Animation with the gridSVG package

Paul Murrell

March 9, 2023

Introduction

The `grid.animate()` function in the gridSVG package allows various features of a **grid** grob to be animated. Calls to this function can be quite straightforward. For example, the following code animates a circle so that it travels from left to right across the screen (the result of this code is the file `"animCircle.svg"`, which can be viewed in a web browser).

```
> library(grid)
> library(gridSVG)
```

```
> grid.circle(.1, .5, r=.1, gp=gpar(fill="black"),
+             name="circle")
> grid.animate("circle", x=c(.1, .9))
> grid.export("animCircle.svg")
```

Things can get more complicated though. For example, in order to animate a `polyline` grob, it is necessary to specify a *vector* of `x` and/or `y` locations for each time point *and* it may even be necessary to specify *multiple* vectors at each time point if the `polyline` grob specifies more than one polyline (via its `id` argument).

To give a concrete example, consider the result of the following **grid** code, which draws two polylines from a single call to `grid.polyline()` (see Figure 1).

```
> grid.rect()
> grid.polyline(c(.2, .3, .4, .6, .7, .8),
+               c(.7, .5, .7, .3, .5, .3),
+               id=rep(1:2, each=3),
+               gp=gpar(lwd=5),
+               name="polyline")
```

The task is to animate the two polylines so that they appear to "flap" (the left will transition to look like the right one and the right one will transition to look like the left one, and repeat).

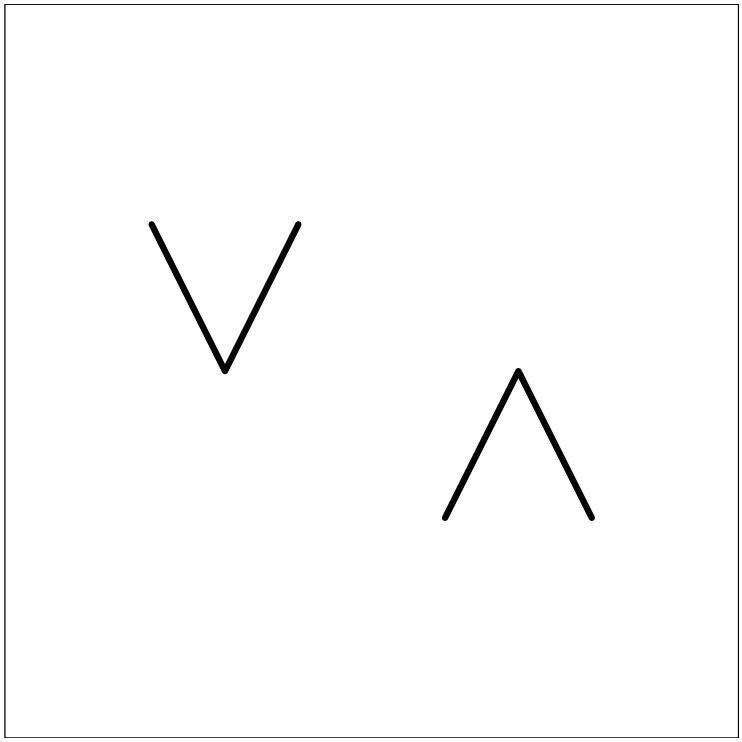

Figure 1: Two polylines drawn from a single call to `grid.polyline`.

The y-values for the animation look something like this (the x-values are not animated):

```
$id1
$id1$t1
[1] 0.7npc 0.5npc 0.7npc

$id1$t2
[1] 0.3npc 0.5npc 0.3npc

$id1$t3
[1] 0.7npc 0.5npc 0.7npc


$id2
$id2$t1
[1] 0.3npc 0.5npc 0.3npc

$id2$t2
[1] 0.7npc 0.5npc 0.7npc

$id2$t3
[1] 0.3npc 0.5npc 0.3npc
```

For the first polyline (`id1`), at the first time point (`t1`), the y-values are just the original y-values, `c(.7, .5, .7)`. At the second time point, the y-values for the first polyline are `c(.3, .5, .3)`, and at the third time point the y-values are back to the original `c(.7, .5, .7)`. The y-values for the second polyline are the reverse of the y-values for the first polyline.

