Data visualization

COSC 480B

Reyan Ahmed

rahmed1@colgate.edu

Lecture 12

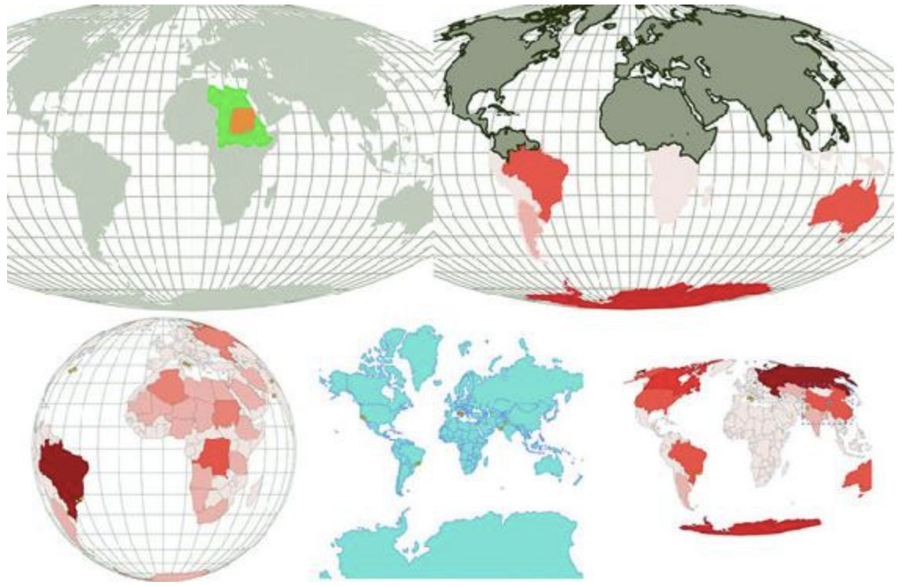
Geospatial information visualization

Overview

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

Overview

Mapping with D3 takes many forms and offers many options, including topology operations like merging and finding neighbors, globes, spatial calculations, and data-driven maps using novel projections



Basic mapmaking

CSS

```
path.countries {
   stroke-width: 1;
   stroke: #75739F;
   fill: #5EAFC6;
}
circle.cities {
   stroke-width: 1;
   stroke: #4F442B;
   fill: #FCBC34;
}
circle.centroid {                 1
   fill: #75739F;
   pointer-events: none;
}
```

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

Basic mapmaking

```
rect.bbox {
   fill: none;
   stroke-dasharray: 5 5;
   stroke: #75739F;
   stroke-width: 2;
   pointer-events: none;
}
path.graticule {                    2
   fill: none;
   stroke-width: 1;
   stroke: #9A8B7A;
}
path.graticule.outline {
   stroke: #9A8B7A;
}
```

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

Basic mapmaking

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

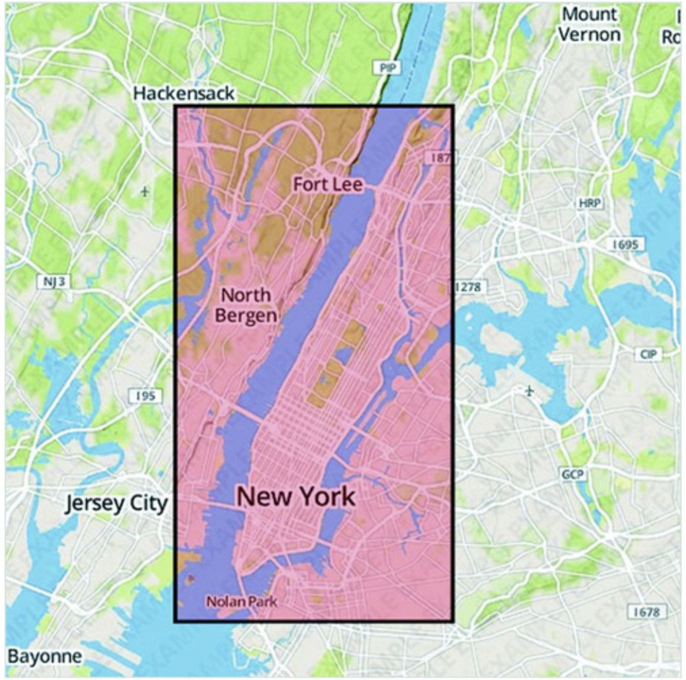
Basic mapmaking

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
```

Basic mapmaking

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



Basic mapmaking

GeoJSON example of Luxembourg

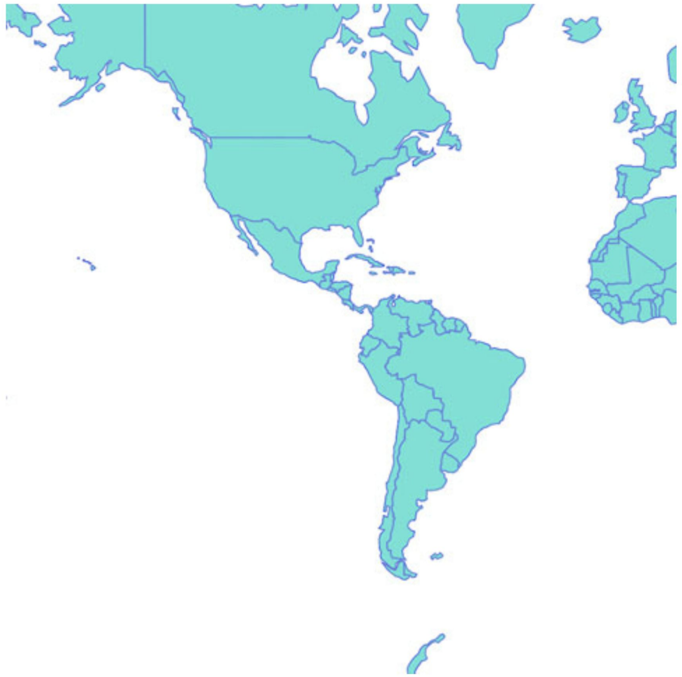
```
{
   "type": "FeatureCollection",
   "features": [
        {
            "type": "Feature",
            "id": "LUX",
            "properties": {
                "name": "Luxembourg"
            },
            "geometry": {
                "type": "Polygon",
                "coordinates": [
                    [
                        [
                            6.043073,
                            50.128052
                        ],
```

Basic mapmaking

```
                    [
                        6.242751,
                        49.902226
                    ],
                    [
                        6.18632,
                        49.463803
                    ],
                    [
                        5.897759,
                        49.442667
                    ],
                    [
                        5.674052,
                        49.529484
                    ],
                    [
                        5.782417,
                        50.090328
                    ],
                    [
                        6.043073,
                        50.128052
                    ]
                ]
            ]
        }
    }
  ]
}
```

Basic mapmaking

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.



Basic mapmaking

Initial mapping function

