

# Mapping with d3.js

Another string to the bow of d3.js is the addition of a set of powerful routines for handling geographical information.

In the same sense that a line graph is a simple representation of data on a document, a map can be regarded as a set of points with an underlying coordinate system. When you say it like that it seems obvious that it should be applied as a document for display. However, I don't want to give the impression that this is some sort of trivial matter for either the original developers or for you, the person who wants to display a map. Behind the scenes for this type of work, the thought that must have gone into making the code usable and extensible must have been enormous.

Mike Bostock has lauded the work of Jason Davies in the development of the latest major version of d3.js (version 3) for his work on improving mapping capability. A visit to his [home page](#) provides a glimpse into Jason's expertise and no visit would be complete without marvelling at his work with [geographic projections](#).

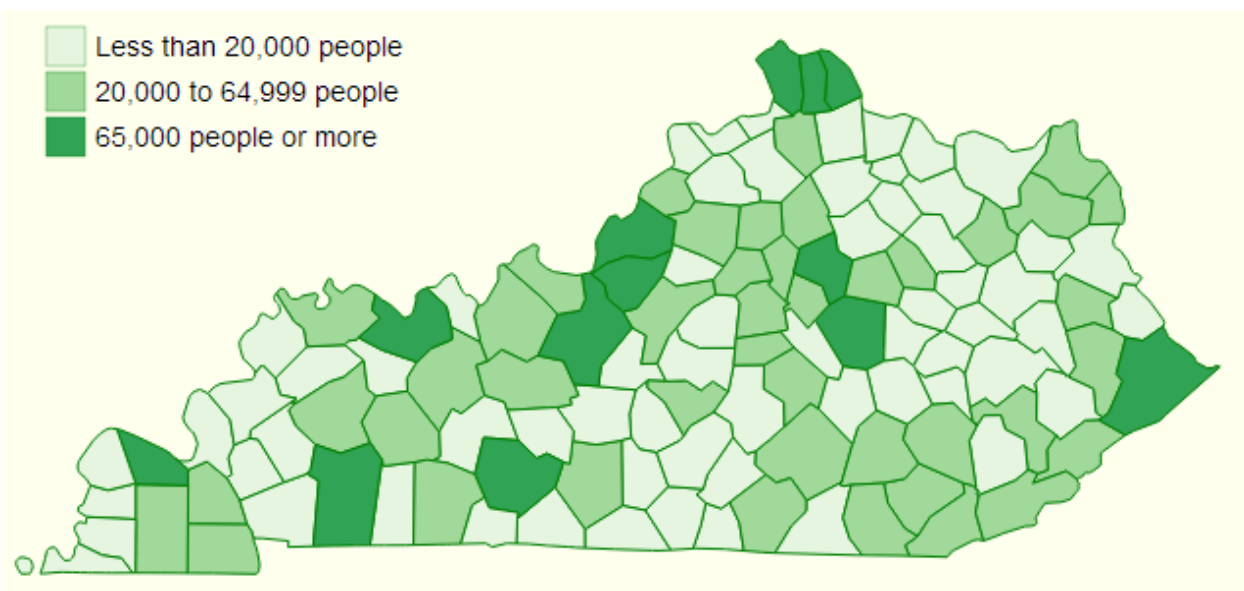
## Examples

I am firmly of the belief that mapping in particular has an enormous potential for adding value to data sets. The following collection of examples gives a brief taste of what has been accomplished by combining geographic information and D3 thus far. (The screen shots following have been sourced from the [biovisualize gallery](#) and as such provide attribution to the best of my ability. If I have incorrectly attributed the source or author please let me know and I will correct it promptly.)



