend pattern fills
R> legendNames <- paste0("plot_01.key.rect.2.", 1:2)
R> grid.patternFill(legendNames[1], label = "val0pat",
R+                  group = FALSE)
R> grid.patternFill(legendNames[2], label = "val1pat",
R+                  group = FALSE)
R> grid.export("pattern-barchart.svg")
```

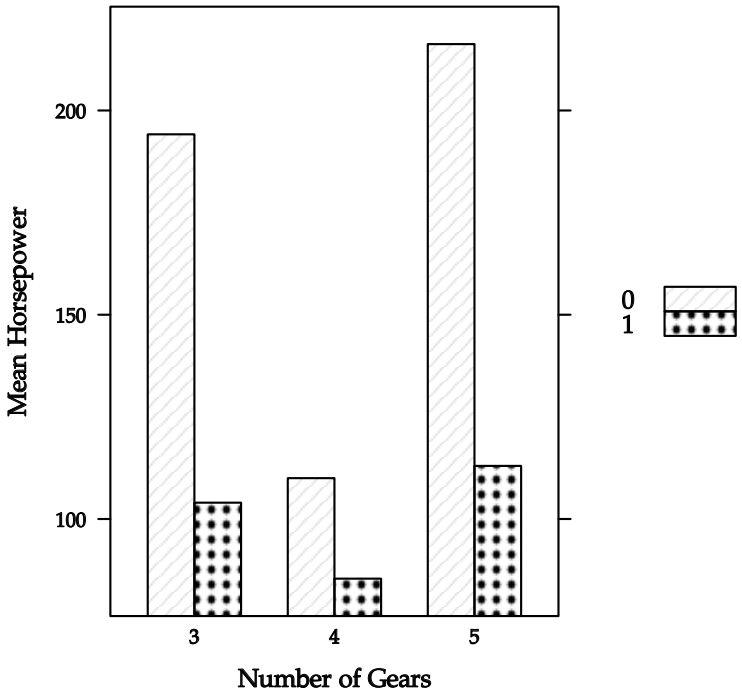


Figure 7.6: A lattice bar chart with patterns applied for each group.

To explain Figure 7.6 in more detail, when `grid.patternFill()` is being called the first argument we give it is the name of the grob that we want to fill with a pattern. The `label` argument refers to the pattern we registered earlier (either `val0pat` or `val1pat`), while the `group` argument being set to `FALSE` means that we want the pattern to be applied to the SVG `<rect>` element produced by gridSVG and not the grob's grouping element (see section 2.4 for more details).

Instead of using colour to differentiate between two groups, gridSVG has demonstrated an alternative option for performing this task by using patterns in SVG.

7.2 Gradients

Gradients are often used to show a linear colour scale in plots, particularly when the data the scale describes is a continuous variable. SVG allows us to create both linear and radial gradients in a declarative manner and gridSVG provides a high-level interface for creating and using SVG gradients.

7.2.1 Linear Gradients

Linear gradients are often used in existing statistical software, particularly when drawing plots such as two-dimensional density plots. Examples of gradients being used are shown in Figure 7.7, taken from the lattice and ggplot2 packages.

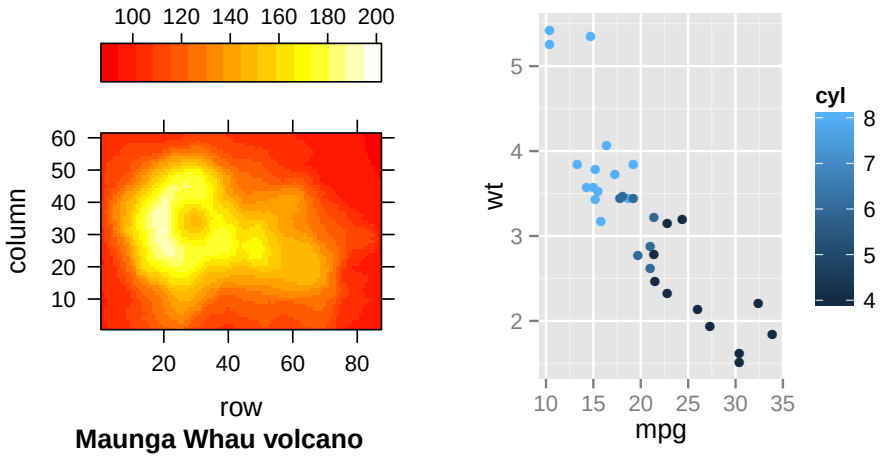


Figure 7.7: A lattice plot and a ggplot2 plot, both using gradients.

The first thing to consider when creating a linear gradient is what units are used to position gradient stops. They can either be relative to the filled graphics object or relative to a viewport's coordinate system. For example, consider a rectangle that is 5 centimetres wide, and another that is 10 centimetres wide. If we allow a gradient to be relative to the referring object, then when the same gradient is applied to both rectangles, the gradient is scaled to be twice as wide on the 10cm wide rectangle than the 5cm wide rectangle. The other alternative is to position the gradient relative to the coordinate system, in which case the gradient will appear exactly the same on both rectangles, but the smaller rectangle may not be able to display as much of the gradient as the larger rectangle can. The difference between the two alternatives is illustrated in Figure 7.8.

Figure 7.8: The two options for positioning gradients.

By default, gridSVG will scale the linear gradient so that the start of the gradient and the end of the gradient cover the entire graphical object that is being filled.

Once it has been established whether our gradients are going to be relative to our grobs relative or to our viewport coordinate system we can begin to define our gradients. The `x0`, `x1`, `y0`, and `y1` arguments to our gradient function are particularly important because they define three things: position, magnitude, and direction.

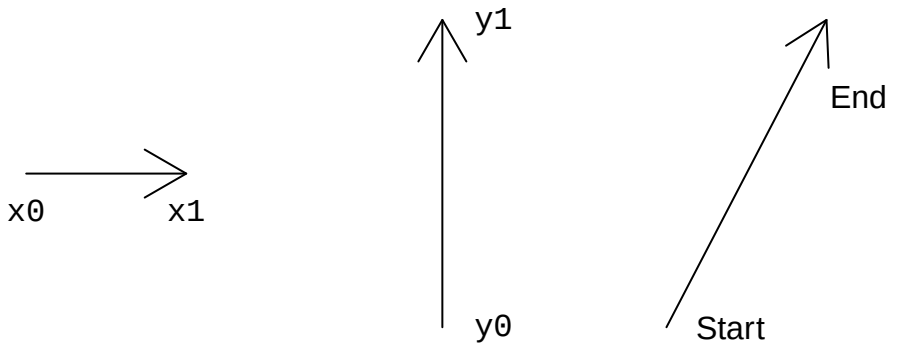


Figure 7.9: Defining a gradient vector using four points.

The x and y vectors not only define a direction, but also a magnitude. When these vectors are added together, the result is the vector shown on the right of Figure 7.9. When the gradient is defined to be relative to a grob these vectors will be converted to normalised parent coordinates. This means that the location $(0, 0)$ refers to the bottom-left corner of the grob and $(1, 1)$ refers to the top-right corner. If we are absolutely positioning a gradient, then all `grid` units can be used.

In addition to x and y vectors, a gradient also requires gradient stops. Gradient stops are colours defined at control points that a gradient must interpolate through. Each gradient stop must define both a colour and a position. For example, a gradient could start at the colour black, then at halfway it must be red, then at the end of the gradient it is white. This example uses three gradient stops and is illustrated in Figure 7.10.

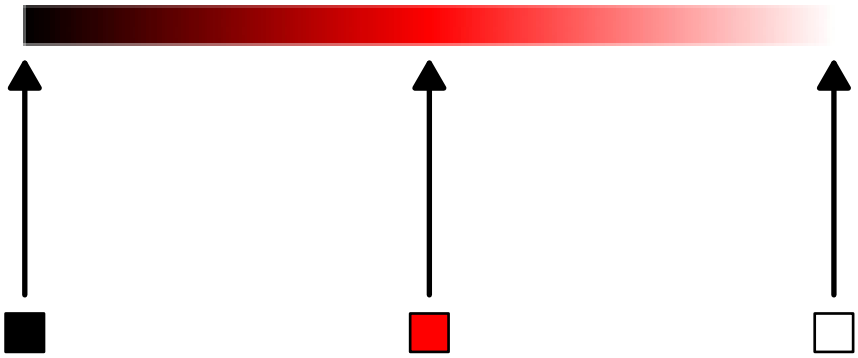


Figure 7.10: A linear gradient featuring three gradient stops.

Gradient stops give us a way of specifying the colours used in a gradient but the way in which they are positioned requires further explanation. We have already stated how the start and end points of the gradient can be positioned relative to a graphics object, or relative to a viewport coordinate system. This means that a gradient stop needs to be positioned relative to the gradient as it is being used. The way in which this is done in gridSVG is to provide a relative location for each gradient stop. Locations can range from (but are not limited to) 0 to 1, where 0 represents the start of the gradient and 1 is the end of the gradient. We can now see how the gradient stops were positioned for the example shown earlier: black at 0, red at 0.5 and white at 1.

A final consideration with the definition of linear gradients is when the gradient region is not defined for the the region filling the entire referring graphics object. In other words, what should appear when a gradient defines colours for a region shorter than the entire length of our fill region? SVG gradients can handle this case using what is known as a "spread method". There are three possible options:

Pad Use the colours at the ends of the gradient to fill the remaining region.

Reflect Reflect the gradient pattern start-to-end, end-to-start, start-to-end, etc. continuously until the remaining region is filled.

Repeat Repeat the gradient pattern start-to-end, start-to-end, start-to-end, etc. continuously until the remaining region is filled.

To show how these might be used, consider the earlier example where we had three gradient stops. If we define the gradient region to only cover half the width of the referring rectangle, we are left with a fill region that is only defined between the start and halfway points. With this incomplete gradient fill, we can see the effect of the different spread methods.

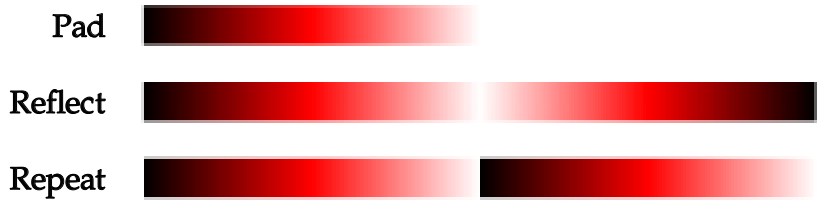


Figure 7.11: The effect of the spread method on gradients.

We have covered the key components of a linear gradient: the gradient region (and its direction), gradient stops, and the spread method. To demonstrate how to apply a linear gradient in gridSVG, we will attempt to fill a simple rectangle. This rectangle will be filled with black, then blue, then white and the gradient will be applied from the top-left to the bottom-right of the rectangle. The first thing we will do is draw the rectangle, shown in Figure 7.12.

```
R> grid.rect(name = "myrect")
```

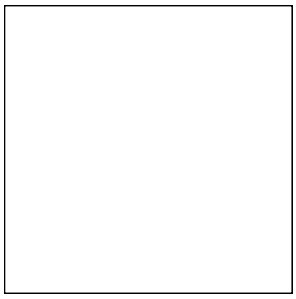


Figure 7.12: A simple grid rectangle.

The rectangle is now on grid's display list, meaning that we can refer to the name `myrect`, and apply a gradient fill to it. Before the gradient can be applied it must first be defined using the `linearGradient()` function.

```
R> myLinGrad <-
R+     linearGradient(c("black", "dodgerblue4", "white"),
R+                    x0 = unit(0, "npc"), x1 = unit(1, "npc"),
R+                    y0 = unit(1, "npc"), y1 = unit(0, "npc"))
```

The linear gradient definition is now stored in the variable `myLinGrad`. One thing to note is that by default the gradient stops are positioned evenly across the gradient region i.e. at the start, halfway and end of the gradient. Given that a linear gradient definition has been created, we can apply it to the rectangle named `myrect`.

```
R> grid.gradientFill("myrect", myLinGrad)
R> grid.export("rect-with-linear-gradient.svg")
```

Figure 7.13: A rectangle with a linear gradient.

In order to apply a gradient fill, all that was necessary was to refer to the drawn grob by name, and also supply the gradient definition as arguments to the `grid.gradientFill()` function. This is just a simple demonstration showing how linear gradients can be defined and applied in gridSVG.

A more complicated example demonstrating the use of linear gradients will now be shown attempting to recreate the lattice plot shown earlier (Figure 7.7), but using SVG gradients instead of rectangles. First, let us draw the lattice plot and display it in Figure 7.14.

```
R> library(lattice)
R> levelplot(volcano, colorkey = list(space = "top"),
R+           sub = "Maunga Whau volcano", aspect = "iso")
```

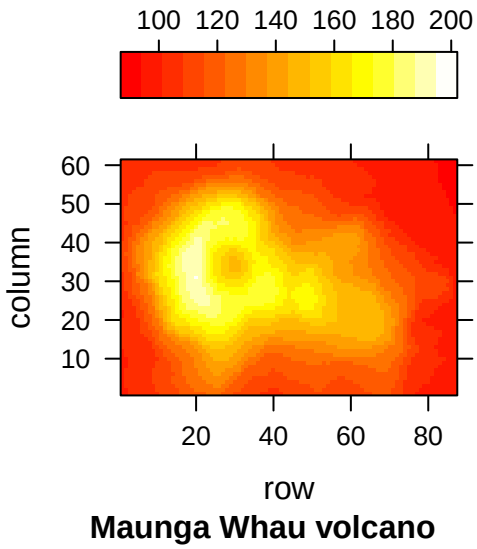


Figure 7.14: A lattice plot featuring a colour scale.

```
R> grid.ls(fullNames = TRUE)
...
frame[plot_01.colorkey.frame]
  cellGrob[GRID.cellGrob.174]
    rect[plot_01.colorkey.image]
  cellGrob[GRID.cellGrob.175]
    rect[plot_01.colorkey.border]
  cellGrob[GRID.cellGrob.176]
    segments[plot_01.colorkey.ticks]
  cellGrob[GRID.cellGrob.177]
    text[plot_01.colorkey.labels]
```

The use of a naming scheme in lattice makes the task of identifying any component of a plot easy. We can observe from the relevant output from `grid.ls()` that the colour scale is stored in a rectangle grob named `plot_01.colorkey.image`. With knowledge of the name of the colour scale, we can collect the information we need to construct a gradient, namely the colours used to fill it.

```
R> rect <- grid.get("plot_01.colorkey.image")
R> gradCols <- rect$gp$fill
R> gradCols
 [1] "#FF0000FF" "#FF1800FF" "#FF2D00FF" "#FF4500FF" "#FF5A00FF"
 [6] "#FF7200FF" "#FF8A00FF" "#FF9F00FF" "#FFB700FF" "#FFCB00FF"
[11] "#FFE300FF" "#FFFC00FF" "#FFFF2EFF" "#FFFF75FF" "#FFFFB3FF"
[16] "#FFFFFAFF"
```

We can see that 16 colours were used to fill the colour scale. It is also necessary to modify the rectangle so that instead of drawing many small rectangles, it draws a single large rectangle. This step is not shown but once completed we can create and apply a linear gradient.

