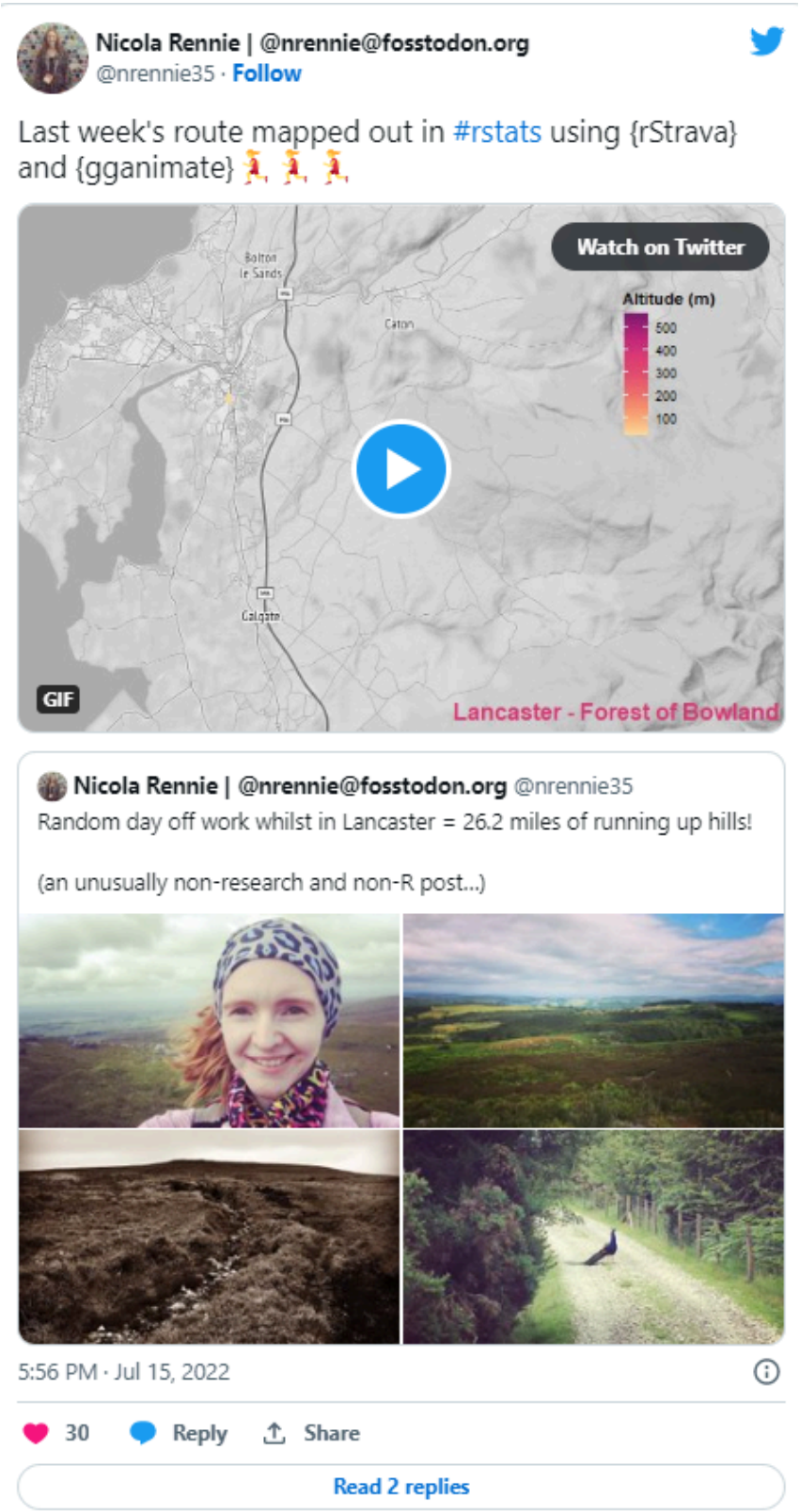
Mapping a marathon with {rStrava}

This tutorial blog will walk through the process of getting data from Strava using {rStrava}, making a map of it, and animating the map with {gganimate}.