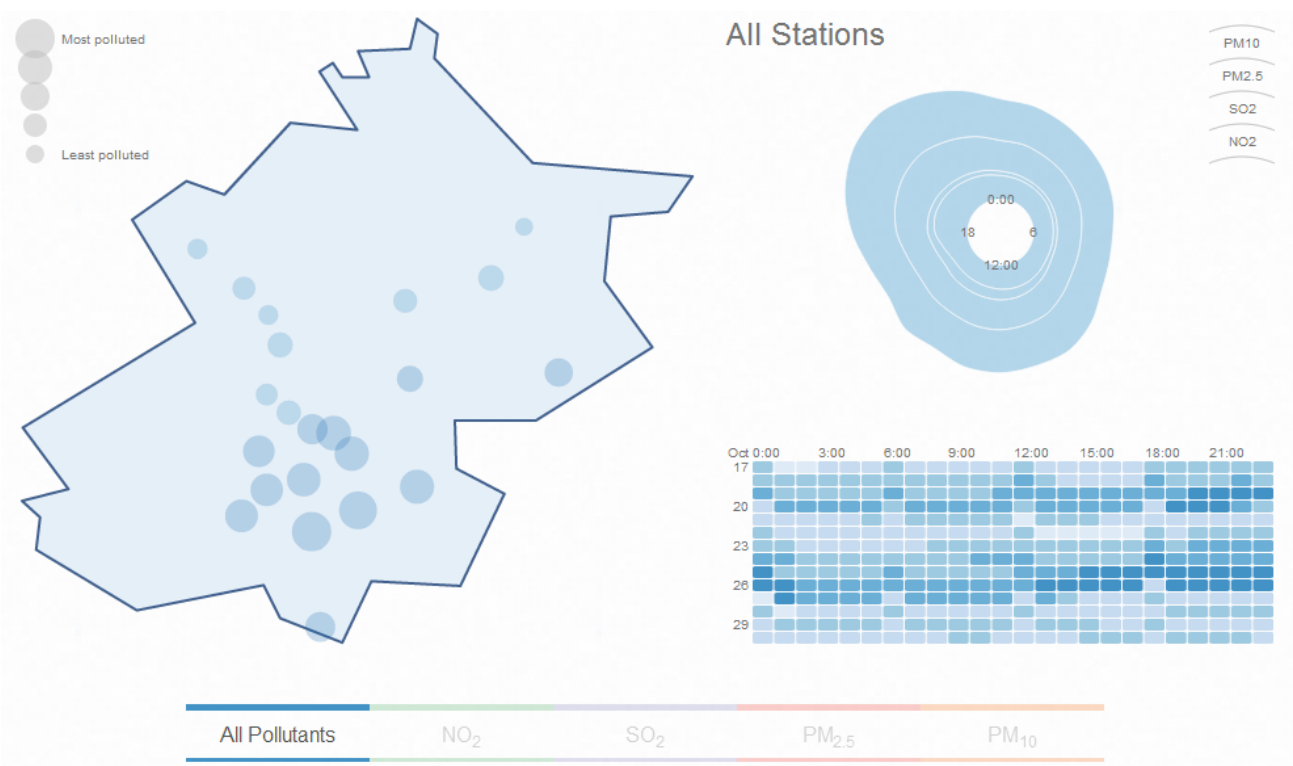
Faux D3 3d globe integrated with Mapbox / Open Street Map

Above is an interactive visualization showing the position of the main map on a faux D3 3d globe with a Mapbox / Open Street Map main window. Source [dev.geosprocket.com](#) Source Bill Morris.



Kentucky Count Population from the 2010 census

This is a breakdown of population in Kentucky Counties from the 2010 census. Source: [ccarpenterg.github.com](https://github.com/ccarpenterg) by Cristian Carpenter.



Beijing air pollution

This map visualizes air pollution in Beijing. Source: [scottcheng.github.com](https://github.com/scottcheng) by Scott Cheng.



#### Shuttle Radar Topography Mission tile downloading

This is a section of the globe that is presented on the Shuttle Radar Topography Mission tile downloading web site. This excellent site uses the interactive globe to make the selection of SRTM tiles easy. Source [dwtkns.com](http://dwtkns.com) by Derek Watkins.



#### Animated World tour

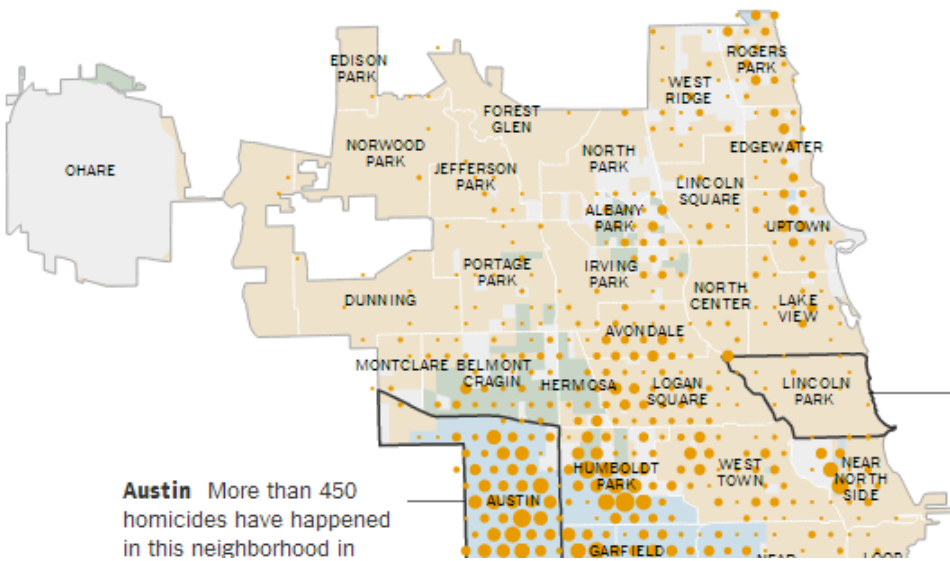
This is a static screen-shot of an animated tour of the Worlds countries. Source [bl.ocks.org](http://bl.ocks.org) by Mike Bostock.