```
d3.json("world.geojson", createMap);
  function createMap(countries) {
    var aProjection = d3.geoMercator();                          1
    var geoPath = d3.geoPath().projection(aProjection);          2
    d3.select("svg").selectAll("path").data(countries.features)
      .enter()
      .append("path")
      .attr("d", geoPath)                                        3
      .attr("class", "countries");
    };
```

1 Projection functions have many options that you'll see later
2 d3.geoPath() defaults to albersUSA, which is a projection suitable only for maps of the United States
3 d3.geoPath() takes properly formatted GeoJSON features and returns SVG drawing code for SVG paths

Basic mapmaking

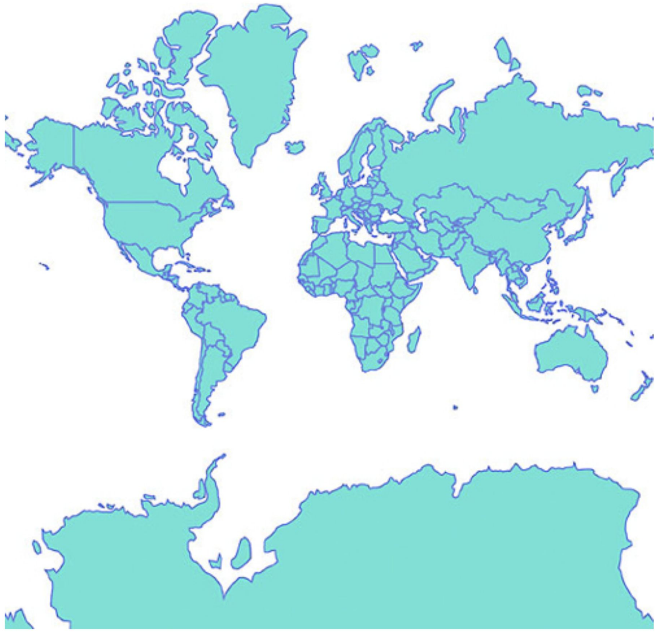
By default we only see a part of the map in the screen. My changing the scale we can fit the whole map.

```
d3.geoMercator().scale()         1
d3.geoAlbersUsa().scale()         2
```

1 150
2 1070

Basic mapmaking

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.



Basic mapmaking

Simple map with scale and translate settings

```
function createMap(countries) {
     var aProjection = d3.geoMercator()
    .scale(80)                                                    1
    .translate([250, 250]);                                       2
    var geoPath = d3.geoPath().projection(aProjection);
    d3.select("svg").selectAll("path").data(countries.features)
     .enter()
     .append("path")
     .attr("d", geoPath)
     .attr("class", "countries");
  };
```

1 Scale values are different for different families of projections—80 works well in this case
2 Moves the center of the projection to the center of our canvas

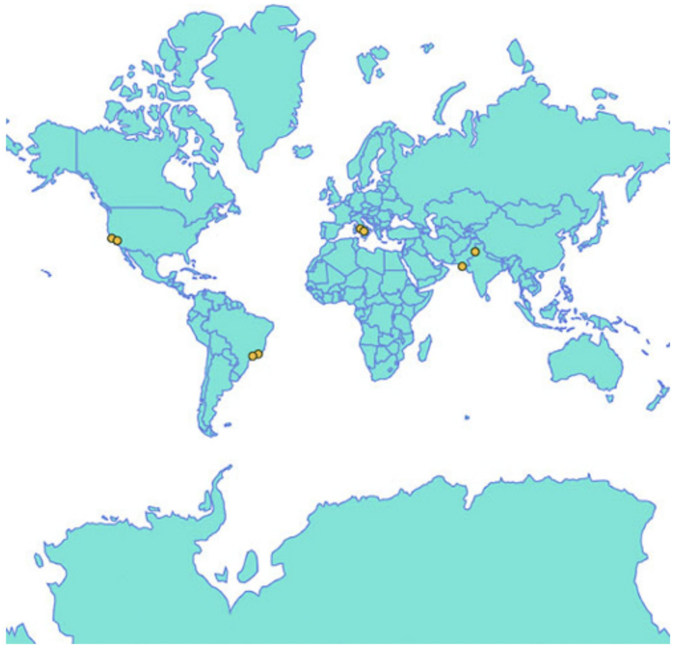
Basic mapmaking

- One can think the projection as a scaling
- It maps the longitude and latitude to the screen coordinates
- For example, to map -122, 37 in the screen, call:

```
aProjection([-122,37])
```

Basic mapmaking

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.



Basic mapmaking

Loading point and polygon geodata

```
var PromiseWrapper = (xhr, d) => new
    Promise(resolve => xhr(d, (p) => resolve(p)))                1

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

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

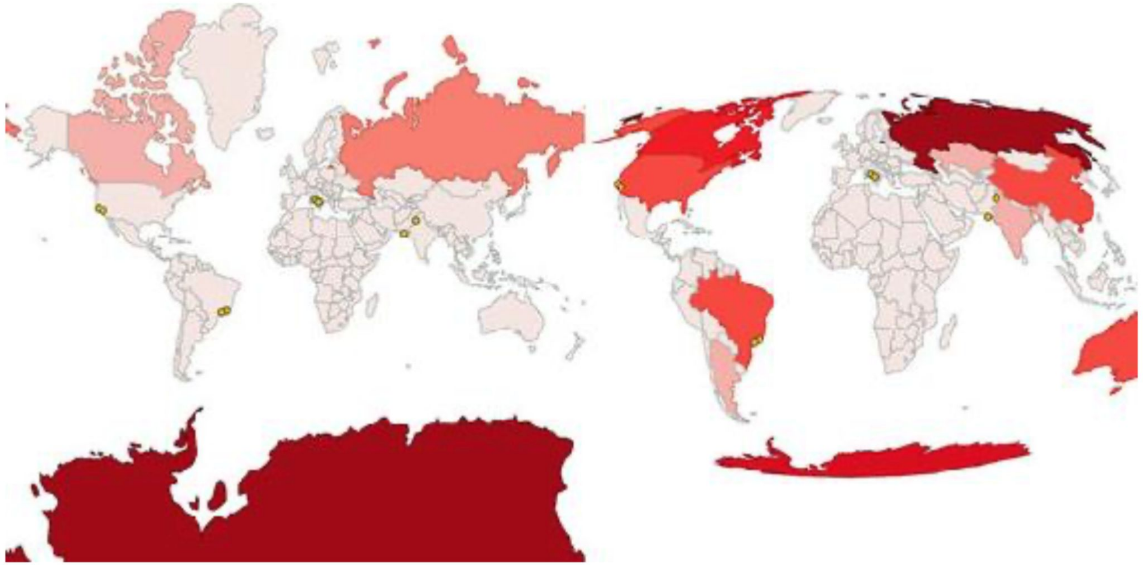
Basic mapmaking

```
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)            2
        .enter()
        .append("circle")
        .attr("class", "cities")
        .attr("r", 3)
        .attr("cx", d => projection([d.x,d.y])[0])              3
        .attr("cy", d => projection([d.x,d.y])[1])
}
```

2 You want to draw the cities over the countries, so you append them second.
3 Projection returns an array, which means you need to take the [0] value for cx and the [1] value for cy

Basic mapmaking

Mercator (left) dramatically distorts the size of Antarctica so much that no other shape looks as large. In comparison, the Mollweide projection maintains the physical area of the countries and continents in your geodata, at the cost of distorting their shape and angle. Notice that geo.path.area measures the graphical area and not the physical area of the features.