July 18, 2022

After a long run in the Forest of Bowland when visiting Lancaster for a few days, I decided to try out the {rStrava} package to make some maps of where I'd been. This tutorial blog will walk through the process of getting the data from Strava, making the map, and animating it with {gganimate}.



What is {rStrava}?

Strava is a an app for tracking physical activities, mostly used for running and cycling. The {rStrava} package lets you access data through the Strava API.

Since {rStrava} is available on CRAN, you can install it in the usual way using:

Copy

```
1 install.packages("rStrava")
2 library(rStrava)
```

There are two levels of data you can get using {rStrava} depending on whether or not you have an authentication token. If you don't have a token, you can access some basic summary information on athletes with public profiles and their recent activities. You don't even need a Strava account to access this data. You can see which functions don't require a token by running:

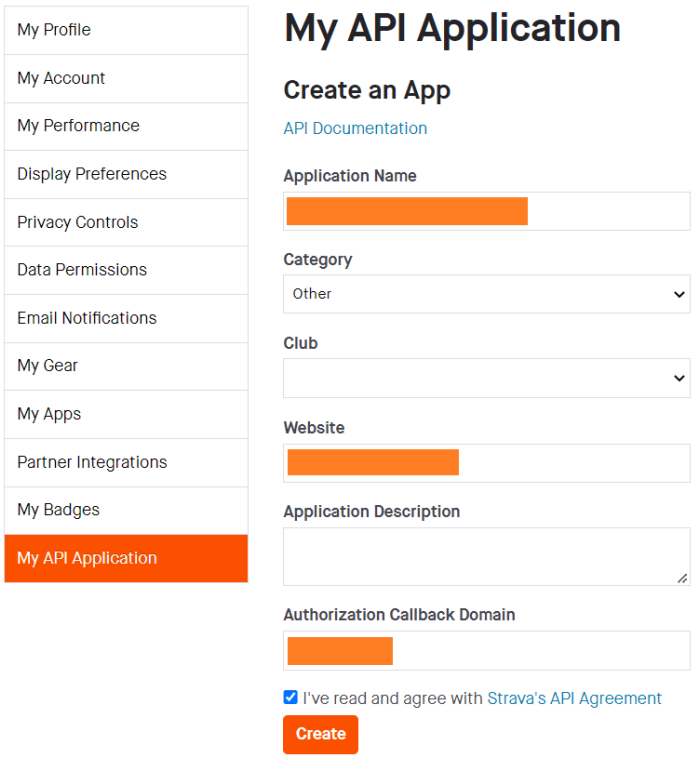
Copy

```
1 help.search('notoken', package = 'rStrava')
```

However, if you want detailed data on your activities or if your profile is private (like mine) you need an authentication token to get the data into R. To get an authentication token you do need a Strava account of your own. The instructions on the {rStrava} README are pretty easy to follow to set up your authentication token.

Setting up an authentication token

On the Strava website, in the settings, you can make an API application. There are three pieces of information you need to fill out:



- **Application Name**: the name of your app (this can be almost anything). I used the blog title.
- **Website**: Must be a valid URL, but can otherwise be pretty much anything.
- **Authorization Callback Domain**: change to `localhost` or any domain. If deploying an app that uses the Strava API, you'll need to update this.

After you click "Create", you'll be prompted to upload an icon (can be any image), and this will generate a token for you.

Now, you need to add this token into R. You can do this using the `config()` function from {httr}, and the `strava_oauth()` function from {rStrava}. The `strava_oauth()` function needs four pieces of information, all provided as character strings.