# A Chicago Divided by Killings

Related Article »

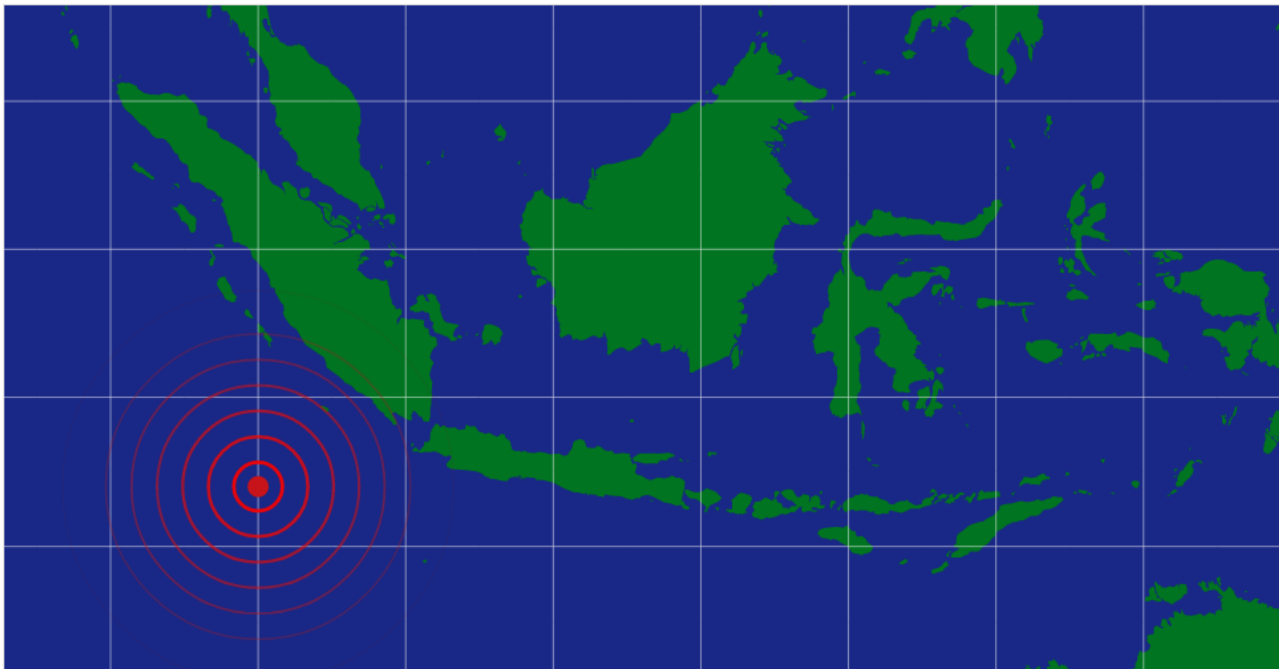
A New York Times analysis of homicides and census data in Chicago compared areas near murders to those that were not. Residents living near homicides in the last 12 years were much more likely to be black, earn less money and lack a college degree.

	NEAR HOMICIDES	NOT NEAR HOMICIDES
Population	1.3 mil.	1.4 mil.
Income	\$38,318	\$61,175



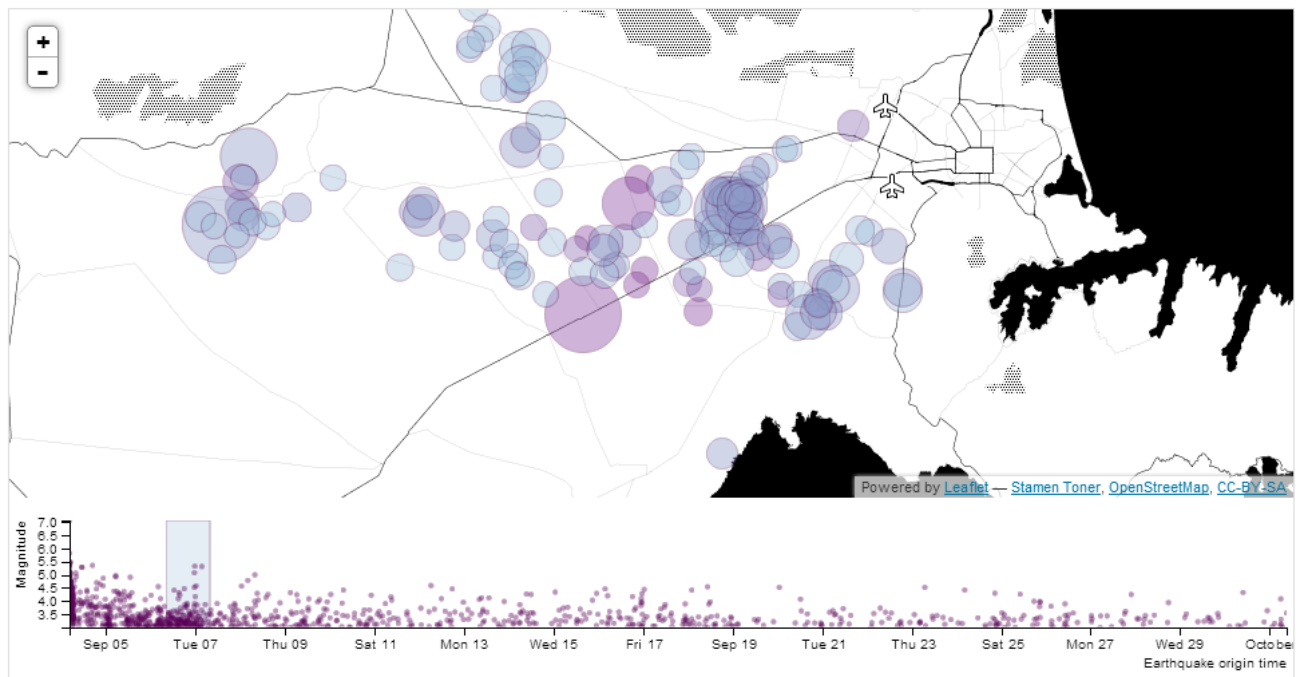
A Chicago Divided by Killings: New Your Times

