We explored interactive visualizations of [simple features](https://r-spatial.github.io/sf/) (i.e., interactive maps) via **ggplot2**’s geom\_sf() and **plotly**’s ggplotly(). This time we’ll make similar visualizations using **plotly**’s “non-ggplot2” mapping interfaces (namely plot\_ly(), plot\_geo(), and plot\_mapbox()). Here’s a quick example of reading a shape file into R as simple features via [st\_read()](https://r-spatial.github.io/sf/reference/st_read.html), then plotting those features (in this case, North Carolina counties) using each one of the four mapping approaches **plotly** provides. Notice how all of these options auto-magically know how to render simple features:

# to replicate these examples, you currently need the dev version

library(plotly)

nc <- sf::st\_read(system.file("shape/nc.shp", package = "sf"), quiet = TRUE)

class(nc)

#> [1] "sf" "data.frame"

subplot(nrows = 2,

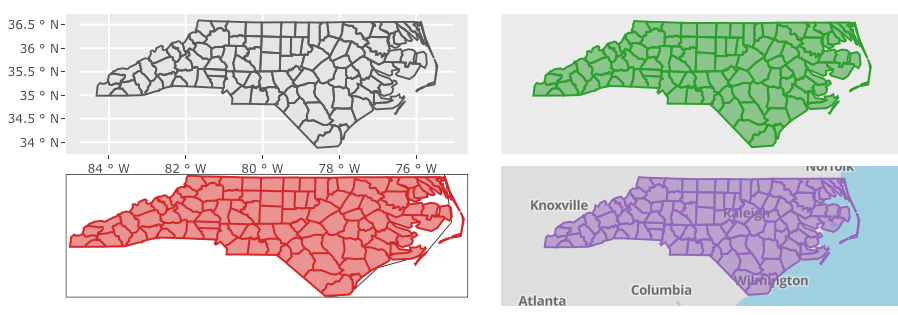
ggplot(nc) + geom\_sf(),

plot\_ly(nc),

plot\_geo(nc),

plot\_mapbox(nc)

) %>% hide\_legend()

[[](https://blog.cpsievert.me/06-sf-options.html)](https://blog.cpsievert.me/06-sf-options.html)

You might be wondering, “What can **plotly** offer over other interactive mapping packages such as [**leaflet**](https://cran.r-project.org/package=leaflet), [**mapview**](https://cran.r-project.org/package=mapview), [**mapedit**](https://cran.r-project.org/package=mapedit), etc?”. One big feature is the [linked brushing framework](https://plotly-book.cpsievert.me/linking-views-without-shiny.html), which works best when linking **plotly** together with other **plotly** graphs (i.e., only a subset of brushing features are supported when linking to other [crosstalk-compatible htmlwidgets](https://rstudio.github.io/crosstalk/widgets.html)). Another is the ability to leverage the [plotly.js API](https://plot.ly/javascript/plotlyjs-function-reference/) to make efficient updates in shiny apps [via plotlyProxy()](https://blog.cpsievert.me/2017/07/13/plotly-4.7.1-now-on-cran/). Speaking of efficiency, [plotly.js](https://plot.ly/javascript/) keeps on improving the performance of their WebGL-based rendering, so I recommend trying plot\_ly() (with toWebGL()) and/or plot\_mapbox() if you have lots of graphical elements to render. Also, by having a consistent interface between these various mapping approaches, it’s much quicker and easier to switch from one approach to another when you need to leverage a different set of [strengths and weaknesses](https://moderndata.plot.ly/visualizing-geo-spatial-data-with-sf-and-plotly/#strengths-weaknesses).

Speaking of strengths, plotly.js also has -class 3D rendering support. When used in combination with **plotly**’s linking framework, we can do some nifty things – all inside self-contained HTML! For example, [here are linked 3D & 2D views](https://blog.cpsievert.me/sf-plotly-3D-globe.html) of tropical storm paths which is useful for querying anomalies and provides some insight into the relationship between distance traveled, altitude, latitude, and longitude:[1](https://moderndata.plot.ly/visualizing-geo-spatial-data-with-sf-and-plotly/#fn1)