Basic mapmaking

Mollweide projected world

```
function createMap(countries, cities) {
var projection = d3.geoMollweide()
 .scale(120) #A
 .translate([250, 250]);                                    1
var geoPath = d3.geoPath().projection(projection);
var featureSize = d3.extent(countries.features, d =>
geoPath.area(d))     2
var countryColor = d3.scaleQuantize()                                    2
             .domain(featureSize).range(colorbrewer.Reds[7]);
```

1 For a Mollweide projection, these settings show the entire world on a 500 px map
2 Measures the features and assigns the size classes to a color ramp

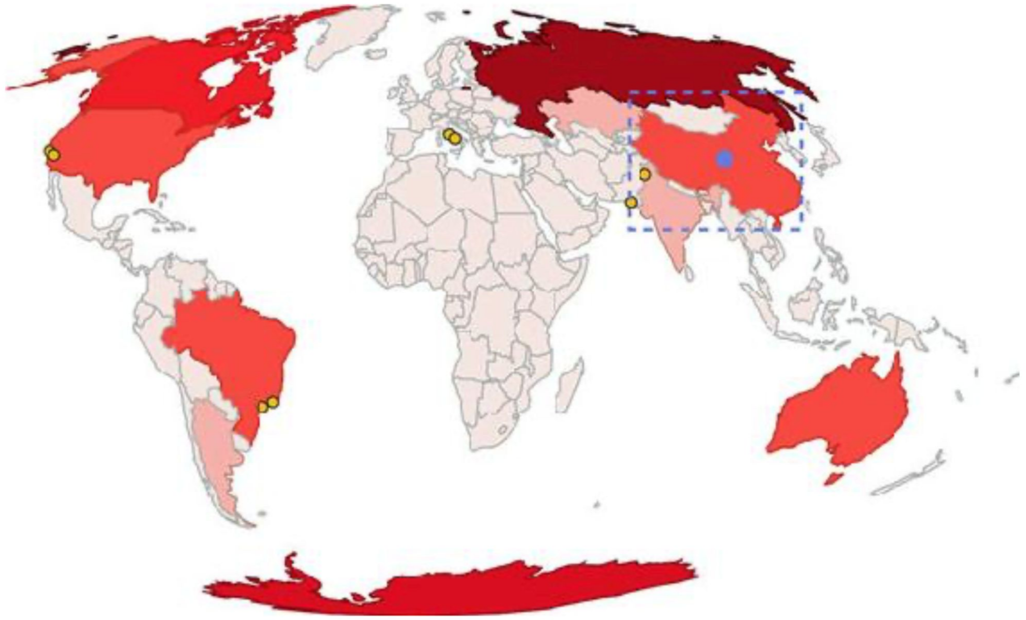
Basic mapmaking

```
d3.select("svg").selectAll("path").data(countries.features)
   .enter()
   .append("path")
   .attr("d", geoPath)
   .attr("class", "countries")
   .style("fill", d => countryColor(geoPath.area(d)))                    3
   .style("stroke", d =>
d3.rgb(countryColor(geoPath.area(d))).darker())
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]);
};
```

3 Colors each country based on its size

Basic mapmaking

Your interactivity provides a bounding box around each country and a red circle representing its graphical center. Here you see the bounding box and centroid of China. The D3 implementation of a centroid is weighted, so that it's the center of most area, and not only the center of the bounding box.



Basic mapmaking

Rendering bounding boxes with geodata

```
d3.selectAll("path.countries")
  .on("mouseover", centerBounds)
  .on("mouseout", clearCenterBounds);
function centerBounds(d) {
  var thisBounds = geoPath.bounds(d);                    1
  var thisCenter = geoPath.centroid(d);                  1
  d3.select("svg")
    .append("rect")
    .attr("class", "bbox")
    .attr("x", thisBounds[0][0])                    2
    .attr("y", thisBounds[0][1])                    2
    .attr("width", thisBounds[1][0] - thisBounds[0][0])          2
    .attr("height", thisBounds[1][1] - thisBounds[0][1])         2
  d3.select("svg")
    .append("circle")
    .attr("class", "centroid")
    .attr("r", 5)
    .attr("cx", thisCenter[0]).attr("cy", thisCenter[1])         3
}

function clearCenterBounds() {
  d3.selectAll("circle.centroid").remove();                4
  d3.selectAll("rect.bbox").remove();
}
```

Functions of geo.path that give results based on the associated projection
2 Bounding box is the top-left and bottom-right coordinates as an array
3 Centroid is an array with the x and y coordinates of the center of a feature
4 Removes the shapes when you mouse off a feature