This is one of the great infographics published by the [New York Times](http://www.nytimes.com). Source: [www.nytimes.com](http://www.nytimes.com) by Mike Bostock, Shan Carter and Kevin Quealy.



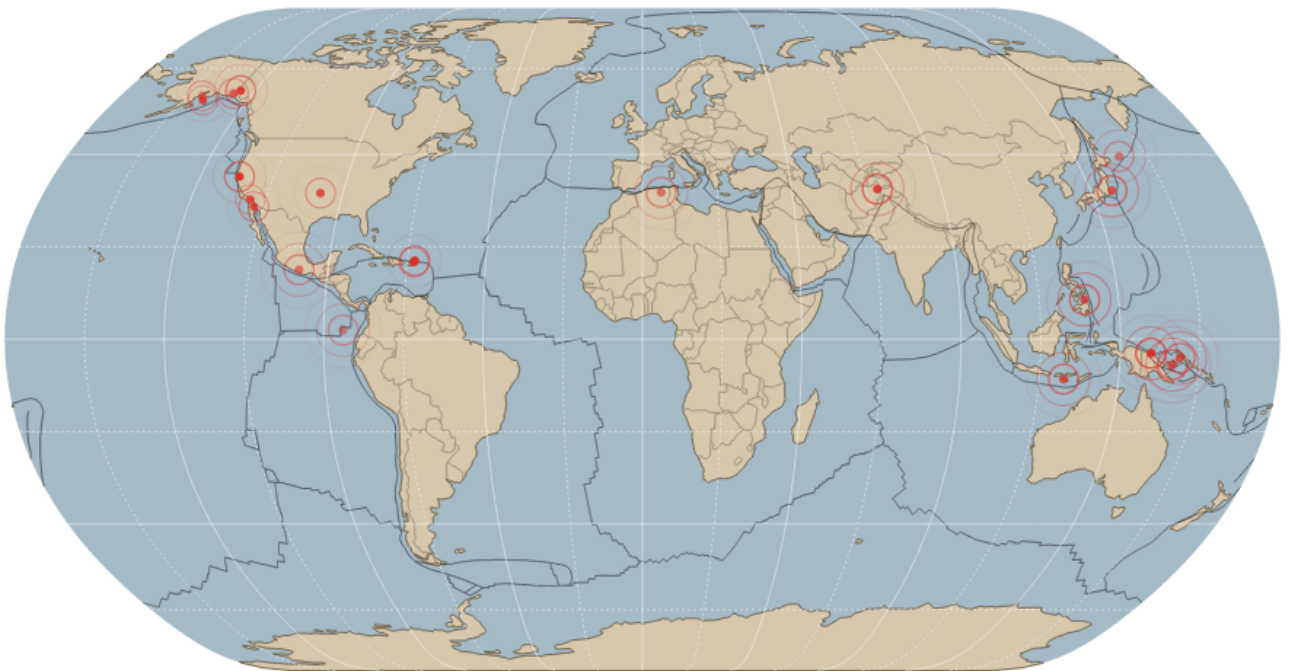
Concentric circles emanating from glowing red dot

This is an animated graphic showing a series of concentric circles emanating from glowing red dot which was styled after a [news article in The Onion](http://www.theonion.com). Source: [blocks.org](http://blocks.org) by Mike Bostock.