Specifying this set of animation values is more complex and can get pretty confusing, but at the same time, we still want to be able to specify the simple animation values (like in the first example) in a simple manner. The gridSVG package provides a consistent, but flexible system for specifying animation values that is based on a single, coherent data structure, but which allows the user to use simpler specifications where possible.

The `animUnit` class

An `animUnit` object has three components: a vector of `values` (as a `unit` object), a `timeid` vector (default `NULL`), and an `id` vector (default `NULL`).

The `animUnit()` function is used to create an `animUnit`, with only the `values` as a required argument. The following code generates a single value at four different time points.

```
> animUnit(unit(1:4, "cm"))
```

```
$t1
[1] 1cm

$t2
[1] 2cm

$t3
[1] 3cm

$t4
[1] 4cm
```

As this example shows, the default interpretation of a `NULL` `timeid` is that each value belongs to a separate time period (and the default interpretation of a `NULL` `id` is that there is only one shape to be animated). This example also demonstrates the `print` method for `animUnit` objects, which is useful for seeing which animation values belong to different time periods.

This simple sort of `animUnit` is sufficient for specifying something like the x-location of a single data symbol (where there is exactly one x-value required per time point).

For the slightly more difficult situation of animating multiple data symbols (where we need several x-values per time period, one for each different data symbol), the `id` argument can be explicitly specified. The following code generate values for two shapes (`id1` and `id2`) with values at two time points (`t1` and `t2`) for each shape.

```
> animUnit(unit(1:4, "cm"), id=rep(1:2, 2))

$id1
$id1$t1
[1] 1cm

$id1$t2
[1] 3cm


$id2
$id2$t1
[1] 2cm

$id2$t2
[1] 4cm
```

In the case where we have a single shape, but that shape is described by multiple x-values (e.g., a single polygon), we need multiple x-values per time point *for each shape*, the `timeid` argument can be used to associate multiple x-values with a single time point. The following code generates six values at each of two time points (`t1` and `t2`) for a single shape.

```
> animUnit(unit(1:12, "cm"), timeid=rep(1:2, 6))

$t1
[1] 1cm  3cm  5cm  7cm  9cm  11cm

$t2
[1] 2cm  4cm  6cm  8cm  10cm 12cm
```

And in the worst case, we have multiple shapes, each requiring multiple x-values per time period (e.g., multiple polygons from a single `polygon` grob), so we need to specify *both* `id` and `timeid`. The following code generates three values at two different time points for two different shapes.

```
> animUnit(unit(1:12, "cm"),
+          timeid=rep(1:2, 6), id=rep(1:2, each=6))

$id1
$id1$t1
[1] 1cm 3cm 5cm

$id1$t2
[1] 2cm 4cm 6cm


$id2
$id2$t1
[1] 7cm  9cm  11cm

$id2$t2
[1] 8cm  10cm 12cm
```

The following code uses the `animUnit()` function to produce the flapping polylines example from the previous section. It produces a file called `"animPolyline.svg"` that can be viewed in a browser.

```
> grid.newpage()
> grid.rect()
> grid.polyline(c(.2, .3, .4, .6, .7, .8),
+               c(.7, .5, .7, .3, .5, .3),
+               id=rep(1:2, each=3),
+               gp=gpar(lwd=5),
+               name="polyline")
> polylineY <- animUnit(unit(c(.7, .5, .7, .3, .5, .3,
+                              .3, .5, .3, .7, .5, .7,
+                              .7, .5, .7, .3, .5, .3),
+                            unit="npc"),
+                       timeid=rep(1:3, each=6),
+                       id=rep(rep(1:2, each=3), 3))
```

```
> grid.animate("polyline", y=polylineY, rep=TRUE)
> grid.export("animPolyline.svg")
```

The `as.animUnit()` function

The `animUnit` class gives us the range of possible specifications that we require, but it is overkill for simple cases, and may be less convenient even for more complex cases.