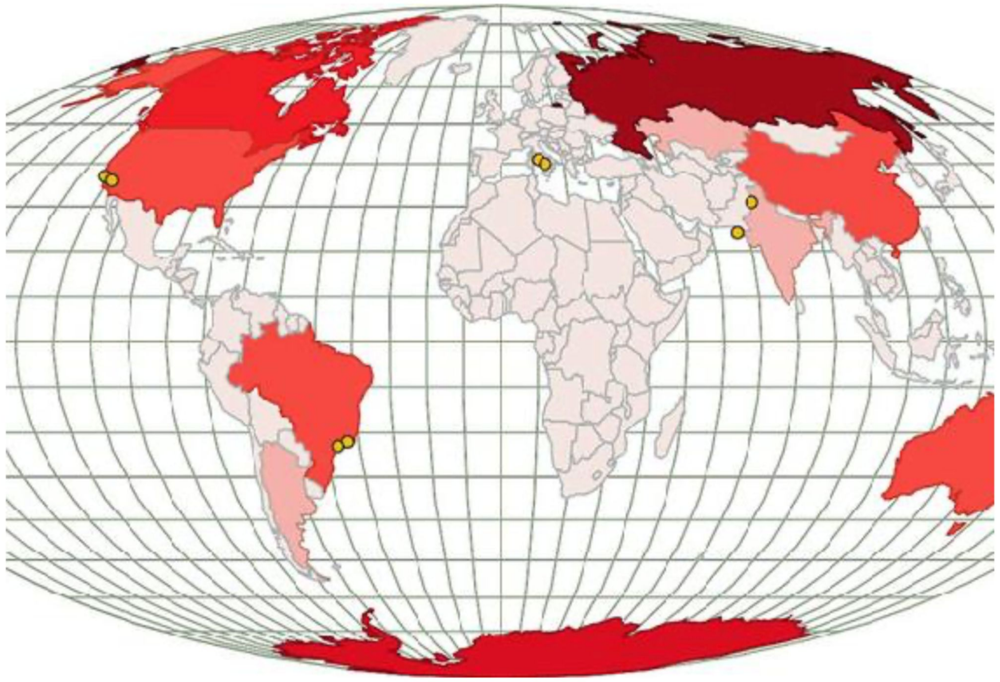
Better mapping

Adding a graticule

```
var graticule = d3.geoGraticule();
d3.select("svg").insert("path", "path.countries")
  .datum(graticule)
  .attr("class", "graticule line")
  .attr("d", geoPath)
d3.select("svg").insert("path", "path.countries")
  .datum(graticule.outline)
  .attr("class", "graticule outline")
  .attr("d", geoPath)
```

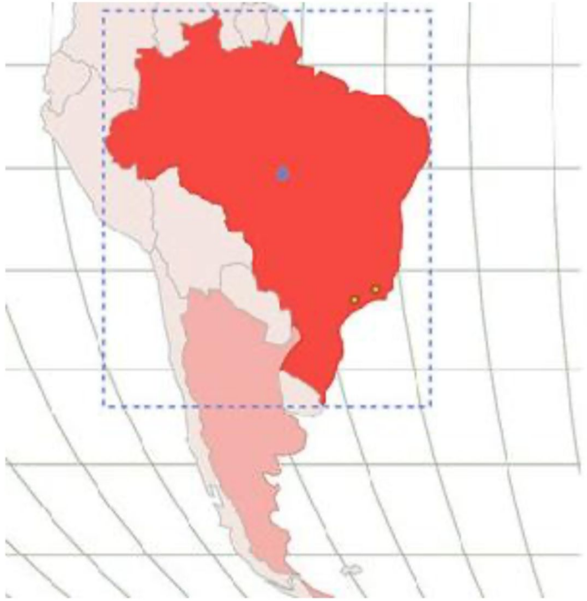
Better mapping

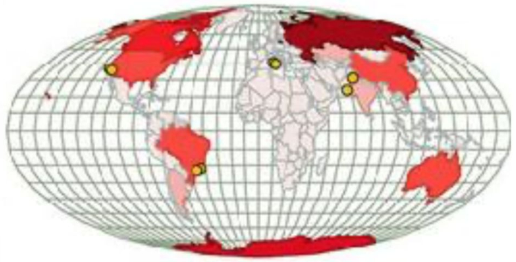
Our map with a graticule (in light gray) and a graticule outline (the black border around the edge of the map)



Better mapping

Our map with zooming enabled. Panning occurs with the drag behavior and zooming with mousewheel and/or double-clicking. Notice that the bounding box and centroid functions still work because they're based on our constantly updating projection.





Better mapping

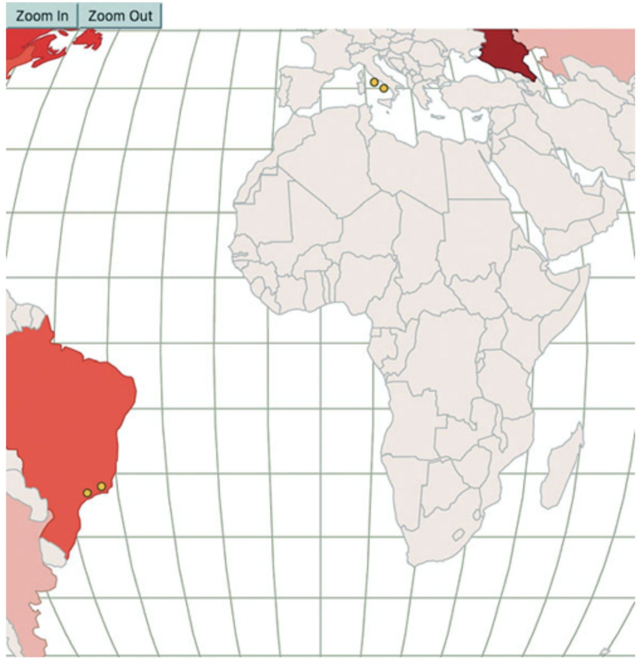
Zoom and pan with maps

```
var mapZoom = d3.zoom()
   .on("zoom", zoomed)

var zoomSettings = d3.zoomIdentity                                        1
   .translate(250, 250)
   .scale(120)

d3.select("svg").call(mapZoom).call(mapZoom.transform, zoomSettings)      1

function zoomed() {
  var e = d3.event                                                        2
    projection.translate([e.transform.x, e.transform.y])                  3
      .scale(e.transform.k)                                               3
    d3.selectAll("path.graticule").attr("d", geoPath)
    d3.selectAll("path.countries").attr("d", geoPath)                     4
    d3.selectAll("circle.cities")
   .attr("cx", d => projection([d.x,d.y])[0])                             5
   .attr("cy", d => projection([d.x,d.y])[1])
}
```

1 We use zoomIdentity to overwrites the translate and scale of the zoom to match the projection
2 Get the zoom settings from the event
3 Whenever the zoom behavior is called, this overwrites the original projection values to match the updated zoom values
4 Any path will be properly redrawn by calling the d3.geoPath associated with the updated projection
5 Also calls the now-updated projection

Better mapping

Zoom buttons and the effect of clicking Zoom In twice. Because the zoom buttons modify the zoom behavior's translate and scale, any mouse interaction afterward reflects the updated settings.