```
R> # Defining the gradient so that it draws from left to right
R> grad <- linearGradient(col = gradCols,
R+                        x0 = unit(0, "npc"), x1 = unit(1, "npc"),
R+                        y0 = unit(0, "npc"), y1 = unit(0, "npc"))
R> # Applying the gradient to our grob
R> grid.gradientFill("plot_01.colorkey.image", grad, group = FALSE)
R> grid.export("lattice-gradient.svg")
```

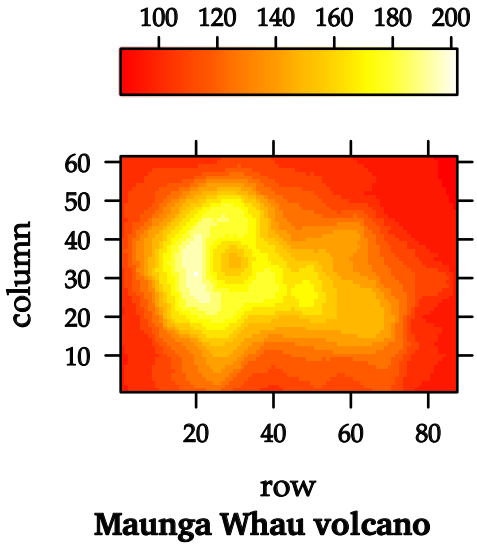


Figure 7.15: A lattice plot with a linear gradient used for a colour scale instead of a series of rectangles.

We can see in Figure 7.15 that this is an improvement over the original implementation because the colour scale is now smooth instead of "blocky". The linear gradient was simple to create because the colours were already available from the existing implementation and the gradient vector was directed from $(0, 0)$ to $(0, 1)$. When applying the gradient fill, the only additional complication was to set the `grid.gradientFill()` function's `group` argument to `FALSE`. If `group` is `TRUE` (this is the default value), then the gradient fill operation is applied to the SVG `<g>` that wraps the grob. In this case because there are already colours applied directly to the rectangle grob, these will take precedence over any applied to the wrapping `<g>` element. By setting `group` to `FALSE`, we are ensuring that the colours on the grobs are directly replaced with a gradient fill.

7.2.2 Radial Gradients

We have discussed linear gradients but SVG also allows us to fill a graphical element with a radial gradient. In many ways these are similar to linear gradients because the way in which gradient stops are defined and the spread methods are exactly the same. Again, like linear gradients, radial gradients can also be relative to a graphics object, or relative to a viewport coordinate system. The only significant difference between the types of gradients is in how the gradient region is defined.

The gradient region for a linear gradient is determined by a vector and its start and

end points; radial gradients instead define their gradient region by a starting location, called the focal point, and the radius of a circle dictates the end of the gradient region.

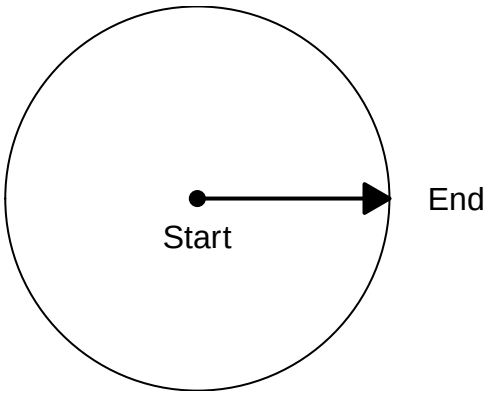


Figure 7.16: A radial gradient starts at the focal point and ends at the edge of the circle.

Additionally, it is not required that the focal point must be in the centre of the circle. It is possible to have the focal point in a non-central location, the effect is shown in Figure 7.17.

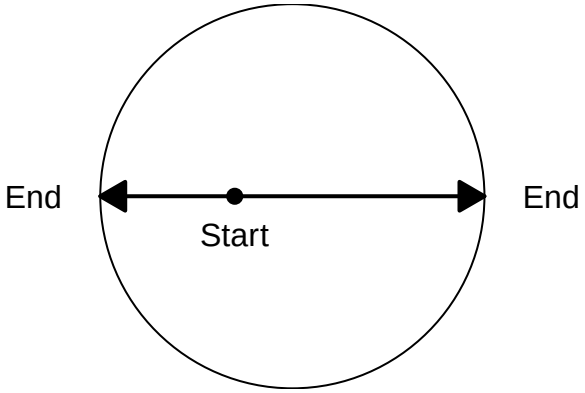


Figure 7.17: A radial gradient with a non-central focal point.

This means that we can have the gradient "radiate" from a non-central focal point. In this example, the gradient will appear more compressed as it radiates left from the focal point in comparison to when it radiates right from the focal point. To demonstrate this, a simple radial gradient will be drawn where the gradient radiates from white to blue to black.

```
R> grid.circle(name = "radial-example")
```

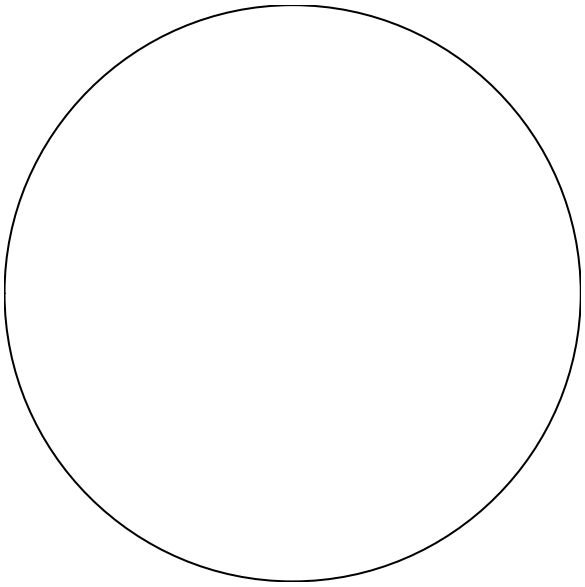


Figure 7.18: A simple grid circle.

```
R> grid.ls()
radial-example
```

We have started by drawing a simple circle in Figure 7.18 named `radial-example`. Indeed, the output from `grid.ls()` tells us that the circle is now on the grid display list because its name is present. A radial gradient definition must also be created in order to draw a gradient. We can do this using the `radialGradient()` function.

```
R> radGradDef <-
R+     radialGradient(c("white", "deepskyblue", "black"),
R+                    # Setting circle location and dimensions
R+                    x = 0.5, y = 0.5, r = 0.5,
R+                    # Setting focal point slightly to the left
R+                    fx = 0.4, fy = 0.5)
```

The definition that we have created is intended to cover the entire circle that we drew earlier. Note that the focal point of this gradient definition is slightly to the left with an x-location of 0.4, compared to a central x-location of 0.5.

```
R> grid.gradientFill("radial-example", radGradDef)
R> grid.export("circle-with-radial-gradient.svg")
```

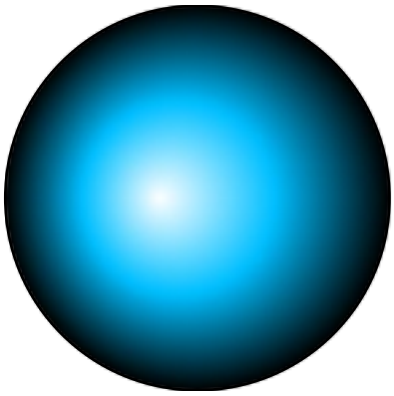


Figure 7.19: The circle from Figure 7.18 with a radial gradient fill applied.

The `grid.gradientFill()` function searches for the grob named `radial-example` and applies a gradient fill using the newly created radial gradient definition `radGradDef`.

This simple example shows how the use of a non-central focal point allows us to create a subtle directed lighting effect on a circle. A more complex example would be to use this effect on each plotting symbol in a grid plot. First, let us draw a plot using the lattice package that is exhibited in Figure 7.20.

```
R> xyplot(mpg ~ wt, data = mtcars, pch = 16, cex = 2)
```

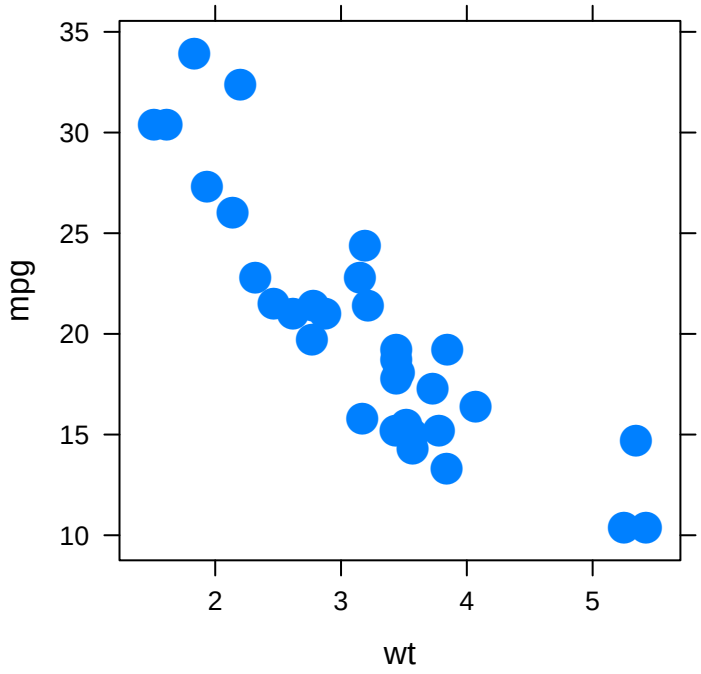


Figure 7.20: A basic lattice scatter plot.

```
R> grid.ls()
plot_01.background
plot_01.xlab
plot_01.ylab
plot_01.ticks.top.panel.1.1
plot_01.ticks.left.panel.1.1
plot_01.ticklabels.left.panel.1.1
plot_01.ticks.bottom.panel.1.1
plot_01.ticklabels.bottom.panel.1.1
plot_01.ticks.right.panel.1.1
plot_01.xyplot.points.panel.1.1
plot_01.border.panel.1.1
```

We can see from the structured labelling of the grid grobs that the points are drawn using the name `plot_01.xyplot.points.panel.1.1`. The points in this plot are also drawn twice as large as normal for emphasis. Rather than having a simple blue colour for

each of these points, we want to apply the radial gradient effect that was shown earlier. Given that we already have a radial gradient definition stored in `radGradDef`, we can apply it to our existing points grob that we know is named `plot_01.xyplot.points.panel.1.1`.

```
R> registerGradientFill("mrg", radGradDef)
R> grid.gradientFill("plot_01.xyplot.points.panel.1.1",
R+                   label = rep("mrg", nrow(mtcars)), group = FALSE)
R> grid.export("points-with-rad-gradient.svg")
```

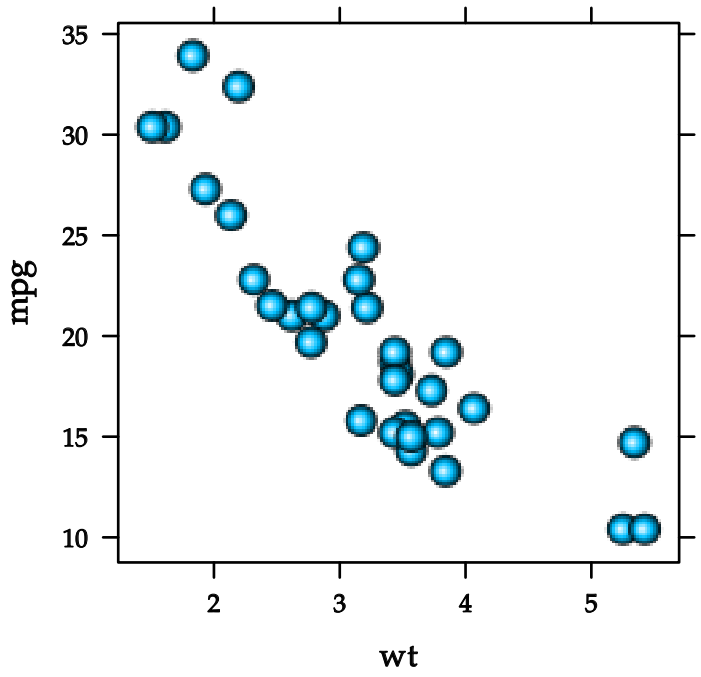


Figure 7.21: The lattice plot from Figure 7.20 with a radial gradient applied to data points.

In the code fragment shown in Figure 7.21, we have performed two key tasks. Firstly, we have registered the radial gradient by giving it a label. For the purposes of this example it simply means that we can use the gradient multiple times. The second task is to apply the radial gradient fill to the plotted points. We want to replace the existing fill colour for all points, which requires that the `group` argument is set to `FALSE`. If this operation is not performed then the gradient fill will be applied to the SVG `<g>` element which wraps all of the points but its effect will not be visible. Additionally, we want the

same gradient fill to be applied to all of the points so the label that the gradient was registered under needs to be repeated for all points and given to the `label` argument.

The final exporting step shows the result of applying the radial gradient fill to the plot and demonstrates a simple example where radial gradients can be utilised.

7.3 Clipping Paths and Masks

7.3.1 Clipping Paths

In grid it is possible to restrict the drawing of grobs to a rectangular region. This region is called the clipping region, and is typically the boundaries of a grid viewport. What this means is any grobs that attempt to draw outside the clipping region will not be visible beyond that region. To demonstrate this concept, a grid plot will be drawn with text intentionally drawn large enough to exceed its clipping region.

```
R> # Creating a new viewport in the middle of the plot
R> pushViewport(viewport(width = 0.75, clip = "on"))
R> # Showing the size of the viewport
R> grid.rect(gp = gpar(lty = "dashed"))
R> # Drawing large text that exceeds the size of the viewport
R> grid.text("hello, world!", gp = gpar(fontsize = 40))
R> # Leaving the viewport
R> popViewport()
```

Figure 7.22: Simple usage of viewport clipping in grid.

Figure 7.22 shows that the text "hello, world!" is not entirely visible because it has been *clipped* to the viewport boundary.

A limitation of R graphics is that clipping paths must be rectangular. This limitation is not present in SVG as it is possible to clip to an arbitrary region, e.g. a circle or a

complex path. What this means is that the example shown earlier could be clipped to any region. To show how this can be done in gridSVG, we shall re-implement the example from Figure 7.22 but will use a circular clipping region instead of a rectangular clipping region. The first thing to do is draw the text that will be clipped.

```
R> grid.text("hello, world!", gp = gpar(fontsize = 36),
R+           name = "example")
```


Figure 7.23: A piece of text that is not yet clipped.

```
R> grid.ls()
example
```

Given that there is a named text grob, we now need to define what the clipping region should look like. This is done via the use of the `clipPath()` function, which takes a grid grob and allows it to be used as a clipping path definition. In our case, we want a grid circle grob to be used as our definition, but with a smaller radius than the entire drawing region.

```
R> circ <- circleGrob(r = 0.3)
R> clipRegion <- clipPath(circ)
```

With a named grob on the display list and a clipping path definition, we can now clip the existing text to our circular region using the `grid.clipPath()` function.

```
R> grid.clipPath("example", clipRegion)
R> # Showing the clipping region (invisible usually)
R> grid.circle(r = 0.3, gp = gpar(lty = "dashed"))
grid.export("text-with-circular-clip.svg")
```

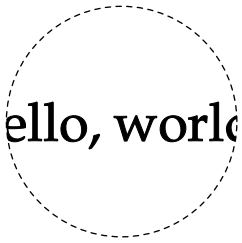


Figure 7.24: The text from Figure 7.23 is clipped to a non-rectangular path.

We have shown in Figure 7.24 that it is possible to clip to a circle, but the fact that SVG allows for most graphical content (including animated content) to be used as a clipping path means that more interesting clipping regions can be defined. The following example will show that a rectangle can be clipped to a region defined by some text, in addition to a circle. Firstly, before this rectangle is clipped, we will first just draw the text and circle grobs to show what the clipping region looks like. Conveniently, grid allows us to store the resulting image by providing the `grid.grab()` function.

```
R> grid.newpage()
R> grid.circle(r = 0.3)
R> grid.text("hello, world!", gp = gpar(fontsize = 24))
R> image <- grid.grab()
```

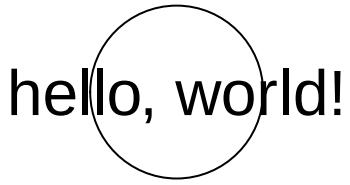


Figure 7.25: The clipping region that will be applied to a grid rectangle.

```
R> # Defining as a clipping region
R> clipRegion <- clipPath(image)
```

Now we can draw the rectangle that will later be clipped.

```
R> grid.newpage()
R> # Starting rectangle that will later be clipped
R> grid.rect(gp = gpar(fill = "grey"), name = "example")
```

Figure 7.26: A grid rectangle that has not been clipped.

Again, in order to apply the clipping path definition we use the `grid.clipPath()` function.

```
R> grid.clipPath("example", clipRegion)
R> grid.export("complex-clip-region.svg")
```

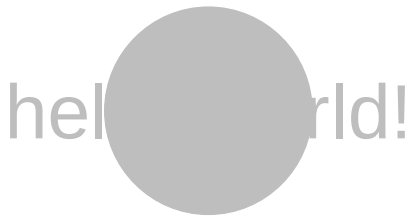


Figure 7.27: The grid rectangle from Figure 7.26 is clipped to the region shown in Figure 7.25.

The rectangle that was drawn has now been clipped to a complicated clipping region. In fact, we can see in Figure 7.27 that the clipping region defined by our circle and text is the union of the area covered by the two grobs. This is a simple demonstration to show how complicated clipping paths can be applied, even if the example is not particularly useful. In practice, non-rectangular clipping paths are beneficial when drawing on geographic maps. Consider Figure 7.28, which shows an example that was contributed to the R Journal (Murrell, 2012a).

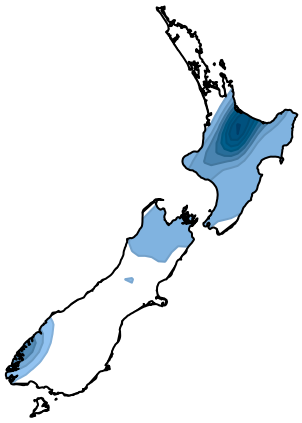


Figure 7.28: A map with an overlaid contour. A path has been used to obscure the contour where it does not overlap with land.

Figure 7.28 shows a contour of earthquake events overlaid across a map of New Zealand. The contours appear to be clipped to the New Zealand coastline but in fact, the contours are not clipped at all. Cleverly, the contours are drawn first and to obscure the unwanted contour regions, the *inverse* of the region covered by the New Zealand coastline is drawn over the top in white. The process that was taken to draw the earlier example is shown in Figure 7.29.

Figure 7.29: A map with a path used to obscure unwanted drawing.

If non-rectangular clipping regions are able to be used instead then this trickery is no longer necessary. We will recreate this example with gridSVG by applying the clipping

functions that we have already described earlier. For the sake of brevity, the code used to prepare and reshape the data will not be shown.

