Interactive Data Visualisation

Geospatial Information Visualised

Dr Ruairi O'Reilly

What we'll cover

- Creating points and polygons from GeoJSON and TopoJSON data
- Using Mercator, Mollweide, orthographic, and satellite projections
- Understanding advanced TopoJSON neighbor and merging functionality

Geospatial Information Visualised

One of the most common categories of data you'll encounter is geospatial data. This can come in the form of administrative regions like states or counties, points that represent cities or the location of a person when sending a tweet, or satellite imagery of the surface of the earth.

D3 provides enough core functionality to make any kind of map you've seen on the web (examples of maps created in this chapter using D3 can be seen in figure 8.1). Because you're already working with D3, you can make that map far more sophisticated and distinctive than the out-of-the-box maps you typically see. The major reason to use a dedicated library like Google Maps API is because of the added functionality that comes from being in that ecosystem, such as Street View. Another reason is if you want to do the cool 3D mapping you can accomplish with WebGL-based mapping libraries like MapboxGL. But if you're not looking for those features, then it may be a smarter move to build the map with D3. You won't have to invest in learning a different syntax and abstraction layer, and you'll have the greater flexibility D3 mapping affords.

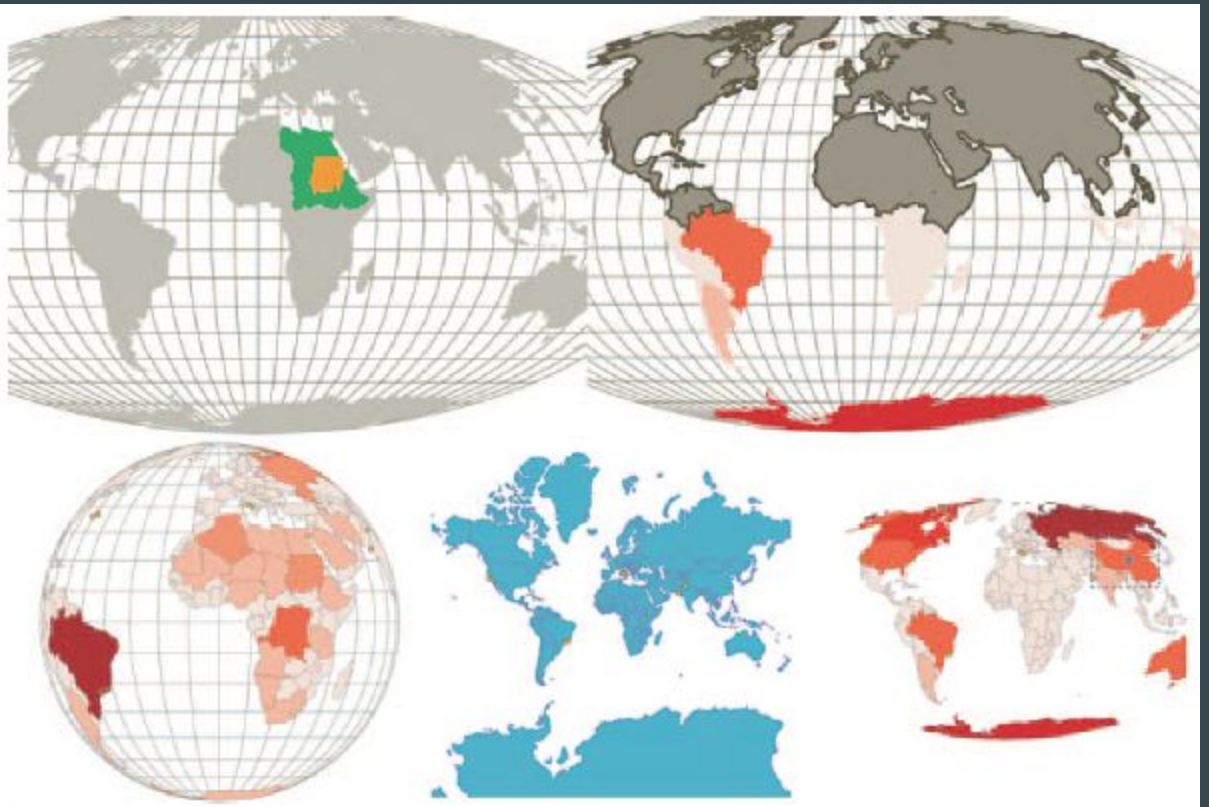

Figure 8.1 Mapping with D3 takes many forms and offers many options, including topology operations like merging and finding neighbors (section 8.4), globes (section 8.3.1), spatial calculations (section 8.1.4), and data-driven maps (section 8.1) using novel projections (section 8.1.3).

