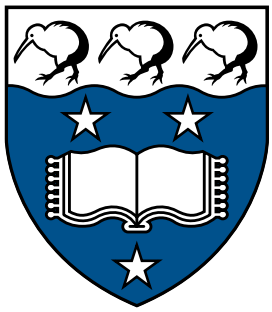


University of Auckland

Department of Statistics

Web-based Interactive Graphics with gridSVG

Author:
Simon J. Potter

Supervisor:
Dr. Paul Murrell

June 27, 2011

Abstract

We describe an approach taken for the creation of animated and interactive graphics using the grid graphics engine for the statistical software package R. In order to create these graphics an existing package for R, gridSVG, is extended from a proof-of-concept into a usable solution. gridSVG produces SVG images that are animated, interactive and can be included within web pages.

Contents

List of Figures

List of Listings

List of Tables

1 Aim

The statistical software package R (R Development Core Team, 2011) is freely available and widely used amongst statisticians. The intention of this project is to produce animated and interactive plots on web pages using R. These plots can convey more information and engage a reader better than regular static graphics. Currently, the creation of these plots is not possible in R without difficulty. A package for R does exist that can create the type of plots we want, but it is not capable of producing any but the most basic of statistical graphics. This package, gridSVG (Murrell, 2011), needs to be improved upon in order to create useful animated and interactive graphics.

As a first step the gridSVG package requires extending so that it is capable of reproducing the appearance of plots created by R's native graphics devices. Once this is accomplished, further work can be done to improve gridSVG's capacity to include animation and interactivity to its plots.

2 Introduction

2.1 What are web-based interactive graphics?

It must be established what are web-based interactive graphics in order to illustrate the intended goal of this project. Web-based graphics are images that are able to be viewed within a web page by a web browser. The graphics we intend to produce are not only web-based, but also have the properties of animation and interactivity. Interactivity involves changing the behaviour or appearance of an image, most commonly by the use of a mouse or keyboard.

2.2 Existing solutions

There are currently a few notable packages for R that do allow for the creation of web-based interactive graphics. A description of how these packages work follows, in order to explain why gridSVG is being improved upon.

The animation package (Xie, 2011) can create animated graphics in many image and video formats, most of which are not provided by R. In order to produce animated graphics, the animation package generates a series of static plots. Each plot shows the animation at a specific point in time. This means that long and fluid animations will generate a large amount of static plots in comparison to shorter, "choppy" animations. By piecing all of the static plots together, the illusion of animation is created.

animation relies heavily on the use of software not present within the package to produce many of the different graphics formats it supports. In fact, the only formats that do not have any dependencies on third-party software are on-screen animations and HTML pages. On-screen animations have the drawback of being unable to be stored in any way. The GIF, Flash, PDF and video formats that animation supports all require software additional to R.

Other packages have been released but they leverage other graphics systems to implement any animation or interactivity. These packages include webvis, googleVis and gWidgetsWWW.

The webvis package (Conway, 2010) currently uses the Protovis JavaScript library to produce its plots. The approach that webvis takes is to translate the graphical functions that R provides into equivalents that utilise the Protovis library. Use of the `plot.webvis` function is expected to produce similar results to a plot that is created using R's `plot` function. It is also possible to construct a graph using Protovis-specific functions, e.g. using `pv.line` to add a line. This approach requires knowledge of Protovis and the code to produce plots within R will be very similar to equivalent JavaScript code.

The googleVis package (Gesmann and de Castillo, 2011) provides an interface for R to Google's Visualisation API. googleVis allows the creation of plots that use Google's graphics library. This means that any plot that can be created with this library can be used by googleVis. An example of the graphics that googleVis can create is an interactive map that places markers at locations on the map. Many of the types of plots available in googleVis are widely used in highly visible Google products. This ensures that the plots are going to be highly polished in both appearance and behaviour.

gWidgetsWWW (Verzani, 2011) is a package that provides an HTML and JavaScript implementation of the gWidgets package for R. gWidgets implements a generic interface for creating interactive GUIs allowing the same R code to work in multiple GUI toolkits. This means that we can use gWidgetsWWW to create a graphical user interface that responds to user input. When used in conjunction with RApache, an R module for the Apache HTTP Server, gWidgetsWWW can add interactivity to a web page.

A package developed by Lang (2010) that generates animated, interactive graphics via the R graphics system is SVGAnnotation. It leverages R's `svg()` graphics device by post-processing its output to see which SVG elements correspond with specific components of a plot. After performing the post-processing, animation can occur along with interactivity via JavaScript. Many functions have been provided that allow for interactivity in processed plots. Examples of these functions include adding tooltips, animating graphical elements and linking related points.

2.3 Motivation for gridSVG

Here we discuss why the currently available R packages are not suitable for our needs when creating web-based interactive graphics.

We deem animation to be unsuitable for our needs because it does not provide any means of interactivity, only animation. Moreover, the animation is does produce is unsuitable. Rather than drawing several plots to generate an animated image, we would rather draw a plot only once that has animation embedded within it. The distinction is similar to the difference between a cartoonist who has to draw every single frame, and a director that simply tells people what to do.

The packages webvis and googleVis are also unsuitable because they do not use R's

graphics system to create their plots. We would like to use the facilities R provides with its powerful graphics system to create animated and interactive graphics. This method ensures that the graphics we see in R are what is actually going to be drawn when we create our interactive plots.

gWidgetsWWW does create the kind of graphics we want to create, with facilities present for powerful interactivity. What it does not provide is any means of animating a plot once it has been created. It can only be done in a manner similar to animation, which is impractical to serve frame by frame over the internet due to latency and network speed.

SVGAnnotation can also create the kind of graphics we desire, but it is difficult to understand how it works. The approach SVGAnnotation takes to match SVG elements with graphical objects in R requires knowledge of the expected SVG output of the `svg()` device. This introduces a lack of transparency because it relies on reverse engineering R's `svg()` device's output. Because of this, it is challenging to extend SVGAnnotation's features to create animated and interactive graphics that are not provided out-of-the-box. Another issue to consider is that if SVGAnnotation does not work as expected then there is little that a user can do.

None of the existing solutions produce the kind of graphics we wish to create. However, a package exists that has the potential to create animated and interactive graphics in a transparent manner. This package, gridSVG, can only produce basic plots as it is currently little more than a proof-of-concept. By extending gridSVG to be capable of producing more complex plots we should be able to create sophisticated web-based interactive graphics.

3 The design of gridSVG

3.1 What is grid? How does it work?

In order to explain how gridSVG works, we must first explain how grid works. grid (Murrell, 2005) is a graphics system that is provided by R along with the base graphics system. Two key features of grid separate it from the base graphics system. The first of these features is the concept of viewports. Viewports are a convenient way of defining a plotting region and setting a drawing context. When using viewports, all drawing is relative to the coordinate system within the current viewport.

```
> library(grid)
> grid.newpage()
> grid.rect(gp = gpar(lty = "dashed")) # Showing viewport size
> grid.circle() # Draws a circle as large as the root viewport
> # Moving into a new viewport
> pushViewport(viewport(x = 0.5, y = 0.5, height = 0.5,
+                       width = 0.5, just = c("left","bottom")))
> grid.rect(gp = gpar(lty = "dashed")) # Showing viewport size
> grid.circle() # Draws a circle as large as the current viewport
> popViewport() # Leaving the viewport
```

Listing 3.1: Using viewports to change the location and dimensions of the plotting region.

Listing 3.1 demonstrates how the code used to produce a circle remains constant, but the position and dimensions of the circles are dependent on the viewports they were drawn in. This is shown by first drawing a dashed rectangle to show the size of the entire plotting region. A circle is then drawn as large as the size of the viewport it occupies. The root viewport is as large as the entire plotting region, because of this, so is the circle that is being drawn by `grid.circle()`. Following this, a new viewport is created that occupies the top-right quadrant of the plot. Now when we use the same code to produce the dashed rectangle and the circle, it only draws within the top-right quadrant. The final line, calling `popViewport()`, leaves the viewport positioned in the top-right quadrant

and returns to the root viewport. The code in Listing 3.1 produces the plot shown in Figure 3.1.

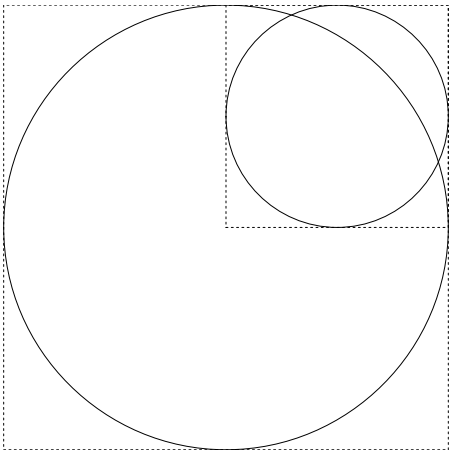


Figure 3.1: An example of grid viewports, produced by Listing 3.1.

The application of viewports allows trellis graphics to be created relatively easily. Several packages provide high level functions that can create these complex plots, notably lattice (Sarkar, 2008) and ggplot2 (Wickham, 2009). lattice uses viewports to create strips and panels within its plots, where each strip and panel is a viewport for graphics objects to be drawn in.

Another feature of grid is that grid graphics functions also produce graphics objects. A graphics object stores all of the information necessary for the object to be drawn. For example, `grid.rect()` creates a graphics object that gives grid enough information to draw a rectangle, then draws it. Associated with every graphics object is a name, this name is an identifier that we can use to inspect or modify a graphics object. Each time a graphics object is drawn, it is recorded on grid's display list. This display list stores all of the graphics objects necessary for an image to be drawn.

```
> library(grid)
> grid.newpage()
> grid.rect()
> grid.ls()
GRID.rect.1
```

Listing 3.2: Inspecting grid's display list.

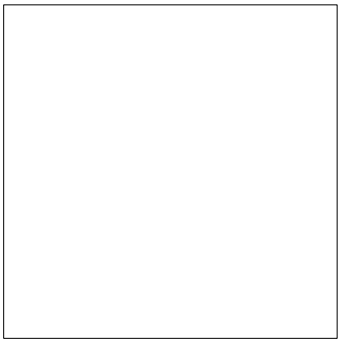
We can inspect grid's display list using the command `grid.ls()` to see which graphics objects have been drawn. The output of `grid.ls()` shows the names of these objects.

Observing the example in Listing 3.2 we can see that the name of the rectangle object that was produced by `grid.rect()` is in fact `GRID.rect.1`.

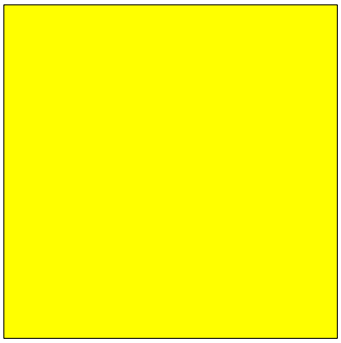
```
> grid.ls()
GRID.rect.1
> grid.edit("GRID.rect.1", gp = gpar(fill = "yellow"))
```

Listing 3.3: Modifying a grid graphics object.

The record of graphics objects in the display list also allows us to modify them. Listing 3.3 demonstrates object modification by changing the colour of the rectangle that was drawn in Listing 3.2. This is possible through the use of `grid.edit()`, which modifies existing graphics objects by specifying the name of the object to be modified as its first parameter. Any parameters following the object name are properties of the graphics object that being modified. In Listing 3.3, the rectangle called `GRID.rect.1` will be filled with yellow instead of being transparent. Figure 3.2 shows the effect of `grid.edit()` on `GRID.rect.1`.



(a) A rectangle produced by `grid.rect()`.



(b) The same rectange modified to be filled with yellow.

Figure 3.2: Modifying an existing graphics object, produced by Listing 3.3.

The naming of graphics objects is particularly important, as we have a way of identifying graphics objects. If we can identify an object, then modifying it is possible. This is the main reason why we are targetting the grid graphics engine and cannot use the base graphics engine. To demonstrate this, if we were to draw a plot using functions provided by the graphics package, e.g. the `plot()` function, we cannot modify elements of the plot once it is drawn. Figure 3.3 creates a plot of random variates but if we wanted to change the x and y axis labels we are forced to draw a new plot.

```
> plot(rnorm(10), rnorm(10))
```

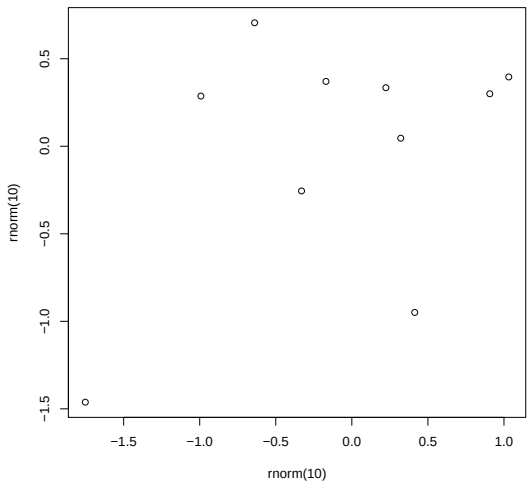


Figure 3.3: A basic plot created using the graphics package.

When saving grid plots, we would like to retain the ability to identify graphics objects on plots we have saved. There are few image formats that R supports that can do this, and only one of these formats is viable for use on the web. This format is SVG.

3.2 What are SVG and JavaScript?

SVG (Scalable Vector Graphics) is an XML-based format for describing two-dimensional vector images. SVG images are described using text, therefore we can use the facilities present in R to write text to a file to create SVG images.

```
<svg height="100" width="100">
  <rect id="example"
        x="30" y="20" height="50" width="30" />
</svg>
```

Listing 3.4: A basic SVG image.

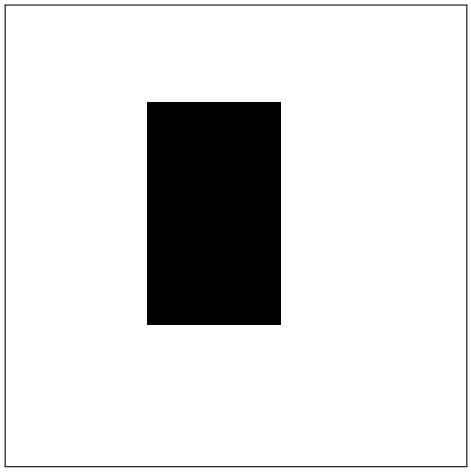


Figure 3.4: The rendered image produced by Listing 3.4.