```
R> # Push into a viewport to establish a coordinate system
R> pushViewport(viewport(xscale = xbox, yscale = ybox))
R> # Draw the contours
R> grid.polygon(polyxs, polyys, id.lengths = eachlevel,
R+              name = "contours", default.units = "native",
R+              gp = gpar(fill = fillColours,
R+                        col = adjustcolor(fillColours,
R+                                          1, 0.9, 0.9, 0.9)))
R> # Create a path of the NZ islands
R> nzpath <- pathGrob(pathxs, pathys, id.lengths = groupns,
R+                    default.units = "native")
R> # Draw the path and also clip the contours to it
R> grid.draw(nzpath)
R> grid.clipPath("contours", clipPath(nzpath))
R> popViewport()
R> # Export to SVG
R> grid.export("map-with-clipped-contours.svg")
```

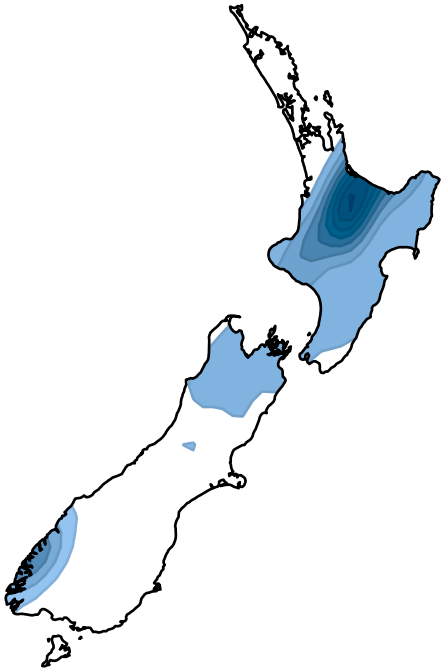


Figure 7.30: Clipping contours to land regions with a non-rectangular clipping path.

We can see in Figure 7.30 that the advantage of the approach provided by gridSVG is that not only do we need to write less code, but also that the code sufficiently captures the intent of what we are trying to draw. Furthermore, the order in which we draw the contours and the map of New Zealand no longer makes a difference because the clipping path is still applied to the contours regardless of the order in which the grobs are drawn.

7.3.2 Masks

In grid it is possible to make any grob semi-transparent in three ways. Firstly, the border on a grob may have a colour with an alpha channel present. Similarly, a grob may be filled with a semi-transparent colour. The final option is to apply additional semi-transparency to both the border colour and the fill colour at the same time. The one thing all approaches have in common is that the opacity they apply is uniform. In other words we cannot have for example a fill opacity that varies across different parts of

the grob. SVG allows for such behaviour to be possible with opacity masks. When a grob is drawn, its opacity can be determined by the opacity defined by a mask.

An opacity mask is a collection of graphical objects whose luminance is transferred to the opacity of another graphics object. This means that when the grobs in the mask are bright, then any grobs using this mask will be opaque where the grobs in the mask are bright. The way masks are defined is that they are assumed to be drawn on a black canvas, a canvas with no luminance present.

To illustrate the concept of masking, we will create a mask that is composed of two grobs: a rectangle and a circle. Both grobs are drawn in Figure 7.31.

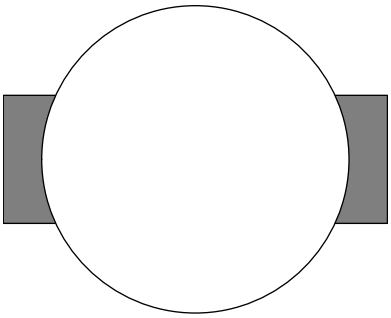


Figure 7.31: Two grid grobs that comprise a mask definition.

It is perhaps more illustrative to show exactly how the mask is defined by drawing on a black background. Any luminance on the page is therefore going to be the relative level of opacity on any grob using this mask.

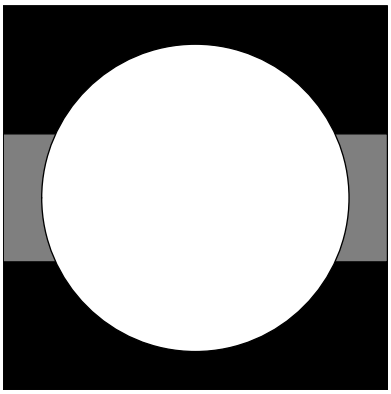


Figure 7.32: The mask in Figure 7.31 reduced to its luminance.

What Figure 7.32 shows is that anything drawn outside the circle and the grey rectangle will not be visible once the mask has been applied. Furthermore, anything within the circle will be opaque and anything within the grey rectangle will have semi-transparency applied. To show how this can be done in gridSVG, we will show the steps necessary to define a mask, and indeed apply it to a grob. We will first start with a black rectangle that will later be masked.

```
R> grid.rect(gp = gpar(fill = "black"),
R+           name = "blackrect")
```

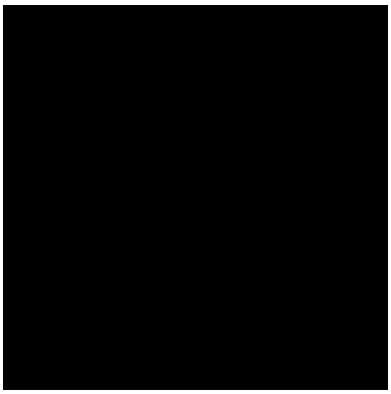


Figure 7.33: A simple black rectangle.

```
R> grid.ls()
blackrect
```

We can now define an opacity mask. For our purposes, all that is necessary is to provide the `mask()` function with a grob that defines what the mask looks like. To do this, we will recreate the mask definition shown earlier.

```
R> # Single grob composed of a rectangle and a circle
R> mygrob <-
R+     gTree(children = gList(
R+         rectGrob(height = 1/3, gp = gpar(fill = "grey50")),
R+         circleGrob(r = 0.4, gp = gpar(fill = "white"))))
R> mymask <- mask(mygrob)
```

All that is needed now is to apply the mask definition to the black rectangle that has already been drawn. This task is performed by the `grid.mask()` function and the result

of masking is shown in Figure 7.34.

```
R> grid.rect(gp = gpar(fill = "black"), name = "blackrect")
R> grid.mask("blackrect", mymask)
R> grid.export("simple-mask.svg")
```

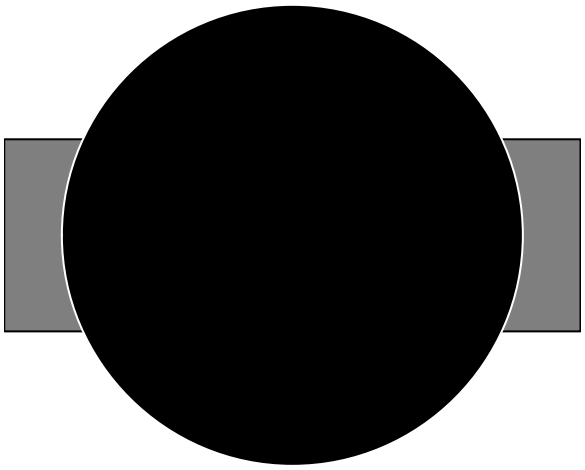


Figure 7.34: The result of masking the rectangle from Figure 7.33 with the mask defined in Figure 7.31.

We can see in Figure 7.34 that masking the rectangle has produced output that is considerably different from the plain black rectangle that was originally visible.

Another feature of masking is that we can also define a rectangular region that controls the area that the mask affects. By default, this region covers the same area as the current viewport. The region of effect is defined as we create the mask definition with the `mask()` function. We will demonstrate this by re-implementing the previous example but having its effect restricted only to the right half of the plot.

```
R> grid.rect(gp = gpar(fill = "black"),
R+           name = "blackrect")
R> mymask <- mask(mygrob, x = 0.5, width = 0.5,
R+                just = "left")
R> grid.mask("blackrect", mymask)
R> grid.export("mask-with-restricted-region.svg")
```

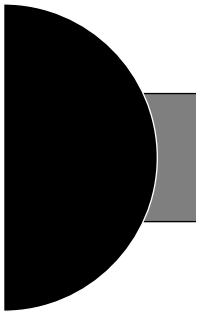


Figure 7.35: A mask with a restricted masking region applied to the rectangle in Figure 7.33.

We can see in Figure 7.35 that outside the mask area the rectangle is no longer visible so only the right hand side of the plot is visible and masked. Another way to think about the mask's area of effect is that masked content is not only masked within that region, but also clipped to it too.

A plot that could benefit from masking can be drawn using the lattice package. Consider the plot in Figure 7.36 which has grid lines present. Unfortunately lines are drawn through the legend, which we would rather avoid.

```
R> plot <-
R+ xyplot(mpg ~ disp, mtcars, group=am, pch=16,
R+   panel=function(...) {
R+     panel.grid(-1, -1)
R+     panel.xyplot(...)
R+   },
R+   key=
R+     list(x=.65, y=.85, corner=c(0, 1),
R+       border=TRUE,
R+       padding.text=3,
R+       text=list(c("automatic", "manual")),
R+       points=
R+         list(pch=16,
R+              col=trellis.par.get("superpose.symbol")$col[1:2])))
R> plot
```

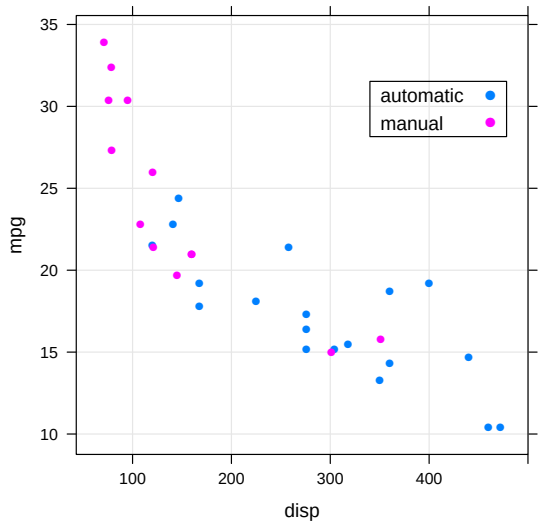


Figure 7.36: A lattice scatter plot with grid lines.

The strategy that we will take is to create a mask based from this image, then apply the mask to the grid lines. We want the masked grid lines to be completely transparent where the legend is located, but opaque in all other locations. The easiest way to define this is to draw a white rectangle over the entire plot and redraw the legend in black.

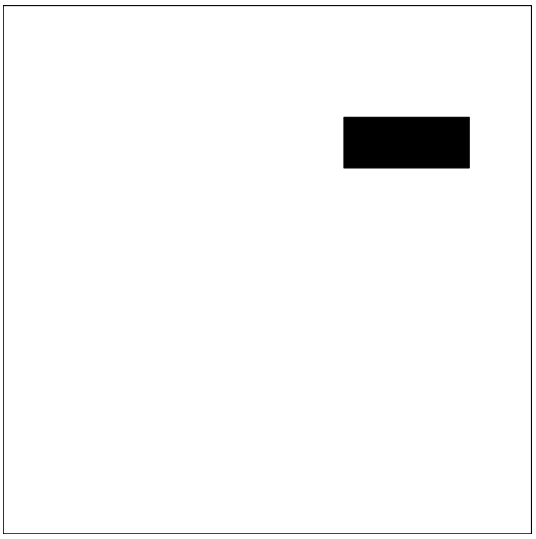


Figure 7.37: The opacity mask used to hide the grid lines that are visible in the legend from Figure 7.36.

When we apply the mask shown in Figure 7.37, the result is shown in Figure 7.38.

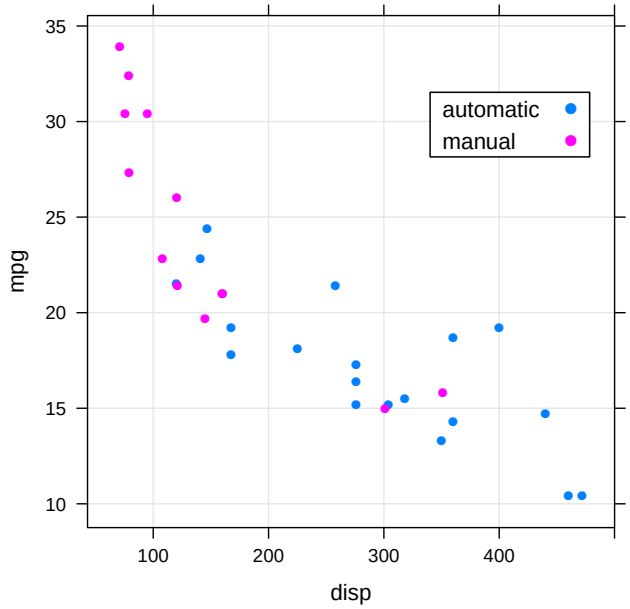


Figure 7.38: The lattice scatter plot with no grid lines present in the legend.

This example has shown how the use of masks allows us to customise an existing plot without having to modify any of its existing graphical components. Rather than altering any of the graphical components, we simply annotate the grid lines to be masked.

7.3.3 Contexts

What we have shown so far with clipping and masking allows for any grob to be clipped or masked. There may be circumstances where repeatedly masking or clipping is desired but repeatedly calling the clipping and masking functions would be a tedious operation. grid provides the `grid.clip()` function to change the clipping region inside a viewport. Anything that is drawn after `grid.clip()` has been called will be clipped to that (rectangular) region.

```
R> grid.clip(width = 2/3, height = 2/3)
R> # The two grobs below will be clipped
```

```
R> grid.rect(gp = gpar(fill = "grey"))
R> grid.text("hello, world!", gp = gpar(fontsize = 48))
```

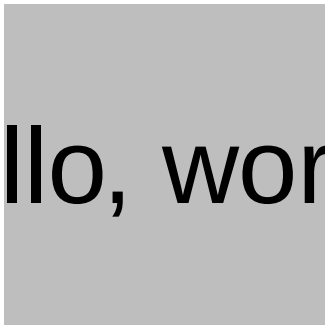


Figure 7.39: Two grid grobs clipped to a region defined by `grid.clip()`.

We can see from Figure 7.39 that the clipping region has been set by `grid.clip()` and that it truncates the drawing of both the text and the rectangle. Given that we are able to clip to a non-rectangular clipping path with `grid.clipPath()`, we might want to use non-rectangular clipping paths in the same way. This can be achieved using the `pushClipPath()` function. It is able to set a clipping *context*. This context ensures that all following drawing operations are clipped to the region defined by `pushClipPath()`. We can re-implement the earlier example using a circle as a simple non-rectangular clipping path instead of the rectangle defined by `grid.clip()`.

```
R> myClipCircle <- clipPath(circleGrob(r = 0.4))
R> pushClipPath(myClipCircle)
R> # The two grobs below will be clipped to a circle instead
R> grid.rect(gp = gpar(fill = "grey"))
R> grid.text("hello, world!", gp = gpar(fontsize = 48))
```

Figure 7.40: Establishing a non-rectangular clipping context with `pushClipPath()`.

Not only is it possible to *push* a clipping context, it is also possible to *pop* them. This is done using the `popClipPath()` function.

```
R> myClipCircle <- clipPath(circleGrob(r = 0.4))
R> pushClipPath(myClipCircle)
R> # The two grobs below will be clipped to a circle instead
R> grid.rect(gp = gpar(fill = "grey"))
R> grid.text("hello, world!", gp = gpar(fontsize = 48))
R> popClipPath()
R> # No longer clipped!
R> grid.text("not clipped", y = 0.1,
R+           gp = gpar(fontsize = 24))
```

Figure 7.41: Entering and leaving a clipping context.

We can see in Figure 7.41 that the clipping context has been removed, allowing the text "not clipped" to be entirely visible. It must be noted that multiple clipping contexts can be pushed, as is also the case with viewports. The behaviour for clipping contexts is the same as with `grid.clip()`, but with slightly more flexibility. When `grid.clip()` establishes a new clipping context, it can only be removed by leaving the viewport that it was established in. We have more flexibility with `pushClipPath()` because we can leave the clipping context without having to leave the viewport in which it was established (using `popClipPath()`).

Not only are clipping contexts able to be set, masking contexts can be pushed into too. This is done using the `pushMask()` function and behaves in much the same way as `pushClipPath()`. Similarly, it is also possible to leave a masking context using the `popMask()` function. A simple demonstration of masking contexts in action is shown in Figure 7.42.

```
R> mymask <- mask(gTree(
R+   children = gList(
R+     rectGrob(gp = gpar(fill = "white")),
R+     circleGrob(r = 0.4, gp = gpar(fill = "grey50")))))
R> pushMask(mymask)
R> grid.rect(gp = gpar(fill = "grey"))
R> grid.text("I am masked", y = 0.6,
R+           gp = gpar(fontsize = 36))
R> popMask()
```

```
R> grid.text("I am not masked", y = 0.4,
R+           gp = gpar(fontsize = 36))
```

Figure 7.42: Establishing and leaving a masking context.

By using contexts we avoid the need to repeatedly call the clipping and masking functions. It also allows us to write more declarative code that clearly shows how masking and clipping are occurring.

7.4 Filter Effects

It is possible to add special effects to SVG content with a feature called filter effects. Filter effects apply an image processing operation to existing graphical content that changes how they appear. Many of the available filter effects are commonly available in raster image editing software, such as Adobe Photoshop or the GNU Image Manipulation Program (GIMP) but are not commonly employed in vector graphics, primarily because of their complexity. With filter effects it is possible to apply complicated effects such as blurring, composition operations, blending and lighting effects.

There are a large number of possible filter effects that can be applied by gridSVG, but many of them are complex and therefore difficult to describe. Consequently, we will only show a simple example where we apply a drop shadow to points in an existing image. It must be noted that this example barely scratches the surface when it comes to the

possible filter effects that can be drawn. The aim is to produce a drop shadow that is located one millimetre below and one millimetre to the right of each point. To do this we first need to draw the plot that will later be filtered (see Figure 7.43).