Copy

```
strava_token <- httr::config(token = strava_oauth(app_name,
                                                  app_client_id,
                                                  app_secret,
                                                  app_scope = "activity:read_all"))
```

The `app_name` is the name you gave to the app when making your token on the Strava website. The `app_client_id` and `app_secret` were generated after you clicked "Create" on the Strava website, and you can simply pass these in. You will also perhaps want to change the `app_scope` argument. By default, this is set to `"public"`, but you may want to get information on your activities which are not public. You can save the token as a variable, to pass into the {rStrava} functions. I've called it `strava_token`.

Reading in the data

With the authentication token set you can now begin to get data into R, directly from the Strava API. First of all, I grabbed the data on my activities using the `get_activity_list()` function, for which I need to pass in my Strava token. I then use the `get_activity_streams()` function to get detailed information on a specific activity. Here the `id` is the activity id i.e., the number that comes at the end of the URL string for the activity: `https://www.strava.com/activities/{id}`.

Copy

```
my_acts <- get_activity_list(strava_token)
id = {id}
strava_data <- get_activity_streams(my_acts,
                                    strava_token,
                                    id = id)
```

This is what the output of `strava_data` looks like:

Copy

```
  altitude cadence distance grade_smooth heartrate      lat       lng moving time velocity_smooth         id
1     24.8      84   0.0027          2.0       105 54.04575 -2.798552  FALSE    0          0.0000 7419225187
2     24.9      85   0.0066          1.3       112 54.04572 -2.798607   TRUE    3          4.6548 7419225187
3     24.9      85   0.0078          1.0       117 54.04572 -2.798626   TRUE    4          4.5180 7419225187
4     24.9      86   0.0078          0.8       117 54.04570 -2.798638  FALSE    5          3.6144 7419225187
5     24.9      86   0.0102          0.9       118 54.04567 -2.798653   TRUE    6          4.4892 7419225187
6     24.9      85   0.0130          1.1       119 54.04565 -2.798678   TRUE    7          5.2812 7419225187
```

There are some nice built-in mapping functions in {rStrava} that I recommend checking out, but since I'm going to build my own here, I don't need to use {rStrava} again. I saved the data as a CSV file so that I could go back and work on it again without having to re-download it using {rStrava}.

Copy

```
write.csv(strava_data, "strava_data.csv", row.names = F)
```

Data wrangling

The data the comes out of the `get_activity_streams()` function is already very clean, so the data wrangling for this example is very minimal. In fact, I only used two functions, neither of which was really necessary. I converted the data frame to a tibble using `as_tibble()` because I prefer working with tibbles. Since all the data is for a single activity in this case, the `id` column is a bit redundant so I also used `select()` from {dplyr} to remove the `id` column.

Copy

```
library(tidyverse)
strava_data %>%
```

```
3   as_tibble() %>%
4   select(-id)
```

Background maps

Now it's finally time to start building a map! Here, I loaded the rest of the R packages I'll be using for mapping and animating.

Copy

```
1 library(sf)
2 library(ggmap)
3 library(osmdata)
4 library(rcartocolor)
5 library(gganimate)
```

Here, {sf} isn't technically necessary but useful if you want to make a geometry object in R (more on that later). {ggmap} and {osmdata} are used for creating a background map. {ggplot2} has already been loaded eariler with the rest of the tidyverse, and along with {rcartocolor} for a nice colour scheme, this will plot the main map. Then, {gganimate} is used for animating the map.

Before I actually mapped my run, I wanted to get a background map. I used the `getbb()` (bounding box) function from {osmdata} to get the approximate coordinates around where I started my run using the place name as input.

Copy

```
1 getbb("Lancaster, UK")
2         min       max
3 x -2.983647 -2.458735
4 y 53.918066 54.239557
```

I then played around to get the exact rectangle I wanted, and specified it manually. Now, `bb` specifies the minimum and maximum latitude and longitude of where my background map should cover.