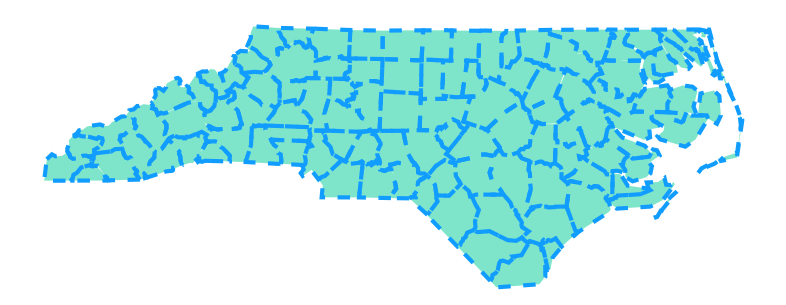
# To see the actual R code that generates the self-contained HTML result,

# enter this in your R console or visit https://github.com/ropensci/plotly/tree/master/demo

demo("sf-plotly-3D-globe", package = "plotly")

As you’ll see in the [NEWS](https://github.com/ropensci/plotly/blob/master/NEWS.md) of the development version, I’ve also added new stroke and span arguments which make it easier to control the outline color and width of filled polygons/markers/bars/etc. These new arguments work in a similar way to existing arguments like color and linetype. In particular, constant values can be specified via I():

plot\_ly(nc, stroke = I("#119dff"), span = I(5), color = I("#00cc96"), linetype = I("dash"))



Furthermore, values not wrapped in I() are mapped to a visual range defined by the plural form of the argument (i.e., strokes and spans). However, to effectively map data to these sorts of visuals, we might want to generate mutliple traces.

**One trace to rule them all?**

One of the trickiest things about [mastering **plotly**](http://consulting.cpsievert.me/#Training) and/or plotly.js is knowing what can and can not be done with just one trace. As I [elude to in the *plotly for R* book](https://plotly-book.cpsievert.me/scatter-traces.html#line-plots), if something can be implemented with a single trace, then it should, because traces don’t scale very well (i.e., can easily lead to a sluggish plot). That’s why, by default, the maps above are implemented with just one trace.[2](https://moderndata.plot.ly/visualizing-geo-spatial-data-with-sf-and-plotly/#fn2) However, when you need certain scalar (i.e., non-data-array) trace properties (e.g. fillcolor) to vary, you might want to use split to create a trace for every level in the split variable. For example, in this map of [territories](https://en.wikipedia.org/wiki/First-level_NUTS_of_the_European_Union) in [Franconia](https://en.wikipedia.org/wiki/Franconia), we ensure one trace per territory (via the split argument), which leads to two fairly obvious features:

1. The ability to hide/show territories by clicking (or double-clicking) legend entries. As we’ll discuss later, there are actually ways to do this sort of thing with *just one trace* via the [crosstalk (i.e. linked views) framework](https://moderndata.plot.ly/visualizing-geo-spatial-data-with-sf-and-plotly/#crosstalk).
2. A different color for each territory. In this case, we’ve used split without specifying color, so plotly.js will impose it’s default coloring rules, but you could easily set a constant color across traces (e.g., color = I("black")) or, as we’ll see shortly, use a custom color scheme.