```
R> xyplot(mpg ~ disp, mtcars, group = am, pch = 16, cex = 1.5)
```

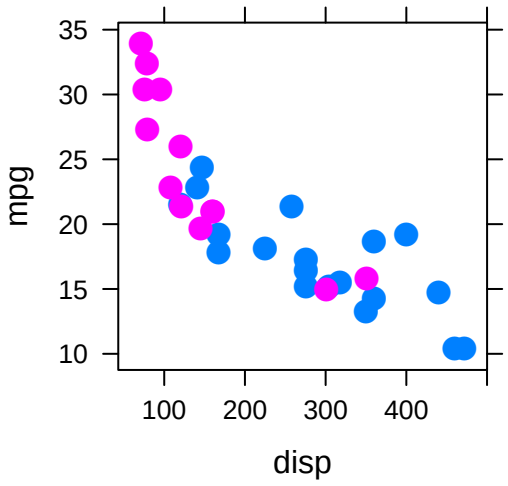


Figure 7.43: A lattice scatter plot that will be enhanced with filter effects.

Again, like many of the previous examples because the lattice package is using a consistent naming scheme, we can easily find the name of the grob that draws the points. Given that the grob is on the display list, we can now create the drop shadow filter effect that will be applied to it.

```
R> dropShadow <- filterEffect(
R+     list(feGaussianBlur(input = "SourceAlpha",
R+                         sd = 2, result = "blur"),
R+          feOffset(input = "blur",
R+                   unit(1, "mm"), unit(-1, "mm"),
R+                   result = "offsetBlur"),
R+          feMerge(list(feMergeNode(input = "offsetBlur"),
R+                       feMergeNode(input = "SourceGraphic")))))
```

What this drop shadow effect describes is that we first apply the `feGaussianBlur()` filter effect primitive. This primitive takes the opacity of the source (in our case a

collection of points) and applies a Gaussian blur with a standard deviation of 5. The result of this operation is named "blur" so that the `feOffset()` primitive can use it. `feOffset()` then uses "blur" and translates it to the right by one millimetre and down one millimetre. The result of the offset operation is called "offsetBlur". Finally, we take "offsetBlur" (i.e. the drop shadow), and merge it with the source image (i.e. our points). The order in which the inputs are "merged" is important because we want the drop shadow to be behind the source image.

It must be noted that the order in which the filter effect primitives is defined can be important when the `input` and `result` arguments are not explicitly given.

The process that is undertaken to draw the drop shadow (before merging) is illustrated with a single point in Figure 7.44.

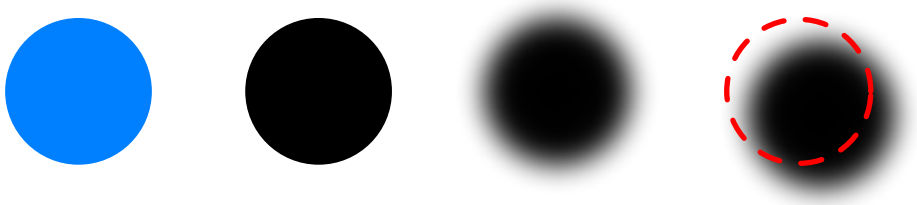


Figure 7.44: The process taken to apply a drop shadow filter effect.

At this point, we now have a filter effect definition, and points that can be filtered. All that is necessary is to apply the filter effect to the points using `grid.filter()` and export to SVG. The result is shown in Figure 7.45.

```
R> grid.filter("points", dropShadow, global = TRUE, grep = TRUE)
R> grid.export("points-with-drop-shadow.svg")
```

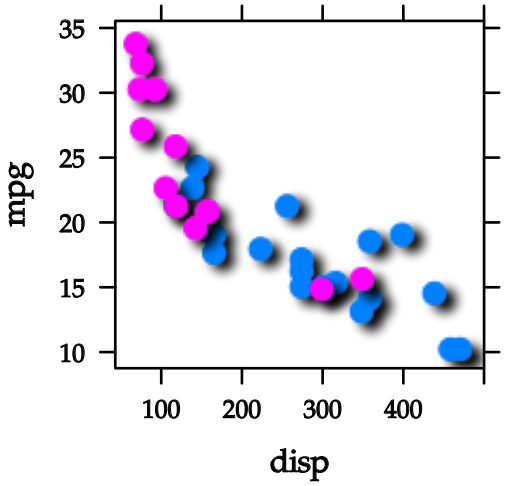


Figure 7.45: The lattice plot from Figure 7.43 with a drop shadow filter effect applied.

This demonstration has shown how the use of filter effects in SVG allows us to create statistical plots with complex embellishment.

7.5 Definition and Registration

A key detail that has been not yet been discussed is the difference between defining and *registering* the advanced SVG features within a gridSVG image. These concepts are crucial for making best use of the SVG features we have mentioned. Indeed, although they have not been discussed, their use underpins the way in which these advanced SVG features are used in gridSVG.

We refer to the definition of a feature object (e.g. a mask or a gradient object) as being the object that describes what the content should look like, for example the object that is created by calling `pattern()`. If any of those definition objects contain grid units (e.g. normalised parent coordinates), then the locations and dimensions that these units are describing are relative to a grid viewport. By registering an image, the locations and dimensions used by feature objects become fixed and relative to the entire drawing canvas. This means that the effect is as if they are drawn to the canvas (but not visible).

To demonstrate the concepts of definition and registration, we will first consider a simple pattern. The pattern is defined in Figure 7.46.

```
R> pat <-
R+     pattern(circleGrob(r = 0.3, gp = gpar(fill = "black")),
R+             width = unit(0.1, "npc"), height = unit(0.1, "npc"))
```

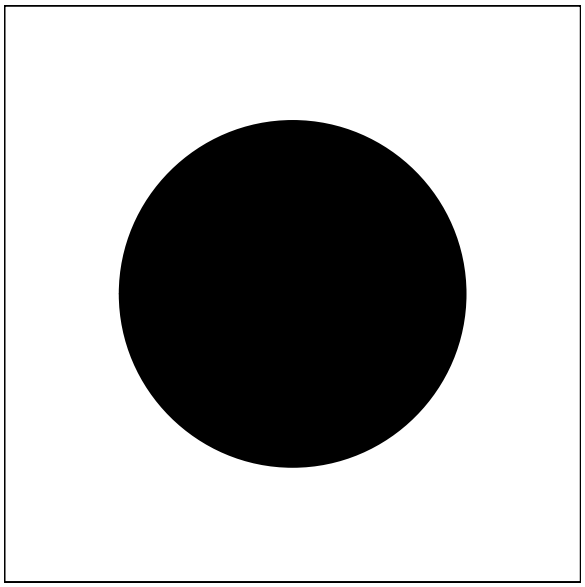


Figure 7.46: A simple pattern definition.

What Figure 7.46 shows is that our pattern definition (`pat`) has a width and height that is one tenth of a viewport's width and height respectively. If we registered this definition, these dimensions would be translated into absolute units. For example, if the viewport was 10 centimetres wide and 10 centimetres high then the pattern would be registered with a width and height of 1 centimetre. Registering a pattern is demonstrated below:

```
R> registerPatternFill("example", pat)
```

The first argument to `registerPatternFill()` is a label that we used to identify the registered pattern object. The pattern object is the second argument that we give to the function and its locations and dimensions have now become fixed.

We shall now illustrate this process by registering the pattern definition in two different viewports. One viewport will have dimensions that are each twice as large as those in the other viewport.

```
R> # Pattern width/height is 1/10th whole page
R> registerPatternFill("pagePattern", pat)
R> # Now pushing into a smaller viewport
R> pushViewport(viewport(width = 0.5, height = 0.5))
R> # Registering a pattern whose width/height is
R> # 1/10th of the current viewport
R> registerPatternFill("smallPattern", pat)
```

We have registered the same pattern definition under two different labels, `pagePattern` and `smallPattern`. An advantage of registering a pattern means that we do not need to provide the pattern definition object when applying it a grob. The registration allows us to refer to the pattern by its label instead of providing a definition object.

We will now draw two rectangles that use the two registered patterns. The effect of the registration process should now be clear to see in Figure 7.47.

```
R> # Creating simple rectangles to fill
R> grid.rect(x = 0, width = 3/8, just = "left",
R+           name = "large")
R> grid.rect(x = 1, width = 3/8, just = "right",
R+           name = "small")
R> # Registering patterns by label
R> grid.patternFill("large", label = "pagePattern")
R> grid.patternFill("small", label = "smallPattern")
R> grid.export("registered-patterns-example.svg")
```

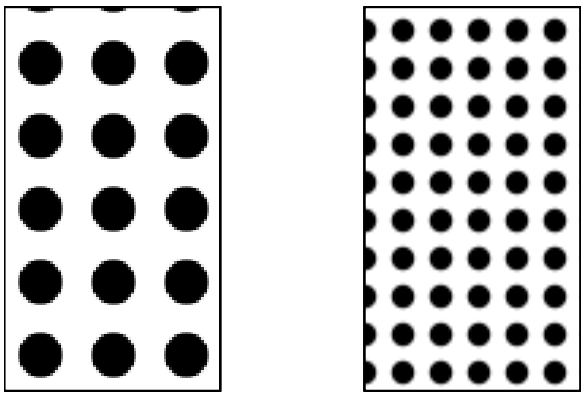


Figure 7.47: Applying a pattern that was registered in two different drawing contexts.

What Figure 7.47 shows is that despite using the same pattern definition, the fact

that we registered it in different viewports means that the registered patterns appear different to each other.

Another point to note about registration of definition objects is that this always occurs, even without explicitly registering a definition object. An example of this when we call any function like `grid.patternFill()` and give it a definition object. It will register the object with an automatically generated label, and then apply the registered definition.

7.5.1 Inspecting Registered Definitions

When any definition object is registered, it is stored in an off-screen display list that can be inspected by calling `listSVGDefinitions()`, just like a grid scene can be inspected by calling `grid.ls()`.

```
R> listSVGDefinitions()
Reference Definitions

Pattern Fills
  pagePattern
  smallPattern
```

This output shows that we currently have two pattern fill definitions registered with the labels `pagePattern` and `smallPattern`. Recall that these are the labels that patterns were registered to when creating Figure 7.47.

This display list is retained even after creating a new grid page. This is because we may want to use the same definition across multiple images without needing to register the definition again. This behaviour may be a hindrance in some circumstances because it may make reproducibility more difficult. To clear the list of registered definitions requires calling `gridSVG.newpage()` instead of `grid.newpage()`.

```
R> listSVGDefinitions()
Reference Definitions

Pattern Fills
  pagePattern
  smallPattern
R> grid.newpage()
R> listSVGDefinitions()
```

```
Reference Definitions

Pattern Fills
  pagePattern
  smallPattern
R> gridSVG.newpage()
R> # No output
R> listSVGDefinitions()
```

This is particularly useful in order to apply many different gradients or patterns (or any other type of registered definition) to a single grob that has multiple sub-grobs. In fact, this has been demonstrated earlier this chapter in Figure 7.6. To explain this in more detail, consider the plot in Figure 7.48.

```
R> xyplot(mpg ~ disp, mtcars, pch = 16, cex = 1.5)
```

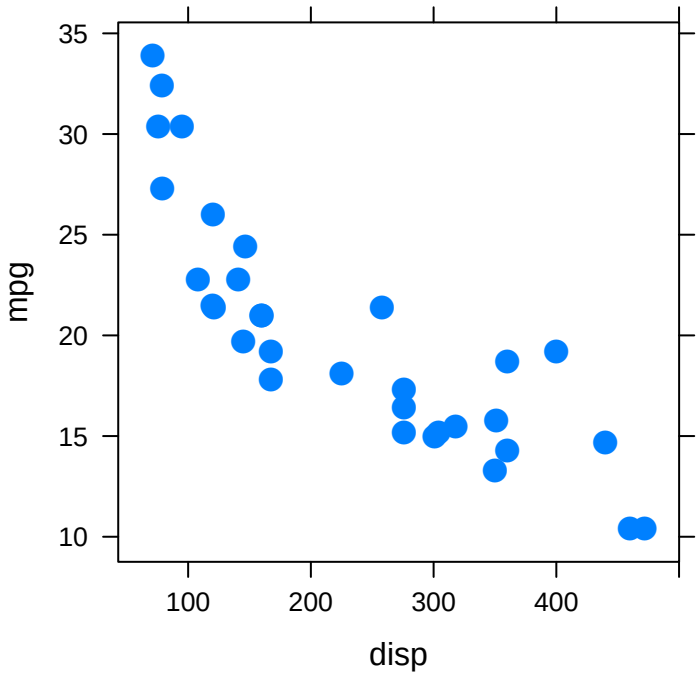


Figure 7.48: A simple lattice scatter plot.

```
R> grid.ls()
plot_01.background
plot_01.xlab
plot_01.ylab
plot_01.ticks.top.panel.1.1
plot_01.ticks.left.panel.1.1
plot_01.ticklabels.left.panel.1.1
plot_01.ticks.bottom.panel.1.1
plot_01.ticklabels.bottom.panel.1.1
plot_01.ticks.right.panel.1.1
plot_01.xyplot.points.panel.1.1
plot_01.border.panel.1.1
```

We can apply radial gradients to the data points, which are drawn by a grob named `plot_01.xyplot.points.panel.1.1`. However, we want to apply a radial gradient with each colour showing whether the data point represents a car with a manual transmission or an automatic transmission. For cars with automatic transmission we will apply a blue radial gradient, while cars with manual transmission will have a red radial gradient applied. The result is shown in Figure 7.49.

```
R> xyplot(mpg ~ disp, mtcars, pch = 16, cex = 1.5)
R> # Defining gradients
R> redGrad <- radialGradient(c("red", "black"))
R> blueGrad <- radialGradient(c("blue", "black"))
R> # Registering
R> registerGradientFill("red", redGrad)
R> registerGradientFill("blue", blueGrad)
R> # Applying
R> grid.gradientFill("plot_01.xyplot.points.panel.1.1",
R+                   label = ifelse(mtcars$am == 0, "blue", "red"),
R+                   group = FALSE)
```

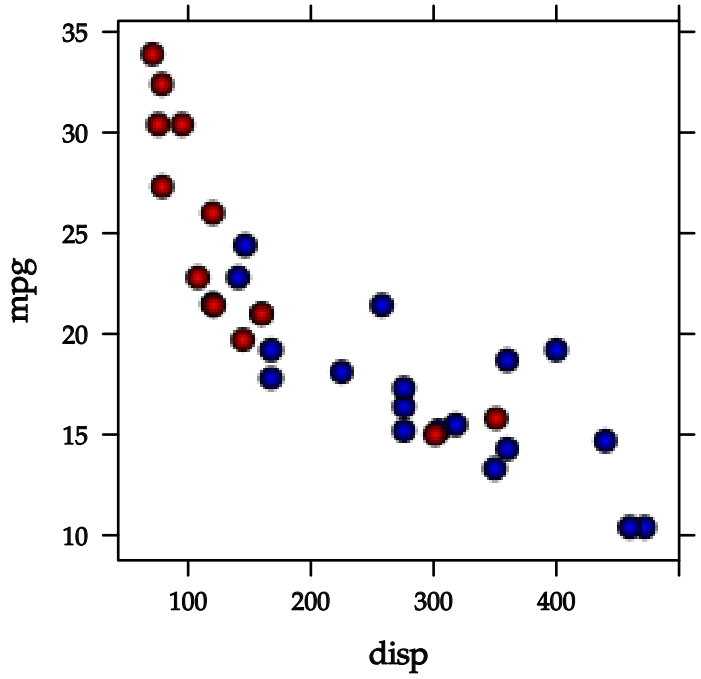


Figure 7.49: Applying registered gradients by label to alter Figure 7.48.

The key point to note is that because we have registered our gradient definitions we are able to create a character vector of labels that selects which gradient will be applied to each data point. This is often necessary if we wish to apply any advanced SVG feature to a sub-grob instead of the grob's SVG grouping element (<g>).

7.6 Element Grobs

We have shown many features of gridSVG that expose graphical functionality to R that would not otherwise be present. Whilst there are many new features available that expose the vast majority of the SVG feature set, it is not possible to provide *all* of the SVG functionality without writing SVG by hand. This is done by using the `grid.element()` function, which draws a grob on the grid display list that is treated as if it were plain SVG. This means that when `grid.export()` is called, the grid grobs are translated directly to SVG elements. Simple examples showing its use are shown below:

```
R> # Single 'example' element
R> grid.element("example")
R> # Now drawing an element with attributes
R> grid.element("second", attrs = list(a = "b", c = "d"))
R> # We can nest elements
R> grid.element("parent", children =
R+     gList(elementGrob("child", attrs = list(e = "f"))))
R> # We can also nest grobs
R> grid.element("grobpar", children = gList(circleGrob()))
```

After exporting, the relevant SVG output is shown below:

```
<example id="GRID.element.452.1"/>

<second a="b" c="d" id="GRID.element.453.1"/>

<parent id="GRID.element.454.1">
  <child e="f" id="GRID.element.455.1"/>
</parent>

<grobpar id="GRID.element.456.1">
  <g id="GRID.circle.457.1">
    <circle id="GRID.circle.457.1.1" cx="108" cy="108" r="108"/>
  </g>
</grobpar>
```

We can see that there is a very direct relationship between a gridSVG element grob and its resulting SVG output. In fact, the only extra annotation that an element grob is given (by default) is an `id` attribute. This is added so that we can still refer to the generated content and may be useful in conjunction with the tools shown in section 2.6.

A case where the `grid.element()` interface can be used is to insert live HTML content into an SVG image. This is achieved by using the `<foreignObject>` SVG element. In this example we are going to draw a simple grid image, but a portion of the image will be showing an HTML document that is loaded from the University of Auckland's Department of Statistics homepage (`http://stat.auckland.ac.nz/`).