Because mapmaking and geographic information systems and science (known as GIS and GIScience, respectively) have been in practice for so long, well-developed methods exist for representing this kind of data. D3 has built-in robust functionality to load and display geospatial data. A related library that you'll get to know in this chapter, TopoJSON, provides more functionality for geospatial information visualization.

In this lecture, we'll start by making maps that combine points, lines, and polygons using data from CSV- and GeoJSON-formatted sources. You'll learn how to style those maps and provide interactive zooming by revisiting d3.zoom() and exploring it in more detail. After that, we'll look at the TopoJSON data format, its built-in functionality that uses topology, and why it provides significantly smaller data files. Finally, you'll learn how to make maps using tiles to show terrain and satellite imagery.

Basic mapmaking

Before you explore the boundaries of mapping possibilities, you need to make a simple map. In D3, the simplest map you can make is a vector map using SVG *<path>* and *<circle>* elements to represent countries and cities. We can bring back cities.csv, which we used in chapter 2, and finally take advantage of its coordinates, but we need to look a bit further to find the data necessary to represent those countries. After we have that data, we can render it as areas, lines, or points on a map. Then we can add interactivity, such as highlighting a region when you move your mouse over it or computing and showing its center.

Before we get started, though, let's look at the CSS for this chapter, as shown in the following listing.

Listing 8.1 ch8.css

```
path.countries {
   stroke-width: 1;
   stroke: #75739F;
   fill: #5EAFC6;
}
circle.cities {
   stroke-width: 1;
   stroke: #4F442B;
   fill: #FCBC34;
}
circle.centroid {
   fill: #75739F;
   pointer-events: none;
}
rect.bbox {
   fill: none;
   stroke-dasharray: 5 5;
   stroke: #75739F;
   stroke-width: 2;
   pointer-events: none;
}
path.graticule {
   fill: none;
   stroke-width: 1;
   stroke: #9A8B7A;
}
path.graticule.outline {
   stroke: #9A8B7A;
}

path.merged {
   fill: #9A8B7A;
   stroke: #4F442B;
   stroke-width: 2px;
}
```

A centroid is the center point of a geographic feature—we'll see them later

Graticules are those background latitude and longitude lines you see on maps—you'll learn how to create them in this chapter

Finding data

Making a map requires data, and you have an enormous amount of data available. Geographic data can come in several forms. If you're familiar with GIS, then you'll be familiar with one of the most common forms for complex geodata, the shapefile, which is a format developed by Esri and is most commonly found in desktop GIS applications. But the most human-readable form of geodata is latitude and longitude (or xy coordinates like we list in our file) when dealing with points like cities, often in a CSV. We'll use cities.csv, shown in the following listing. This is the same CSV we measured in chapter 2 that had the locations of eight cities from around the world.

Listing 8.2 cities.csv

```
"label","population","country","x","y"
"San Francisco", 750000,"USA",-122.431,37.773
"Fresno", 500000,"USA",-119.772,36.746
"Lahore",12500000,"Pakistan",74.329,31.582
"Karachi",13000000,"Pakistan",67.005,24.946
"Rome",2500000,"Italy",12.492,41.890
"Naples",1000000,"Italy",14.305,40.853
"Rio",12300000,"Brazil",-42.864,-22.752
"Sao Paolo",12300000,"Brazil",-46.330,-23.944
```

Make that address data work for you

If you only have city names or addresses and need to get latitude and longitude, you can take advantage of geocoding services that provide latitude and longitude from addresses. These exist as APIs and are available on the web for small batches. You can see an example of these services maintained by Texas A&M .

When dealing with more complex geodata like shapes or lines, you'll necessarily deal with more complex data formats. You'll want to use GeoJSON, which has become the standard for web-mapping data.