Better mapping

Manual zoom controls for maps

```
function zoomButton(zoomDirection) {
  var width = 500
  var height = 500
  if (zoomDirection == "in") {
    var newZoom = projection.scale() * 1.5;                              1
    var newX =
      ((projection.translate()[0] - (width / 2)) * 1.5) + width / 2;      2
    var newY =
      ((projection.translate()[1] - (height / 2)) * 1.5) + height / 2;
  }
  else if (zoomDirection == "out") {
    var newZoom = projection.scale() * .75;
    var newX = ((projection.translate()[0] - (width / 2)) * .75) + width / 2;
    var newY = ((projection.translate()[1] - (height / 2)) * .75) + height / 2;
  }
```

1 Calculating the new scale is easy
2 Calculating the new translate settings isn’t so easy and requires that you recalculate the center

Better mapping

Manual zoom controls for maps

```
var newZoomSettings = d3.zoomIdentity
    .translate(newX, newY)                                              3
    .scale(newZoom)

  d3.select("svg").transition().duration(500).call(mapZoom.transform,
newZoomSettings)                                                    4

}d3.select("#controls").append("button").on("click", () => {
                      zoomButton("in")}).html("Zoom In");
d3.select("#controls").append("button").on("click", () => {
                      zoomButton("out")}).html("Zoom Out");
```

3 Sets the zoom behavior's scale and translate settings to your new settings
4 Calls the zoom function associated with the SVG through a transition, so your zooming is animated

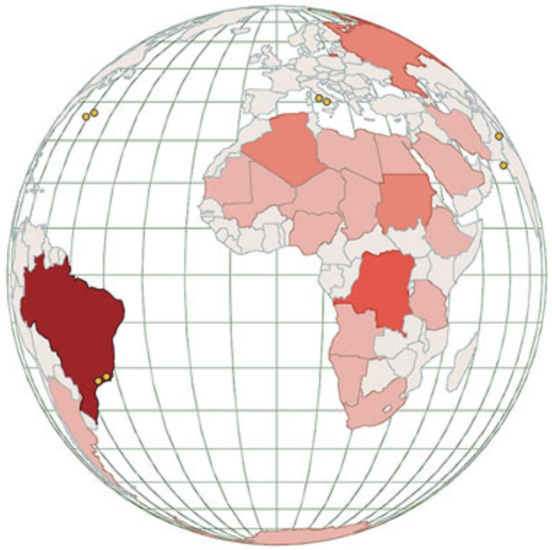
Advanced mapping

Creating a simple globe

```
projection = d3.geoOrthographic()
	.scale(200)
	.translate([250, 250])
	.center([0,0]);
```

Advanced mapping

An orthographic projection makes our map look like a globe. Notice that even though the paths for countries are drawn over each other, they're still drawn above the graticules. Also notice that although zooming in and out works, panning doesn't spin the globe but instead moves it around the canvas. The coloration of our countries is once again based on the graphical size of the country.



Advanced mapping

A draggable globe in D3

```
var zoomSettings = d3.zoomIdentity
    .translate(0, 0)                                        1
    .scale(200)

var rotateScale = d3.scaleLinear()                                  2
  .domain([-500, 0, 500])
  .range([-180, 0, 180]);

d3.select("svg")
  .call(mapZoom)
  .call(mapZoom.transform, zoomSettings)

function zoomed() {
  var e = d3.event
  var currentRotate = rotateScale(e.transform.x) % 360                3

  projection
    .rotate([currentRotate, 0])                                 4
    .scale(e.transform.k)
```

1 Orthographic should start out untranslated
2 This scale will be used to transform for x-zoom to degrees
3 Even though projection.rotate can deal with rotations greater than 360 degrees and less than –360 degrees, we need to filter the points so this will help
4 Rotate the projection only on the x-axis

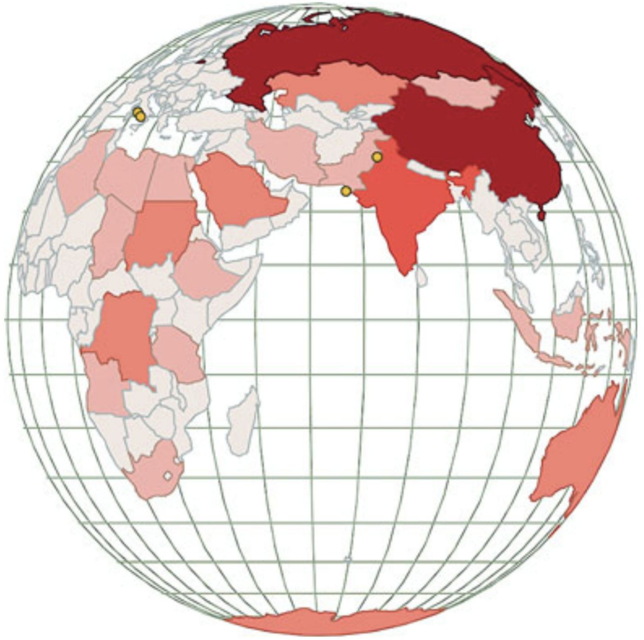
Advanced mapping

```
  d3.selectAll("path.graticule").attr("d", geoPath);
  d3.selectAll("path.countries").attr("d", geoPath)                    5
     .style("fill", d => countryColor(geoPath.area(d)))                 6
     .style("stroke", d => d3.rgb(countryColor(geoPath.area(d))).darker())

  d3.selectAll("circle.cities")
     .each(function (d, i) {
      var projectedPoint = projection([d.x,d.y])
      var x = parseInt(d.x)
      var display = x + currentRotate < 90 && x + currentRotate > -90
|| (x + currentRotate < -270 && x + currentRotate > -450)
|| (x + currentRotate > 270 && x + currentRotate < 450)
? "block" : "none"                                               7
      d3.select(this)
        .attr("cx", projectedPoint[0])
        .attr("cy", projectedPoint[1])
        .style("display", display)
    })
  }
```

5 Paths will be automatically clipped
6 Let’s color the countries based on displayed size
7 For points, we need to hide the points that aren’t in the current view

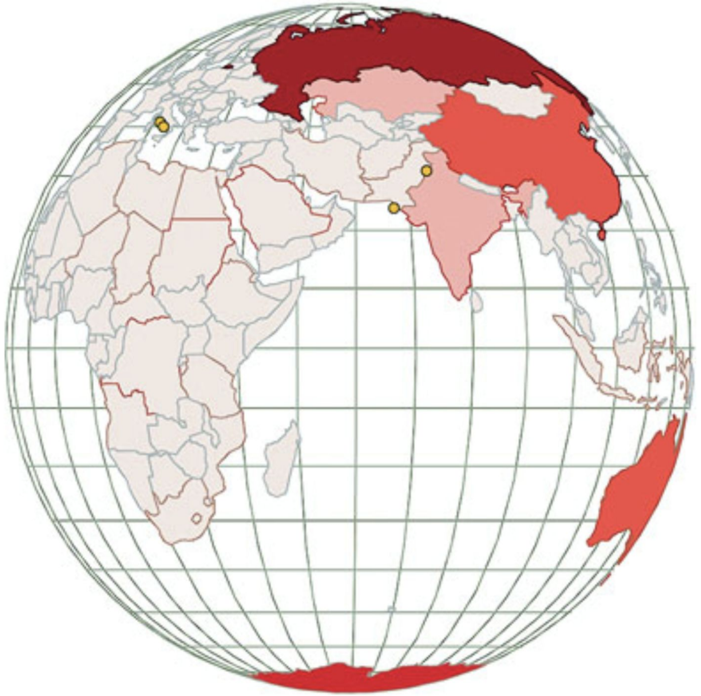
Advanced mapping

A draggable globe that clips the cities based on whether they should be in view and recolors the countries based on their displayed size