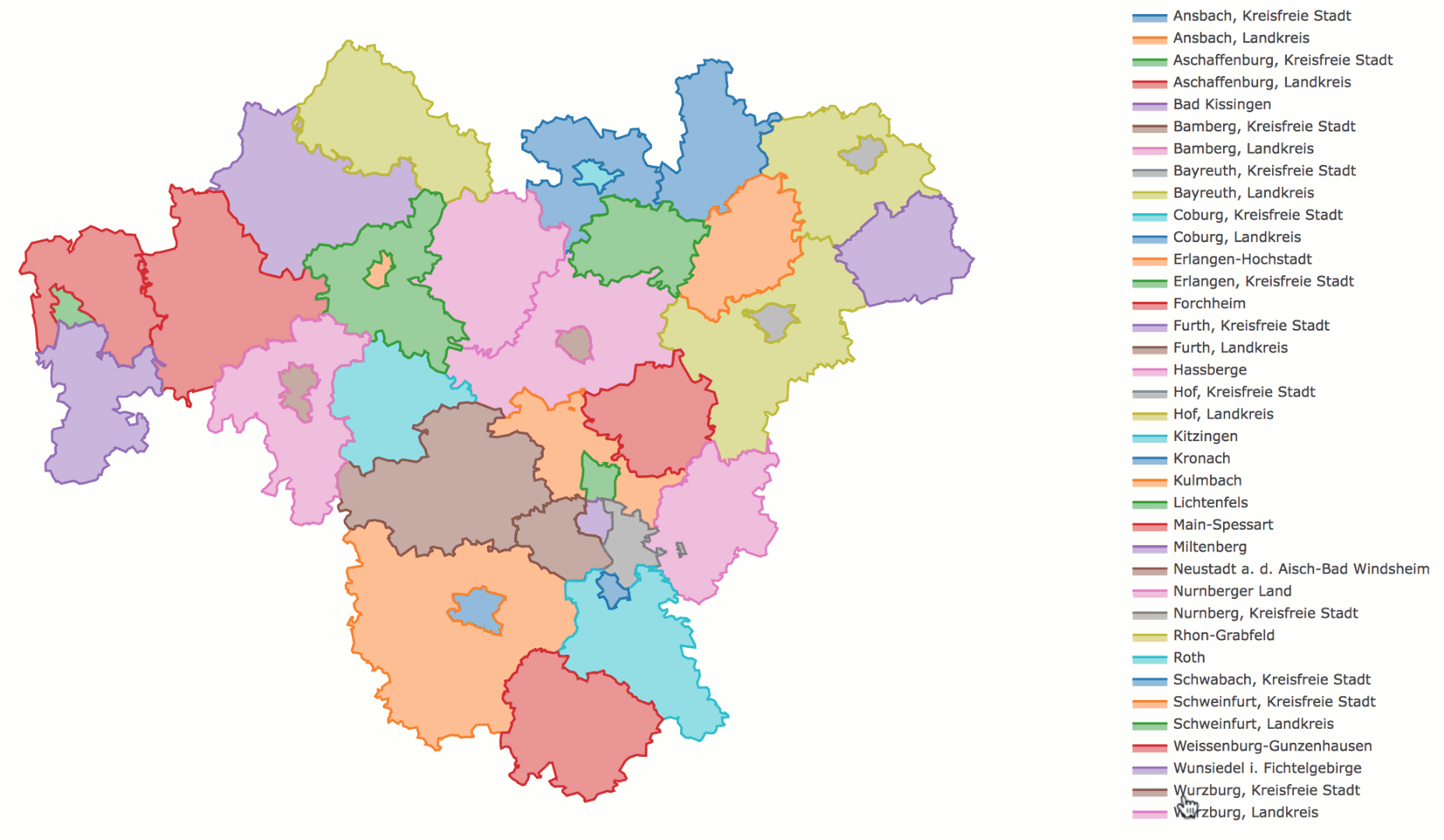
# load trails data (an sf object bundled with the mapview package)

# install.packages('mapview')

data(franconia, package = "mapview")

# Compare this result with: `plot\_ly(franconia, split = ~NAME\_ASCI, color = I("black"))`

plot\_ly(franconia, split = ~NAME\_ASCI)

[[](https://blog.cpsievert.me/franconia-split.html)](https://blog.cpsievert.me/franconia-split.html)

Having a different trace for each territory opens the door for further territory-level customization, such as having a custom color, linetype, fill-specific tooltip, etc. When color is *numeric*, and you want it to set a different fill-color for certain polygon(s), you’ll need split to ensure there is no more than one color per trace:

plot\_ly(

franconia,

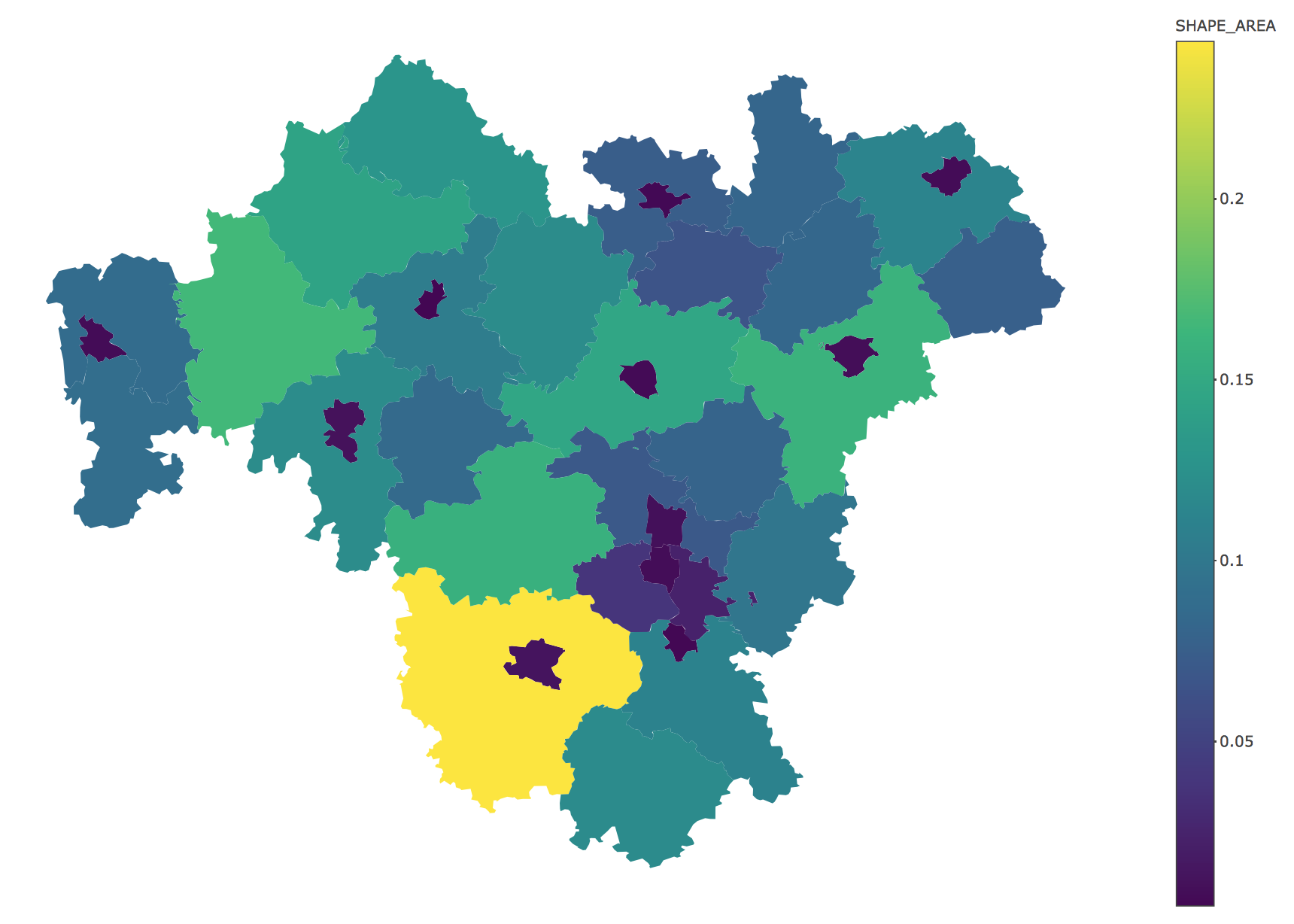
split = ~NUTS\_ID,

color = ~SHAPE\_AREA,

alpha = 1,

showlegend = FALSE

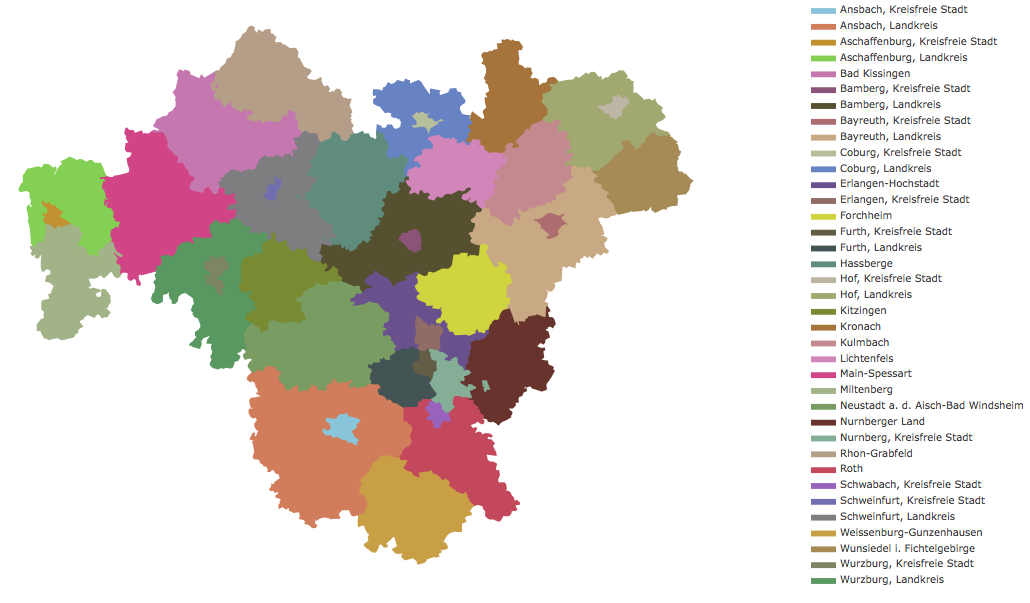
)