GeoJSON

GeoJSON is, like it sounds, a way of encoding geodata in JSON format. Each feature in a featureCollection is a JSON object that stores the border of the feature in a coordinates array as well as metadata about the feature in a properties hash object. For instance, if you wanted to draw a square that went around the island of Manhattan, it would have corners at [–74.0479, 40.6829], [–74.0479, 40.8820], [-73.9067, 40.8820], and [–73.9067, 40.6829], as shown in figure 8.2. You can easily export shapefiles into GeoJSON using QGIS (a desktop GIS application, qgis.org), PostGIS (a spatial database run on Postgres, postgis.net), GDAL (a library for manipulation of geospatial data, gdal.org), and other tools and libraries. A rectangle drawn over a geographic feature like this is known as a bounding box. It's often represented with only two coordinate pairs: the upper-left and bottom-right corners. But any polygon data, such as the irregular border of a state or coastline, can be represented by an array of coordinates like this.

In the following listing, we have a fully compliant GeoJSON "FeatureCollection" with only one feature: the simplified borders of the small nation of Luxembourg.

Figure 8.2 A polygon drawn at the coordinates [–74.0479, 40.8820], [–73.9067, 40.8820], [–73.9067, 40.6829], and [–74.0479, 40.6829].

Listing 8.3 GeoJSON example of Luxembourg

```
{
   "type": "FeatureCollection",
   "features": [
        {
            "type": "Feature",
            "id": "LUX",
            "properties": {
                "name": "Luxembourg"
            },
            "geometry": {
                "type": "Polygon",
                "coordinates": [
                    [
                        [
                            6.043073,
                            50.128052
                        ],
                        [
                            6.242751,
                            49.902226
                        ],
                        [
                            6.18632,
                            49.463803
                        ],
                        [
                            5.897759,
                            49.442667
                        ],
                        [
                            5.674052,
                            49.529484
                        ],
                        [
                            5.782417,
                            50.090328
```

GeoJSON

We're not going to create our own GeoJSON in this lecture, and unless you get into serious GIS, you may never create your own GeoJSON. Instead, you can get by with downloading existing geodata and either use it without editing it or edit it in a GIS application and export it. In our examples in this chapter, we'll use world.geojson (available here), a file that consists of the countries of the world in the same simplified, low resolution representation that you see in figure 8.3.

Figure 8.3 A map of the world using the default settings for D3's Mercator projection. You can see most of the Western Hemisphere and some of Europe and Africa, but the rest of the world is rendered out of sight.

PROJECTION

Entire books have been written on creating web maps, and an entire book could be written on using D3.js for crafting maps. Because this is only one week of lectures, I'll gloss over many deep issues. One of these is projection. In GIS, *projection* refers to the process of rendering points on a globe, like the earth, onto a flat plane, like your computer monitor. You can project geographic data in many different ways for representation on your screen, and in this chapter we'll look at a few different methods.

To start, we'll use one of the most common geographic projections, the Mercator projection, which is used in most web maps. It became the de facto standard because it's the projection used by Google Maps. To use the Mercator projection, you have to include an extension of D3, *d3.geo.projection.js*, which you'll want for the more interesting work you'll do later in the chapter. By defining a projection, you can take advantage of *d3.geoPath*, which draws geodata onscreen based on your selected projection. After we've defined a projection and have *geo.path()* ready, the entire code in listing 8.4 is all we need to draw the map shown in figure 8.3.

Why do you only see part of the world in figure 8.3? Because the default scale and transform of the Mercator projection show only part of the world in your SVG canvas.

If you want to center the map on a different part of the world, you need to change the scale and transform, as we will shortly. Each projection has a .translate() and .scale() that follow the syntax of the transform convention in SVG, but have different effects with different projections.