Advanced mapping

Our globe with countries colored by their geographic area, rather than their graphical area



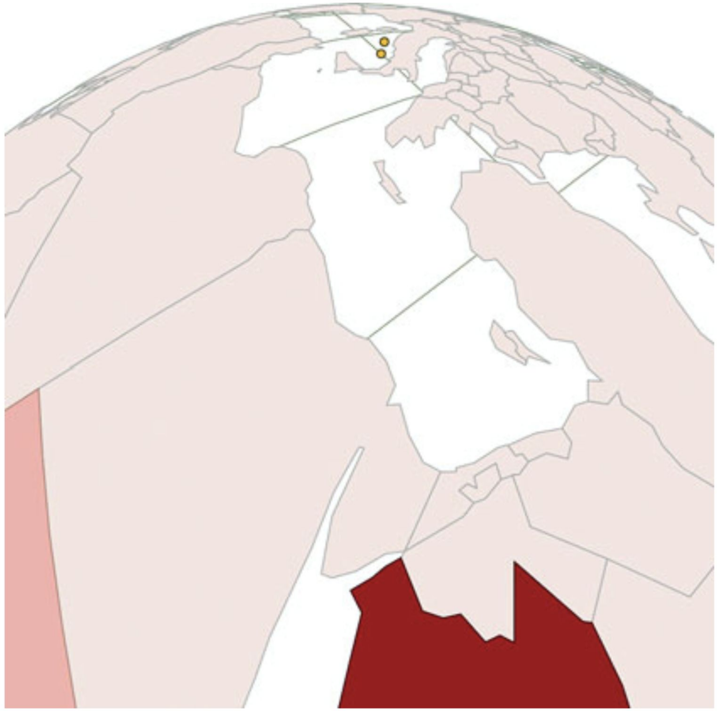
Advanced mapping

Coloring the area:

```
var featureData = d3.selectAll("path.countries").data();
var realFeatureSize =
d3.extent(featureData, d => d3.geoArea(d))
var newFeatureColor =
d3.scaleQuantize().domain(realFeatureSize).range(colorbrew
er.Reds[7]);
d3.selectAll("path.countries")
.style("fill", d => newFeatureColor(d3.geoArea(d)));
```

Advanced mapping

A satellite projection of data from the Middle East facing Europe



Advanced mapping

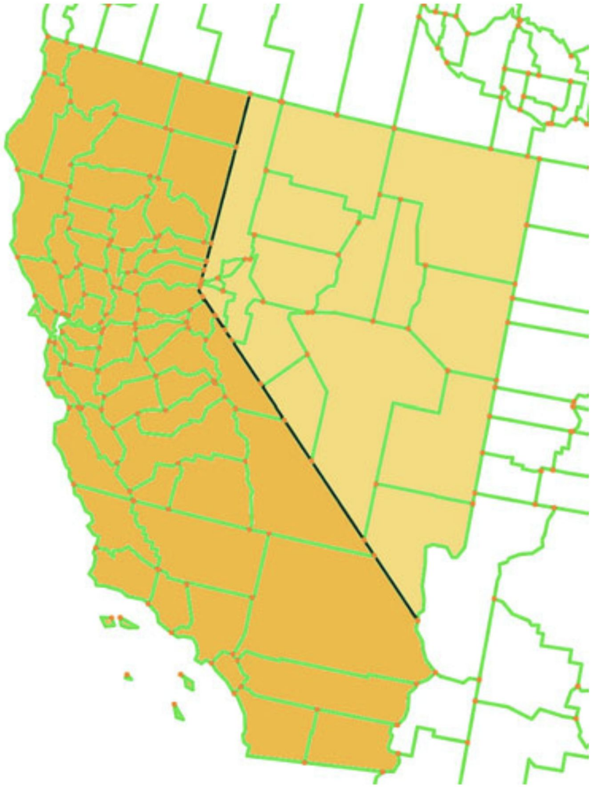
Satellite projection settings

```
projection = d3.geoSatellite()
        .scale(1330)
        .translate([250,250])
        .rotate([-30.24, -31, -56])
        .tilt(30)                                    1
        .distance(1.199)                                2
        .clipAngle(45);
```

1 The angle of the perspective on the geographic features
2 The distance of the surface from your perspective

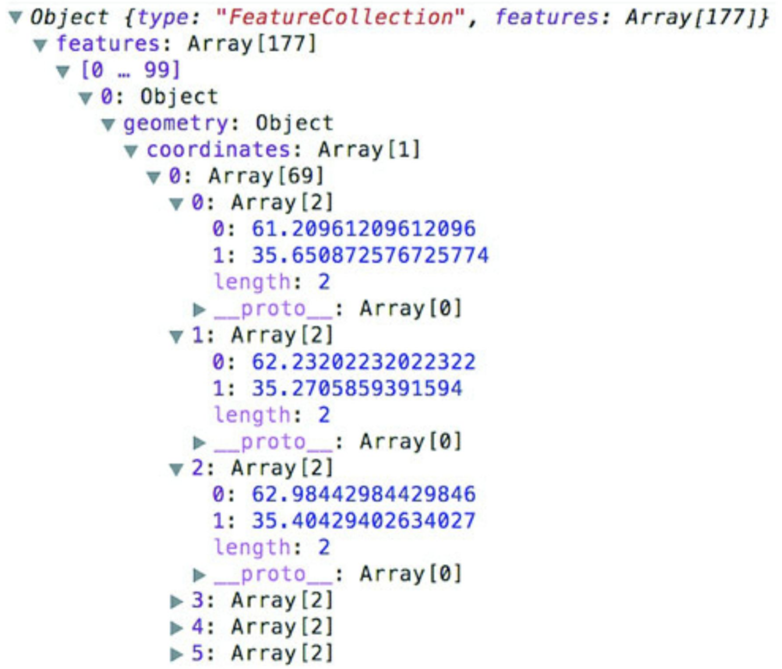
TopoJSON data and functionality

Arcs making up the counties of California and Nevada and neighboring states. You can see that the arcs are split whenever there is a possibility that they could be used in a different polygon. As a result, the 17 arcs making up the border of California and Nevada are used not only in the polygons making California and Nevada but also the polygons making their counties. Because the dataset knows the arcs are shared, it can easily derive neighbors.