### Christchurch earthquakes timeline

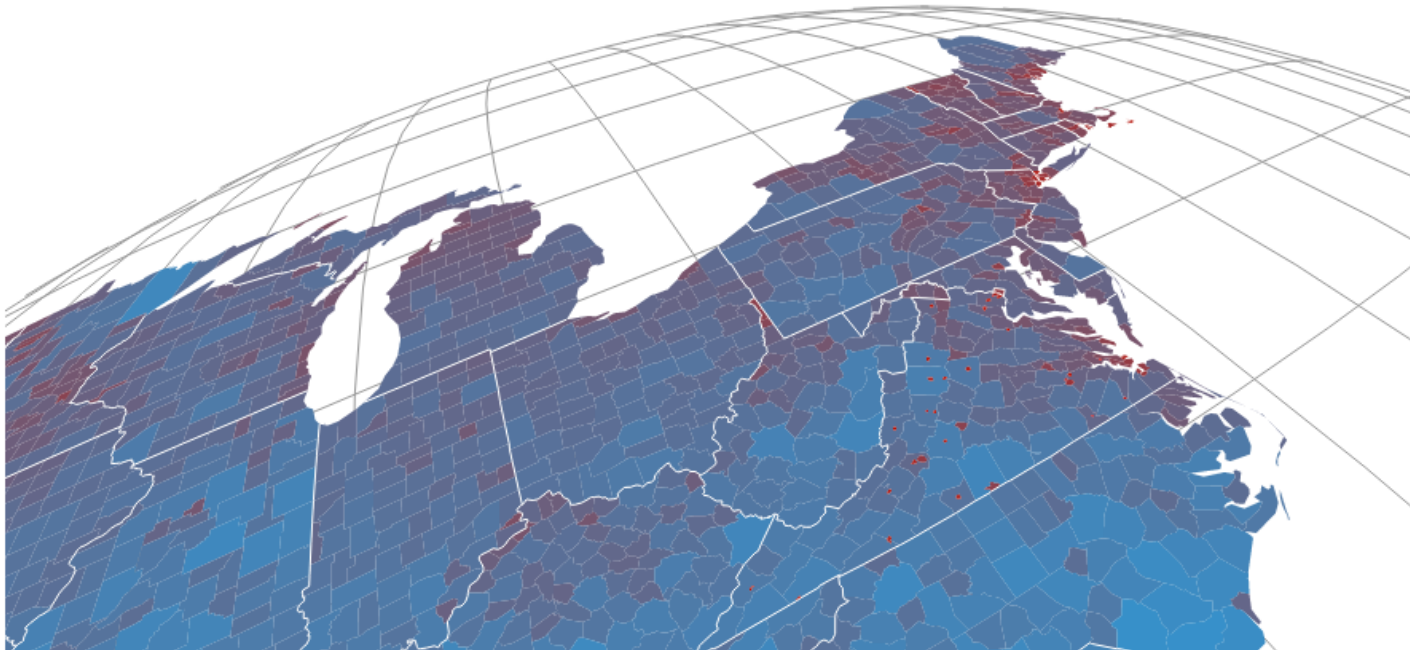
Here we see earthquakes represented on a selectable timeline where D3 generates a svg overlay and the map layer is created using Leaflet. Source: [bl.ocks.org](http://bl.ocks.org) by tnightingale.



### Earthquakes in the past 24 hours

Carrying on with the earthquake theme, this is a map of all earthquakes in the past 24 hours over magnitude 2.5. Source: [bl.ocks.org](http://bl.ocks.org) by benelsen.





Satellite projection

An interactive satellite projection. Source [dev.geosprocket.com](http://dev.geosprocket.com) by Bill Morris.

## GeoJSON and TopoJSON

Projecting countries and various geographic features onto a map can be a very data hungry exercise. By that I mean that the information required to present geographic shapes can result in data files that are quite large. GeoJSON has been the default geographic data file of choice for quite some time, and as the name would suggest it encodes the data in a JSON type hierarchy. Often these GeoJSON files include a significant amount of extraneous detail or incorporate a level of accuracy that is impractical (too detailed).

Enter TopoJSON. Mike Bostock has designed TopoJSON as an extension to GeoJSON in the sense that it has a similar structure, but the geometries are not encoded discretely and where they share features, they are combined. Additionally TopoJSON encodes numeric values more efficiently and can incorporate a degree of simplification. This simplification can result in savings of file size of 80% or more depending on the area and use of compression. Although TopoJSON has only begun to be used, the advantages of it seem clear and so I will anticipate its future use by incorporating it in my example diagrams (not that the use of GeoJSON differs much if at all). A great description of TopoJSON can be found on the TopoJSON wiki on [github](https://github.com).

## Starting with a simple map

Our starting example will demonstrate the simple display of a World map. Our final result will look like this;



## The World

The data file for the World map is one produced by Mike Bostock's as part of his TopoJSON work.

We'll move through the explanation of the code in a similar process to the one we went through when highlighting the function of the Sankey diagram. Where there are areas that we have covered before, I will gloss over some details on the understanding that you will have already seen them explained in an earlier section (most likely the basic line graph example).