Listing 8.4 Initial mapping function

```
d3.json("world.geojson", createMap);
function createMap(countries) {
  var aProjection = d3.geoMercator();
  var geoPath = d3.geoPath().projection(aProjection);
  d3.select("svg").selectAll("path").data(countries.features)
    .enter()
    .append("path")
    .attr("d", geoPath)
    .attr("class", "countries");
  };
```

Projection functions have many options that you'll see later

d3.geoPath() takes properly formatted GeoJSON features and returns SVG drawing code for SVG paths

d3.geoPath() defaults to albersUSA, which is a projection suitable only for maps of the United States

SCALE

You have to do several tricks to set the right scale for certain projects. For instance, with our Mercator projection, if we divide the width of the available space by 2 and divide the quotient by Math.pi, then the result will be the proper scale to display the entire world in the available space. Figuring out the right scale for your map and your projection is typically done through experimenting with different values, but it's easier when you include zooming, as you'll see in section 8.2.2.

Different families of projections have different scale defaults. The d3.geo.albers-Usa projection defaults to 1070, whereas d3.geo.mercator defaults to 150. As with most D3 functions like this, you can see the default by calling the function without passing it a value:

```
d3.geoMercator().scale()      <—— 150
d3.geoAlbersUsa().scale()     <—— 1070
```

By adjusting the translate and scale as in listing 8.5, we can adjust the projection to show different parts of the geodata we're working with—in our case, the world. The result in figure 8.4 shows that we now see the entire world rendered.

Figure 8.4 The Mercator-projected world from our data now fitting our SVG area. Notice the enormous distortion in size of regions near the poles, such as Greenland and Antarctica.

Listing 8.5 Simple map with scale and translate settings

```
function createMap(countries) {
   var aProjection = d3.geoMercator()
  .scale(80)
  .translate([250, 250]);
  var geoPath = d3.geoPath().projection(aProjection);
  d3.select("svg").selectAll("path").data(countries.features)
   .enter()
   .append("path")
   .attr("d", geoPath)
   .attr("class", "countries");
};
```

Scale values are different for different families of projections—80 works well in this case

Moves the center of the projection to the center of our canvas

Drawing points on a map

Projection isn't used only to display areas; it's also used to place individual points. Typically, you think of cities or people as represented not by their spatial footprint (though you do this with particularly large cities) but with a single point on a map, which is sized based on a variable such as population. A D3 projection can be used not only in a geo.path() but also as a function on its own. When you pass it an array with a pair of latitude and longitude coordinates, it returns the screen coordinates necessary to place that point. For instance, if we want to know where to place a point representing San Francisco (roughly speaking, –122 latitude, 37 longitude), we could pass those values to our projection. This code segment will return xy screen coordinates (roughly [79.65, 194.32]: aProjection([-122,37])

We can use this to add cities to our map along with loading the data from cities.csv, as in listing 8.6 and which you see in figure 8.5.

Listing 8.6 Loading point and polygon geodata

```
var PromiseWrapper = (xhr, d) => new
    Promise(resolve => xhr(d, (p) => resolve(p)))

Promise.all([PromiseWrapper(d3.json, "world.geojson"),
    PromiseWrapper(d3.csv, "cities.csv")])
.then(resolve => {
    createMap(resolve[0], resolve[1])
})

function createMap(countries, cities) {
    var projection = d3.geoMercator()
        .scale(80)
        .translate([250, 250]);
    var geoPath = d3.geoPath().projection(projection);
    d3.select("svg").selectAll("path").data(countries.features)
        .enter()
        .append("path")
        .attr("class", "countries")
        .attr("d", geoPath)
    d3.select("svg").selectAll("circle").data(cities)
        .enter()
        .append("circle")
        .attr("class", "cities")
        .attr("r", 3)
        .attr("cx", d => projection([d.x,d.y])[0])
        .attr("cy", d => projection([d.x,d.y])[1])
}
```

Updated our promise wrapper to let us send the specific xhr request

You want to draw the cities over the countries, so you append them second.

Projection returns an array, which means you need to take the [0] value for cx and the [1] value for cy

One thing to note from listing 8.6 is that coordinates are often given in the real world in the order of “latitude, longitude.” Because latitude corresponds to the y-axis and longitude corresponds to the x-axis, you have to flip them to provide the x, y coordinates necessary for GeoJSON and D3.

Figure 8.5 Our map with our eight world cities added to it. At this distance, you can’t tell how inaccurate these points are, but if you zoom in, you see that both of our Italian cities are in the Mediterranean.

References

All content is taken verbatim from Chapter 8 of D3.js in Action 2nd Edition