We can see that the XML code used to produce the image in Figure 3.4 is quite readable. An SVG image has been created that is 100 by 100 units. It contains a `<rect>` element that is positioned at (30, 20) and is 30 units wide by 50 units high. Note that in SVG, the origin is located at the top-left of the plotting region, while in R it is at the bottom-left of the plotting region. This means that higher y values correspond to lower positions on the plotting region. Another important thing to note is the `id` attribute, which gridSVG will use when naming SVG elements.

SVG can also be modified after it has been loaded in a browser through the use of JavaScript. JavaScript is a web-based scripting language that is well-supported by modern web browsers. gridSVG makes use of JavaScript to provide interactivity with the graphics it produces by embedding it within an SVG image. By extending the example from Listing 3.4 we can demonstrate how JavaScript can modify an SVG image.

```
<svg height="100" width="100">
  <rect id="example" onmouseover="setYellow(evt)"
        x="30" y="20" height="50" width="30" />
  <script type="text/ecmascript">
    function setYellow(evt) {
      var rect = evt.target;
      rect.style.setProperty("fill", "yellow");
    }
  </script>
</svg>
```

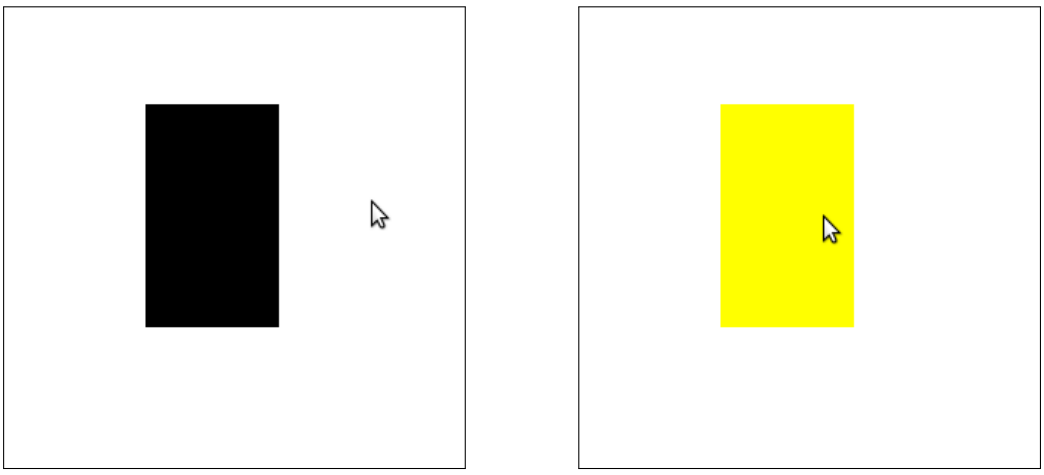
Listing 3.5: An interactive SVG image.

The image in Listing 3.5 features a couple of additions that require explanation. The first of these changes is the addition of a `<script>` element. Within this `<script>` element is the definition of a `setYellow()` function. It takes a single parameter, `evt`, which contains information about the event that triggered its execution. The first line of the function finds out which element triggered the event, and refers to it as `rect`. The second line modifies `rect` so that instead of being coloured black, it is coloured yellow.

The other change to the image is the addition of an `onmouseover` attribute to the rectangle element. This is a special attribute that executes any JavaScript code assigned to it when a mouse cursor hovers over the element. In this example, when a mouse cursor hovers over the `<rect />` element, the JavaScript function `setYellow()` will be executed.

The effect of both of these changes is that this new SVG image appears the same as Figure 3.4 until a mouse hovers over the rectangle. When this occurs its colour changes from black to yellow. This is shown in Figure 3.5.

This combination of JavaScript and SVG is how gridSVG is going to implement interactivity in its plots.



(a) The rendered image prior to any changes.

(b) Triggering a colour change by moving the cursor over the rectangle.

Figure 3.5: The SVG image produced by Listing 3.5.

3.3 Mapping of grid graphics to SVG elements

The task we would like to accomplish is creating SVG images in R with the ability to animate and interact with these images. Unfortunately we cannot use R's `svg()` device to create these plots. The reason for this is that the `svg()` device is only concerned with ensuring that the appearance of SVG output is accurate. This means that plots

created using the `svg()` device are not animated, nor are they able to be modified using JavaScript because the device does not provide the ability to include this information.

gridSVG intends to write to SVG, but with the ability to include information necessary for animation and interactivity to occur. This requires us to write directly from grid to SVG and avoid the `svg()` device altogether. An illustration of how gridSVG differs from most graphics devices is provided in Figure 3.6.

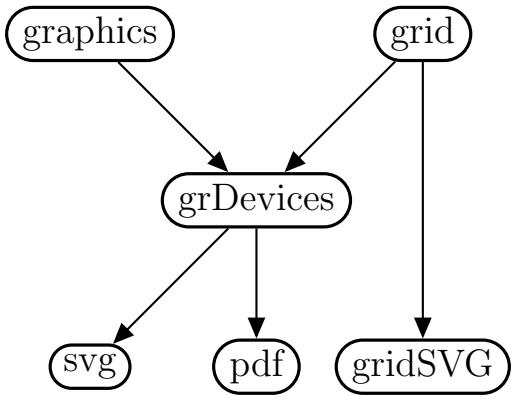


Figure 3.6: Instead of using grDevices to create an SVG image for a grid plot, gridSVG creates the image directly.

The approach gridSVG takes when creating an SVG image is to first capture all of the graphics objects and viewports present in a grid plot. Extra information can then be added to specific graphics objects if interaction or animation is to occur on them. Once the additional annotation of graphics objects has taken place, gridSVG attempts to write out the information it has available to it.

Additional information is added to graphics objects through the provision of functions that modify existing graphics objects. An example of one of these functions is `grid.garnish()` which "garnishes" a graphics object with extra attributes. `grid.garnish()` takes as its first parameter the name of the graphics object that is being modified. Any additional named parameters are additional SVG attributes that the graphics object will have. This is how we can add attributes such as `onmouseover` to implement interactivity.

```
> library(gridSVG)
> grid.rect(gp = gpar(fill = "black"))
> grid.ls()
GRID.rect.1
> grid.garnish("GRID.rect.1", onmouseover = "setYellow(evt)")
```

Listing 3.6: Adding an `onmouseover` attribute to a graphics object.

Listing 3.6 shows how we would typically use `grid.garnish()`. First a black rectangle is being drawn, and we find out that it is named `GRID.rect.1`. We then use `grid.garnish()` to add an `onmouseover` attribute to `GRID.rect.1`. This attribute holds the value of `setYellow(evt)`. Now gridSVG is aware that there is an additional attribute associated with `GRID.rect.1`.

After modifying graphics objects, gridSVG's next task is to translate the graphics objects to SVG. Writing to SVG requires that for each grid graphics object, there is a mapping to SVG elements that adequately represent the grid graphics object. For many graphics objects, this task is simple as there is a direct mapping between a grid graphics object and an SVG element. An example of this is a `rectGrob` (produced by `grid.rect()`) mapping to a `<rect />`. However, for other elements this is not so straightforward because the mapping is not obvious. This will be further explained in Section 4.

When writing to SVG, gridSVG can annotate SVG elements with the same names as the grid graphics objects on the display list. This means that if we have a graphics object named `GRID.rect.1`, we can identify the SVG element(s) that the object translates to. By retaining object names, we can use JavaScript to target these names and interact with the objects the names are associated with.

The extra attributes that are annotated to grid graphics objects also require translating to SVG. This can mean adding attributes on the SVG element that the graphics object maps to. This is the case when "garnishing" a graphics object with extra attributes. However, when animating graphics objects, additional SVG elements must be used to store animation information.

4 gridSVG grows up

While gridSVG was able to produce basic plots, it lacked the ability to handle many of grid's graphics objects. Moreover, gridSVG did not have any understanding of many of the properties that grid graphics objects have. This section details the processes and decisions made when extending gridSVG.

4.1 Mapping of grid graphics objects to SVG elements

4.1.1 Graphics objects and sub graphics objects

When translating grid graphics objects to SVG, there are cases where a one-to-one mapping cannot occur. The reason why this happens is because one grid graphics object can require many SVG elements to represent it. Multiple elements are necessary due to grid graphics objects being able to represent what appears to be multiple graphics objects. This happens when a single call to a grid function produces several visually distinct graphical objects.

Some grid graphics functions provide a way of producing what appears to be multiple graphics objects through the use of an `id` parameter. The reason for this parameter is because multiple graphics are unable to be produced in a single function call without it. An example of one of these functions is `grid.polyline()`. A polyline is a collection of lines, meaning that a single call to `grid.polyline()` can produce the same results as several calls to `grid.lines()`. The `id` parameter is significant in `grid.polyline()` because it provides a way of separating the list of xs and ys into different lines.

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

Listing 4.1: Using the `id` parameter to specify multiple lines.

Listing 4.1 features three notable parameters, `x`, `y` and `id`. Both `x` and `y` are point coordinates for a line to follow. This means that each element of `x` corresponds with an element in `y`. The `id` parameter specifies which line each point belongs to. In this example, the vector that `id` holds is 1..5, repeated four times. This means that for the line with the `id` of 1, we expect the point coordinates to have indices of 1, 6, 11 and 16 because those the the indices where 1 appears in the `id` vector. Given that the `id` parameter has 5 unique values we can determine that 5 lines are being drawn. Using the mechanism described earlier we can determine which points belong to each of these 5 lines. When drawn, the code from Listing 4.1 produces what appears in Figure 4.1.

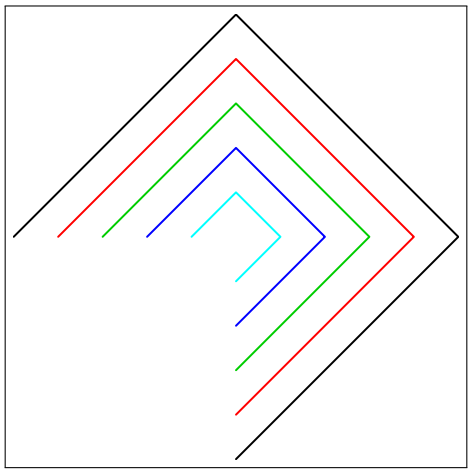


Figure 4.1: Creating a polyline using the `id` parameter, produced by Listing 4.1.

While it is the case that some grid graphics functions are able to create what appear to be multiple objects by using the `id` parameter, this is not the case with most grid graphics functions. Most grid graphics functions are vectorised so that they can handle a vector for a parameter instead of a scalar value. An example of such a function is `grid.circle()`. `grid.circle()` has three key parameters, `x`, `y` and `r`. These parameters govern the x position, y position and radius of the circle respectively. If we provide more than one value for any of these parameters, more than one circle will be drawn. An example of this in action is provided in Listing 4.2

```
> grid.circle(x = c(0.2, 0.7), y = c(0.2, 0.7),
+             r = 0.1, gp = gpar(fill = "black"))
```

Listing 4.2: Using vectorised parameters to create multiple circles.

By providing two values for the `x` and `y` parameters, we are determining the locations of the two circles that are to be drawn. These locations are (0.2, 0.2) for the first circle and (0.7, 0.7) for the second circle. Given that the `r` parameter is a constant of 0.1, we know

that both circles are going to have the same radius. This produces the plot shown in Figure 4.2.

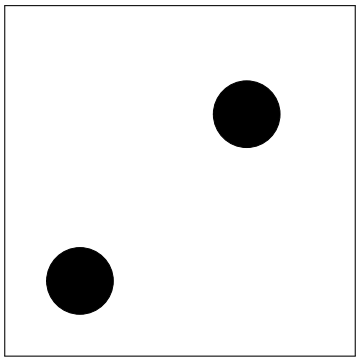


Figure 4.2: Multiple circles created using one grid function call, produced by Listing 4.2.

In both of the cases where a single grid graphics object produces multiple graphical elements we are presented with a problem when writing to SVG. The issue is that SVG has no single elements which can represent the graphics objects that grid produces. We are forced to produce multiple elements for graphics objects where more than graphical element is produced.

The response to the multiple element problem was to create new graphics objects, one for each of the graphical elements that was produced by the function call. These new graphics objects are what is written to SVG, not the object that originally produced it. For example, if a call to `grid.circle()` produces four circles, then we create four circle objects that produce the same result as the call to `grid.circle()`.

Unfortunately this introduces a problem relating to the naming of SVG elements. We would like the names of the SVG elements to match the names of our grid graphics objects. If we create multiple SVG elements for a single grid graphics object, the original name cannot be applied to multiple SVG elements. This is because SVG requires the name of each SVG element to be unique. This problem can be resolved by determining an appropriate naming scheme for cases where multiple SVG elements are produced.

When multiple SVG elements are produced by a grid graphics function with an `id` parameter, the naming scheme is simple. We name each of the resulting elements by using the original name of the graphics object, suffixed by a full stop and the `id` of the graphics object. For example, if a grid `polyline` is named `GRID.polyline.1`, then the line created using an `id` of 2 creates an SVG element with the name `GRID.polyline.1.2`.

grid graphics functions that lack an `id` parameter have a similar naming scheme. Instead of having an `id` determine the new name, we instead use the order in which the new graphical objects are created. For example, returning to Listing 4.2, we know that two circles are being created. The first circle is determined to be the circle located at (0.2,

0.2), because of this it would be given a suffix of 1. If the name of the grid circle object is `GRID.circle.1`, then the first circle that is being produced is going to be named `GRID.circle.1.1`.

These naming schemes do solve the problem of non-unique names but we no longer have the original name of the graphics object available to us. The solution to this is to use an SVG group element (`<g>`). A group element does not change the appearance of an SVG image, but it allows the collection of related elements. This means we can group the multiple elements produced by a graphics object under one group element. The group element can then be assigned the name of the grid graphics object and is guaranteed to be unique.

We can demonstrate this naming scheme by showing the SVG that `gridSVG` produces from Listing 4.2. The relevant subset of the resulting SVG image is shown in Listing 4.3.

```
<g id="GRID.circle.1">
  <circle id="GRID.circle.1.1" ... />
  <circle id="GRID.circle.1.2" ... />
</g>
```

Listing 4.3: Demonstrating the naming scheme applied when creating multiple elements.

To ensure consistency in the SVG output that gridSVG produces, every graphics object will be grouped. This is the case even when multiple elements are not produced from a single grid graphics object. If we observe Listing 4.3, the output would only be slightly different if `GRID.circle.1` only produced one circle. The effect of this would be that the line containing the `<circle>` named `GRID.circle.1.2` will be absent. The modified example does not require multiple elements to be created because only a single circle is created, but grouping it regardless ensures consistent output.