There is an `as.animUnit()` function that can convert vectors, matrices, and lists to `animUnit`s so that we can use those simpler data structures to provide animation values. For example, a single value per time period can be specified with just a vector, as follows.

```
> as.animUnit(1:4, unit="cm")

$t1
[1] 1cm

$t2
[1] 2cm

$t3
[1] 3cm

$t4
[1] 4cm
```

Even better, the `grid.animate()` function makes use of this coercion function and fills in the `unit` based on the units of the feature that is being animated. This means that the call to `grid.animate()` can just specify a vector, like the following (taken from the first example at the beginning of this document).

```
> grid.animate("circle", x=c(.1, .9))
```

When we need to specify values for multiple shapes it can be convenient to use a matrix, where each column provides the values for a different shape. The following code shows an example (again, in a call to `grid.animate()` we can leave out the `unit`).

```
> m <- matrix(1:6, ncol=2)
> m

     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

```
> as.animUnit(m, unit="cm")

$id1
$id1$t1
[1] 1cm

$id1$t2
[1] 2cm

$id1$t3
[1] 3cm


$id2
$id2$t1
[1] 4cm

$id2$t2
[1] 5cm

$id2$t3
[1] 6cm
```

If we have multiple values per time point (e.g., a polygon), we can get `as.animUnit()` to treat columns as different time points rather than as different shapes by specifying the `multVal` argument, as shown below.

```
> as.animUnit(m, unit="cm", multVal=TRUE)

$t1
[1] 1cm 2cm 3cm

$t2
[1] 4cm 5cm 6cm
```

The `grid.animate()` function guesses how it should use the columns of a matrix, depending on the shape that is being animated, so for relatively straightforward cases we should be abe to simply pass a matrix to `grid.animate()`.

Finally, we can specify animation values as a list of units (if that is more convenient than calling `animUnit()`). The following code shows an example, which shows that the default behaviour is to treat each component of the list as animation values for a separate shape.

```
> l <- list(unit(1:3, "cm"), unit(4:6, "cm"))
> l

[[1]]
[1] 1cm 2cm 3cm
```

```
[[2]]
[1] 4cm 5cm 6cm
```

```
> as.animUnit(l)
```

```
$id1
$id1$t1
[1] 1cm

$id1$t2
[1] 2cm

$id1$t3
[1] 3cm


$id2
$id2$t1
[1] 4cm

$id2$t2
[1] 5cm

$id2$t3
[1] 6cm
```

Again, we can specify that the components of the list correspond to separate time points rather than separate shapes (by calling `as.animUnit()` directly and supplying the `multVal` argument).

```
> as.animUnit(l, multVal=TRUE)
```

```
$t1
[1] 1cm 2cm 3cm

$t2
[1] 4cm 5cm 6cm
```

The `animValue` class

Some features of a shape, such as `visibility`, are not numeric locations or dimensions, so they do not need to be specified as unit values. For these cases, there is an `animValue()` function to create the various specifications that we might need, plus an `as.animValue()` function, which `grid.animate()` makes use of to allow convenient variations.

Summary

For relatively simple animations, all we need to do is specify a numeric vector, or possibly a matrix, in the call to `grid.animate()`. For more complex animations, how we specify the animation values depends on what we find most convenient. Specifying a matrix or a list of units may suffice, although this may rely on `grid.animate()` correctly guessing our intention. It may be necessary to directly call `as.animUnit()` on a matrix or a list of units to get the behaviour that we want. Alternatively, a direct call to `animUnit()` should allow us to specify any set of animation values that we need and the `print()` method for the `animUnit` objects that are created by that function should help us to check that we are generating values in the right format.