```
R> grid.rect(gp = gpar(fill = "grey"))
R> grid.text("HTML Document", y = 0.9, gp = gpar(fontsize = 24))
R> # Creating the body of the (X)HTML document
R> htmlBody <-
R+   elementGrob("body",
R+     namespaceDefinitions = "http://www.w3.org/1999/xhtml",
R+     children = gList(
R+       elementGrob("iframe",
R+         attrs = list(src = "http://www.stat.auckland.ac.nz/",
R+                      width = 300, height = 400))))
R> grid.element("foreignObject",
R+              attrs = list(
R+                  x = 30, y = 60,
R+                  width = 300, height = 400,
R+                  transform = "translate(0, 360) scale(1, -1)"),
R+              children = gList(htmlBody))
R> grid.export("svg-with-html.svg")
```

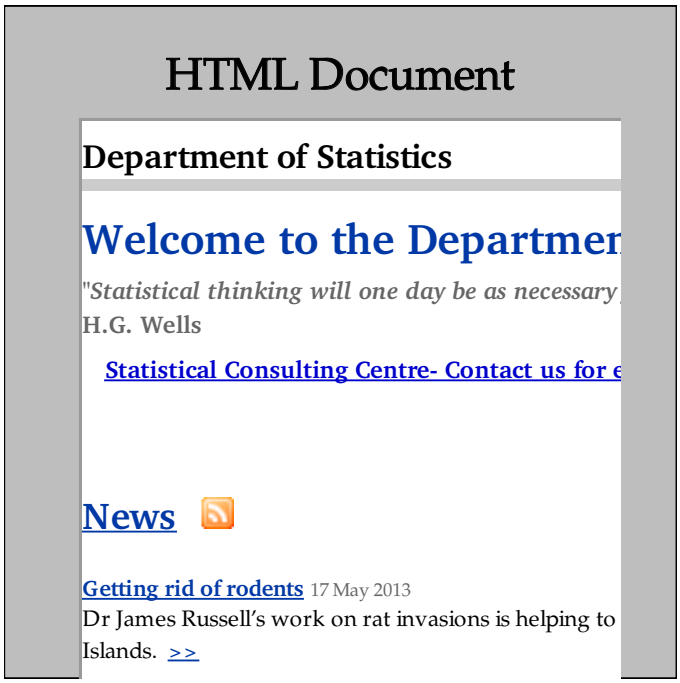


Figure 7.50: An SVG image produced by gridSVG with a live HTML document embedded.

Figure 7.50 demonstrates not only the flexibility of SVG, but also a way in which we can write sophisticated SVG markup in R. A notable limitation in this approach is that any positions or dimensions (e.g. `x`, `y`, `width`, `height`) must be hard-coded, therefore trial and error may be required to correctly position content drawn by `grid.element()`.

7.7 Conclusion

This chapter describes new features of gridSVG that enable us to use graphical features that are not present in the R graphics engine. Many of these features are complicated to do by writing SVG markup ourselves, and are also rarely available in alternative (in particular JavaScript-based) graphics systems. gridSVG not only makes it possible for us to use these SVG features, but it also provides an easy-to-use interface.

8 Examples

In this chapter we will show three examples of the types of interactive graphics that gridSVG is now capable of producing. In particular, these examples demonstrate how gridSVG is used to update a *piece* of a plot, rather than redrawing an entire plot. Unfortunately, due to the static nature of this printed document, the animations and interactivity cannot be adequately captured. Consequently, one must assume that D3 is performing the described animations.

To view these examples as they were intended, an R package is available (at `http://sjp.co.nz/projects/msc-thesis/`) which contains all of the examples and instructions on how to run them. The examples can be viewed in a web browser and are served from an R web server that runs on a user's machine.

8.1 LOESS Smoothing

This interactive LOESS smoothing example was mentioned in chapter 1 as a goal for the types of interactive and animated graphics that we want gridSVG to produce. LOESS smoothing is a commonly used technique that allows for a trend to be made visible in a noisy dataset. The key parameter of interest is the "span" parameter that determines the size of the spanning window used for local regression. By setting a higher value of "span", more data points will be used to construct the smoother at each point along the curve. This means the LOESS curve will be more smooth than one constructed with a lower "span". Figure 8.1 shows a plot with a red LOESS curve added to it.

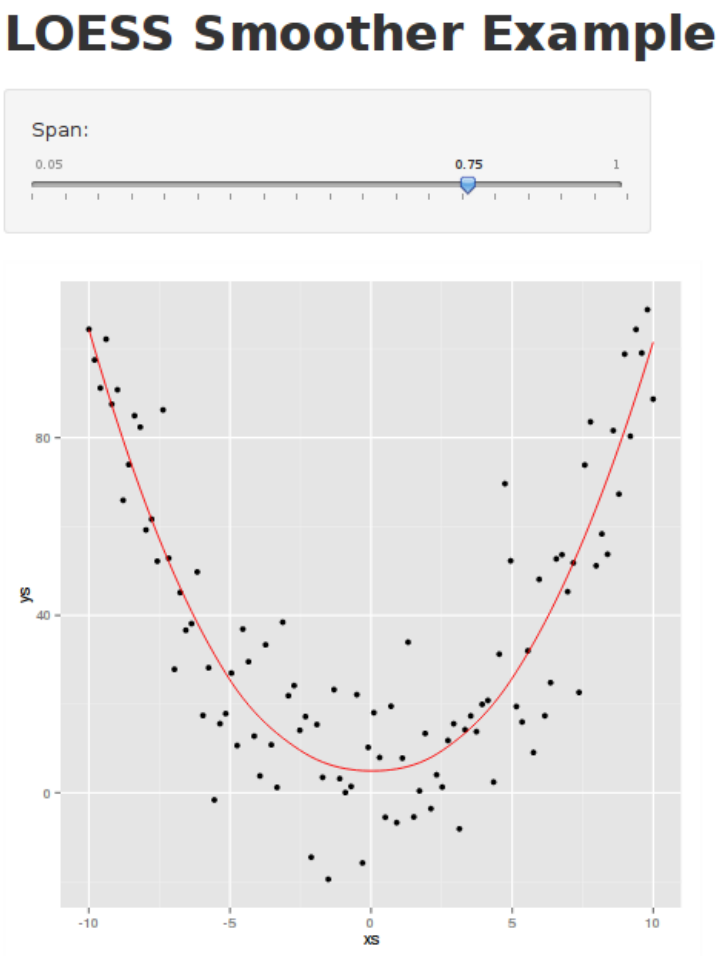


Figure 8.1: A LOESS curve fitted to a dataset with a default span parameter of 0.75.

The HTML slider control allows us to change the value of the “span” parameter in an easy manner. Setting “span” to a low value of 0.05 allows us to observe its effect by transitioning the line to become more “noisy”, as illustrated in Figure 8.2. Indeed, the spanning window is so small that the LOESS curve is no longer a curve but a jagged line that is drawn through each data point.

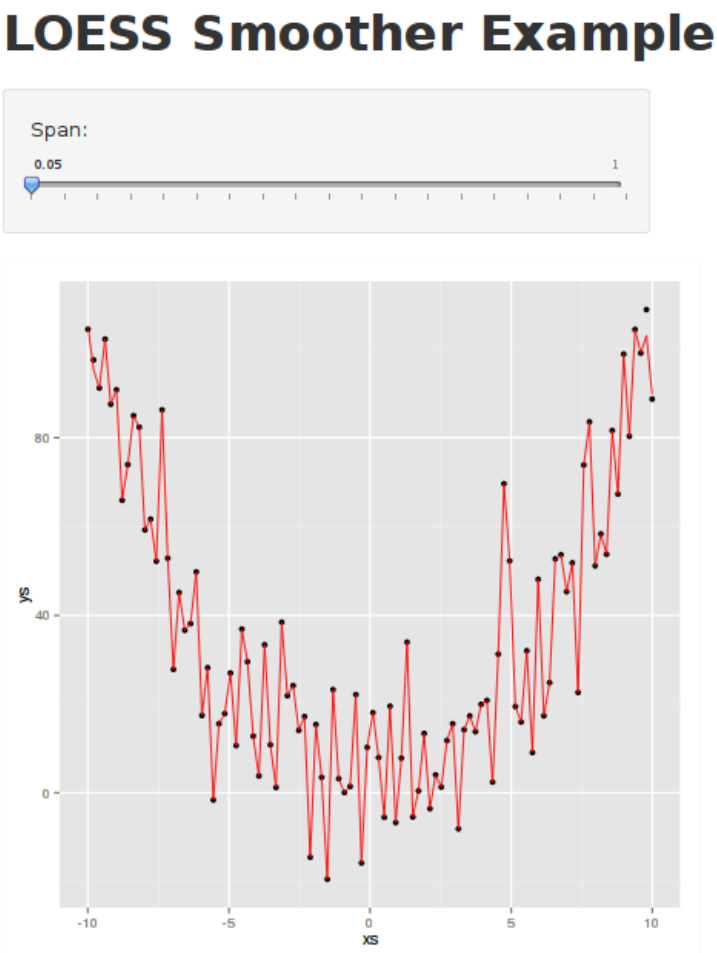


Figure 8.2: Setting the span parameter to a low value of 0.05 causes the line to become more “noisy”.

We can see the effect of changing the span parameter to a more reasonable value of 0.25 in Figure 8.3.

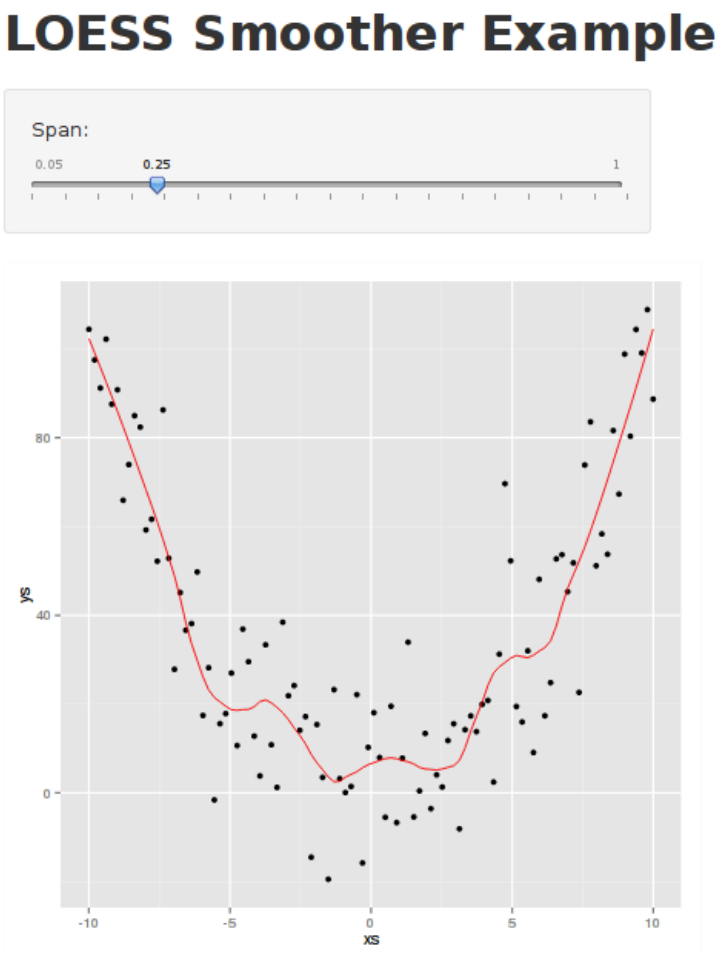


Figure 8.3: The LOESS curve after transitioning to a new span value of 0.25.

The method that by which this interactive demonstration occurs is that a web page is initially loaded with everything shown in Figure 8.1. When the span parameter changes via the HTML slider control, a message is sent back to R requesting new line coordinates for the LOESS curve. R then calculates the new line coordinates using grid graphics, which are then exported to SVG by gridSVG. The resulting line coordinates are retrieved from the newly generated SVG and sent back to the browser as a response to its request message. Once the browser gets this message D3 is used to transition the LOESS curve from its old line coordinates to the newly generated line coordinates.

8.2 Interactive ARIMA Model Diagnostics

This example is a tool created to ease model diagnostics when building ARIMA time series models. There are three key parameters of interest: AR(p), Differencing (d), and MA(q). When building models that have been constructed using these parameters, plots showing the autocorrelation (ACF) and partial autocorrelation (PACF) functions are very useful for assessing whether the constructed models are a good fit.

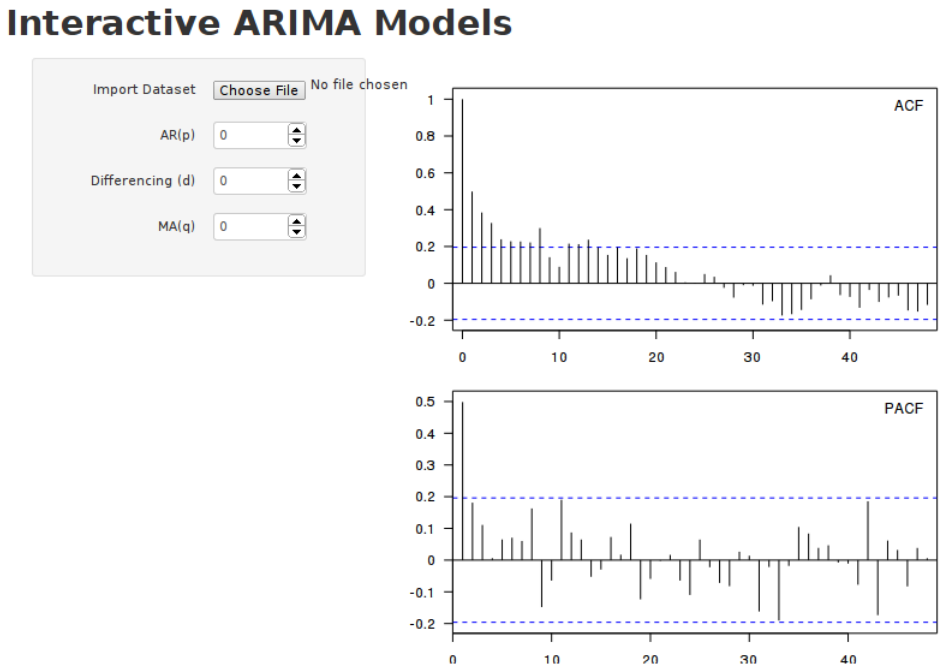


Figure 8.4: An ARIMA$(0,0,0)$ model fitted to data. ACF and PACF plots are drawn from the errors from the model.

There are many different possible models that can be constructed with these parameters, for example even if we restrict p, d and q to only take integer values in the range $[0,3]$, there are 64 possible models. This can make model comparisons difficult if we are assessing them visually using plots of the autocorrelation and partial autocorrelation functions. This tool assists with the process because it not only shows what the ACF and PACF plots look like, but also *transitions* the error lines in each plot to new positions. This allows easy comparison between two models because the human eye is very good at detecting movement. Therefore, we can more easily observe differences between two models when there are transitions present than when comparing static images.

When diagnosing the ARIMA$(0,0,0)$ model in Figure 8.4 we can see that there are

multiple lags with significant errors in the ACF plot that drop off gradually at higher lags. In addition, the PACF plot clearly shows a single significant error at the first lag. This indicates that ARIMA$(1, 0, 0)$ might be an appropriate model. To test this we can simply increment the value of p in the HTML numeric picker control. This causes a message to be sent to R that requests new line coordinates for the errors in each of the plots. In order to get this information a new ARIMA model is fitted to the data. From the errors of this model, ACF and PACF plots are constructed. We collect the relevant pieces of these plots and send the information to the browser where D3 uses this new information to transition the lines to the appropriate positions.

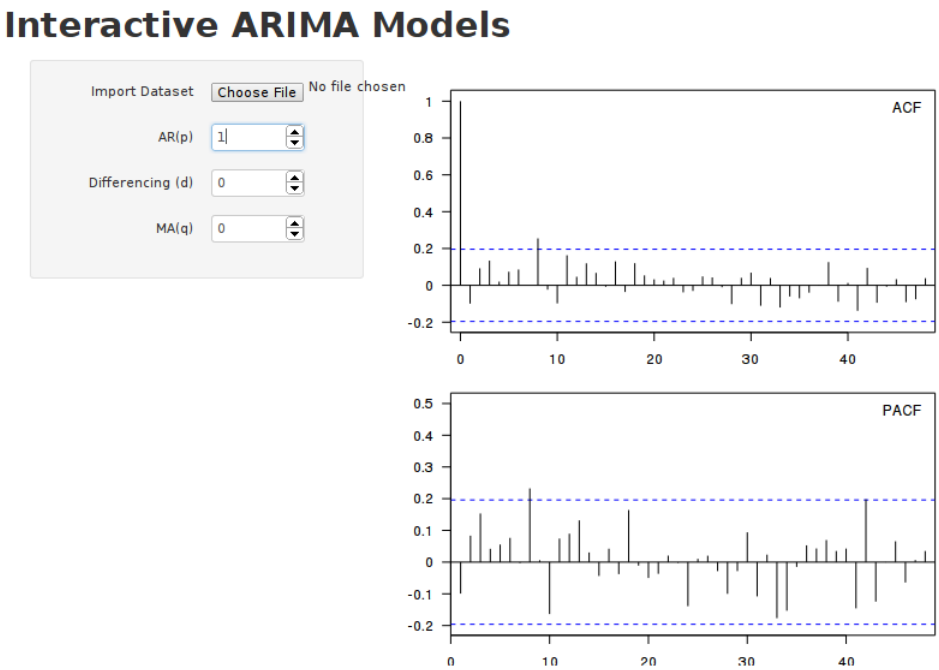


Figure 8.5: An ARIMA$(1, 0, 0)$ model fitted to data. The lines in each plot ACF and PACF plots were transitioned to these locations from their original states in Figure 8.4.

The diagnostic plots shown in Figure 8.5 suggest that ARIMA$(1, 0, 0)$ is an adequate model for this dataset. To test whether this indeed the case we can attempt to add a moving average term, i.e. an ARIMA$(1, 0, 1)$ model.

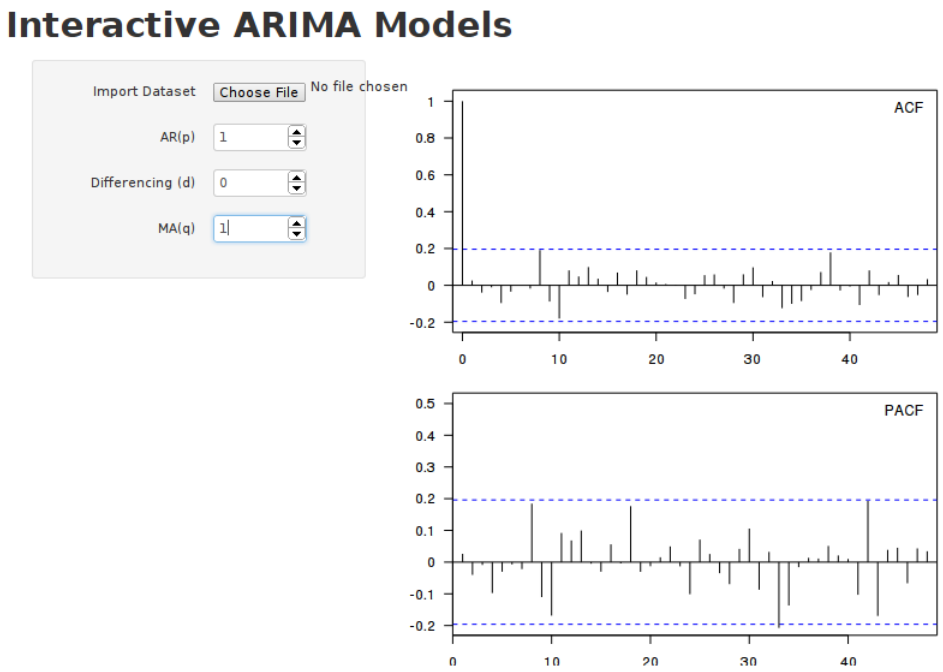


Figure 8.6: An ARIMA$(1, 0, 1)$ model fitted to data. No obvious changes were visible when transitioning from the model used in Figure 8.5.

Because there was no obvious movement evident when adding a moving average term, it is clear to the user that adding a moving average term does not capture any variability that was previously in the errors. Although this dataset is clearly demonstrating an AR(1) process, more complicated datasets may make this tool even more beneficial.

8.3 Sampling Variation Teaching Visualisation

The VIT (Visual Inference Tools) package (Wild et al., 2013) creates animations are used for teaching statistical concepts such as bootstrap confidence interval construction, permutation testing and confidence intervals. One of the animations it can create is used for teaching the concept of sampling variation.

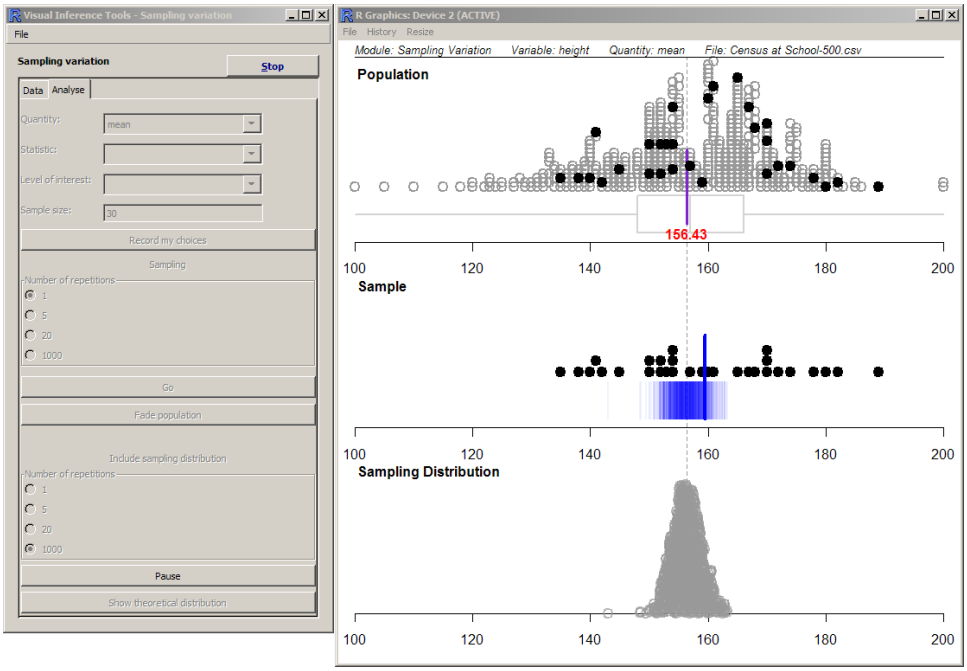


Figure 8.7: Visual Inference Tools running a sampling variation animation.

This example differs from the LOESS and ARIMA examples because it requires far more coordination once data has been given to the browser. This additional complexity is reduced immensely when the animaker package is used to describe and apply the animation sequencing. This is a huge advantage that a gridSVG implementation has over VIT. VIT has no way of keeping track of time because it repeatedly draws as quickly as possible. This means that VIT animations vary in length depending on how complex the grid scene is in addition to being "choppy". These limitations are not present when animating SVG content because we can declaratively say what, when and for how long graphical content is being animated.

The sampling variation animation sequence that needs to be described by animaker is the following:

- Select a sample from a population in the Data panel.
- Pull the currently selected sample into the Sample panel.
- Generate a sample statistic, and leaving a "ghost" statistic behind.
- Pull the sample statistic into the Statistic panel and turn it into a point.
- Repeat to build up a distribution of sample statistics.

A single iteration of this process is described in the following animation sequence and illustrated in Figure 8.8.