Copy

```
1 bb <- matrix(c(-2.9, -2.53, 53.95, 54.10),
2              ncol = 2,
3              nrow = 2,
4              byrow = TRUE,
5              dimnames = list(c("x", "y"), c("min", "max")))
```

This bounding box can be passed into `get_map()` from {ggmap} to get the background map. By default, {ggmap} uses Google Maps, for which an API key is required. Setting the `source = "stamen"` means that you don't have to register a Google API key. You can also choose a `maptype`, and here I chose `"toner-hybrid"`. I'd recommend playing around with the different types to see which one you like - use `?get_map()` for a list of options. You can also choose whether or not you want a colour or black and white background. I opted for a black and white (`"bw"`) background map, as I later found it difficult to get enough contrast between my data points and the background map otherwise.

Copy

```
1 bg_map <- get_map(bb,
2                   source = "stamen",
3                   maptype = "toner-hybrid",
4                   color = "bw")
```

The background map can be visualised using `ggmap()`.

Copy

```
1 ggmap(bg_map)
```

Overlaying the activity data

I'm simply going to use {ggplot2} to overlay the data in `strava_data` on top of my background map. Using {ggplot2}, there are (at least) two different ways we could add the data: either using `geom_point()` or `geom_sf()`. We'll start with `geom_point()`.

Copy

```
g <- ggmap(bg_map) +
     geom_point(data = strava_data,
                inherit.aes = FALSE,
                aes(x = lng,
                    y = lat,
                    colour = altitude),
                 size = 1)
```

Here, we specify `strava_data` as the `data` argument in `geom_point()`. Note that there is no `ggplot()` call here, as it's hidden inside the `ggmap()` function. Therefore, we also want to specify `inherit.aes = FALSE` to make sure that the hidden aesthetics carried through by `ggmap()` don't interfere with our point data. I specify the `x` and `y` coordinates as the longitude and latitude, respectively, and colour the points based on the altitude. I also played around with the size of the points until it looked the way I wanted it to. Note that, alternatively you could use `geom_line()` in exactly the same way.

Since, longitude and latitude are geographic data, it may make sense to instead convert them to a geometry object using the {sf} package. This may be necessary if your background map and coordinate data use different coordinate systems. In this case, it doesn't actually matter. But I'll show you anyway, just in case you need it. First, we convert our `strava_data` tibble into an `sf` object using `st_as_sf()`. We also specify which columns from `strava_data` are the longitude and latitude, with the longitude column coming first. We set the coordinate reference system (`crs`) as `4326` to match the coordinate system used. Setting `remove = FALSE` also keeps the original latitude and longitude columns in the tibble, even after converting to an `sf` object.

Copy

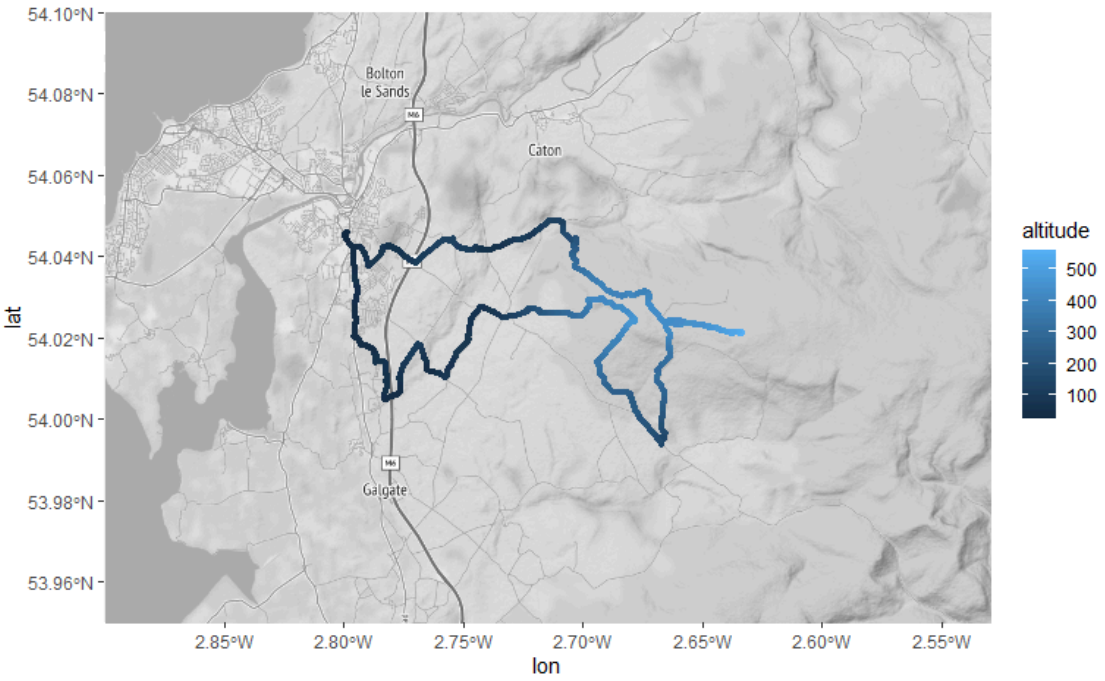
```
strava_sf <- st_as_sf(strava_data,
                      coords = c("lng", "lat"),
                      crs = 4326,
                      remove = FALSE)
```