Here is the full code;

```
<!DOCTYPE html> <meta charset="utf-8"> <style>
path { stroke: white; stroke-width: 0.25px; fill: grey; }
</style>
<body>
<script type="text/javascript" src="d3/d3.v3.js">
</script>
<script src="js/topojson.v0.min.js"></script>
<script>
var width = 960, height = 500;
var projection = d3.geo.mercator().center([0, 5]).scale(900)
.rotate([-180,0]);
var svg = d3.select("body").append("svg").attr("width", width)
.attr("height", height);
var path = d3.geo.path().projection(projection);
var g = svg.append("g"); // load and display the World
d3.json("json/world-110m2.json", function(error, topology) {
g.selectAll("path").data(topojson.object(topology,
topology.objects.countries).geometries).enter()
.append("path").attr("d", path) });
</script> </body> </html>
```

One of the first things that struck me when I first saw the code to draw a map was how small it was (the amount of code, not the World). It's a measure of the degree of abstraction that D3 is able to provide to the process of getting data from a

raw format to the screen that such a complicated task can be condensed to such an apparently small amount of code. Of course that doesn't tell the whole story. Like a duck on a lake, above the water all is serene and calm while below the water the feet are paddling like fury. In this case, our code looks serene because D3 is doing all the hard work :-).

The first block of our code is the start of the file and sets up our HTML.

```
<!DOCTYPE html> <meta charset="utf-8"> <style>
```

This leads into our style declarations.

```
path {    stroke: white;    stroke-width: 0.25px;    fill: grey; }
```

We only state the properties of the path components which will make up our countries. Obviously we will fill them with grey and have a thin (0.25px) line around each one.

The next block of code loads the JavaScript files.

```
</style> <body> <script type="text/javascript" src="d3/d3.v3.js"></script>  
<script src="js/topojson.v0.min.js"></script> <script>
```

In this case it's d3 and topojson. We load `topojson.v0.min.js` as a separate file because it's still fairly new. In other words it hasn't been incorporated into the main d3.js code base (that's an assumption on my part since it might exist in isolation or perhaps end up as a plug-in). Whatever the case, for the time being, it exists as a separate file.

Then we get into the JavaScript. The first thing we do is define the size of our map.

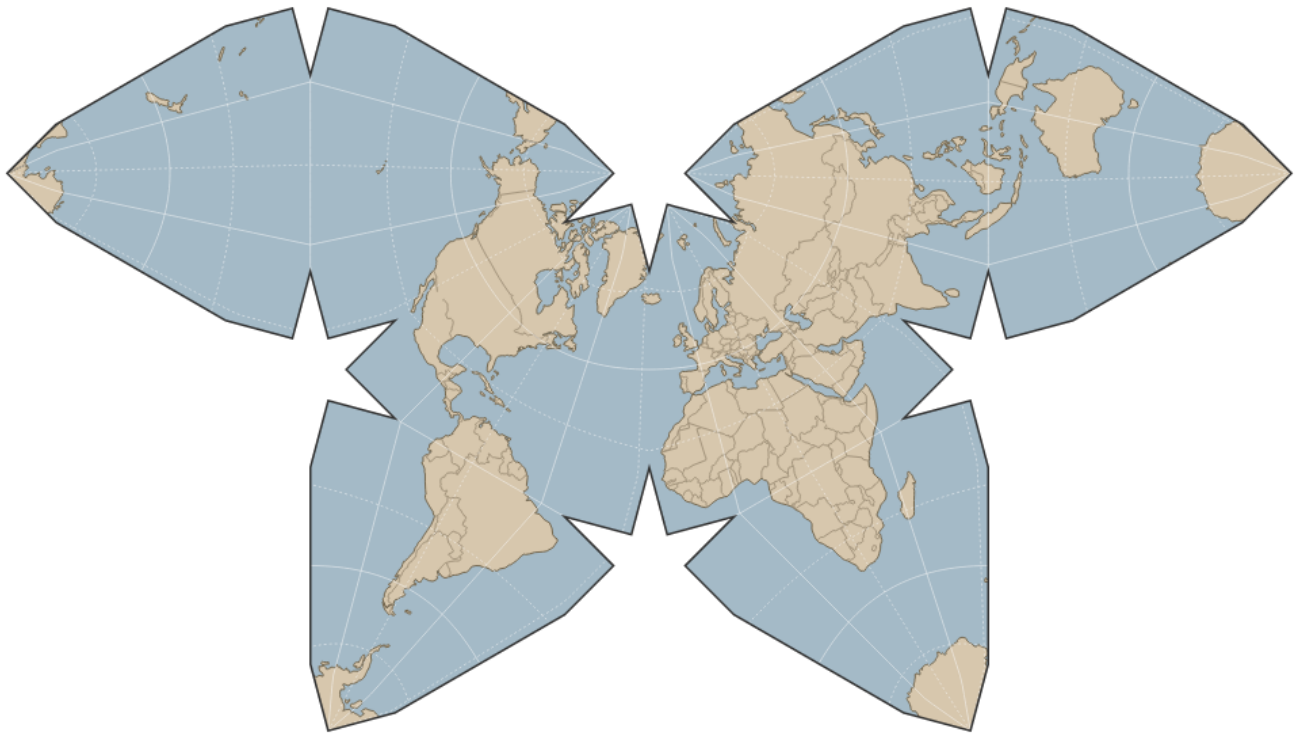
```
var width = 960,    height = 500;
```