[[](https://blog.cpsievert.me/franconia-choro.html)](https://blog.cpsievert.me/franconia-choro.html)

On the other hand, if color is a discrete variable, plot\_ly() always produces one trace per level. Since, in this case, there are over 30 districts, I’m going to use [a color palette](https://stackoverflow.com/questions/21352683/randomising-qualitative-colours-for-large-sets-in-ggplot) that does a fairly good job in distinguishing between a large number of groups:

cols <- c("#89C5DA", "#DA5724", "#74D944", "#CE50CA", "#3F4921", "#C0717C", "#CBD588", "#5F7FC7", "#673770", "#D3D93E", "#38333E", "#508578", "#D7C1B1", "#689030", "#AD6F3B", "#CD9BCD", "#D14285", "#6DDE88", "#652926", "#7FDCC0", "#C84248", "#8569D5", "#5E738F", "#D1A33D", "#8A7C64", "#599861")

plot\_ly(franconia, color = ~NAME\_ASCI, colors = cols, alpha = 1)

[[](https://blog.cpsievert.me/franconia-color-discrete.html)](https://blog.cpsievert.me/franconia-color-discrete.html)

Note that you can still color multiple traces the same color, but if you want to have a different tooltip to appear upon hovering a specific set of polygons (as opposed to along each point), you’ll want to still use split in combination with hoveron = "fill":

plot\_ly(

franconia,

split = ~NUTS\_ID,

color = ~district,

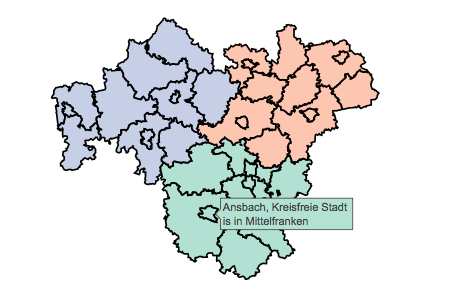
stroke = I("black"),

text = ~paste(NAME\_ASCI, "\n is in", district),

hoverinfo = "text",

hoveron = "fill"

)

[[](https://blog.cpsievert.me/franconia-color-split.html)](https://blog.cpsievert.me/franconia-color-split.html)

**Leveraging mapbox basemaps**

Among the four mapping options, the one that excites me the most is plot\_mapbox()[3](https://moderndata.plot.ly/visualizing-geo-spatial-data-with-sf-and-plotly/#fn3). The primary reason to use plot\_mapbox() (or plot\_geo()) over geom\_sf() (or plot\_ly()) is that these approaches include a base-map layer which updates dynamically on zoom and can quite helpful for providing geographic context. The base-map styling can be easily customized – especially for plot\_mapbox() via the [layout.mapbox.style attribute](https://plot.ly/r/reference/#layout-mapbox-style). In addition to a URL to a [custom style](https://www.mapbox.com/help/create-a-custom-style/), you can provide this attribute with a pre-packaged style name. To get a list of those pre-packaged styles, reach into the plotly.js plot [schema()](https://www.rdocumentation.org/packages/plotly/versions/4.7.1/topics/schema?tap_a=5644-dce66f&tap_s=10907-287229):

styles <- schema()$layout$layoutAttributes$mapbox$style$values

styles

[1] "basic" "streets" "outdoors" "light"