The `strava_sf` object is now an `sf` object so it can be used with `geom_sf()` instead of `geom_point()`. Here, we don't need to specify the `x` and `y` aesthetics as they are automatically detected from the `sf` object. You may get a `Coordinate system already present. Adding new coordinate system, which will replace the existing one.` warning. This is because `geom_sf()` and `ggmap()` are both trying to set the (same) coordinate system.

Copy

```
g <- ggmap(bg_map) +
     geom_sf(data = strava_sf,
             inherit.aes = FALSE,
             aes(colour = altitude),
             size = 1)
g
```

The maps returned using `geom_point()` and `geom_sf()` are essentially the same in this case.



Styling the map

The initial map looks okay, but we can add some styling to make it look better. I'm a big fan of {rcartocolor} for colour palettes. I can get the hex codes of the `"SunsetDark"` palette, and use the same hex codes for the title font later.

Copy

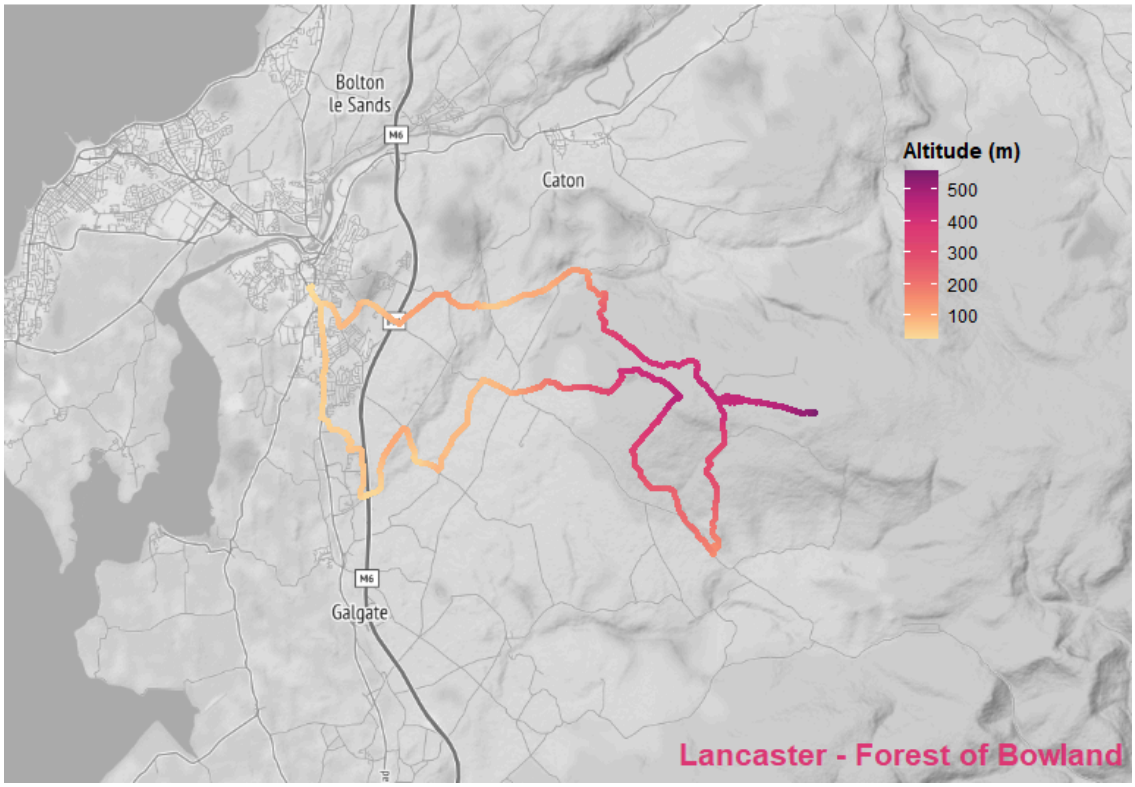
```
my_colors <- carto_pal(7, "SunsetDark")
my_colors
```