4.1.2 Opacity

A feature of grid graphics that gridSVG was not previously aware of is opacity. Without the support of this feature, semi-transparent graphics objects cannot be drawn.

grid has three ways of applying opacity to graphics objects. All of these methods use the `gp` parameter that is present in all grid graphics objects to apply the opacity. The `gp` parameter takes a `gpar()` object that determines the appearance of a graphics object. It is with this `gpar()` object that we can apply semi-transparency to a graphics object.

The `gpar()` object has three parameters that we are concerned with, `col`, `fill` and `alpha`. `col` determines the colour of lines and borders. We can assign to this parameter a colour created by R's `rgb()` function. The `rgb()` function allows us to specify colours by its RGB components, but we also have access to an `alpha` parameter. This is where

we determine how transparent a colour can be. By creating a colour with an alpha value less than 1, a colour can be semi-transparent.

`fill` behaves in some way as `col`, only differing in how the colour is applied. Rather than defining the colour of lines and borders, it defines the colour used to fill graphics objects like rectangles and polygons.

`alpha` is the graphical parameter that applies opacity to the entire graphics object. This parameter is applied on top the colours set for `col` and `fill`. This means that a rectangle with semi-transparent borders will be even more transparent after applying an `alpha` parameter lower than 1.

```
> library(grid)
> grid.newpage()
> grid.circle(r = 0.2, gp = gpar(col = rgb(0, 0, 0, 0.5),
+                                fill = "black",
+                                lwd = 30))
> grid.newpage()
> grid.circle(r = 0.2, gp = gpar(col = "black",
+                                fill = rgb(0, 0, 0, 0.5),
+                                lwd = 30))
> grid.newpage()
> grid.circle(r = 0.2, gp = gpar(col = "black",
+                                fill = rgb(0, 0, 0, 0.5),
+                                alpha = 0.5,
+                                lwd = 30))
```

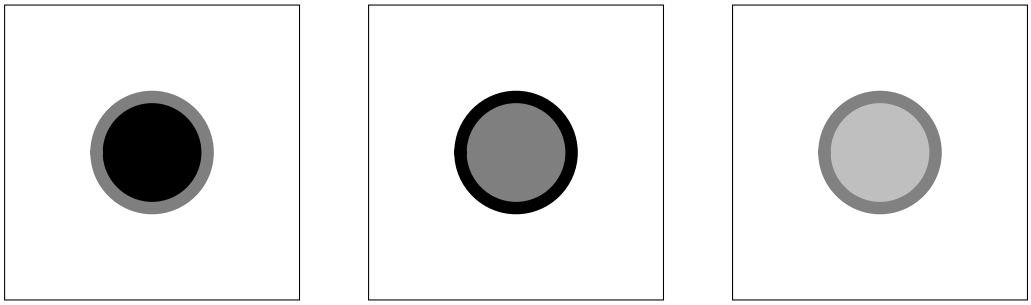
Listing 4.4: Creating circles with semi-transparent components.

Listing 4.4 illustrates the difference between each of these parameters. All three of the circles that are being drawn would be a solid black circle if not for semi-transparency. In all three examples, the background is white, but it could be any colour.

The first circle draws a grey border because it has an alpha component of 0.5. This means that half of its colour is provided by the black colour, and half by the white background. The second circle does the same thing but has a semi-transparent fill instead of a semi-transparent border.

The final circle that is being drawn is the same as the second circle but with the `alpha` parameter set to 0.5. The effect this has is multiplying the alpha components of the `col` and `fill` colours by 0.5. This means that a circle will be drawn as if it had a black border with an alpha component of 0.5. The circle will also be drawn as if the fill colour had alpha component of 0.25.

The circles drawn from Listing 4.4 produce the figures in Figure 4.3.



(a) A circle with a semi-transparent border being applied.

(b) Filling the circle with a semi-transparent colour.

(c) A circle with a semi-transparent fill, with an `alpha` parameter of 0.5.

Figure 4.3: Circles with semi-transparency applied, produced by Listing 4.4.

gridSVG must therefore translate the grid graphical parameters to SVG, but also ensure the same behaviour is applied when semi-transparency is present. Conveniently, the translation is a simple mapping with the same behaviour being applied to semi-transparent elements in SVG. This translation is shown in Table 4.1.

grid Graphical Parameters	SVG Styling Parameters
`col`	`stroke` & `stroke-opacity`
`fill`	`fill` & `fill-opacity`
`alpha`	`opacity`

Table 4.1: Mapping colours and opacities from grid to SVG.

While the mapping from `alpha` is a straightforward and one-to-one, the `col` and `fill` parameters require explanation. SVG requires that colours be specified separately from the associated opacity. This is explains why `col` (and similarly `fill`) needs to be translated to both `stroke` and `stroke-opacity`.

4.1.3 X-splines

The grid graphics system has the capacity to draw a curved line using a set of control points. These lines are known as x-splines. gridSVG was not able to draw x-splines at all prior to extending the functionality this package.

A key feature of x-splines is the ability to define whether an x-spline is open or closed. This has a significant effect on what will be drawn by grid. An open x-spline will draw the line relative to any control points, but no fill colour will be applied. A closed x-spline behaves similarly to an open x-spline, however, a curve is also drawn between the last

control point and the first control point. This closes the spline and allows for a fill colour to be set. A comparison of the two types of splines is shown in Figure 4.4.

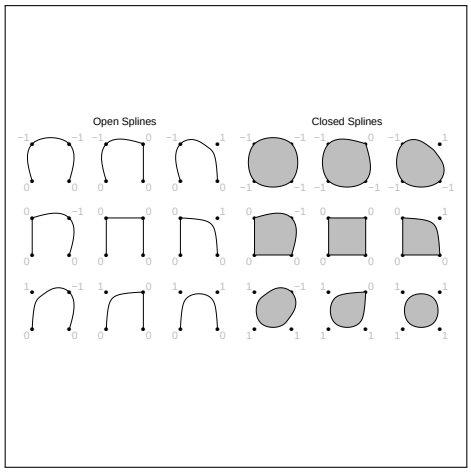


Figure 4.4: Multiple x-splines created using `grid.xspline()`. X-splines on the left and right are almost equivalent, differing only by being open or closed.

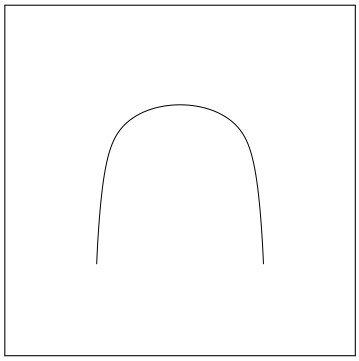
We are presented with a problem when translating the behaviour of x-splines to equivalent SVG code. While SVG paths can draw curved paths, the types of curves that SVG supports are elliptical arcs and variations of Bézier curves. X-splines drawn in grid can support more complex curves. As a result it is not possible to represent all grid x-splines as SVG paths. The solution to this is to use a function provided by grid, `xsplinePoints()`. This function returns a set of points that can be used to draw an approximation of an x-spline as a line.

If the positions that an x-spline passes through is known, emulating the behaviour of the open and closed x-splines is the next logical step. The decision was made to re-use existing gridSVG functionality by implementing an open x-spline as a grid line and a closed x-spline as a grid path. This means that when translating an x-spline graphics object, we inspect the object to see whether it is an open or closed x-spline.

If the x-spline is open, we create a new graphics object that represents a line. Line graphics objects are used because they do not draw a fill colour, and as a result they behave similarly to an evaluated open x-spline. Relevant information from the x-spline graphics object is translated or copied to create the line graphics object. An example of the translation that occurs is using the `xsplinePoints()` function to provide the line coordinates. Most other information is copied, such as the name of the graphics object and the graphical parameters that the x-spline has.

```
> grid.xspline(c(0.25, 0.25, 0.75, 0.75),
+              c(0.25, 0.75, 0.75, 0.25),
+              shape = 1, open = TRUE,
+              name = "openSpline")
```

(a) Creating an open x-spline in grid.



(b) The x-spline in (a) when drawn.

```
<g id="openSpline">
  <polyline id="openSpline.1"
            points="..." />
</g>
```

(c) The SVG code that the open x-spline translates to.

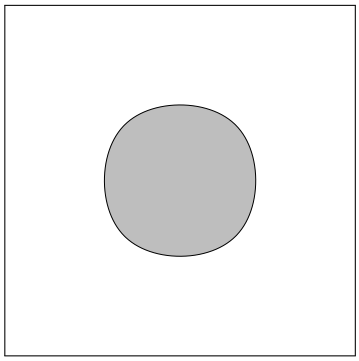
Figure 4.5: An open x-spline being processed by gridSVG.

Figure 4.5 demonstrates the steps that gridSVG takes to translate an open x-spline to SVG code. First, an x-spline is created with four control points. The x-spline is given a name of `openSpline`, but more importantly the `open` parameter is set to `TRUE`. This ensures an open x-spline is drawn and appears as the image shown in Figure 4.5b. The SVG code that was produced uses a `<polyline />` element because that is what a grid line translates to. In creating the `<polyline />` element, the `points` attribute takes values returned from `xsplinePoints()`.

When an x-spline is closed, a path graphics object is created instead of a line graphics object. The path graphics object is created in the same manner as the line graphics object is for open splines. The reason for using grid paths is that paths in grid are closed. A consequence of this is that a line will always be drawn between the last control point and the first control point. If a path is closed, as is the case with grid, then it can be filled with a colour.

```
> grid.xspline(c(0.25, 0.25, 0.75, 0.75),
+              c(0.25, 0.75, 0.75, 0.25),
+              shape = 1, open = FALSE,
+              name = "closedSpline",
+              gp = gpar(fill = "grey"))
```

(a) Creating a closed x-spline in grid.



(b) The x-spline in (a) when drawn.

```
<g id="closedSpline"  >
  <path id="closedSpline.1"
        d="..." />
</g>
```

(c) The SVG code that the closed x-spline translates to.

Figure 4.6: A closed x-spline being processed by gridSVG.

Figure 4.6 provides a similar example to Figure 4.5. The key difference here is the change in the `open` parameter. It is now `FALSE`. This ensures a closed x-spline is drawn and to further illustrate this, the x-spline is filled with grey. As the x-spline is closed and that a grid path is used to represent it, the SVG code uses a `<path />` element. The `d` attribute of the `<path />` element uses output from `xsplinePoints()` to define the path. Comparing figures 4.5 and 4.6, we can see that decision to use different graphics objects has only a minor effect to the resulting SVG code. More importantly, the output that is produced by x-splines is visually accurate.

4.1.4 Arrows

Within the grid graphics system, some graphics objects have the option of being drawn with an arrow. All graphics objects that support arrows are some variation of a line. With an arrow applied a line, it could be used to illustrate a direction or perhaps show an outlier in a plot.

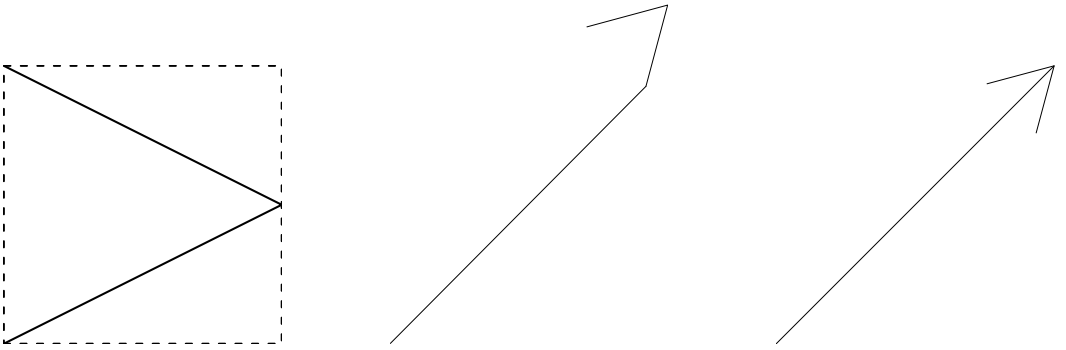
An arrow in grid can appear at the beginning or end of a line or both. There are three ways in which the appearance of an arrow can be changed. The first of these is the angle of the arrow head, this controls how wide an arrow head is. By using larger angles, the arrow head is going to appear wider. The length of an arrow head is defined by the distance from the tip of the arrow to the base. Lastly, an arrow can be either open or

closed which indicates whether the arrow head is a closed triangle or not.

When translating arrows, there is a clear mapping to an SVG element, the `<marker>` element. A useful attribute of the `<marker>` element is `orient`, which takes a value of `auto` by default. This means gridSVG can offload the work of orienting the marker to an SVG renderer. The task for gridSVG is therefore to define how the marker is to appear and linking a `<marker>` element to the corresponding SVG element.

In order to define how the marker appears we use an SVG `<path />` element. Therefore, when we refer to a marker, it is actually a path that is drawn relative to a marker's position and orientation.

SVG provides a slightly different definition for a marker than grid's arrows. A marker is placed at the end of a line, while in grid an arrow is positioned within a line. This means that a marker's position will need to be adjusted to match grid's behaviour.



(a) How a marker is defined, prior to orientation with a line. Dashes indicate marker boundaries.

(b) Applying the marker to a line.

(c) A grid line with an arrow.

Figure 4.7: Comparing the definition and application of the SVG `<marker>` element to grid's arrows.

We can see in Figure 4.7 that although the SVG marker is oriented correctly, it is not positioned at the right place. In order to position the marker correctly, there are two attributes we can use. These attributes are `refX` and `refY`. The effect that these attributes have is they modify the origin of the coordinates used in a marker. For example if we set `refX` to be -3, then the x values of the points in our marker will be increased by 3. The formula we use to reposition markers depends on whether the marker is placed at the end of a line or the beginning of a line. The formulae used are shown in Table 4.2.

Line end	`refX` and `refY` Formula
Start	$\left(-\text{width}, \frac{-\text{height}}{2}\right)$
End	$\left(\text{width}, \frac{\text{height}}{2}\right)$

Table 4.2: Formulae used to reposition a marker. Width and height refer to the dimensions of the marker.

An issue when repositioning is that markers by default do not draw outside of their defined dimensions. This means that when we use `refX` and `refY` to change where the marker path is drawn, at least some of the path will be drawn outside of the defined dimensions. This leaves us with an arrow that is partially obscured. To correct this, we change an attribute on the `<marker>` element. This attribute is `overflow`, which is `hidden` by default and we will change this to `visible`. The effect is that now a marker can be repositioned and will also be completely visible.