Then we get into one of the simple, but cool parts of making any map. Setting up the view.

```
var projection = d3.geo.mercator()    .center([0, 5 ])    .scale(900)  
    .rotate([-180,0]);
```

The projection is the way that the geographic coordinate system is adjusted for display on our flat screen. The screen is after all a two dimensional space and we are trying to present a three dimensional object. This is a big deal to cartographers in the sense that selecting a geographic projection for a map is an exercise in compromise. You can make it look pretty, but in doing so you can grievously distort the land size / shape. On the other hand you might make it more accurate, in size / shape but people will have trouble recognising it because they're so used to the standard Mercator projection. For example, the awesome [Waterman Butterfly](#).





### The Waterman Butterfly

There are a lot of alternatives available. Please have a browse on the [wiki](#) where you will find a huge range of options (66 at time of writing).

In our case we've gone with the conservative Mercator option.

Then we define three aspects of the projection. Center, scale and rotate.

#### **center**

If center is specified, this sets the projection's center to the specified location as a two-element array of longitude and latitude in degrees and returns the projection. If center is not specified the default of (0°,0°) is used.

Our example is using [ 0 , 5 ] which I have selected as being in the middle (I use 0) for longitude (left to right) and 5 degrees North of the equator (for latitude, North is positive, South is negative). This was purely to make it look aesthetically pleasing. Here's the result of setting the center to [ 100 , 30 ].



Center set to [ 100 , 30 ]

The map has been centered on 100 degrees West and 30 degrees North. Of course, it's also been pushed to the left without the right hand side of the map scrolling around. We'll get to that in a moment.

### **scale**

If scale is specified, this sets the projection's scale factor to the specified value. If scale is not specified, it returns the current scale factor which defaults to 150. It's important to note that scale factors are not consistent across projections.

Our current map uses a scale of 900. Again, this has been set for aesthetics. Keeping our center of [ 100 , 30 ], if we increase our scale to 2000 this is the result.



Scale set to 2000

### **rotate**

If rotation is specified, this sets the projection's three-axis rotation to the specified angles for yaw, pitch and roll (equivalently longitude, latitude and roll) in degrees and returns the projection. If rotation is not specified, it sets the values to  $[0, 0, 0]$ . If the specified rotation has only two values, rather than three, the roll is assumed to be  $0^\circ$ .

In our map we have specified  $[-180, 0]$  so we can assume a roll value of zero. Likewise we have rotated our map by -180 degrees in longitude. This has been done specifically to place the map with the center on the anti-meridian (The international date line in the middle of the Pacific Ocean). If we return the value to  $[0, 0]$  (with our original values of `scale` and `center` this is the result.



Rotate set to `[ 0, 0 ]`

In this case the centre of the map lines up with the meridian.

The next block of code sets our svg window;

```
var svg = d3.select("body").append("svg")      .attr("width", width)
          .attr("height", height);
```

The following portion of code creates a new geographic path generator;

```
var path = d3.geo.path()      .projection(projection);
```

The path generator (`d3.geo.path()`) is used to specify a projection type (`.projection`) which was defined earlier as a Mercator projection via the variable `projection`. (I'm not entirely sure, but it is possible that I have just set some kind of record for use of the word 'projection' in a sentence.)

We then declare `g` as our appended svg.

```
var g = svg.append("g");
```

The last block of JavaScript draws our map.

```
d3.json("json/world-110m2.json", function(error, topology) {
g.selectAll("path")      .data(topojson.object(topology,
topology.objects.countries)      .geometries)      .enter()
.append("path")      .attr("d", path) });
```

We load the TopoJSON file with the coordinates for our World map (`world-110m2.json`). Then we declare that we are going to act on all the path elements in the graphic (`g.selectAll("path")`).

Then we pull the data that defines the countries from the TopoJSON file (`.data(topojson.object(topology, topology.objects.countries).geometries)`). We add it to the data that we're going to display (`.enter()`) and then we append that data as path elements (`.append("path")`).

The last html block closes off our tags and we have a map!



The World map centered on the Pacific

The code and data for this example can be found as [World Map Centered on the Pacific](#) on blocks.org.

## Zooming and panning a map

With our map displayed nicely we need to be able to move it about to explore it fully . To do this we can provide the functionality to zoom and pan it using the mouse.