I change the colour of my points using `scale_colour_carto_c()` from {rcartocolor}, and change the title that appears in the legend at the same time. I also add a caption using the `labs()` function. Finally, I edit the theme. The `theme_void()` function is really useful for maps because it removes most of the theme elements which aren't very useful on maps like this e.g. axis labels, axis ticks, grid lines. I use the `theme()` function to bring the legend and the plot caption (used as a title here) inside the plot area. This create a little bit of white space at the bottom of the plot, so I remove it using `plot.margin`. I also edit the colour and size of the caption text.

Copy

```
g <- g +
  scale_colour_carto_c(name = "Altitude (m)", palette = "SunsetDark") +
  labs(caption = "Lancaster - Forest of Bowland ") +
  theme_void() +
```

```
  theme(legend.position = c(0.85, 0.7),
        legend.title = element_text(face = "bold", hjust = 0.5),
        plot.caption = element_text(colour = "#dc3977", face = "bold", size = 16,
                                    vjust = 10),
        plot.margin = unit(c(0, 0, -0.75, 0), unit = "cm"))
g
```



Animating with {gganimate}

I was pretty happy with the final static image, but why not animate it? {gganimate} makes it really easy to animate ggplot objects. For this example, I'd strongly recommend using the `geom_point()` version of the map.

Copy

```
g <- ggmap(bg_map) +
  geom_point(data = strava_data,
           inherit.aes = FALSE,
           aes(colour = altitude,
               x = lng,
               y = lat),
           size = 1) +
  scale_colour_carto_c(name = "Altitude (m)", palette = "SunsetDark") +
  labs(caption = "Lancaster - Forest of Bowland ") +
  theme_void() +
  theme(legend.position = c(0.85, 0.7),
        axis.title = element_blank(),
        legend.title = element_text(face = "bold", hjust = 0.5),
        plot.caption = element_text(colour = "#dc3977", face = "bold", size = 16,
                                    vjust = 10),
        plot.margin = unit(c(0, 0, -0.75, 0), unit = "cm"))
```

Although you *can* animate plots with `sf` data using {gganimate}, it's a little bit trickier and it takes longer to render. So why not make our lives a little easier? There are two functions we need to animate our map:

- `transition_time()` specifies which variable in `strava_data` we want to animate over.
- `shadow_mark()` means the animation plots points cumulatively over time rather than just plotting a single point for each time.