Now that we know how to draw markers that appear the same as grid arrows, implementing this functionality is the next logical step. The way gridSVG approaches this task is to first identify if an arrow is present on a graphics object. If there is an arrow present, we note the name of the graphics object that has the arrow. This allows us to give the arrow a reasonable name so that a graphics object can refer to the resulting marker.

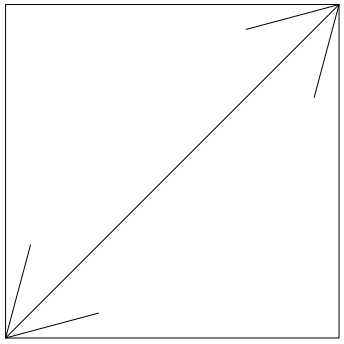
To define how the SVG marker appears we use the angle and length properties of grid arrows. By applying basic trigonometry these properties allow us to calculate the marker's height and width. We can create a `<path />` element that reproduces the arrow using this information. This path would appear similar to Figure 4.7a.

As an arrow can be applied to either end of a line, the markers that we draw in SVG should be able to handle both line ends. This requires producing a `<marker>` element for each line end that arrows appear on. The decision was made to always create two `<marker>` elements so that it is possible in JavaScript to enable or disable arrows at either end.

Once again the issue of a naming scheme is raised as we cannot create two markers with the same name. The naming scheme that is applied to markers is we first take the name of graphics object that contains arrows and suffix it with one of `markerStart` or `markerEnd`. These refer to arrows positioned at the start of a line and end of a line respectively.

```
> grid.lines(arrow =
                arrow(ends = "both"),
              name = "example-line")
```

(a) A grid line with arrows at both ends.



(b) A line with arrows at both ends, produced by Figure 4.8a.

```
<defs>
  <marker id="example-line.1.markerStart" ... >
    <path d="..." />
  </marker>
  <marker id="example-line.1.markerEnd" ... >
    <path d="..." />
  </marker>
</defs>
```

(c) The SVG code that produces arrows.

Figure 4.8: Demonstrating the naming scheme applied to arrows.

Figure 4.8 shows how a grid line named `example-line` is created with arrows at both ends of the line. When this is written out to SVG, the `<marker>` elements created have the name `example-line`, but are suffixed with `markerStart` or `markerEnd`. We can see that paths are drawn within each of the `marker` elements. In our implementation, the paths are in fact identical to each other. An SVG element that has yet to be mentioned is the `<defs>` element. This element allows us to define elements that are able to be used by other elements. This allows us to refer to markers from our line elements.

Given that a reasonable method of creating arrows and referring to them has been established, they must be applied to lines. The way in which SVG allows this to occur is by adding SVG attributes to our line elements. The two attributes we are concerned with are `marker-start` and `marker-end`. These attributes correspond with the arrow that is positioned at the start of the line and the end of the line respectively. If an arrow is only defined as existing at the end of a line, then the `marker-start` attribute is not included. An example of how these attributes are used is included in Listing 4.5

```
<g id="example-line">
  <polyline id="example-line.1"
            marker-start="url(#example-line.1.markerStart)"
            marker-end="url(#example-line.1.markerEnd)"
            ... />
</g>
```

Listing 4.5: SVG code that applies markers to the line produced by Figure 4.8a.

In Listing 4.5 we can see the use of the `url()` function in the `marker-*` attributes. The purpose of this function is to be able to refer to another element within the SVG image. In this case we are referring to the `<marker>` elements that we created earlier in Figure 4.8. Knowing that a consistent naming scheme was in use allowed us to know in advance what the names of the `<marker>` elements would be. Note that the `#` within the `url()` function simply means to search for an element with the given `id`.

We have been able to extend gridSVG to be able to include arrows on line graphics objects. By applying `<marker>` elements and referring to these elements through SVG attributes, the result appears equivalent to what grid draws.

4.1.5 Multi-line text

When drawing text in grid through the `grid.text()` function, there is the ability to split text over multiple lines. This is achieved by including a newline character (`\n`) within a text label. Every time a newline character is encountered, a line break occurs. gridSVG lacked the ability to handle line breaks when manipulating grid text objects. The effect of `\n` was it introduced a newline in the SVG code, splitting the text over multiple lines in SVG. However, whitespace is not significant in SVG code and as a result the SVG text appeared as if it was all one line. This incorrect behaviour is demonstrated in Figure 4.9.

```
> grid.text("Hello,\nworld!")
```

(a) A grid command that produces text with two lines.

(b) The expected image produced by Figure 4.9a.

```
<text ...>
  <tspan>Hello,
  world!</tspan>
</text>
```

(c) The effect that the newline character had on the SVG code.

(d) The output that was produced by gridSVG instead of Figure 4.9b.

Figure 4.9: Demonstrating the incorrect newline behaviour previously present in gridSVG.

The reason for the incorrect behaviour demonstrated in Figure 4.9 is that gridSVG treated every character in a text label as literal text. This means that gridSVG assumed that `\n` simply meant the characters `\n` and not a newline character. gridSVG therefore had to remove this assumption and recognise that some characters have a special meaning associated with them.

To implement multi-line text in SVG, the SVG specification offers two solutions. One solution is to use multiple `<text>` elements to hold the text; one element per line of text. The other solution is to use one `<text>` element but use a `<tspan>` element for each line of text. Both of these methods require the position of each line to be calculated, however the latter option is going to be used. The reason is by having a single `<text>` element, the SVG images that gridSVG creates will be able to support text selection over multiple lines. The first option, using multiple `<text>` elements, cannot provide this feature. Shown below in Figure 4.10 is a comparison of the expected output from the two alternatives.

```
<text>Hello,</text>
<text>world!</text>
```

(a) Multi-line text solution using multiple `<text>` elements.

```
<text>
  <tspan>Hello,</tspan>
  <tspan>world!</tspan>
</text>
```

(b) Using multiple `<tspan>` elements to achieve multi-line text.

Figure 4.10: Comparing the expected SVG code produced from using two different multi-line text solutions.

It is now known which elements gridSVG is going to use and how they are going to be applied. The general approach is that each time a newline character is encountered, create a new `<tspan>` element. This method ensures that each line holds a line of text. The key step remaining is that each line now needs to be positioned correctly.

The issue of line positioning requires mitigating two problems. The first of these is the calculation of a line's height, which depends on a few factors. Secondly, because SVG does not provide a way of vertically justifying multiple lines of text, we must perform this in gridSVG.

Calculating line heights appeared to be reasonably simple. This is because grid's graphical parameters defines a line's height to be `fontsize` × `cex` × `lineheight`. These represent the font size in points, the character expansion and the line height multiplier. The character expansion is a multiplier applied to `fontsize` that determines the height of the characters in a text label. After calculating the character height, the result is multiplied by the value of `lineheight` to get a line's height. These are all numeric values that result in a size measured in points. While this calculation is straightforward, the resulting line height was incorrect for three reasons.

One reason why the line height calculation is inaccurate is due to how fonts are handled in SVG. When a font is specified in SVG, there is no guarantee that it is the font that is used when the image is rendered. This is due to SVG using Cascading Style Sheets (W3C, 2011) for font selection. Font specification in SVG is merely a declaration of the preferred font for text. If we cannot guarantee which font is being used, we can only make a best guess at positioning with the fonts that we know are available in R. Consequently the spacing between lines may be relatively larger or smaller, depending on the font selected by the web browser.

When the size of a line is calculated in terms of points, as is the case in grid, we run into a problem. Although there is a typographical definition for the size of a point ($^1/_{72}$ inches), points as a unit of measurement do not translate well to computer displays. In order to get the correct height of a line we need to apply a workaround in gridSVG.

We first obtain the height of the characters in the line, hereafter referred to as the character height. This involves creating a grid text graphics object with a label consisting

solely of an "M". The new text graphics object is given the same `fontsize` and `cex` as the text we wish to draw. The reason why an "M" is used is because it allows us to approximate the size of the largest character in a line. In other words, the "M" will have the same height as what grid calls a `char`. By using grid's unit conversion functions we can translate the actual height of the "M" into usable SVG units. The resulting height of the "M" text object is the character height within a line.

While grid defines `lineheight` to be a multiplier on the height of the characters in a text label, doing this in SVG produces incorrect output. An example of this is shown in Figure 4.11.

```
> grid.text("Hello,\nworld!",
+           gp = gpar(fontsize = 144,
+                     cex = 1,
+                     lineheight = 1))
```

(a) A grid command that produces text with two lines.

(b) An SVG image using line heights calculated from graphical parameters, using Figure 4.11a.

(c) The output from Figure 4.11a as it appears in R.

Figure 4.11: Comparing line heights between gridSVG creates and what is expected.

Observing the two images in Figure 4.11, we can see that the line height from the SVG image is too small. This is because the character height has been set to 144 points, and the line height is set to be the same as the character height. While the calculation used to arrange text in Figure 4.11b is correct according to our earlier definition, we need it to match the output shown in Figure 4.11c. The reason for the discrepancy in the line heights is due to the behaviour of the R graphics engine. As a result, the line height

does not match what is calculated using graphical parameters. By using the R graphics engine's method of calculating line height we can create the correct output.

While the issue of line height has been discussed, the other key problem in multi-line text is the correct justification of text. grid text graphics objects can be justified vertically to the top, centre and bottom; and horizontally to the left, centre and right. SVG does not provide a way of vertically justifying text so this must be calculated in gridSVG. Curiously, horizontal justification for multi-line text is supported in SVG and gridSVG required no changes to be able to support it. Given that vertical justification is not provided by SVG, gridSVG needed to implement justification algorithms to be able to justify text for three key cases: top, centre and bottom.

The way in which justification occurs in gridSVG requires some explanation. The method of line breaking used is shown in Figure 4.10b. Because we are using a `<tspan>` element for each line, we have useful SVG attributes available to us. These attributes, `dx` and `dy`, refer to the offset relative to the previous line. A `dy` value of 10 on a `<tspan>` element means that we position it 10 units lower than the previous line. Given this information, we only need to justify the position the first line because every line after the first line will have a `dy` equal to the line height. The formulae used to justify the first line of text is shown in Table 4.3.

Justification	First Line Offset Formula
Top	$\texttt{charheight}$
Centre	$-\left(\frac{((\texttt{n}-1)\times\texttt{lineheight})-\texttt{charheight}}{2}\right)$
Bottom	$-(\texttt{n}-1)\times\texttt{lineheight}$

Table 4.3: Formulae used to vertically position the first line of text. `charheight` and `lineheight` refer to the character height and line height respectively. `n` is the number of lines in a text label.

After implementing solutions to the two key problems relating to multi-line text we are able to accurately draw grid text graphics objects. We can observe the SVG output that gridSVG produces given the code from Figure 4.9a. This is shown in Listing 4.6.

```
<text text-anchor="middle" ... >
  <tspan dy="-4.65">Hello,</tspan>
  <tspan dy="19.44">world!</tspan>
</text>
```

Listing 4.6: SVG code that appears as multi-line text, produced by Figure 4.9a.

The SVG code shown in Listing 4.6 has text that is justified to the centre of the image, both horizontally and vertically. Horizontal justification to the centre is declared by

gridSVG setting the `text-anchor` attribute to `middle`. Vertical justification relative to the position of the text is demonstrated by the use of the `dy` attributes on the `<tspan>` elements. The value of 19.44 for the second `dy` attribute indicates that the line height that gridSVG calculated was 19.44. The first `dy` attribute was also calculated to be -4.65 after applying the formula for centre justification in Table 4.3. Upon rendering, the image produces correctly positioned text, as shown in Figure 4.11c.

4.1.6 Fonts

A feature of grid's text graphics objects that has been briefly mentioned is that of fonts. A text graphics object can be drawn using a font specified by the `fontfamily` graphical parameter. The font can also be modified to be drawn in **bold**, *italic*, *oblique* or both ***bold and italic***. To map these features to SVG, we translate grid graphical parameters to CSS code. This CSS code is then applied to `<text>` elements through the use of the `style` attribute.

The `fontfamily` graphical parameter provides the font that is used to draw a text graphics object. There is a direct mapping from the `fontfamily` parameter to a CSS property, `font-family`. This means that the translation is quite transparent, and an example of the translation is shown in Figure 4.12.

```
> grid.text("Hello, world!",
+           gp = gpar(fontfamily = "Helvetica"))
```

(a) grid text using the Helvetica font.

```
<text style="font-family: Helvetica;">
  <tspan>Hello, world!</tspan>
</text>
```

(b) SVG text that uses the Helvetica font.

Figure 4.12: Demonstrating the translation between grid and CSS within SVG.

A key point to note about fonts in SVG is that they are not embedded within the document at any point. The `font-family` property simply declares the preferred font to use, if it is available. This means if you were to view the image that contains the code in Figure 4.12b on a different machine, you may not see the Helvetica font in use.

To ensure that a suitable font is used when viewing SVG text we can assign multiple fonts to the `font-family` property. The set of fonts that we assign to the `font-family` is known as a font stack. The reason we use a font stack is because of the behaviour of the `font-family` property when multiple fonts are present. The first font in the stack is applied if it is possible, otherwise the next font is used, and so on until the stack is exhausted.

The Helvetica font that was referenced earlier can only be guaranteed in an Apple OSX operating system. Fonts with a similar appearance exist on other platforms, such as Arial for Microsoft Windows and FreeSans for open source operating systems like Linux. With this information, we can construct a font stack that makes a reasonable attempt at drawing text that looks similar on most platforms. This improves our example from Figure 4.12b to Listing 4.7.

```
<text style="font-family: Helvetica,
                          Arial,
                          FreeSans,
                          sans-serif;">
  <tspan>Hello, world!</tspan>
</text>
```

Listing 4.7: SVG text that uses a Helvetica-like font on most platforms.

The application of font stacks, as shown in Listing 4.7 allows us to be confident that the appearance of our text is consistent across most platforms. The example shows that Helvetica is first attempted to be used, then Arial, then FreeSans. If none of these fonts are present on a system, we use the web browser's default for a sans serif font. Because we would like gridSVG images to appear consistent across all platforms, a default set of font stacks has been provided. These font stacks cater for the three common types of fonts that are used within R, sans serif, serif and monospace. The output from gridSVG using these fonts stacks appears similar to Figure 4.13.

Figure 4.13: The appearance of different types of fonts in SVG.