[5] "dark" "satellite" "satellite-streets"

A nice feature to include with these maps is a dropdown to dynamically change the base-map styling. To do so, use the [layout.updatemenus.buttons attribute](https://plot.ly/r/reference/#layout-updatemenus) to control the value of the layout.mapbox.style attribute. Since we wish to modify the layout.mapbox.style attribute, which is part of the plot’s layout, we’ll want each dropdown “button” to trigger a [relayout](https://plot.ly/javascript/plotlyjs-function-reference/) event with the appropriate args. Here is one way to create a list of buttons, one for each of the default styles:

# generate plot.js buttons, one for every style

style\_buttons <- lapply(styles, function(s) {

list(label = s, method = "relayout", args = list("mapbox.style", s))

})

With our list of style\_buttons, we’re now ready to visualize something. Here I’ll leverage the trails data from the [**mapview**](https://r-spatial.github.io/mapview/) package, which has a number of hiking trails of Franconia, Germany.

data(trails, package = 'mapview')

plot\_mapbox(trails, color = I("black")) %>%

layout(

title = "Selected hiking trails in Franconia",

mapbox = list(style = "satellite-streets"),

updatemenus = list(list(y = 0.8, buttons = rev(style\_buttons)))

)

[[](https://blog.cpsievert.me/07-satellite.html)](https://blog.cpsievert.me/07-satellite.html)

As a side note, you could also [use **shiny** to modify the basemap layer](https://blog.cpsievert.me/2017/07/13/plotly-4.7.1-now-on-cran/) without redrawing the entire plot, but the result here is self-contained HTML/SVG, which is **much easier** to share, host, scale, and maintain.

**Linking views without shiny**

In terms of linked brushing *in self-contained HTML*, **plotly** is definitely the most advanced [**crosstalk**-compatible htmlwidget](https://rstudio.github.io/crosstalk/widgets.html). You’ll be able to do more when linking **plotly** to **plotly**, , let’s demonstrate linking **plotly** with **DT**. To link views via **crosstalk**, you’ll want to supply a data frame of interest to the SharedData$new() method and route the resulting object to any plots that you want to link. By default, the row index (which, in this case, is a simple feature) is used to define the graphical queries, but you can also reference a (discrete) variable to achieve [generalized selections](http://vis.berkeley.edu/papers/generalized_selection/) (e.g., you could query all trails in a district via SharedData$new(trails, ~district)).[4](https://moderndata.plot.ly/visualizing-geo-spatial-data-with-sf-and-plotly/#fn4) Linking **plotly** with **DT** in this way gives us a pretty powerful way to identify simple features both [directly and indirectly](https://plotly-book.cpsievert.me/linking-views-without-shiny.html#selection-via-indirect-manipulation).

library(crosstalk)

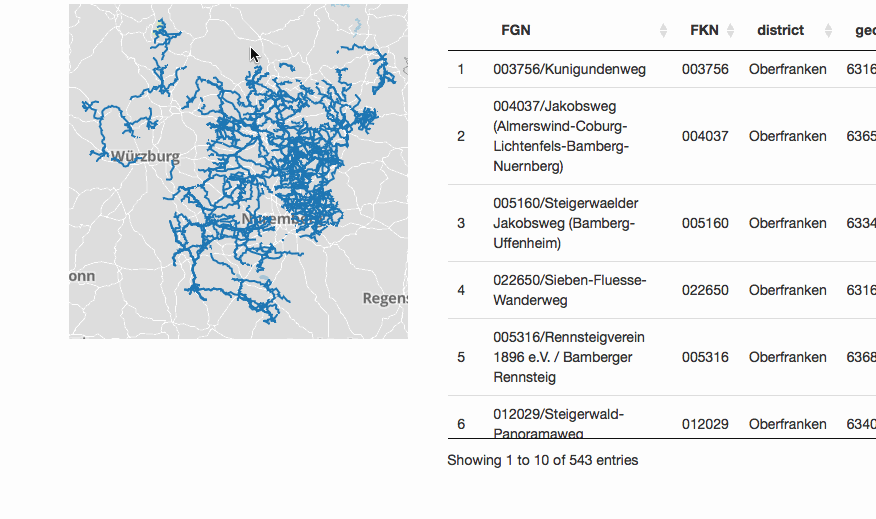
tsd <- SharedData$new(trails)

bscols(

plot\_mapbox(tsd, text = ~FKN, hoverinfo = "text"),

DT::datatable(tsd)