```
R> library(animaker)
R> sampSelect <- atomic(label = "sampleSelect",
R+                      start = 1.5,
R+                      durn = 1.5)
R> sampDrop <- atomic(label = "sampleDrop",
R+                    durn = 1.5)
R> sampStat <- atomic(label = "sampleStat",
R+                    durn = 1)
R> sampStatDrop <- atomic(label = "sampleStatDrop",
R+                        durn = 1.5)
R> sampRemove <- atomic(label = "sampleRemove")
R> statPoint <- atomic(label = "statPoint",
R+                     durn = 0.5)
R> # Collect all of the animations into a single iteration
R> singleIter <- vec(sampSelect,
R+                   sampDrop,
R+                   sampStat,
R+                   sampStatDrop,
R+                   sampRemove,
R+                   statPoint,
R+                   label = "singleIter")
R> plot(singleIter)
```

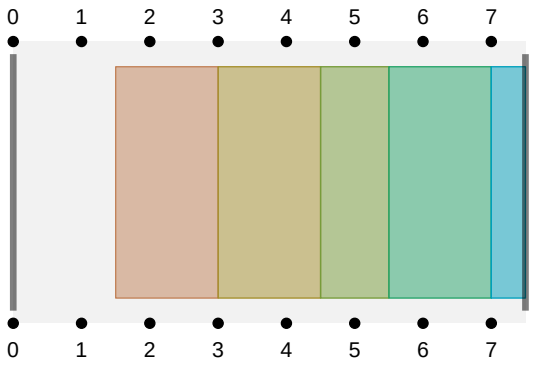
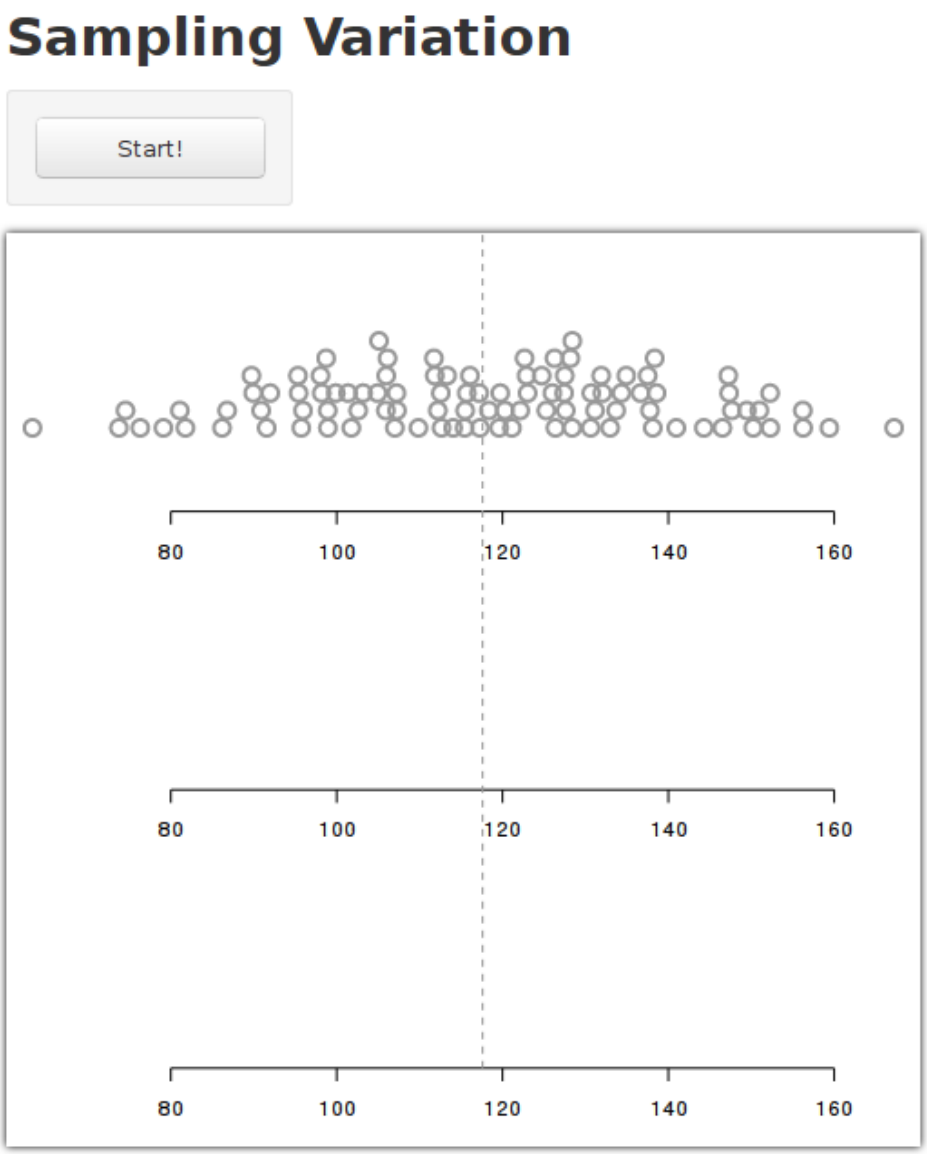
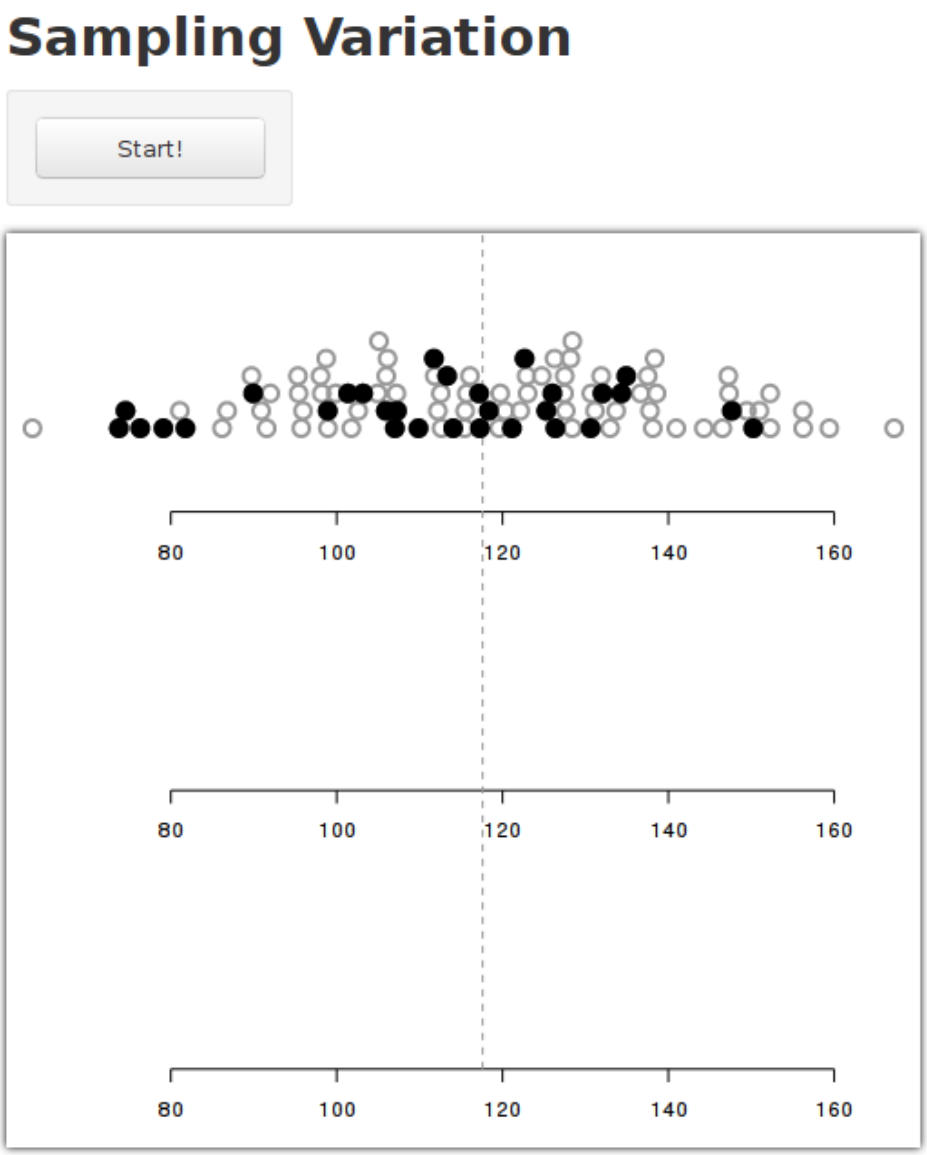


Figure 8.8: A single iteration of the Sampling Variation animation sequence.

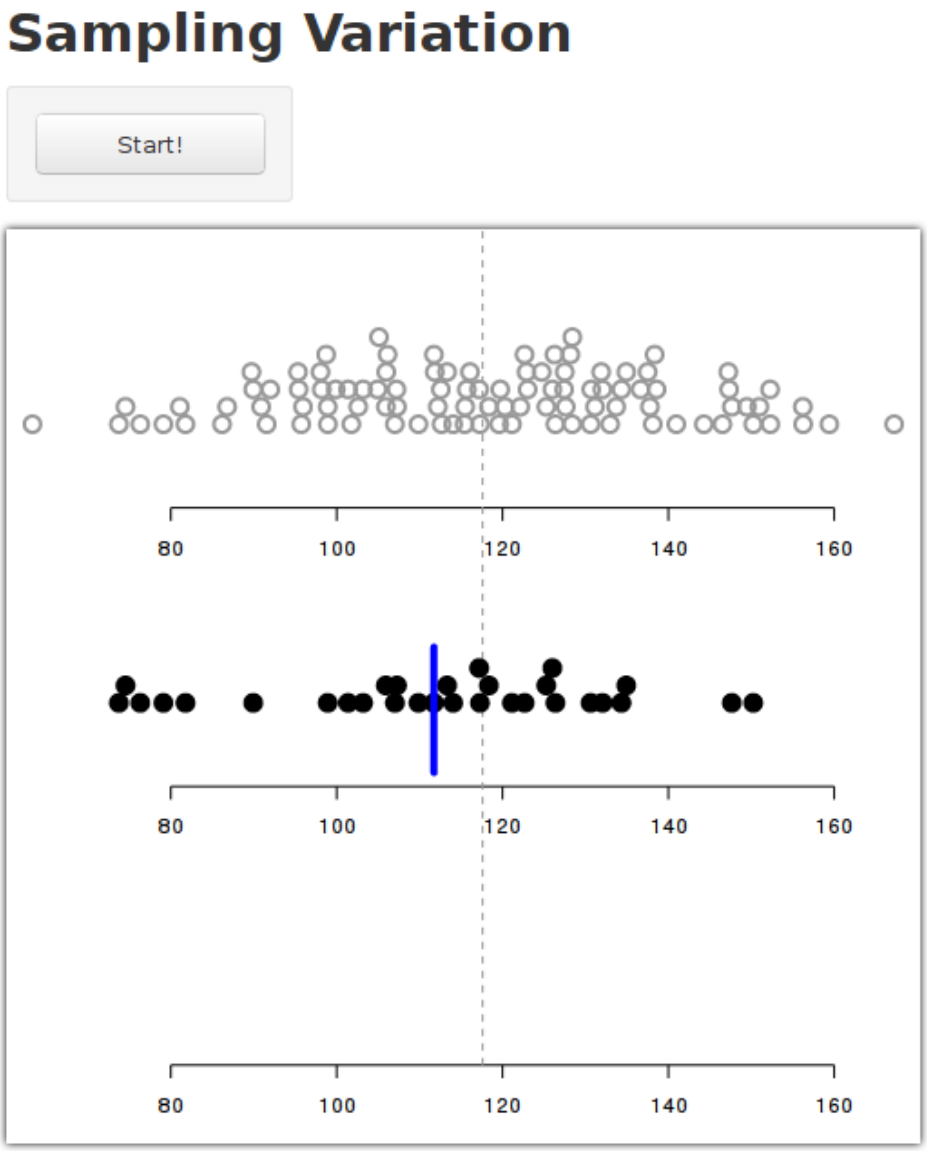
Given that an animation sequence has now been described it can be applied in the browser using the TimingManager library. This step is not shown but the resulting animation is shown in Figure 8.9. Although difficult to observe, the key idea here is that we know exactly when an animation occurs, what it is doing, and how long the animation runs for. The animation runs smoothly and improves upon some of the limitations of the source implementation.



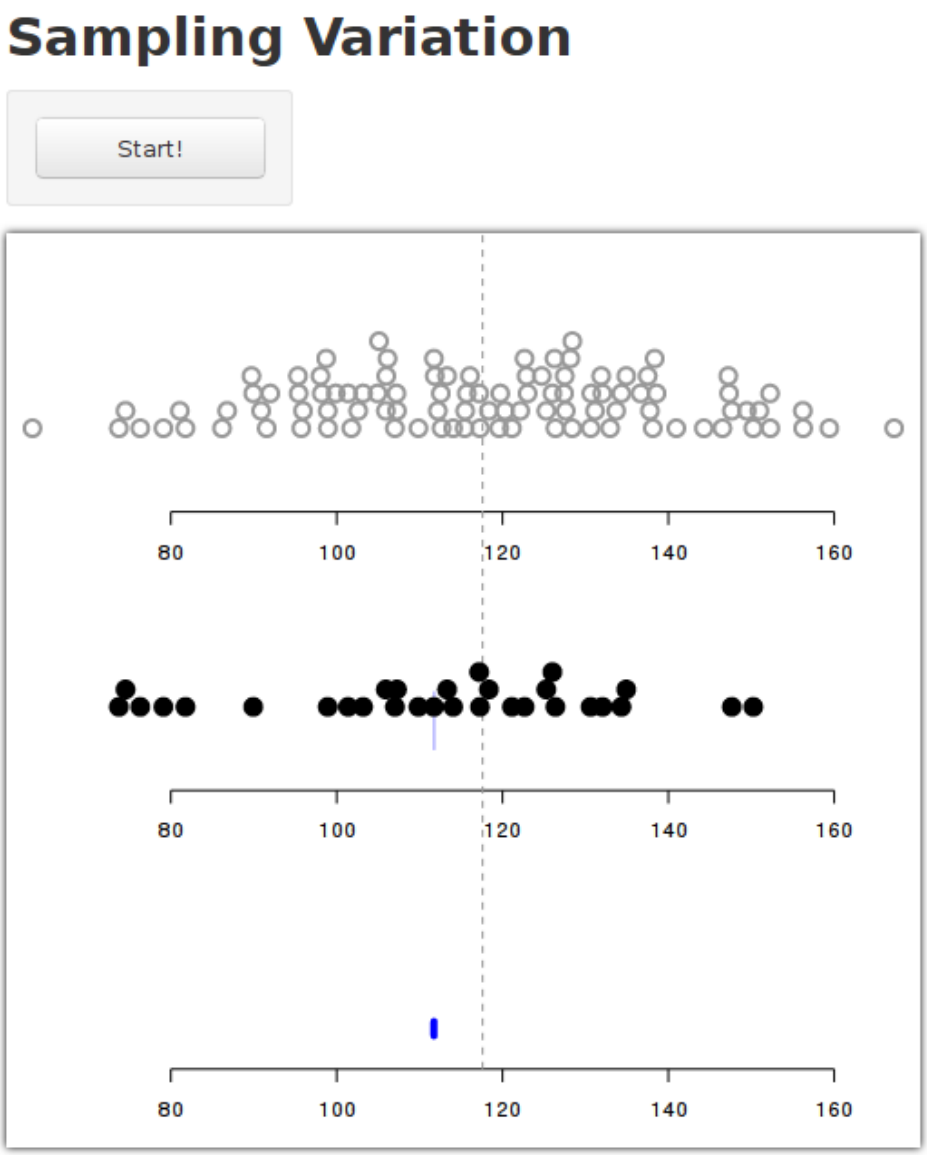
(8.9a) The web page before the “Start” button has been clicked.



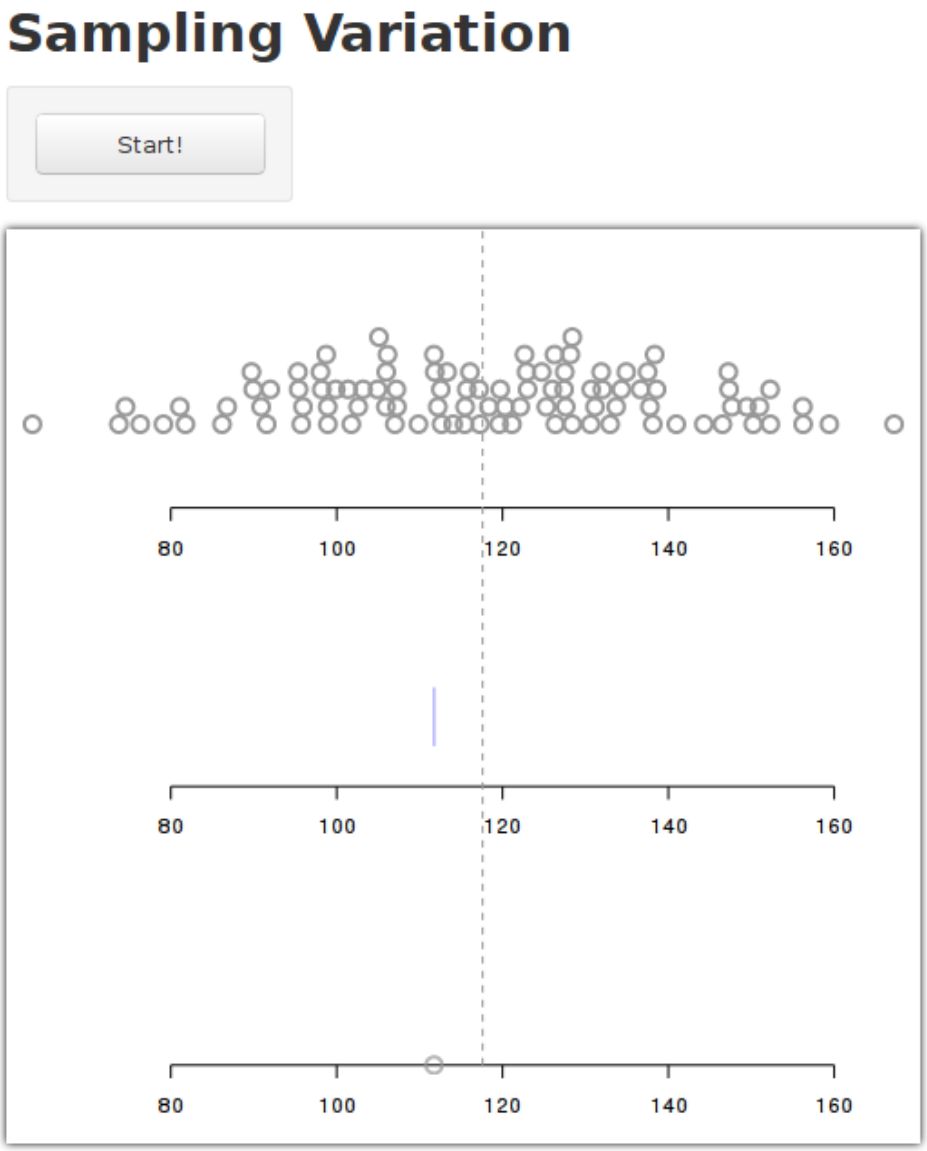
(8.9b) Selecting a sample from a population.



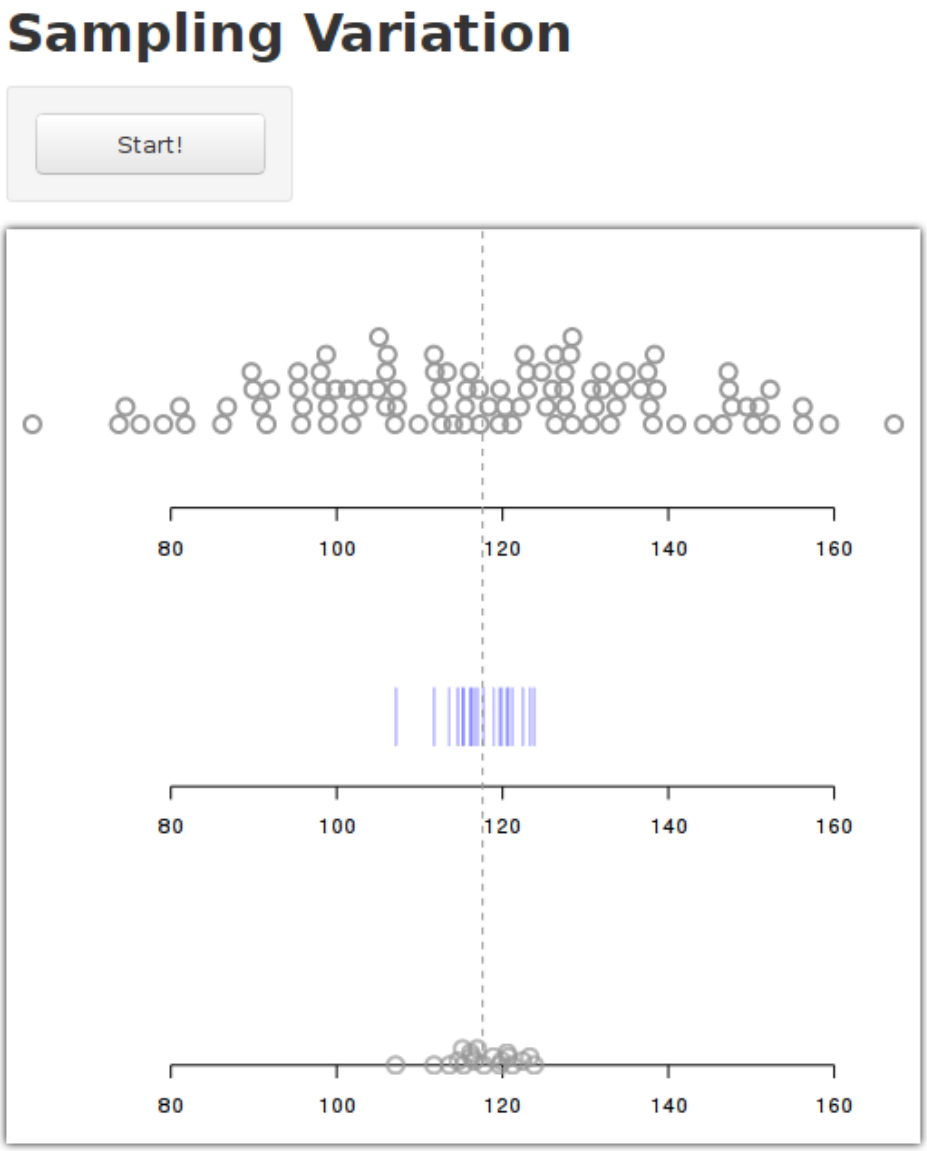
(8.9c) Dropping the selected sample to the “Sample” panel and generating a sample statistic (the sample mean).



(8.9d) Dropping the sample statistic to the “Statistic” panel.



(8.9e) The sample statistic has become a point and a "ghost" sample statistic has been left behind in the "Sample" panel.



(8.9f) The page after 20 iterations.

Figure 8.9: A sampling variation teaching animation using gridSVG and D3.

9 Discussion

9.1 Improvements

9.1.1 Naming

In the past, it was not guaranteed that a gridSVG plot would generate plots where each component was uniquely identifiable. The straightforward approach previously performed was simply assigning the name of a grid grob (or viewport) to the `id` attribute of the associated SVG content. Not only was the uniqueness of SVG `id` attributes not guaranteed within the SVG image, it was also not guaranteed to be the case if the SVG image was embedded inline in an HTML document. The ability to apply a consistent naming scheme that can be controlled by a user provides us with a means of safely and uniquely naming SVG elements.

Named SVG elements are of no use unless we can identify them. We know in advance (from a grid scene) what the names of grobs and viewports are, so mapping grob and viewport names to their equivalent SVG `id` attributes is an essential step for being able to manipulate gridSVG output. gridSVG now allows name mapping information to be exported as it generates an SVG image. By exporting name mapping information along with SVG output, gridSVG makes it possible to *query* this information. This provides us with the essential step necessary to identify exported pieces of a plot.

The identification of graphical components is rarely a consideration by other software that create SVG images (Murrell and Potter, 2013a). This is because they typically do not intend for the resulting image to be manipulated in any way. This gives gridSVG an advantage over many other SVG generating implementations because it is easier to manipulate, at the cost of having to give sensible (and not automatically generated) names to grid grobs.

9.1.2 Coordinate Systems

The ability to retain coordinate information is an essential component for building a reactive gridSVG plot. It is necessary for retaining the drawing contexts that pieces of a plot are drawn in. This allows us to redraw (for example) a LOESS curve and only a LOESS curve without having to redraw the entire image that it originally appeared in. This is because we can revisit the drawing context that the curve was created within. It is also useful because coordinate information can be used in JavaScript without needing to query R to obtain a location or dimension from a particular viewport.

In JavaScript it is rare for any complex coordinate systems to be used, or even viewports. In SVG it is possible to specify a location or dimension by using pixels or an absolute unit (e.g. inches, centimetres), or a relative unit (e.g. a percentage) but no complex units — units composed of more than one type of measurement — can be used. Many JavaScript-based graphing libraries provide a facility to determine a location via a numeric scale that generates the correct SVG location. A limitation of the approach these JavaScript libraries take is that often the only units that can be used are either the equivalent of grid's "native" units, or an absolute unit. Using an exported grid coordinate system enables us to use any grid unit and allows for more complex units to be used. This means that it is possible to plot a point with an x-location of 0.5 "npc" plus 2 centimetres, relative to a particular viewport. Without the ability to use viewports (or a similar concept) or complex units in a JavaScript implementation, we cannot perform the described positioning (similarly for lengths and widths).

9.1.3 Node-based SVG

The ability to generate an SVG image as a collection of in-memory XML nodes has several advantages. The initial motivation for rewriting gridSVG's output layer to use the XML package was to ensure that images could be generated in memory. This is not strictly necessary but it is advantageous when attempting to serve a gridSVG image directly from a web server because it avoids the need to use a hard disk for temporary storage that is an additional and unnecessary complication.

There have been further benefits from the switch to the node-based approach. Many of these are a convenient consequence of using the XML. For example, when JavaScript code and SVG metadata is inserted into the SVG image by gridSVG, it is performed after the rest of the image has been generated. This demonstrates the flexibility of the new approach by allowing us to work with a workable document tree, rather than simply some text in a file.

The greatest advantage to using node-based SVG generation is that it allows us to *query* a document that is resident in memory. This means we can read attributes from specific nodes using standardised and easy to use tools instead of regular expressions. Being able to get a piece of a gridSVG-generated image is also easy. In essence, we now immediately have a DOM to work with, which gridSVG previously lacked. This is important because getting a piece of a plot or indeed specific attribute values from a plot is necessary if we want to send that information to a web browser.

9.1.4 Content Selection

It is now possible to select content in R in the same way that is possible to select any content in a web page, using CSS selectors. The selectr package allows us to use the same selectors on a gridSVG image in R as we would in a web browser with JavaScript. This avoids the need to write two different selector queries — one XPath expression and one CSS selector — which may be prone to error and adds unnecessary complexity. Content manipulation can more easily be replicated between R and an equivalent operation in JavaScript because the selection of SVG content is expressed in the same way.

9.1.5 Animation Sequencing

One of the key advantages of animation with SVG is that we can declaratively animate content, clearly describing an animation action, when that action occurs and for how long it occurs. gridSVG is able to animate SVG content using `grid.animate()`. This makes use of SMIL animation (Synchronized Multimedia Integration Language) (W3C SYYM Working Group, 2001) which allows us to encode animations within an SVG image. Using SMIL animations is preferred to any frame-based alternatives because it is faster in most browsers, in addition to being more flexible. For example, when describing how a circle moves from its original x-location of x_0 to x_t, where x_t is the new location at time t, a naive frame-based implementation may interpolate between the locations linearly. This describes what is called an easing function, specifically this is a linear easing function. There are many easing functions that can be used but the important thing to note is that changing between them in SVG is as simple as changing the value of an attribute in an `<animate>` element. This is exposed in `grid.animate()` via the `interpolate` argument. Performing the same task using frame-based animations is complex because it often requires writing the easing functions and calculating each interpolated step of the animation ourselves in addition to procedurally applying the animation.