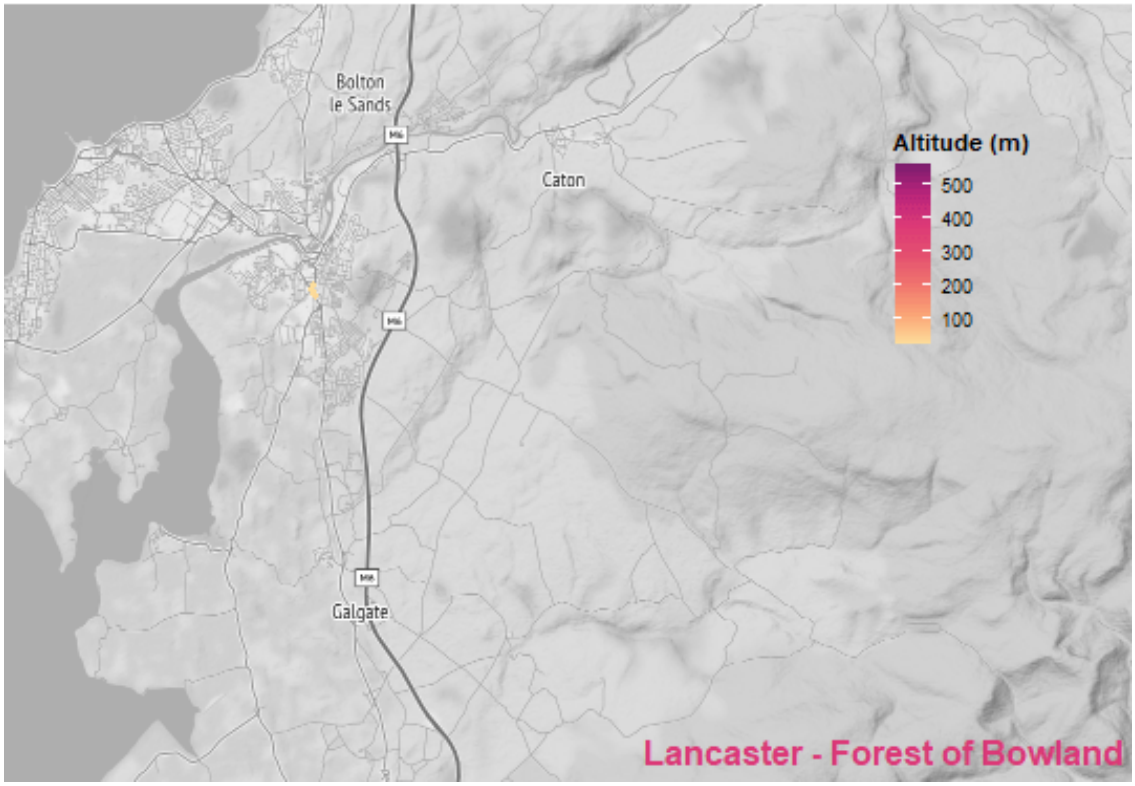
Copy

```
g = g +
  transition_time(time = time) +
  shadow_mark()
```

The `animate()` function then actually builds the animation. Usually `renderer = gifski_renderer()` should be the default, but I kept getting individual images instead of a gif unless I specified it manually - to investigate later. Here, I also specified the width and height (using a little bit of trial and error to avoid white space caused by the fixed ratio from `ggmap()`). `anim_save()` then saves the gif to a file (analogously to `ggsave()` from {ggplot2}).

Copy

```
animate(g, renderer = gifski_renderer(), height = 372, width = 538, units = "px")
anim_save("mapping_marathon.gif")
```



And that's it! You now have an animated map of your Strava recorded run (or cycle, or walk, or …)! If you want to create a map of your own, you can find the R code used in this blog on my website. Thanks very much to the creators of {rStrava} for such an easy to use package!

For attribution, please cite this work as:

> **Mapping a marathon with {rStrava}**.
> Nicola Rennie. July 18, 2022.
> nrennie.rbind.io/blog/mapping-a-marathon-with-rstrava

Licence: creativecommons.org/licenses/by/4.0
← Creating flowcharts with {ggplot2} Designing #TidyTuesday visualisations for mobile (with Quarto) →