TopoJSON data and functionality

TopoJSON data formatted using Topojson.feature(). The data is an array of objects, and it represents geometry as an array of coordinates like the features that come out of a GeoJSON file.



```
▼ Object {type: "FeatureCollection", features: Array[177]}
  ▼ features: Array[177]
    ▼ [0 … 99]
      ▼ 0: Object
        ▼ geometry: Object
          ▼ coordinates: Array[1]
            ▼ 0: Array[69]
              ▼ 0: Array[2]
                  0: 61.20961209612096
                  1: 35.650872576725774
                  length: 2
                ▶ __proto__: Array[0]
              ▼ 1: Array[2]
                  0: 62.23202232022322
                  1: 35.2705859391594
                  length: 2
                ▶ __proto__: Array[0]
              ▼ 2: Array[2]
                  0: 62.98442984429846
                  1: 35.40429402634027
                  length: 2
                ▶ __proto__: Array[0]
              ▶ 3: Array[2]
              ▶ 4: Array[2]
              ▶ 5: Array[2]
```

TopoJSON data and functionality

Loading TopoJSON

```
var PromiseWrapper = (xhr, d) => new Promise(resolve => xhr(d, p =>
   resolve(p)))
Promise.all([PromiseWrapper(d3.json, "world.topojson"),
   PromiseWrapper(d3.csv, "cities.csv")])
  .then(resolve => {
    createMap(resolve[0], resolve[1])
  })

function createMap(countries, cities) {
  var worldFeatures = topojson.feature(countries,
countries.objects.countries)                                   1
  console.log(worldFeatures)
}
```

1 Notice that our TopoJSON file has a property “objects”, which all TopoJSON files have, but “countries” is specific to this file and might be “rivers” or “land” or other property names in other files

TopoJSON data and functionality

Rendering and merging TopoJSON

```
function createMap(topoCountries) {
   var countries =
     topojson.feature(topoCountries, topoCountries.objects.countries)    1

   var projection = d3.geoMollweide()
     .scale(120)
     .translate([250, 250])
     .center([20,0])
   var geoPath = d3.geoPath().projection(projection);
   var featureSize =
     d3.extent(countries.features, d => geoPath.area(d))
   var countryColor = d3.scaleQuantize()
           .domain(featureSize).range(colorbrewer.Reds[7]);
   var graticule = d3.geoGraticule();
   d3.select("svg").append("path")
     .datum(graticule)
     .attr("class", "graticule line")
     .attr("d", geoPath)
   d3.select("svg").append("path")
     .datum(graticule.outline)
     .attr("class", "graticule outline")
     .attr("d", geoPath)
```

1 After processed by Topojson.features, we use exactly the same methods to render the features

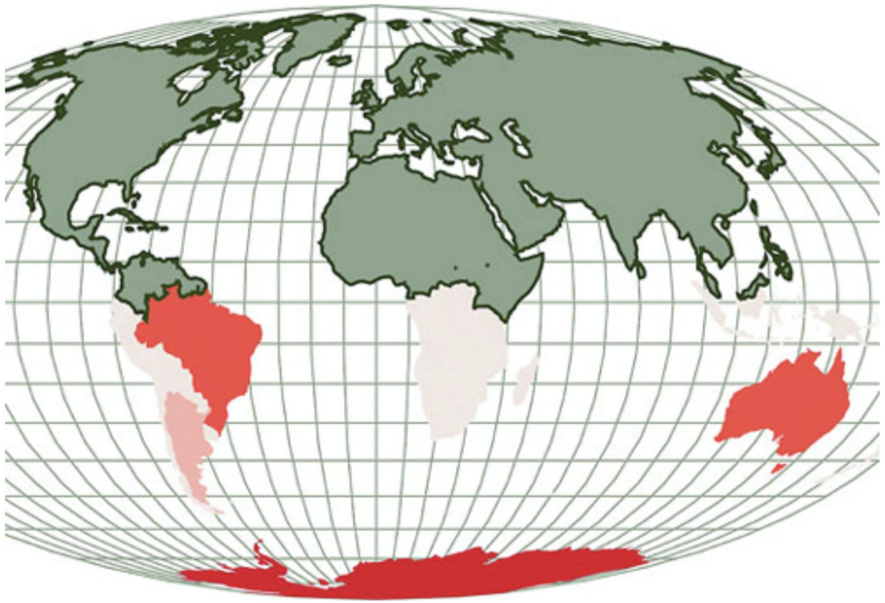
TopoJSON data and functionality

```
    d3.select("svg").selectAll("path.countries")
      .data(countries.features)
      .enter()
      .append("path")
      .attr("d", geoPath)
      .attr("class", "countries")
      .style("fill", d => countryColor(geoPath.area(d)))
      .style("stroke", "none")
   mergeAt(0)
   function mergeAt(mergePoint) {                                   2
     var filteredCountries = topoCountries.objects.countries.geometries   3
         .filter(d => {
           var thisCenter = d3.geoCentroid(
             topojson.feature(topoCountries, d) );                  4
           return thisCenter[1] > mergePoint? true : null;
          })                                                 5
      d3.select("svg").append("g")
       .datum(topojson.merge(topoCountries, filteredCountries))          6
       .insert("path")
       .attr("class", "merged")
       .attr("d", geoPath)
   }
}
```

2 Our merge function
3 We’re working with the TopoJSON dataset
4 To use geoCentroid, we convert each feature into GeoJSON
5 Results in an array of only the corresponding geometries
6 Uses datum because merge returns a single multipolygon

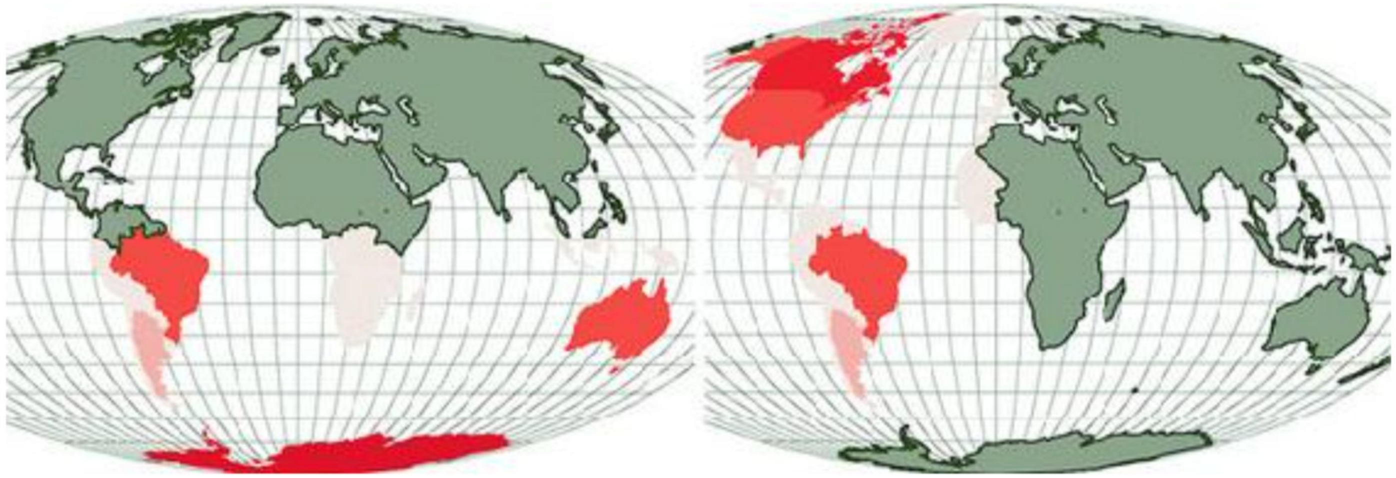
TopoJSON data and functionality

The results of merging based on the centroid of a feature. The feature in gray is a single merged feature made up of many separate polygons.