The main limitation of SMIL animation is that all information regarding animations must be known at the time that the SVG image was generated. CSS transitions allow us to modify the image using animation with all of the same advantages that SMIL animation has but transitions are not required to be known at the time the SVG image was generated. The D3 library is particularly powerful in performing this task so gridSVG images are easily animated post-creation using D3. This alone demonstrates the benefits of leveraging JavaScript as we can "stand upon the shoulders of Giants" but it is not sufficient for accurately describing (and applying) a complex animation sequence. When constructing animation sequences (such as those from the Visual Inference Tools package) the animaker package is particularly useful because it captures the intent of the animation in an easy and flexible manner.

animaker is designed only to describe an animation sequence, delegating any animation actions to other software. By exporting descriptions created by animaker to JSON, we allow alternative systems to use the timing scheme generated by animaker. To make this easy to use in JavaScript, the TimingManager library was created. This means that gridSVG graphics can now systematically build and apply animation sequences that leverage SMIL animation and CSS transitions.

Without animaker and TimingManager to describe animation sequences, and D3 to apply them, animations acting upon existing gridSVG graphics would be far more difficult to perform.

9.1.6 Advanced SVG Content

gridSVG is no longer limited to drawing content that grid can draw in R. By leveraging features of SVG that are not available in grid graphics we can produce plots that appear more sophisticated. The features that can now be drawn are patterns, opacity masks, non-rectangular clipping paths, filter effects and gradients. Instead of simply being a thin wrapper to SVG elements, gridSVG provides an interface that is familiar to users of grid graphics. This means that these advanced features can make use of grid facilities like grobs, viewports and units.

The advantage of exposing these features to a user, even if they cannot view them in R, is that implementing these features in other systems (for example in D3) is very difficult. This may go some way towards explaining why they are uncommon in SVG plots, particularly *statistical* SVG plots.

Finally, it is now possible to directly add SVG elements to gridSVG images using `grid.element()`. Anything that cannot drawn using the available gridSVG features can

now be drawn using a grob-like interface to SVG elements. This is unlikely to be necessary for most use cases but in the rare circumstance where more extensive customisation of a gridSVG plot is necessary it may prove useful.

It remains to be seen exactly how popular these features will be. This is primarily because they are not commonly used by existing SVG plots on the web. Because these features are not available in R graphics, there may be users who want to utilise the features using R but, until now, could not do so. It may be the case that people are not using these features because they are not useful, or perhaps that they cannot or find it difficult to do so. Regardless, the fact that gridSVG now provides an interface to advanced SVG content means that it is now possible to draw these things in grid graphics, which was not the case previously.

9.2 Complexity

We have achieved the goal of creating plots whose pieces can be manipulated, but one of the problems is that there are many separate pieces of software that are necessary to create them. For a typical gridSVG web application that we can now create, we require someone to know all of the following tools and technologies: HTML, CSS, R, grid, gridSVG, an R web server, and D3 (and/or other JavaScript libraries). Although R is commonly used amongst the statistical community, the web-based technologies are not. In particular, JavaScript is unlikely to be known by many statisticians. However, if the goal is to create web applications with responsive gridSVG graphics, then it is a reasonable request that people understand the technologies that they are working with.

The broad programming knowledge required to create these applications may limit its adoption. However, if people do possess the necessary knowledge the potential applications are vast — far more is possible than the simple examples shown in chapter 8.

9.3 Limitations

Although it is possible to create sophisticated and responsive grid graphics for use in a web browser there are several limitations that are present.

One of the key limitations is that gridSVG is limited to only exporting grid graphics. This means that the vast library of plots that have been created in R's base graphics engine are not able to be enhanced by gridSVG. The only option for exposing these plots to a web browser containing features similar to those provided by gridSVG are with

SVGAnnotation or animation. Graphics that are dynamically generated by packages like shiny can also use base graphics because they often rely upon the png() graphics device.

It is also the case that gridSVG is not a complete export of a grid scene to SVG either. Some features are not yet supported, such as viewport rotation, but the vast majority of grid content can be exported without issue. One of the key issues present is when custom grobs are created and used by packages other than grid. In order to be exported to SVG correctly, they need to support "forcing" (Murrell, 2012c). This means that custom grobs need to support the grid makeContent() and makeContext() methods. If these are not present and are necessary, then the grobs will not be exported by gridSVG. Unfortunately, to correct this problem requires package authors to update their grid code to support these methods, which may be a time-consuming process for someone who simply wishes to use gridSVG in the same way they can see content in grid.

Related to the previous limitation is that gridSVG is heavily reliant on grid grobs having *useful* names. This is because automatically generated grob and viewport names are not reproducible (or at least should not be relied upon) and also that it is difficult to determine exactly what a particular grob name refers to. For example, without a reasonable grid naming scheme in place, such as the scheme present in lattice (Murrell, 2012b), it is difficult to work out what the names GRID.text.1 and GRID.text.2 might refer to — are they titles or legend text or other plot annotations? We cannot quickly establish the answer to this question without reasonably named grid grobs. If we have reasonably named grobs, image manipulation in both gridSVG and JavaScript is simplified greatly.

Another limitation of gridSVG is one that can only be worked around and not fixed — gridSVG is slow. In general, most images can be produced within a few seconds but it is possible to spend a long time generating images. This is partially a consequence of R being a slow environment to execute code within, but also that gridSVG's operations depend on the XML package. XML is sufficiently fast for most operations but it can be slow to generate some images. In general, the limitation of gridSVG being slow also applies to vector graphics in general, i.e. when a grid scene has a sufficiently large amount of grobs (or sub-grobs) it is appropriate to use a raster image format instead (e.g. PNG). The raster image in this case will create an image faster, in addition to producing a smaller image file. However, with raster images we do not have the benefits of what SVG and indeed gridSVG can offer. A trade-off is clearly present; we can generate an image quickly without any additional features, or choose to have dynamic and interactive images where the cost is time.

It is known that gridSVG plots can be interactive by using JavaScript to manipulate

SVG content. However, it is difficult to modify anything to a great degree without communicating with R. This step is quite slow in comparison to systems with interactive graphics. Desktop software such as Mondrian (Theus and Urbanek, 2008) and GGobi (Cook and Swayne, 2007), and the R package rgl (Adler and Murdoch, 2013) are designed to be able to manipulate a 2D or 3D scene simply by using a mouse cursor and/or keyboard shortcuts. These rely on the use of hardware accelerated drawing and a custom graphics system to create highly interactive scenes. Alternatively, D3 also provides the ability to generate highly interactive plots such as a spinning globe with cartographic projections being generated immediately. Such plots are impossible with gridSVG because it requires live code to generating graphical content, and therefore a request-response model is inappropriate as it is too slow. By restricting ourselves to enhancing R graphics we not only limit ourselves to 2D graphics, but also to plots that are not as interactive as those on alternative systems.

A final limitation is that it can be a time-consuming process to generate something as simple as a plot with tooltips. This is because there are no high level functions that automate this process. gridSVG only provides the *tools* to generate highly interactive and reactive graphics.

9.4 Future Directions