Towards the end of the script, just before the close off of the script at the `</script>` tag we can add in the following code;

```
var zoom = d3.behavior.zoom()
.on("zoom",function() {g.attr("transform","translate("+
d3.event.translate.join(",")+")scale("+d3.event.scale+")");
g.selectAll("path")
.attr("d", path.projection(projection)); });  svg.call(zoom)
```

This block of code introduces the `behaviors` functions. Using the `d3.behavior.zoom` function creates event listeners (which are like hidden functions standing by to look out for a specific type of activity on the computer and in this case mouse actions) to handle zooming and panning gestures on a container element (in this case our map). More information on the range of zoom options is available on the [D3 Wiki](#).

We begin by declaring the `zoom` function as `d3.behavior.zoom`.

Then we instruct the computer that when it ‘sees’ a ‘zoom’ event to carry out another function (`.on("zoom", function() {})`).

That function firstly gathers the (correctly formatted) `translate` and `scale` attributes in...

```
g.attr("transform", "translate("+  
d3.event.translate.join(",")+")scale("+d3.event.scale+")");
```

... and then applies them to all the path elements (which are the shapes of the countries) via...

```
g.selectAll("path").attr("d", path.projection(projection));
```

Lastly we call the zoom function.

```
svg.call(zoom)
```

Then we relax and explore our map!



The World map with zoom and pan

The code and data for this example can be found as [World Map with zoom and pan](#) on bl.ocks.org.

## Displaying points on a map

Displaying maps and exploring them is pretty entertaining, but as anyone who has participated in the improvement of our geographic understanding of our world via projects such as [Open Street Map](#) will tell you, there’s a whole new level of cool to be attained by adding to a map.

With that in mind, our next task is to add some simple detail in the form of points that show the location of cities.

To do this we will load in a csv file with data that identifies our cities and includes latitude and longitude details. Our file is called `cities.csv` and looks like this;



```

code,city,country,lat,lon
ZNZ,ZANZIBAR,TANZANIA,-6.13,39.31
TYO,TOKYO,JAPAN,35.68,139.76
AKL,AUCKLAND,NEW ZEALAND,-36.85,174.78
BKK,BANGKOK,THAILAND,13.75,100.48
DEL,DELHI,INDIA,29.01,77.38
SIN,SINGAPORE,SINGAPOR,1.36,103.75
BSB,BRASILIA,BRAZIL,-15.67,-47.43
RIO,RIO DE JANEIRO,BRAZIL,-22.90,-43.24
YTO,TORONTO,CANADA,43.64,-79.40
IPC,EASTER ISLAND,CHILE,-27.11,-109.36
SEA,SEATTLE,USA,47.61,-122.33

```

While we're only going to use the latitude and longitude for our current work, the additional details could just as easily be used for labeling or tooltips.

We need to place our code carefully in this case because while you might have some flexibility in getting the right result with a locally hosted version of a map, there is a possibility that with a version hosted in the outside World (*gasp* the internet) you could strike trouble.

The code to load the cities should be placed inside the function that is loading the World map as indicated below;

```

d3.json("json/world-110m2.json", function(error, topology) {
g.selectAll("path")
.data(topojson.object(topology, topology.objects.countries)
.geometries) .enter() .append("path") .attr("d", path)
// <== Put the new code block here });

```

Here's the new code;

```

d3.csv("data/cities.csv", function(error, data) {g.selectAll("circle")
.data(data) .enter().append("circle").attr("cx", function(d) {
return projection([d.lon, d.lat])[0]; })
.attr("cy", function(d) {return projection([d.lon, d.lat])[1]; })
.attr("r", 5) .style("fill", "red");

```

We'll go through the code and then explain the quirky thing about it.

First of all we load the `cities.csv` file (`d3.csv("data/cities.csv", function(error, data) {})`). Then we select all the circle elements (`g.selectAll("circle")`), assign our data (`.data(data)`), enter our data (`.enter()`) and then add in circles (`.append("circle")`).

Then we set the x and y position for the circles based on the longitude (`([d.lon, d.lat])[0]`) and latitude (`([d.lon, d.lat])[1]`) information in the csv file.

Finally we assign a radius of 5 pixels and fill the circles with red.

The quirky thing about the new code block is that we have to put it inside the code block that loads the world data (`d3.json("json/world-110m2.json", function(error, topology) {})`). We could place the two blocks one after the others (load / draw the world data, then load / draw the circles). And this will probably work if you run the file from your local computer. But when you host the files on the internet, it takes too long to load the world data compared to the city data and the end result is that the city data gets drawn before the world data and this is the result.



The cities under the World

To avoid the problem we place the loading of the city data *into* the code that loads the World data. That way the city data doesn't get loaded until the World data is loaded and then the circles get drawn on top of the world instead of under it :-).



The cities on top of the World

The code and data for this example can be found as [World map with zoom / pan and cities](#) on bl.ocks.org.

Additionally the full code can be found in the appendix section at the rear of the book.