)

[[](https://blog.cpsievert.me/08-trails-dt.html)](https://blog.cpsievert.me/08-trails-dt.html)

To provide a sneak peak into the power of the linking framework in **plotly**, let’s leverage a fairly recent feature: brushing of aggregated traces. This example demonstrates how to brush a histogram, but a similar approach could be used to brush other aggregated traces (e.g., add\_histogram2d(), add\_boxplot(), etc). In fact, one could replicate this example with add\_bars() (instead of add\_histogram()) by pre-computing bars heights and [using a list-column key](https://plotly-book.cpsievert.me/linking-views-without-shiny.html#fig:density2scatter) to assign multiple counties to each bar. In either case (add\_bars() or add\_histogram()), it is usually a good idea to set layout.barmode = "overlay" so that newly added bars don’t use the plotly.js default of dodging the existing bars. In the case where we let plotly.js dynamically compute aggregates (i.e., add\_histogram()) it’s also a good idea to also define xbins (or ybins) so that binning of new bars use the same rules as the existing (i.e. initial) bars.

ncsd <- SharedData$new(nc)

bscols(

plot\_mapbox(ncsd) %>%

highlight(dynamic = TRUE, persistent = TRUE),

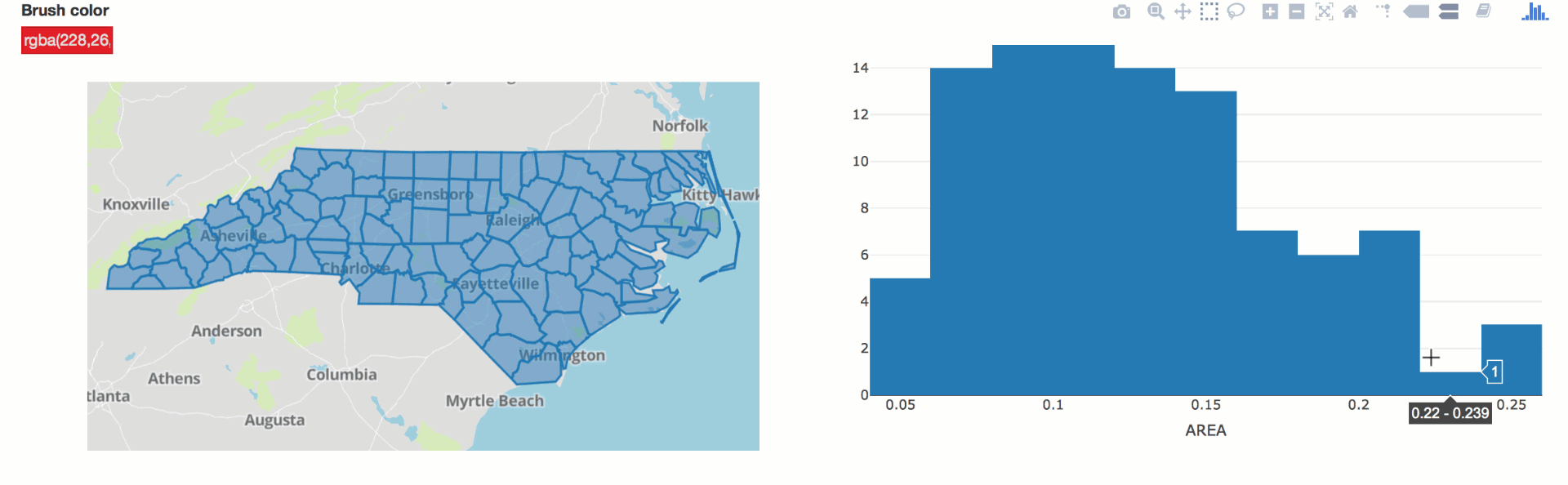
plot\_ly(ncsd, x = ~AREA) %>%

add\_histogram(xbins = list(start = 0, end = 0.3, size = 0.02)) %>%

layout(barmode = "overlay") %>%

highlight("plotly\_selected", persistent = TRUE)

)

[[](https://blog.cpsievert.me/crosstalk-aggregates.html)](https://blog.cpsievert.me/crosstalk-aggregates.html)

If you’re interested in understanding the full power of the linking framework, my [2 day plotly for R workshop](https://plotcon.plot.ly/r-shiny-dash-r/) is the best way to learn it effectively. I also offer this workshop as an on-site training course, so [please get in touch](https://carsonsievert.typeform.com/to/FKUSbW) if you have any interest!