While it is useful for gridSVG to be able to handle common fonts, when a font unknown to gridSVG is encountered unexpected behaviour may occur. The reason why this happens

is because it is not known what type of font is being used. If the type of the font is unknown, we cannot provide reasonable fallback fonts. We therefore assume the font is sans serif and will show a reasonable sans serif font in the event that the user-specified font is not present. Figure 4.14 demonstrates this behaviour using the non-existent font "Example".

```
> grid.text("Hello, world!",
+           gp = gpar(fontfamily = "Example"))
```

(a) grid text using the nonexistent font "Example".

```
<text style="font-family: Example, Helvetica,
                          Arial, ..., sans-serif;" >
  <tspan>Hello, world!</tspan>
</text>
```

(b) SVG code that attempts to use the "Example" font prior to rest of the sans-serif font stack.

Figure 4.14: Demonstrating the behaviour when using an unknown font.

Rather than assuming that all fonts unknown to gridSVG are sans serif in nature, two functions were created that allow font stacks to be modified. These functions are `getSVGFonts()` and `setSVGFonts()`.

`getSVGFonts()` returns a list of font stacks currently in use for sans serif, serif and monospace fonts. By editing these font stacks we can choose which fonts we want to appear in our SVG image. Once the list has been modified, we apply the changes by passing the list to `setSVGFonts()`. This allows us to know in advance the effect setting a font has on the resulting CSS applied to an SVG `<text>` element. A typical example of the usage of these functions is shown in Figure 4.15.

```
> fonts <- getSVGFonts()
> # Showing the names of the font stacks we can modify
> names(fonts)
[1] "sans"  "serif" "mono"
> # Setting the monospaced font to be Inconsolata
> fonts$mono <- "Inconsolata"
> # Applying the modified font stack
> setSVGFonts(fonts)
> grid.text("Hello, world!", gp = gpar(fontfamily = "mono"))
> gridToSVG()
```

(a) A basic workflow for modifying a font stack.

(b) The "Inconsolata" font as it appears in SVG.

```
<text style="font-family:
    Inconsolata, monospace;">
  <tspan>Hello, world!</tspan>
</text>
```

(c) The SVG code produced from Figure 4.15a.

Figure 4.15: Setting SVG images to use "Inconsolata" as the default monospaced font.

The code in Figure 4.15a shows how the use of `getSVGFonts()` and `setSVGFonts()` can influence gridSVG's output. The first step taken is to first grab the font stacks that gridSVG is currently using. We then inspect the list of font stacks to see the types of font stacks we can modify. Because we wish to draw monospaced text using the Inconsolata font, we set the monospaced font stack to store only Inconsolata. The font changes are then applied to gridSVG and then written out to SVG. The image shown in Figure 4.15b is the result of the operations, showing that the Inconsolata font is in use. This is confirmed by a subset of the output visible in Figure 4.15c, where the `font-family` property shows Inconsolata being present.

The other grid graphical parameter that controls the appearance of fonts is `fontface`. This is the parameter that determines whether a font is **bold** or *italic*. The mapping from grid to SVG is quite clear, though it requires two CSS properties to be used instead of just `fontface`. These properties are `font-weight` and `font-style`. `font-weight` determines the thickness of a font's text. This is used to embolden text. The `font-style` property is used to modify the shape of the text, for example italicise text. Table 4.4

shows the complete mapping from `fontface` to CSS properties in SVG.

`fontface` Parameter	`font-weight` Value	`font-style` Value
`plain`	`normal`	`normal`
`bold`	`bold`	`normal`
`italic`	`normal`	`italic`
`oblique`	`normal`	`oblique`
`bold.italic`	`bold`	`italic`

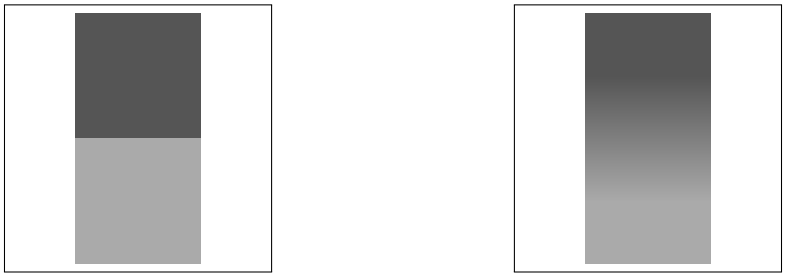
Table 4.4: Mapping the `fontface` parameter to CSS properties.

With the added support of two grid graphical parameters, `fontfamily` and `fontface`, gridSVG can now draw text in SVG that reproduces the appearance of grid text.

4.1.7 Raster Graphics Objects

grid has the ability to include raster images into its plots. This presents a problem when writing to SVG because raster images cannot be represented as a vector graphic, even in SVG. Fortunately, SVG can import raster images (e.g. PNG, JPEG, GIF) through the use of the `<image />` element. This element imports a raster image that is not embedded within the SVG image itself.

When translating grid's raster graphics objects to SVG, producing the SVG code is reasonably simple. However, complications in the translation process arise when we attempt to store raster graphics objects in a raster image format. The reason for the complication is the due to interpolation of raster images. Interpolation is the process of approximating the appearance of an image when it is scaled. Figure 4.16 shows the effect of interpolation using the `interpolate` parameter that is available on raster graphics objects.



(a) A raster image that is resized without interpolation.

(b) A raster image after resizing with interpolation.

Figure 4.16: The appearance of resized raster images.

We can see in Figure 4.16 the behaviour of grid's interpolation. It turns the two pixel image into a gradient. When interpolation is absent, the two pixels are simply scaled to larger dimensions. If we were to scale the same image in a web browser, interpolation is always applied. There is no way of telling a browser, through SVG, not to interpolate an image.

The approach that is taken by gridSVG when there is no interpolation on raster graphics objects is to first write the raster to a PNG image. The PNG image is assumed to have the same dimensions as the space that is occupied by the largest appearance of the raster image in SVG. Despite SVG being resolution independent, we can make this assumption because the SVG images gridSVG produces nominally use pixels for dimensions. The result of the decision to write to SVG dimensions results in PNG images that are not as small as possible. For example, in Figure 4.16 the raster object has only two pixels of data. This could be written to a PNG image that is two pixels high by one pixel wide. Instead, to avoid interpolation we write to a PNG image with larger dimensions, e.g. 200 pixels high by 100 pixels wide.

When a raster graphics object has interpolation enabled, we use the same method of writing to a PNG file as if it were disabled. The reason for this is because a web browser may not interpolate an image in the same way that R does. This is why we don't simply write a PNG image with the same dimensions as the raster graphics object.

The PNG file that is produced is given the same file name as the grid raster object, suffixed by `.png`. This is a clear naming scheme that allows us to reliably know the location that a raster image is saved to. Knowing the location of the raster image is necessary for SVG's `<image />` element, as it imports a raster image into the SVG image from a known location. A demonstration of the SVG image that gridSVG produces from raster graphics objects is shown in Figure 4.17.

```
> grid.raster(matrix(1:2/3, ncol=1))
```

(a) A basic raster image consisting of two pixels, identical in appearance to Figure 4.16b.

```
<g id="GRID.rastergrob.1">
  <image id="GRID.rastergrob.1.1"
         width="284" height="566"
         xlink:href="GRID.rastergrob.1.png" />
</g>
```

(b) The SVG code produced from Figure 4.17a.

Figure 4.17: The mapping of a grid raster image to SVG.

The example in Figure 4.17 shows how a two pixel image, stored in a raster graphics object named `GRID.rastergrob.1` translates to SVG. Firstly, because there may be multiple appearances of the same raster image, we need a group element. The `<image />` element

is given the name `GRID.rastergrob.1.1` because it is the first (and only) appearance of the raster image. The raster image has been turned into a PNG file with the name `GRID.rastergrob.1.png`, with a height and width of 566 and 284 respectively. In order to import this image into SVG, the `xlink:href` attribute uses the predetermined file name.

4.1.8 gTrees, viewports, frames and cellGrobs

In the grid graphics system, almost all graphics objects directly produce graphical content. However, a graphics object that does not do so is a `gTree`. A `gTree` is a graphics object that contains other graphics objects. These graphics objects may be common graphics objects like rectangle objects, but they can also be `gTree`s. An example where this might be applied is drawing text with a rectangular border. This is illustrated in Figure 4.18.

```
> rg <- rectGrob(height = 0.1, width = 0.3)
> tg <- textGrob("gTree Example")
> ex <- gTree(children = gList(rg, tg),
+             name = "example-gTree")
> grid.draw(ex)
> grid.ls()
example-gTree
  GRID.rect.1
  GRID.text.2
```

(a) A basic `gTree` consisting of two graphics objects.

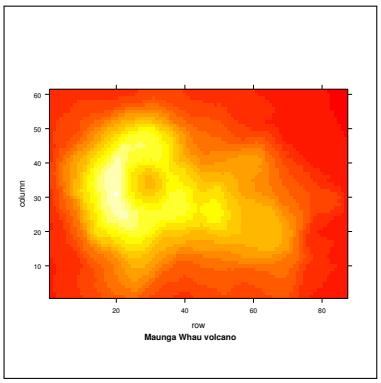
(b) The image produced from Figure 4.18a.

Figure 4.18: Using a `gTree` to create bordered text.

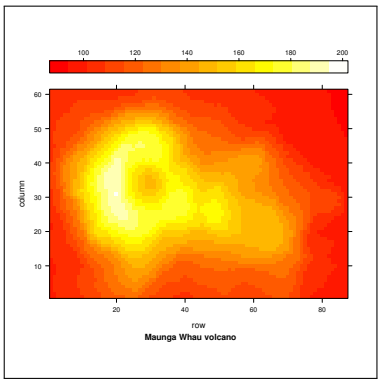
Figure 4.18 requires explanation in order to understand the behaviour of `gTree`s. In the first two lines of Figure 4.18a we are creating a rectangle graphics object and a text graphics object. These graphics objects, `rg` and `tg`, have not been drawn yet and consequently are not on grid's display list. When creating the `gTree` on line 3, we use the `children` parameter to include the graphics objects in the `gTree`. The `gList()` function that is used on the `children` parameter simply groups the graphics objects together. Because the `gTree` graphics object we created has not been drawn yet, we draw it using `grid.draw()`. This gives the appearance of drawing both the rectangle and the text using one graphics object. Given that the `gTree` has been drawn, we can inspect it in grid's display list. The display list shows that our `gTree` has in fact been drawn, however, the names of the rectangle and text graphics objects are also present. The extra indentation applied to `GRID.rect.1` and `GRID.text.2` shows that they are children of our `gTree` named `example-gTree`.

When using a `gTree`, we can determine the viewport it is drawn in using the `vp` parameter. This parameter, while supported on regular graphics objects, was not supported correctly on `gTree` objects. Without this support, plots created using the ggplot2 library would fail to draw at all. This is because ggplot2 plots are created by drawing a `gTree` that contains everything necessary to draw the entire plot. The detailed solution to the `vp` problem will not be discussed here.

There are common grid graphics objects that are also `gTree`s. These graphics objects are `frame`s and `cellGrob`s. The lattice package often uses `frame`s and `cellGrob`s to create legends. In order to support these two graphics objects, only slight modifications needed to be made given that generic `gTree`s have been implemented. This was to support the parameters `framevp` and `cellvp` that are used instead of `vp`. The effect of this change is shown in Figure 4.19.



(a) A lattice demo plot before supporting `frame` and `cellGrob` graphics objects.



(b) A lattice demo plot with a correct legend.

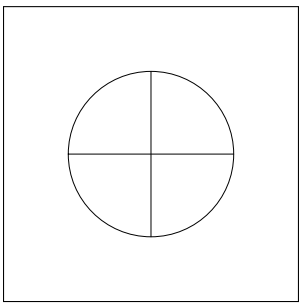
Figure 4.19: Demonstrating the effect of `frame` and `cellGrob` support.

We can see in Figure 4.19b that several graphics objects are being drawn. There is in fact one `frame` that contains several `cellGrobs`, each containing the graphics objects that are components of the legend.

4.1.9 Points

A common feature of plots is that they often use points to represent observations. gridSVG was lacking in its support of points for a few reasons. Firstly, it only supported two plotting characters, whereas grid is able to support over 100. Secondly, it was assumed that when drawing a set of points, all points would have the same plotting character. Finally, gridSVG was deficient in its support of points in that the size of its points were often incorrect.

To fix the first problem, the task of observing plotting characters present in R and translating them to grid graphical objects was undertaken. The reason why this translation is necessary is because SVG does not have an equivalent of a plotting character. This means that to draw a plotting symbol like a dot, we must use a grid circle, which can map to an appropriate SVG element. For some plotting characters, it was necessary to produce multiple graphics objects. In this case we apply the same grouping rules as in Section 4.1.1. A demonstration of this is shown in Figure 4.20.



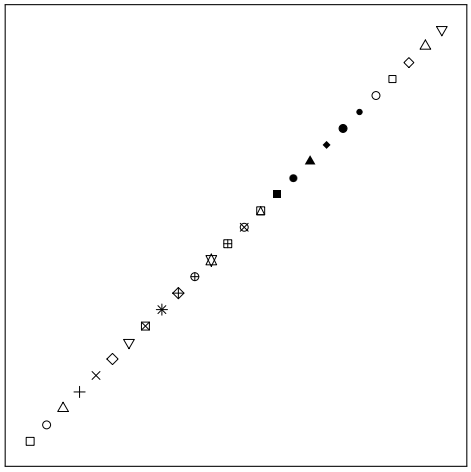
(a) Plotting character #10.

```
<g id="GRID.points.4">
  <g id="GRID.points.4.1">
    <polyline id="GRID.points.4.1.1" ...>
    <polyline id="GRID.points.4.1.2" ... />
    <circle id="GRID.points.4.1.3" ... />
  </g>
</g>
```

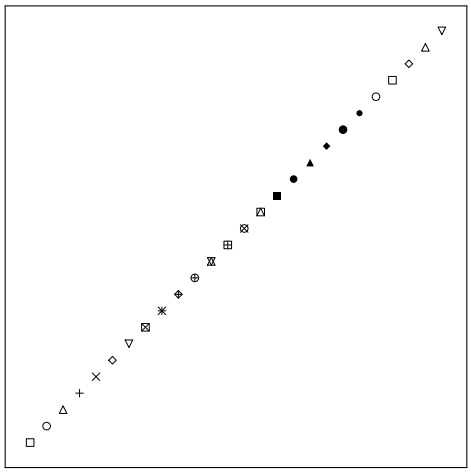
(b) SVG code used to produce the plotting character in Figure 4.20a.

Figure 4.20: Demonstrating the use of multiple elements when translating a plotting character.