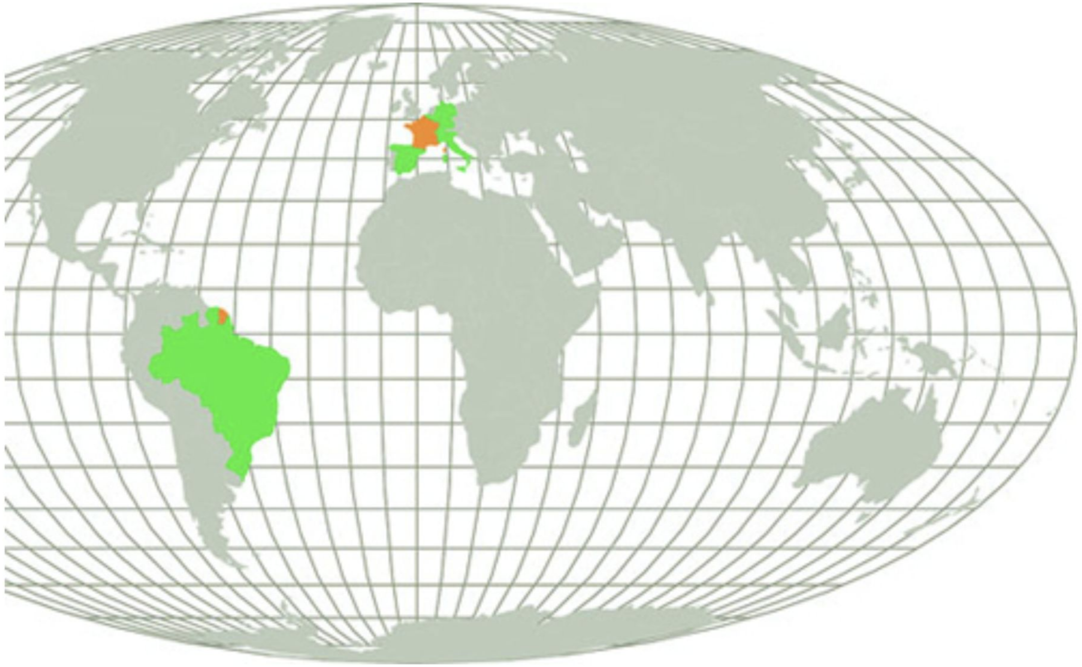
TopoJSON data and functionality

By adjusting the merge settings, we can create something like northern and eastern hemispheres as merged features. Notice that because this is based on a centroid, we can see at the bottom a piece of Eastern Russia as part of our merged feature, along with Antarctica.



TopoJSON data and functionality

Hover behavior displaying the neighbors of France using TopoJSON's neighbor function. Because Guyana is an overseas department of France, France is considered to be neighbors with Brazil and Suriname. This is because France is represented as a multipolygon in the data, and any neighbors with any of its shapes are returned as neighbors.



TopoJSON data and functionality

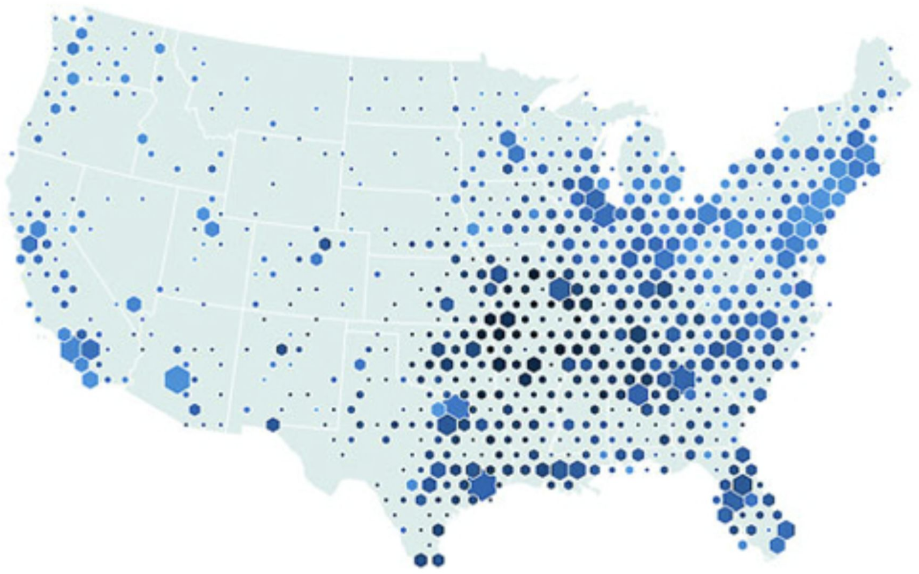
Calculating neighbors and interactive highlighting

```
var neighbors =
  topojson.neighbors(topoCountries.objects.countries.geometries)       1
d3.selectAll("path.countries")
  .on("mouseover", findNeighbors)
  .on("mouseout", clearNeighbors)
function findNeighbors (d,i) {
  d3.select(this).style("fill", "#FE9922")                             2
  d3.selectAll("path.countries")
    .filter((p,q) => neighbors[i].includes(q))                         3
    .style("fill", "#41A368")
}
function clearNeighbors () {
  d3.selectAll("path.countries").style("fill", "#C4B9AC")              4
}
```

1 Creates an array indicating neighbors by their array position
2 Colors the country you hover over orange
3 Colors all neighbors green
4 Colors all countries gray to “clear” results

Further reading for web mapping

An example of hexbinning by Mike Bostock showing the locations of Walmart stores in the United States (available at http://bl.ocks.org/mbostock/4330486)



Further reading for web mapping

An example of a Voronoi diagram used to split the United States into polygons based on the closest state capital (available at www.jasondavies.com/maps/voronoi/us-capitals/).