**More learning resources**

At least currently, the best examples of using **sf** with **plotly** are within the package demos. Any demo names that are prefixed with ‘sf’ when you look at the list provided by demo(package = "plotly") are relevant. For example, demo("sf-dt", package = "plotly") gives an example of querying simple feature data by linking plot\_mapbox() with **DT** via **crosstalk**. Also be on the look-out for updates to the [mapping section](https://plotly-book.cpsievert.me/maps.html) of the [plotly for R](https://plotly-book.cpsievert.me/) book as well as examples in some of my more recent [talks](https://talks.cpsievert.me/).

**Future work**

**Appendix: strengths and weaknesses**

Below is a list of strengths (blue) and weaknesses (red) for each mapping approach in **plotly**. Note that plotly.js is still under development, so this list is likely change a bit (please [let me know](mailto:cpsievert1@gmail.com) if I’m missing anything):

**1. plot\_ly() and geom\_sf()**

* Render in SVG or WebGL (toWebGL() makes SVG -> WebGL easy) 
  + fill='toself' is [still buggy](https://github.com/plotly/plotly.js/issues/2291) in WebGL 
* Make 3D visuals by adding a z attribute 
  + With a bit of math, you can project any lat/lon data onto a sphere (i.e. globe) and color the globe’s surface according to some measure, [like radiation](https://github.com/cpsievert/Weather_Stuff/blob/ba80004/radiation-plot-3D.R#L33-L41) 
  + Unlike 2D lines, 3D lines support a color gradient 
  + Using subplot() with 3D [doesn’t currently rescale domains correctly](https://github.com/ropensci/plotly/issues/711) 
* Works with any coordinate system 
  + Currently no way to update a graticule on zoom 

**2. plot\_mapbox()**

* Full customization of base-maps that provide geo-graphic context 
* Simple features can be rendered as [data](https://github.com/ropensci/plotly/blob/master/demo/sf-mapbox-data.R) or [layout](https://github.com/ropensci/plotly/blob/master/demo/sf-mapbox-layout.R) components 
* [Numerous trace rendering limitations](https://github.com/plotly/plotly.js/issues/815) including no marker.line hoveron='fill' (i.e., stroke and span won’t do anything) 
* WebGL infrastructure allows one to render lots of graphical elements 

**3. plot\_geo():**

* Somewhat customization base-maps provide geo-graphic context 
  + Compared to plot\_mapbox(), base-map styling is quite limited 
* Supports [orthographic](https://en.wikipedia.org/wiki/Orthographic_projection) (3D) projections as well as a number of other 2D projections 

1. I got this idea thanks to [this post](https://medium.com/@plotlygraphs/how-to-create-2d-and-3d-interactive-weather-maps-in-python-and-r-77ddd53cca8) and [plotly user Emilia](https://plot.ly/~empet).[](https://moderndata.plot.ly/visualizing-geo-spatial-data-with-sf-and-plotly/#fnref1)
2. Check for yourself by doing plotly\_json(plot\_ly(nc)).[](https://moderndata.plot.ly/visualizing-geo-spatial-data-with-sf-and-plotly/#fnref2)
3. The plot\_mapbox() builds on [mapbox-gl.js](https://www.mapbox.com/mapbox-gl-js/api/) which requires an access token. Once you have one, inform **plotly** about your token via Sys.setenv('MAPBOX\_TOKEN' = 'secret token') (or via an .Renviron file).[](https://moderndata.plot.ly/visualizing-geo-spatial-data-with-sf-and-plotly/#fnref3)
4. To learn more, read <https://plotly-book.cpsievert.me/linking-views-without-shiny.html#the-shareddata-plot-pipeline>[](https://moderndata.plot.ly/visualizing-geo-spatial-data-with-sf-and-plotly/#fnref4)