We can see the grouping rules from Section 4.1.1 being applied through the use of the <g> element along with the naming scheme. The name of the graphics object must therefore be `GRID.points.4` and the point that we see is the first (and only) point produced from this graphics object. We then see three SVG elements in use, two of these are lines and one is a circle. The reason is because the plotting character is implemented as a grid circle along with a vertical grid line and a horizontal grid line. By implementing this method for several of the plotting characters, gridSVG gains the ability to plot all characters to a reasonable degree of accuracy. A comparison of grid's implementation of plotting characters versus gridSVG's is shown in Figure 4.21.



(a) Plotting characters $0 - 25$ as shown in grid.



(b) gridSVG drawing plotting characters $0 - 25$.

Figure 4.21: Comparing the implementations of plotting characters between grid and gridSVG.

Although not all of the plotting characters shown in Figure 4.21 are identical, we can

now draw any plotting character and have an implementation of it in SVG.

Now that we can draw many plotting characters, it would be useful to be able to use more than one when plotting a set of points. The correction to gridSVG required vectorising the code so that it checked which plotting character to use for each point. Previously it would only check for the plotting character that would be used on the first point and applied it to every point.

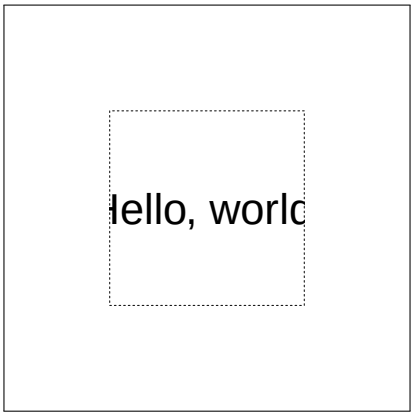
The final fix to points is the point sizes. Often point sizes are defined using grid's `char` unit, as is the case with lattice plots. This is reasonable because after all, it's a plotting *char*acter that is being drawn. However, when the `char` unit is used, we cannot use the information present using a graphics object's graphical parameters to determine its actual size. This issue arose when we tackled the problem of multi-line text and the solution is much the same. Again, to find the height of a character we create a text object with the text "M". The point object's graphical parameters are applied to the text object for correct sizing. We then use grid's unit conversion functions to turn the height of the "M" into a more usable unit. This process allows us to produce more accurately sized points.

4.2 Viewport clipping

While viewports in gridSVG were supported, they lacked the ability to apply the `clip` parameter. This parameter allows for clipping to occur on viewports. If a viewport enables clipping, then anything that attempts to draw beyond the boundaries of a viewport will not be shown. An example of clipping is shown in Figure 4.22.

```
> # Creating a new viewport in the middle of the plot
> pushViewport(viewport(width = 0.5,
+                       height = 0.5,
+                       clip = "on"))
> # Showing the size of the viewport
> grid.rect(gp = gpar(lty = "dashed"))
> # Drawing large text that exceeds the size of the viewport
> grid.text("Hello, world!", gp = gpar(fontsize = 56))
> # Leaving the viewport
> popViewport()
```

(a) Code used to produce a viewport with clipping enabled.



(b) An example of viewport clipping in grid, using code from Figure 4.22a.

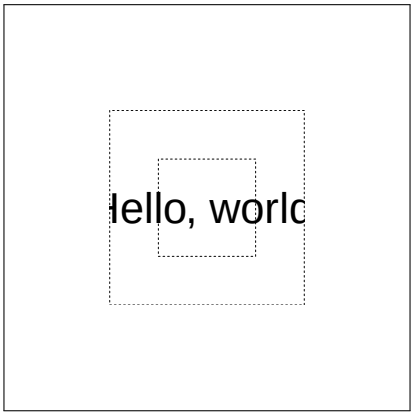
Figure 4.22: Demonstrating viewport clipping in grid.

We can see that because the text "Hello, world!" is drawn large enough to exceed the dimensions of the viewport, it is partially obscured. The application of clipping on viewports allows drawing to be restricted to a defined region.

There are three options that a grid viewport can take on its `clip` parameter. These options are "on", "inherit" and "off". If `clip` is set to "on", then no output will be drawn outside the dimensions of the viewport. When `clip` is set to "off", all graphical output will be drawn regardless of whether it appears within the regions of the viewport. Lastly, the `clip` option of "inherit" means that a viewport will clip to the same region as any viewports it exists within. For example if a viewport has `clip` set to "on", then any viewport created within that viewport which uses "inherit" will clip to the same region as the parent viewport. We can demonstrate this by modifying Figure 4.22 to produce Figure 4.23.

```
> # Creating a new viewport in the middle of the plot
> pushViewport(viewport(width = 0.5, height = 0.5, clip = "on"))
> # Showing the size of the viewport
> grid.rect(gp = gpar(lty = "dashed"))
> # Creating aviewport in the middle of the current viewport
> pushViewport(viewport(width = 0.5, height = 0.5,
+                       clip = "inherit"))
> # Showing the size of the viewport
> grid.rect(gp = gpar(lty = "dashed"))
> # Drawing text larger than the current viewport
> grid.text("Hello, world!", gp = gpar(fontsize = 56))
> # Leaving both viewports
> popViewport()
> popViewport()
```

(a) Code used to demonstrate a viewport that inherits a clipping region.



(b) An example of viewport clipping using "inherit" in grid, using code from Figure 4.23a.

Figure 4.23: Demonstrating viewport clipping using "inherit" in grid.

We can see in Figure 4.23b that the text does not get clipped to the viewport it is in. However, the viewport the text was drawn within inherited the clipping region of its viewport. This is why the text is drawn outside its viewport dimensions but is still partially clipped.

When gridSVG begins its process of creating SVG images, it parses what is present in the current plot and attempts to recreate it. In doing so, when gridSVG moves into a viewport, its `clip` parameter is inspected. If the parameter is set to "on", then we need to implement the clipping region in SVG. This does not require much extra work because gridSVG has already created a <g> element to group all of the graphics objects together that are drawn within it. The reason this is useful is because SVG allows us to define a clipping region on a <g> element, which affects all of the elements within the <g> element.

A clipping region implemented in gridSVG produces output similar to Listing 4.8.

```
<defs>
  <clipPath id="VP.clipPath">
    <rect x="142" y="142" width="283" height="283" />
  </clipPath>
</defs>
<g id="VP" clip-path="url(#VP.clipPath)">
   ...
</g>
```

Listing 4.8: SVG code that clips a group of elements to the area of a rectangle.

What is being produced by gridSVG looks similar to the code used to apply arrows to lines. We can see the use of the `<defs>` element, which allows the use of its contents to be referenced. Following this, we observe the `<clipPath>` element. This is the element that we reference when we want to clip a `<g>` element. The clipping path is defined as a rectangle that is, in this case, positioned at (142, 142) and has a height and width of 283. This means that anything outside this region will be clipped and therefore not drawn. We apply this clipping path to the `<g>` element in the same way we applied markers to line elements, using the `url()` function. Again we are required to implement a naming scheme so that we can reference the clipping path from the `<g>` element. The naming scheme used is to take the viewport's name and suffix it with `clipPath`.

However, a problem is encountered due to the fact that when we create a group for a viewport, we would be giving it a non-unique identifier. This is because a viewport may be used multiple times by gridSVG when drawing a plot. When a viewport was applied more than once, the result is multiple `<g>` elements with the same name. In doing so, when we create multiple clipping paths they all receive the same name. When an SVG renderer attempts to work out which `<g>` element to clip, it will make a choice, but this behaviour is undefined. This means that gridSVG would only be able to clip to a single usage of a given viewport.

The solution to this problem requires creating another naming scheme to ensure a unique identifier is applied to a `<g>` element. gridSVG does this by keeping track of how many times a viewport has been used. The name that is applied to a `<g>` element now becomes the name of the viewport suffixed by the number of times the viewport has been used. For example, the first appearance of a viewport called `VP` will produce the name `VP.1`. A demonstration of this behaviour is shown in Figure 4.24.

```
> # Storing a viewport named "example"
> vp <- viewport(width = 0.5, height = 0.5, name = "example")
> # First use of the "example" viewport
> pushViewport(vp)
> grid.rect()
> popViewport()
> # Second use of the "example" viewport
> pushViewport(vp)
> grid.circle()
> popViewport()
```

(a) Code used to demonstrate a viewport that is used more than once.

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

(b) A subset of gridSVG's output after processing Figure 4.24a.

Figure 4.24: Demonstrating gridSVG's viewport naming scheme.

Using Figure 4.24 we can observe that the first use of the viewport named "example" produced an SVG `<g>` element with the name of `example.1`. The second use of the viewport produced a `<g>` element with the name `example.2`. This confirms that gridSVG is now creating unique identifiers for viewports. We can now clip to `<g>` elements safely, using SVG code similar to what is shown in Listing 4.8.

Although gridSVG can now clip to elements, the only cases that have been implemented are when a viewport's `clip` parameter is `on` or `off`. This is because we can only get to inspect one viewport at a time. The only clipping region information we have available to us when inspecting a viewport is whether or not graphical output can exceed the viewport's dimensions. In order to support `inherit` we need to know the clipping region of a parent viewport, which gridSVG cannot do at the moment. Further work in this area may be explored in future.

4.3 Animation

While most of the work involved in the development of gridSVG involved creating accurate graphical output, a feature of gridSVG that was also improved was its ability to animate graphical objects. When animating a graphical object, gridSVG is able to write an

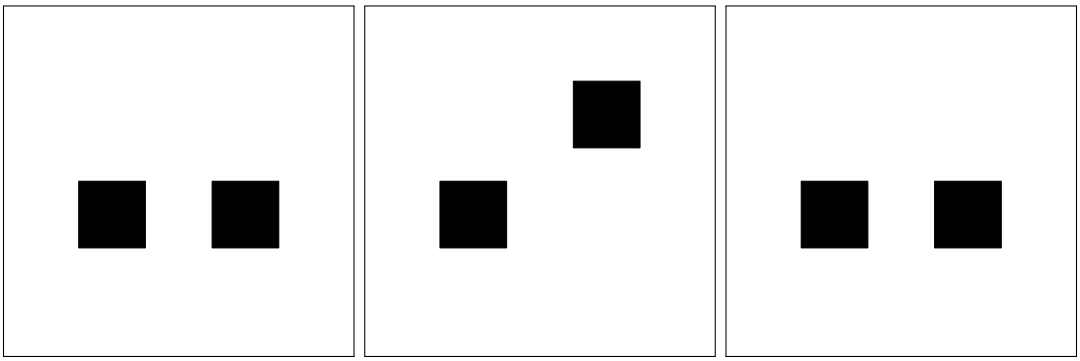
`<animate>` element that describes the behaviour of the animation. The `<animate>` element requires three key pieces of information: the name of an element, the attribute to be animated and values that the attribute is going to animate through. For example, in order to move a rectangle right from $(2, 0)$ to $(5, 0)$ the animate element needs to know the name of the circle, that the x attribute is being changed and that it is being changed from 2 to 5. The function that gridSVG provides that handles this task is `grid.animate()`.

Some of the changes that were made to gridSVG when improving its ability to draw plots caused `grid.animate()` to no longer function correctly. The most important of these changes are that there is no longer the assumption that one graphics object produces one SVG element. The naming scheme applied to SVG elements means that when we want to animate a graphics object like a rectangle, we need to know exactly how many visible rectangles the graphics object produces. For example, assume two rectangles have been produced from one graphics object named `GRID.rect.1`. When attempting to animate this graphics object using gridSVG, the problem we encounter is that although the names given to SVG elements are known, grid is not aware of them. This means that when animating the first rectangle in a rectangle object named `GRID.rect.1`, we cannot simply attempt to animate `GRID.rect.1.1` as it does not exist.

The solution that was applied to gridSVG is that a matrix of values can now be used to animate a graphics object, rather than a vector. The columns of each matrix refer to each of the graphical elements that a graphics object produces. Each of the rows is the value to be shown at a given time. Because a matrix is being used, R's behaviour regarding matrices requires us to provide values for every column of the matrix. Consequently, whenever animation occurs on a graphics object we need to animate every element produced by that object. An example of animation using a matrix is shown in Figure 4.25.

```
> # Creating two rectangles using one graphics object
> grid.rect(x = c(0.3, 0.7), y = 0.4,
+           width = 0.2, height = 0.2,
+           gp = gpar(fill = "black"))
> # Finding out the name of the object
> grid.ls()
GRID.rect.1
> # A matrix of y values to animate through
> ymat <- matrix(c(rep(0.4, 3), 0.4, 0.7, 0.4), ncol = 2)
> ymat
     [,1] [,2]
[1,]  0.4  0.4
[2,]  0.4  0.7
[3,]  0.4  0.4
> # Applying the animation
> grid.animate("GRID.rect.1", y = ymat)
```

(a) Code used to animate a single rectangle vertically.



(b) A rectangle moving vertically from $y = 0.4$ to $y = 0.7$ and back to $y = 0.4$.

```
<animate xlink:href="#GRID.rect.1.1"
         attributeName="y" values="170;170;170" ... />
<animate xlink:href="#GRID.rect.1.2"
         attributeName="y" values="170;340;170" ... />
<g id="GRID.rect.1"  >
  <rect id="GRID.rect.1.1" ... />
  <rect id="GRID.rect.1.2" ... />
</g>
```

(c) SVG code produced by gridSVG.

Figure 4.25: Demonstrating the animation of a graphics object that produces more than one SVG element.

The rectangle graphics object that was created in Figure 4.25 produces two visual rectangles. A matrix was then created for the purpose of moving the second rectangle from $y = 0.4$ to $y = 0.7$ and back to $y = 0.4$. In order to keep the first rectangle stationary we repeated 0.4 three times, producing the first column of the matrix. The second column of the matrix is simply the y values we wish to animate the second rectangle through. We then call `grid.animate()` in order to declare that animation is to occur on a graphics object. The first parameter given is the name of the graphics object that is to be animated, any parameters following this are properties of the graphics object to be animated. In this case we are animating `GRID.rect.1` by its y attribute using values in our matrix. When gridSVG writes this to SVG we can see that the `<animate>` elements refer to each of our rectangle elements. The `attributeName` states that we are animating by y, and the `values` indicate the values we are animating through, separated by semicolons.

The use of matrices to animate properties of a graphics object is a usable solution to our initial problem. However, there are some consequences. Our application of matrices assumes when animating positional or sizing attributes that they fulfil three requirements. The first of these requirements is that all values of the attribute have the same grid unit. This means we can't correctly animate through a set of x values where some of the xs are being specified in inches while others are in centimetres. Another requirement is that the units themselves cannot be complex units, i.e. units composed of more than one type of unit. An example of a complex unit would be where a height is specified as being 3 inches plus 2 centimetres. The third restriction is that the animation values use the same unit as the attribute they are animating. If a rectangle is one inch wide, anything that animates the rectangle's width must be measured in inches.