There are some areas of the SVG specification and indeed grid graphics that gridSVG does not fully support. gridSVG could become a more complete piece of software if it filled these gaps in functionality. These missing features include viewport rotation and also some of the more complicated SVG animation features. Viewport rotation may be particularly complicated to implement due to features such as animation, which may become a lot more complicated as a result. The SVG animation features, implemented by the `<set>`, `<animateMotion>` and `<animateColor>` elements could be supported by gridSVG's `grid.animate()` function. This would allow, for example, a colour to animate between red and blue as part of an animation sequence without requiring any knowledge of SVG. Currently this is possible but it requires knowing what the appropriate SVG attribute names are, which we would like to avoid.

The clearest direction for gridSVG in the future is for additional R packages to build upon the existing tools that have been provided. This means that instead of having to write a significant amount of code to get a simple web application with an interactive gridSVG plot, a user should simply be able to call a high level function that performs this task instead. Such extensibility is one of R's great assets and many packages take

advantage of it. In fact, is it rare to find a contributed package that does not build upon other contributed packages.

What would be ideal is if there were packages that built directly upon gridSVG in the same way that shiny builds upon the websockets package. In other words, a package that is able to conveniently provide high level functionality not only to gridSVG but also for gridSVG-based web applications would be a good avenue for future development.

9.5 Conclusion

The development of the gridSVG package has made it possible to create a web application where pieces of plots generated by R can be created, updated, or removed. Not only is it now possible to perform these tasks, many tools and functionality have been created to make this a relatively easy process. The development of the gridSVG, selectr and animaker packages enable us to produce reactive web-based graphics that are both interactive and animated.

By combining the facilities present in a web browser with computational and graphical facilities provided by R, we can produce interactive R graphics that react to changes in the browser. As a result, animation can be applied to these changes, clearly demonstrating the visual effects of a change in state.

The development of gridSVG has achieved the goal of enabling gridSVG to be a bridge between R graphics and the facilities provided by a web browser.

Bibliography

Adler, Daniel and Murdoch, Duncan (2013). *rgl: 3D visualization device system (OpenGL)*. R package version 0.93.935. URL: http://CRAN.R-project.org/package=rgl.

Becker, Gabriel and Temple Lang, Duncan (2013). *RFirefox*. URL: https://github.com/gmbecker/RFirefox.

Blejec, Andrej (2011). 'animatoR: Dynamic Graphics in R'. In: *useR! 2011 Conference*. URL: http://web.warwick.ac.uk/statsdept/user2011/TalkSlides/Contributed/16Aug_1600_FocusII_3-Visual_1-Blejec.pdf.

Bostock, Mike (2013). *D3: Data Driven Documents*. Version 3.1.10. URL: http://d3js.org/.

Cook, Dianne and Swayne, Deborah F. (2007). *Interactive and Dynamic Graphics for Data Analysis*. Springer New York. ISBN: 978-0-387-71761-6.

Crockford, Douglas (2006). *JSON: JavaScript Object Notation*. URL: http://json.org/.

CSS Working Group, W3C (2011). *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification*. URL: http://www.w3.org/TR/CSS2/.

FFmpeg Team (2013). *FFmpeg*. URL: http://ffmpeg.org.

Heinkel, Ralph (2013). *pyRserve*. Version 0.7.0. URL: https://pypi.python.org/pypi/pyRserve/.

Horner, Jeffrey (2012). *rApache: Web application development with R and Apache*. URL: http://www.rapache.net/.

— (2013). *Rook: Rook - a web server interface for R*. R package version 1.0-9. URL: http://CRAN.R-project.org/package=Rook.

Ignac, Marcin (2001). *Timeline.js*. URL: http://marcinignac.com/blog/timeline-js/.

ImageMagick Studio LLC (2013). *ImageMagick: Convert, Edit, and Compose Images*. URL: http://www.imagemagick.org/.

Lewis, B. W. and Horner, Jeffrey (2012). *websockets: HTML 5 Websocket Interface for R*. R package version 1.1.7. URL: http://CRAN.R-project.org/package=websockets.

Love, Jamie (2013). *R-Node*. Version 0.2.0. URL: http://squirelove.net/r-node/.

Mozilla Foundation (2013). *Popcorn.js*. URL: http://popcornjs.org/.

Murrell, Paul (2012a). 'It's Not What You Draw, It's What You Don't Draw'. In: *The R Journal* 4.2, pp. 13–18. URL: http://journal.r-project.org/archive/2012-2/RJournal_2012-2_Murrell2.pdf.

— (2012b). 'What's in a Name?' In: *The R Journal* 4.2, pp. 5–12. URL: http://journal.r-project.org/archive/2012-2/RJournal_2012-2_Murrell.pdf.

— (2012c). *Writing grid Extensions*. Tech. rep. University of Auckland. URL: http://www.stat.auckland.ac.nz/~paul/Reports/CustomGrobs/custom-grob.html.

Murrell, Paul and Potter, Simon (2012). *Generating Animation Sequence Descriptions*. Tech. rep. Auckland, New Zealand: The University of Auckland. URL: http://stattech.wordpress.fos.auckland.ac.nz/2012-11-generating-animation-sequence-descriptions/.

— (2013a). *Generating Structured and Labelled SVG*. Tech. rep. University of Auckland. URL: http://www.stat.auckland.ac.nz/~paul/Reports/labels/labels.html.

— (2013b). *gridSVG: Export grid graphics as SVG*. R package version 1.2-0. URL: http://r-forge.r-project.org/projects/gridsvg/.

Nolan, Deborah and Temple Lang, Duncan (Jan. 2012). 'Interactive and Animated Scalable Vector Graphics and R Data Displays'. In: *Journal of Statistical Software* 46.1, pp. 1–88. ISSN: 1548-7660. URL: http://www.jstatsoft.org/v46/i01.

Potter, Simon (2012). *Introducing the selectr Package*. Tech. rep. Auckland, New Zealand: The University of Auckland. URL: http://stattech.wordpress.fos.auckland.ac.nz/2012-10-introducing-the-selectr-package/.

R Development Core Team (2013). *R: A Language and Environment for Statistical Computing*. ISBN 3-900051-07-0. R Foundation for Statistical Computing. Vienna, Austria. URL: http://www.R-project.org/.

Reinholdsson, Thomas (2013a). *rHighcharts: An R wrapper for Highcharts JS*. R package version 1.0. URL: https://github.com/metagraf/rHighcharts/.

— (2013b). *rVega: An R wrapper for Vega*. R package version 0.1. URL: https://github.com/metagraf/rVega/.

RStudio, Inc. (2013). *shiny: Web Application Framework for R*. R package version 0.5.0. URL: http://www.rstudio.com/shiny.

Santini, Alberto (2013). *node-rio*. Version 0.8.1. URL: https://github.com/albertosantini/node-rio/.

Sapin, Simon and Bicking, Ian (2013). *cssselect*. Version 0.8.0. URL: https://pypi.python.org/pypi/cssselect.

Sarkar, Deepayan (2008). *Lattice: Multivariate Data Visualization with R*. ISBN 978-0-387-75968-5. New York: Springer. URL: http://lmdvr.r-forge.r-project.org.

Temple Lang, Duncan (2013). *RJSONIO: Serialize R objects to JSON, JavaScript Object Notation*. R package version 1.0-3. URL: http://CRAN.R-project.org/package=RJSONIO.

Temple Lang, Duncan and Nolan, Deborah (2011). *XML: Tools for parsing and generating XML within R and S-Plus*. R package version 3.96-1.1. URL: http://CRAN.R-project.org/package=XML.

The jQuery Foundation (2013). *jQuery: Write Less, Do More*. Version 2.0.2. URL: http://jquery.com/.

Theus, Martin and Urbanek, Simon (2008). *Interactive Graphics for Data Analysis: Principles and Examples (Computer Science and Data Analysis)*. Chapman & Hall/CRC. ISBN: 1584885947, 9781584885948.

Turbelin, Clément (2013). *rserve-php*. URL: https://code.google.com/p/rserve-php/.

Urbanek, Simon (2003). 'Rserve: A Fast Way to Provide R Functionality to Applications'. In: *Directions in Statistical Computing*. R package version 1.7-0.

— (2013). *REngine*. URL: http://www.rforge.net/Rserve/.

Urbanek, Simon and Horner, Jeffrey (2013). *FastRWeb: Fast Interactive Framework for Web Scripting Using R*. R package version 1.1-0. URL: http://www.rforge.net/FastRWeb/.

Vaidyanathan, Ramnath (2013). *rNVD3: Interactive Charts from R using NVD3.js*. R package version 0.0.1. URL: https://github.com/ramnathv/rNVD3/.

Vaidyanathan, Ramnath and Reinholdsson, Thomas (2013). *rCharts: Interactive Charts using Polycharts.js*. R package version 0.3.0. URL: https://github.com/ramnathv/rCharts/.

Veillard, Daniel (2012). *The XML C parser and toolkit of Gnome*. Version 2.9.0. URL: http://www.xmlsoft.org/.

W3C CSS Working Group (2011). *Selectors Level 3*. W3C Recommendation. URL: http://www.w3.org/TR/css3-selectors/.

— (2013). *CSS Transitions*. W3C Working Draft. URL: http://www.w3.org/TR/css3-transitions/.

W3C SVG Working Group (2011). *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*. W3C Recommendation. W3C. URL: http://www.w3.org/TR/SVG/.

W3C SYYM Working Group (2001). *SMIL Animation*. W3C Recommendation. W3C. URL: http://www.w3.org/TR/smil-animation/.

W3C Web Applications Working Group (2004). *Document Object Model (DOM) Level 3 Core Specification*. W3C Recommendation. W3C. URL: http://www.w3.org/TR/DOM-Level-3-Core/.

W3C XML Working Group (2008). *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. W3C Recommendation. W3C. URL: http://www.w3.org/TR/XML/.

W3C XSL and XML Linking Working Groups (1999). *XML Path Language (XPath) Version 1.0*. W3C Recommendation. http://www.w3.org/TR/xpath/. W3C.

Waldron, Rick (2010). *burst-core*. URL: https://github.com/rwldrn/burst-core.

Wild, Chris et al. (2013). *vit: Visual Inference Tools*. R package version 0.1.0. URL: http://www.stat.auckland.ac.nz/~wild/VIT/.

Xie, Yihui (2013). 'animation: An R Package for Creating Animations and Demonstrating Statistical Methods'. In: *Journal of Statistical Software* 53.1, pp. 1–27. ISSN: 1548-7660. URL: http://www.jstatsoft.org/v53/i01.