The reason for these restrictions is the fact that matrices can be composed of either numeric, boolean or character values. There is therefore no way of using grid's unit type to specify values to animate through. These restrictions do not apply to parameters of graphics objects that have character or boolean values.

When improving `grid.animate()` another issue that had to be mitigated is that points can be composed of several types of graphics objects. Consequently, a parameter of a grid graphics object can map to many different SVG attributes. An example of this is a plotting character composed of a circle and two lines (see Figure 4.20a). When animating its x values, we need to move the circle along its `cx` attribute in SVG, while the line must be animated by its `points` attribute. This means when extending animation support for points graphics objects, we need to know the structure of the plotting character that is being animated. The current solution is simply to implement support for each plotting character individually. This means any change in how gridSVG draws a plotting character may break animation support for that plotting character.

The method of animating grid graphics objects has been revised to account for recent developments in gridSVG. In addition, several graphics objects can now be animated along many of their properties.

5 Demonstrations

In order to show the progress that has been made and the possibilities of gridSVG, some demonstrations of animated and interactive graphics have been made. All figures showing demonstrations are screenshots taken while viewing the images in a modern web browser.

The first example, Figure 5.1, is a lattice plot that demonstrates interactivity. When the mouse cursor hovers over a graphical element, it shows a tooltip that bears the name of the grid graphics object that produced the element.

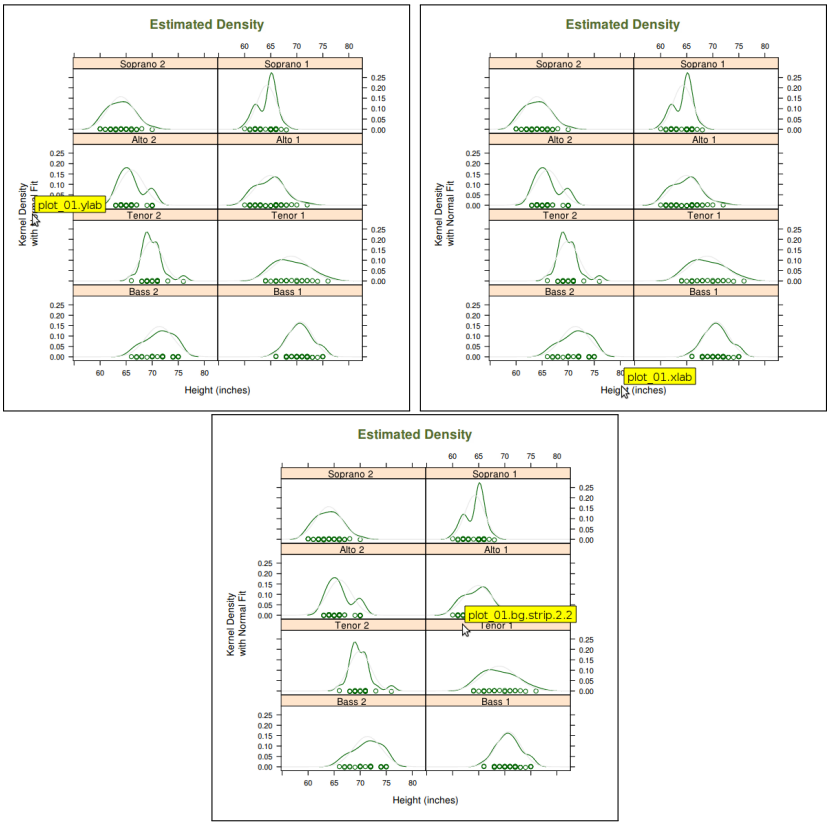


Figure 5.1: A plot showing a tooltip of each graphics object's name.

The example shown in Figure 5.2 is another demonstration of interactivity within a lattice plot. In this case, it is used to show additional information about an observation in the plot. When hovering over a point on the plot, the point itself doubles its radius while the text at the bottom of the plot shows the name relating to the observation.

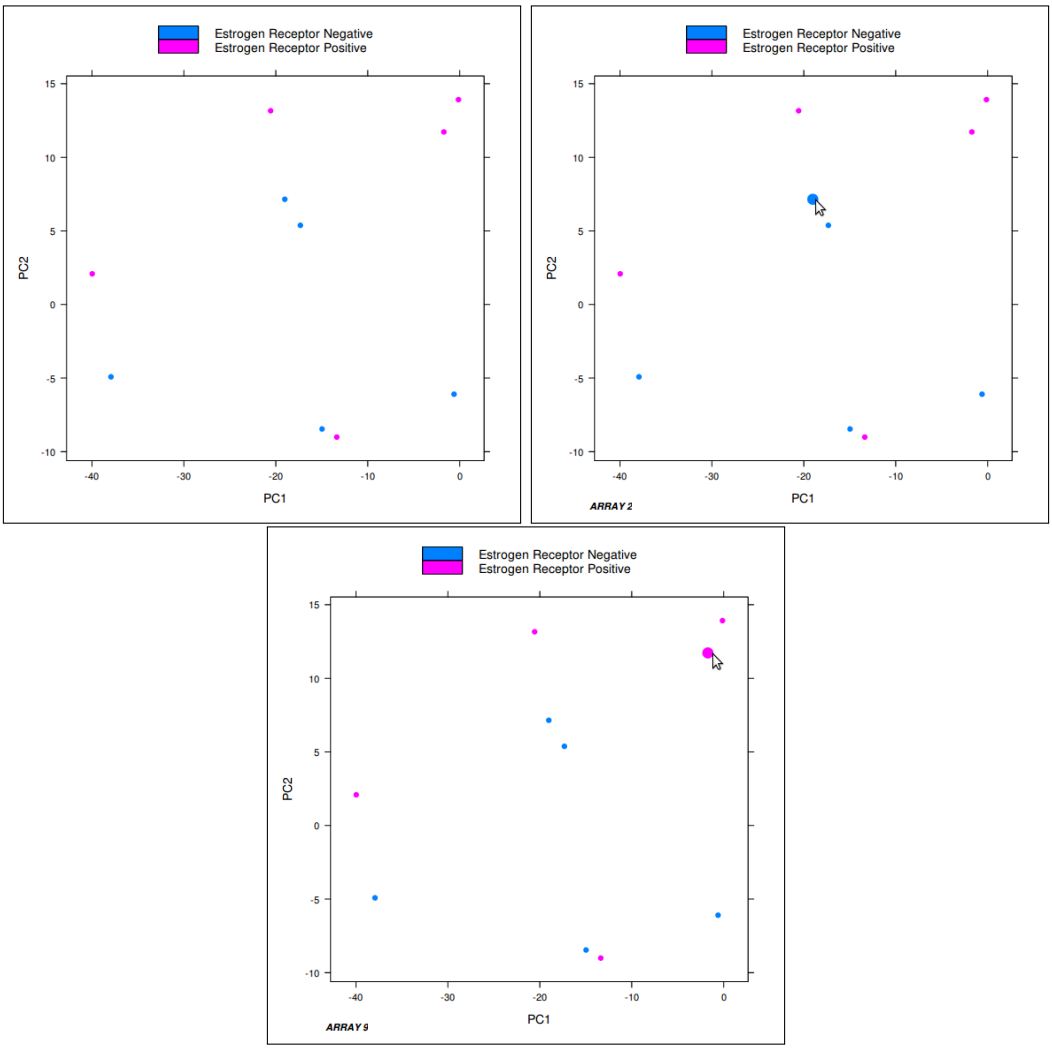


Figure 5.2: Showing the effect of hovering over a point in an interactive plot.

While interactivity has been shown, animation is a key feature of the types of plots gridSVG can create. In order to illustrate this, an example, Figure 5.3, was created that shows a sample being gathered from a population of data. A boxplot is then drawn from this sample to summarise the data.

The animation shows points from the population "falling" into the sample until all of the sample points have moved down. The boxplot is then drawn on the sample, and moves down to its final position.

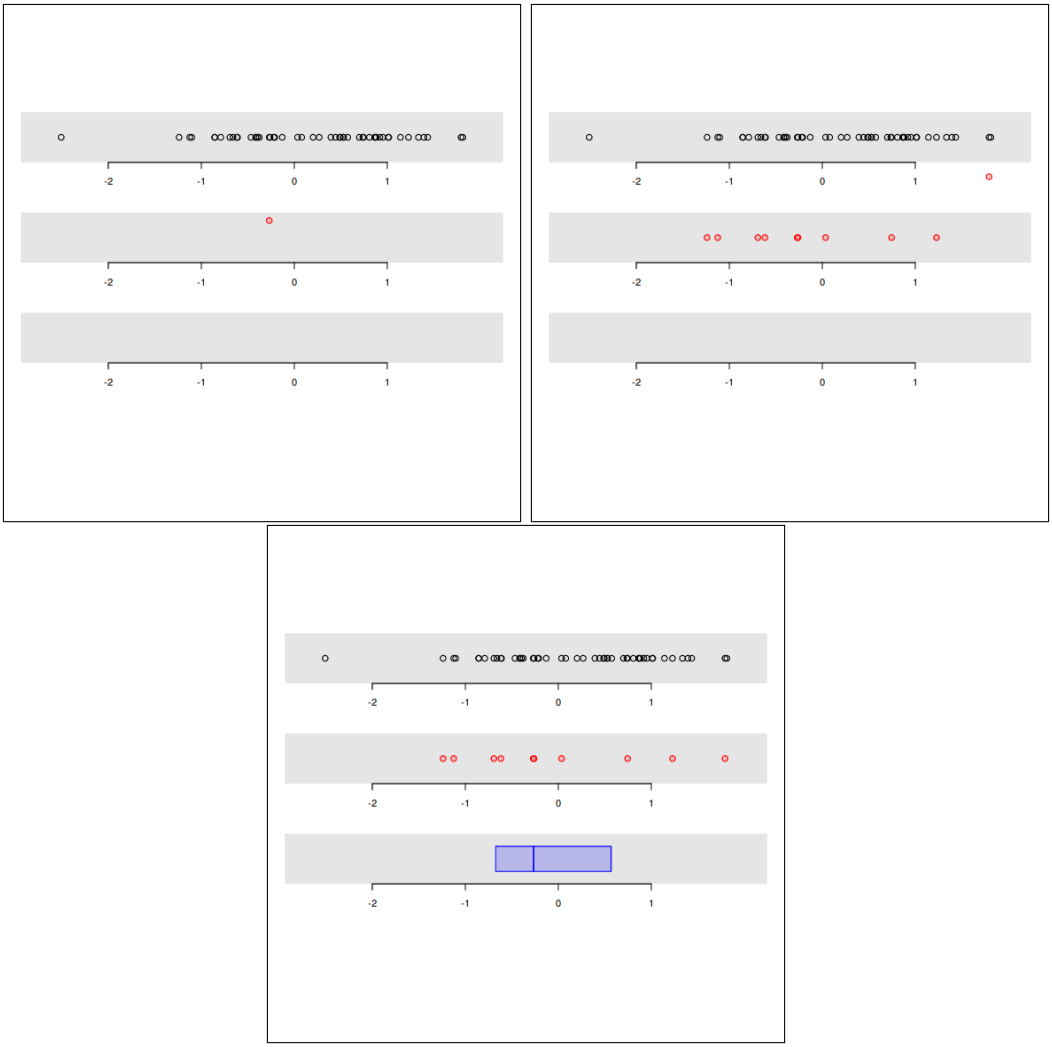
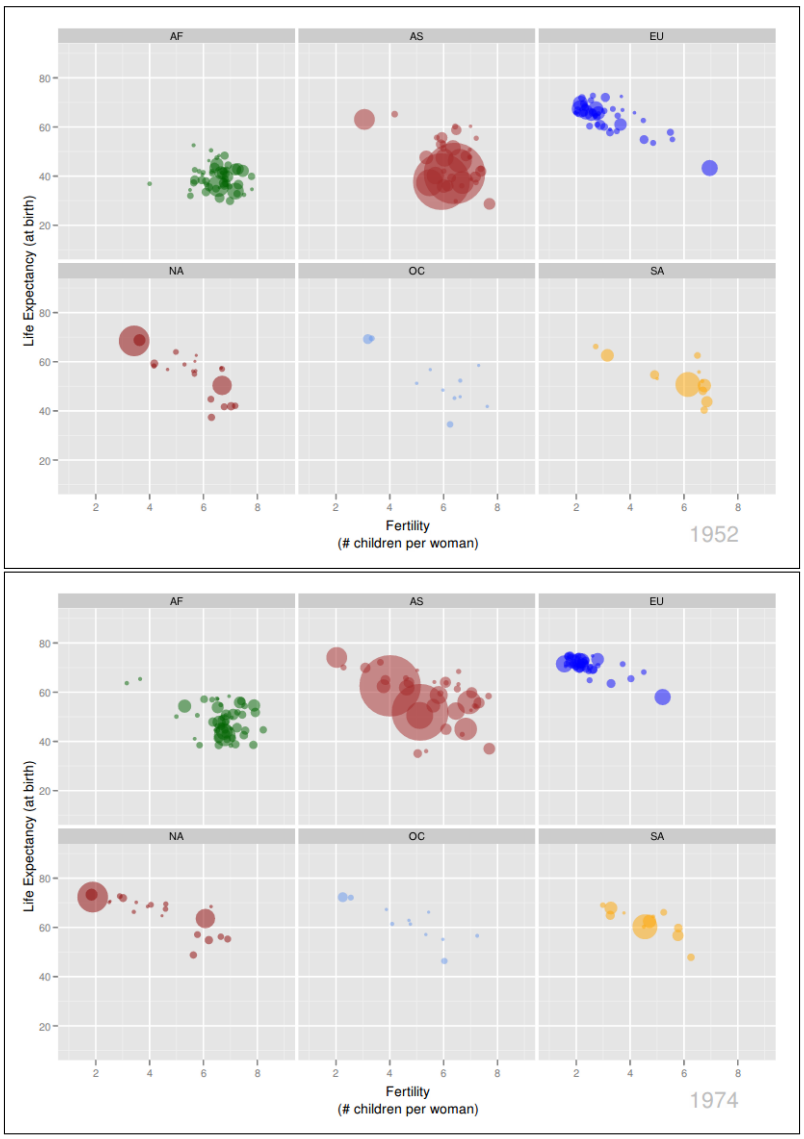


Figure 5.3: An animated example of points being samples from a population, then summarised using a boxplot.

Time series information is an obvious area in which animation can be applied. An animated plot, Figure 5.4, has been created that plots the log of stock prices for well known technology companies over a year. The lines that represent the stock prices of the companies draw over time until the end of the year of data.

Figure 5.4: An animated example of time series data. The lines appear to draw themselves over time.

The final demonstration is an implementation of the type of plot made famous by Hans Rosling and the Gapminder software package (Gapminder Foundation, 2011). In Figure 5.5 we have several variables to consider. We are plotting life expectancy against the number of children per woman for each country. This is animated over time so each "bubble" moves over time. The size of each "bubble" is determined by the population of the country it represents. There are six plotting regions, one for each continent. We observe that over time the "bubbles" move towards the top-left, indicating a trend towards an improvement in life expectancy and a reduction in the number of children per woman.



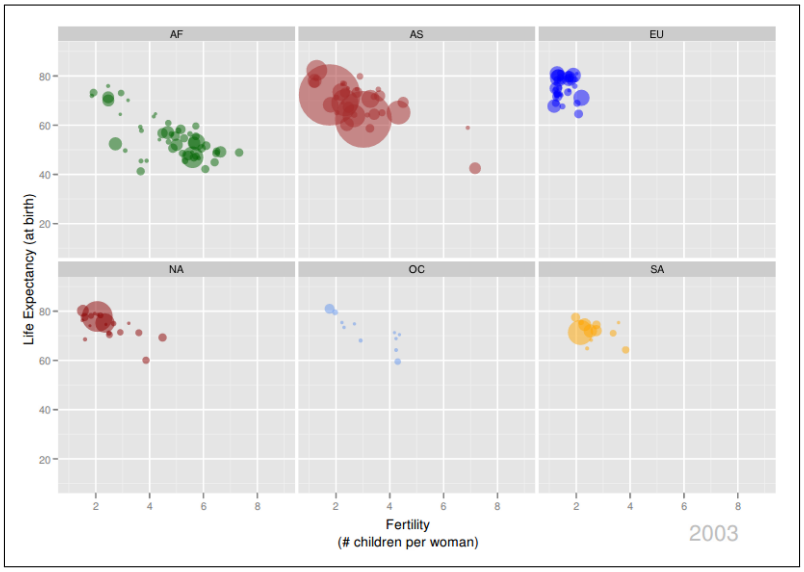


Figure 5.5: An example of a Gapminder-like "bubble" plot.

6 Discussion

Comparison to SVGAnnotation

The development of gridSVG has led not only to visible improvements in the images it produces, but also to the ability to apply it to real-world grid graphics. It has become a viable alternative in the area of web-based interactive graphics as it can capably produce animated and interactive R graphics. The only other solution that offered anything similar to what has been accomplished is SVGAnnotation. The key issue with SVGAnnotation, that it lacks transparency, can now be compared against gridSVG.

Transparency can be demonstrated simply by comparing the output of R's `svg()` device when against gridSVG's output. Because SVGAnnotation uses images that the `svg()` device produces, we can see what information must be used to create animated and interactive plots. When drawing the text "Hello, world!" in grid, we compare the different SVG images using Figure 6.1.

```
> grid.text("Hello, world!")
> grid.ls()
GRID.text.1
```

(a) A simple example using grid graphics.

```
<defs>
  <g>
    <symbol id="glyph0-0">
      <path d="..." />
    </symbol>
    ...
    <symbol id="glyph0-9">
      <path d="..." />
    </symbol>
  </g>
</defs>
<g>
  <use xlink:href="#glyph0-0" x="218" y="256" />
  ...
  <use xlink:href="#glyph0-9" x="281" y="256" />
</g>
```

(b) A subset of the SVG code produced by the `svg()` device to create the image described by Figure 6.1a.

```
<g id="GRID.text.1">
  <g id="GRID.text.1.1">
    ...
    <text>
      <tspan>Hello, world!</tspan>
    </text>
    ...
  </g>
</g>
```

(c) A subset of gridSVG's output from Figure 6.1a.

Figure 6.1: Comparing the output produced by the `svg()` device and gridSVG.

Observing the different SVG code produced, we can see that SVGAnnotation has little information to work with. There is no way of knowing directly from the output that the text object we used was in fact `GRID.text.1`. We can clearly see in gridSVG's SVG code what `GRID.text.1` has been translated to. This means that anyone who wishes to write JavaScript that interacts with the SVG image will know in advance exactly what they need to target — the name of the grid graphics object.

The output produced by the `svg()` device will be more accurate to the image shown when viewing a plot in R. However, there are several advantages in the approach that gridSVG takes. Firstly, it maps graphics objects to appropriate SVG elements, while the `svg()` device always maps to a `<path />` element. By using appropriate SVG elements we gain features that SVGAnnotation simply cannot provide, primarily regarding text. Among the additional features are text selection, text search and the ability to use fonts unknown to R.

Another issue with the `svg()` device always using the `<path />` element is when we attempt to manipulate it either through animation or JavaScript. An example of a graphics object where gridSVG makes this task easier is with circles. If we wish to manipulate the radius of a circle, we can use the fact that SVG's `<circle />` element has a radius attribute (`r`) to perform the manipulation. This makes animation straightforward for `grid.animate()` and it makes writing JavaScript easy as we just have to change the value of the attribute. To perform the same manipulation on a `<path />` element, it is necessary to rewrite path data which is a non-trivial task.

The approach that gridSVG takes appears to be more beneficial than the potential cost of having less accurate images. However, SVGAnnotation is the only option when using R's base graphics engine as gridSVG is restricted solely to grid graphics. This is a downside to gridSVG. However, the fact that popular plotting libraries lattice and ggplot2 use grid means that many plots will still be able to benefit from gridSVG.

Processing Time

An issue when creating SVG images using gridSVG is the large amount of time taken to produce an image. This is largely due to the amount of graphics object manipulation that gridSVG performs. There is no easy solution to this problem. A consequence of the amount of processing required is that dynamically generated images created and delivered by a web server are currently infeasible.

Grouping Issues

When developing gridSVG one of the key design decisions was the choice of grouping elements relating to a graphics object. This meant that in SVG, the name of a graphics object doesn't refer to a graphical SVG element, merely a set of graphical elements. A consequence of this decision is that a user cannot write JavaScript that targets the name of a graphics object. Instead the children of the element with the name of the graphics object must be used instead. Because of the naming scheme that is applied, it must be known in advance exactly which child element is the target. Despite this, the naming scheme provides a transparent and reliable means of accomplishing this task.

A task that was made easier by the introduction of grouping graphics objects is when applying JavaScript event attributes to a graphics object. When an attribute like

`onmouseover` is added to a `<g>` element, any children of the `<g>` element implicitly have the same event attribute applied along with its associated value. This process is known in JavaScript as event capturing. An example where this might be useful is if you have a set of points that you want to highlight when you hover a mouse cursor over them. If the points object is given the name `GRID.points.1`, then all we need to do is garnish the `GRID.points.1` object to include the appropriate event handling code. The JavaScript that is written can then simply change the colour of the element that triggered the event to perform highlighting.

It is not yet known whether this decision to perform grouping is the best solution. It provides a means of mitigating many of the problems we have encountered and appears to have useful and reliable properties for our uses.

X-splines to Lines and Paths

A design decision that may be re-evaluated in future is the decision for x-splines to become lines or paths, depending on whether the x-spline is open or closed. We used a line graphics object for open splines because they cannot be filled and because grid path objects are always closed. It may prove beneficial to always use paths for consistency. This would mean that the only difference in SVG output between open and closed splines are attributes of the resulting `<path />` element. In this case an open spline would be an open path with no fill, while a closed spline would be a closed path with a fill. This is an alternative to the current implementation.

Animation

Although a large amount of progress has been made on gridSVG, there is still room for improvement. Some of this is due to the incompleteness of the current implementation, especially with regards to animation. There are still some of graphics objects that do not support animation, along with their associated parameters. The possible improvements that will be considered relate to design decisions.

Points as Paths

gridSVG currently implements each plotting character for points objects as a grouped set of grid graphics objects. Rather than using several graphics objects to implement a plotting character, we could instead use a single grid path. The benefits of this choice would be that we would be simplifying the SVG output. It would also make interactivity via JavaScript a simpler task.

The downsides of this option are that `<path />` elements are difficult to modify. This is because there are no parameters which dictate size and position, only path data is present.

Therefore, in order to animate the size and position of a point, path data must be parsed and modified. Alternatively, a `<path />` could be scaled and translated appropriately. This is a viable solution, but it does require some care in ensuring that the line width of a path does not change after scaling.

Viewports

Viewports are another area in which improvements could be made. Currently there is no support for clipping on viewports when the `clip` parameter is set to `inherit`. This is due to a viewport not containing information about the region it is clipping to, only whether it defines its own clipping region. The grid graphics system does internally know this information. A possible solution may be to expose this information to allow gridSVG to inspect it. This would make it possible for clipping to occur on all parameters.

Viewports are also written out every time they are visited. While gridSVG works fine using this approach, an alternative may be to store graphics objects for each viewport in a queue. This would mean that each time a graphics object is drawn within a viewport, the graphics object is stored on a viewport-specific queue. The reason why this might be a better approach is that we would only end up writing out the viewport and its contents once. This would also mean that a clipping path would only have to be written once. By implementing this possible solution, the SVG code would be made more concise, but no visual improvement would occur.

Using the XML package

We could also introduce the use of the XML package (Lang, 2011). This would improve gridSVG's ability to write out SVG. Currently the way in which SVG code is produced is by writing out a string of text for each of the graphics objects and viewports. Rather than using hard-coded strings of text which are prone to errors, we could use the XML package to do this for us. This would make the process of writing to SVG a lot more reliable within gridSVG.

Improvements to `grid.garnish()` and `grid.hyperlink()`

The previous suggestions for future improvements have all been regarding cosmetic improvement to SVG code and not the usage of gridSVG. The functions `grid.garnish()` and `grid.hyperlink()` could be modified to handle sub graphics objects in the same way that `grid.animate()` does. Currently they apply their garnishing and hyperlinking to the `<g>` element that groups together the graphical elements that are produced from a graphics object. There is no way to use these functions to affect any of the elements which are children of the `<g>` element. In other words, you either garnish and hyperlink all elements (via the `<g>` element) or not at all. By using matrices or some other R data

structure, we could use the functions in the same way that `grid.animate()` does. This would provide a consistent interface for modifying graphical objects along with being able to provide the functionality that is currently lacking.

The `grid.*()` functions that gridSVG provides either accept a matrix of input for its parameters or have been suggested to do so when modifying graphical objects. It is not yet known whether this is the best approach for handling input. A list as input may prove to be more appropriate, or perhaps a combination of lists and matrices, or something different altogether. Further investigation into the merits of these approaches could certainly lead to an improvement in the usability of these functions.

File sizes

An issue with the output of gridSVG is that file sizes increase with the complexity of the plot. It is easy to produce a plot that is several megabytes in size. Images of this size are unsuitable for distribution on the web due to the time taken to download such a large file.

Because SVG is text based and there is a lot of repeated text, SVG compresses well with tools like GNU zip. This is not immediately useful for distribution because web browsers cannot view compressed SVG images and simply download the image instead. However, all the popular web server applications currently in use can be configured to compress SVG on-the-fly. This means that if a web browser is able to accept compressed text, it will be sent compressed text, and decompressed automatically by the browser. Now we can still show and store large SVG files, but they are delivered as if they were compressed.

A demonstration of the benefits of compression uses the stock ticker demonstration in Figure 5.4. This image is 8.5MB in size, but it compresses down to just 86kB. This makes an image that would take several seconds to deliver on a standard internet connection be delivered almost immediately.

HTML

An interesting feature that was noticed during the development of gridSVG is the behaviour of JavaScript when SVG is used within an HTML document. JavaScript that originates from an HTML document can affect the appearance and behaviour of an SVG image. Conversely, JavaScript that is inserted into an SVG image can affect the state of a web page. An implication of this is that we can use the user interface controls that are present in HTML to affect an SVG image. This could be useful if you wish to selectively show some data in an SVG image, and could accomplish this using a combination of HTML checkboxes and JavaScript. For example, the stock ticker example in Figure 5.4 could show and hide each line using checkboxes.

Implications

The implications of this project are that we can now create our own animated and interactive grid graphics. These graphics can be extended more easily than previous methods and allow the possibility of new R packages being created to build upon gridSVG. These packages could automate the creation of plots like those demonstrated in Section 5. By using gridSVG we can also create statistical reports for distribution on the web that engage the reader better than with static graphics.

7 Conclusion

The gridSVG package for R has been extended to create animated and interactive graphics for the web. It is now capable of producing plots from the lattice and ggplot2 packages with the ability to animate and interact with them.

We have described the methods and decisions made during the development of gridSVG. Primarily, the process of mapping grid graphics objects and viewports to SVG code was discussed.

Applications of gridSVG have demonstrated features that the R graphics engine cannot produce. These features provide a more engaging method of presenting quantitative and qualitative information than static graphics.

References

Conway, S. (2010). *webvis: Create graphics for the web from R.* `http://code.google.com/p/rwebvis/`. R package version 0.0.1.

Gapminder Foundation (2011). Gapminder: Unveiling the beauty of statistics for a fact based world view. `http://www.gapminder.org/`.

Gesmann, M. and de Castillo, D. (2011). *googleVis: Using the Google Visualisation API with R.* `http://code.google.com/p/google-motion-charts-with-r/`. R package version 0.2.4.

Lang, D. T. (2010). *SVGAnnotation: Tools for post-processing SVG plots created in R.* `http://www.omegahat.org/SVGAnnotation/`. R package version 0.9-0.

Lang, D. T. (2011). *XML: Tools for parsing and generating XML within R and S-Plus.* `http://www.omegahat.org/RSXML/`. R package version 3.4-0.

Murrell, P. (2005). *R Graphics.* Chapman & Hall/CRC.

Murrell, P. (2011). *gridSVG: Export grid graphics as SVG.* `http://r-forge.r-project.org/projects/gridsvg/`. R package version 0.7-0.

R Development Core Team (2011). *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0.

Sarkar, D. (2008). *Lattice: Multivariate Data Visualization with R.* Springer, New York. ISBN 978-0-387-75968-5.

Verzani, J. (2011). *gWidgetsWWW: Toolkit implementation of the gWidgets API for use with web pages.* `http://gwidgets.r-forge.r-project.org/`. R package version 0.0-22.

W3C (2011). *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification.* `http://www.w3.org/TR/CSS2/`.

Wickham, H. (2009). *ggplot2: Elegant Graphics for Data Analysis.* Springer New York.

Xie, Y. (2011). *animation: A Gallery of Animations in Statistics and Utilities to Create Animations.* `http://animation.yihui.name/` R package version 2.0-4.