Create attractive web-based data visualizations using the amazing JavaScript library D3.js

D3.js By Example

Michael Heydt

What you will learn from this book

- Install and use D3.js to create HTML elements within a document
- Use development tools such as JSBIN and Chrome Developer Tools to create D3.js applications
- Retrieve JSON data and use D3.js selections and data binding to create visual elements from data
- Create and style graphical elements, such as circles, ellipses, rectangles, lines, paths, and text, using SVG
- Turn your data into bar and scatter charts and add margins, axes, labels, and legends
- Use D3.js generators to perform the magic of creating complex visualizations from data
- Add interactivity to your visualizations, including tooltips, sorting, hover-to-highlight, and the grouping and dragging of visuals

Who this book is written for

Whether you are new to data and data visualization, a seasoned data scientist, or a computer graphics specialist, this book will provide you with the skills you need to create web-based and interactive data visualizations. This book assumes some knowledge of coding and, in particular, experience coding in JavaScript.


Michael Heydt

$ 44.99 US
£ 28.99 UK

Prices do not include local sales tax or VAT

D3.js By Example

Community Experience Distilled

Create attractive web-based data visualizations using the amazing JavaScript library D3.js

Michael Heydt

D3.js By Example

What you will learn from this book

- Install and use D3.js to create HTML elements within a document
- Use development tools such as JSBIN and Chrome Developer Tools to create D3.js applications
- Retrieve JSON data and use D3.js selections and data binding to create visual elements from data
- Create and style graphical elements, such as circles, ellipses, rectangles, lines, paths, and text, using SVG
- Turn your data into bar and scatter charts and add margins, axes, labels, and legends
- Use D3.js generators to perform the magic of creating complex visualizations from data
- Add interactivity to your visualizations, including tooltips, sorting, hover-to-highlight, and the grouping and dragging of visuals

Who this book is written for

Whether you are new to data and data visualization, a seasoned data scientist, or a computer graphics specialist, this book will provide you with the skills you need to create web-based and interactive data visualizations. This book assumes some knowledge of coding and, in particular, experience coding in JavaScript.


Michael Heydt
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 12 'Creating Maps with GeoJSON and TopoJSON'
- A synopsis of the book’s content
- More information on D3.js By Example
About the Author

Michael Heydt is an independent consultant, programmer, educator, and trainer. He has a passion for learning and sharing his knowledge of new technologies. Michael has worked in multiple industry verticals, including media, finance, energy, and healthcare. Over the last decade, he worked extensively with web, cloud, and mobile technologies and managed user experience, interface design, and data visualization for major consulting firms and their clients. Michael’s current company, Seamless Thingies (www.seamlessthingies.tech), focuses on IoT development and connecting everything with everything.

He is the author of numerous articles, papers, and books, such as *Instant Lucene. NET*, *Learning Pandas*, and *Mastering Pandas for Finance*, all by Packt Publishing, on technology. Michael is also a common speaker at .NET user groups and various mobile, cloud, and IoT conferences and delivers webinars on advanced technologies. He can be reached through his website e-mails, mike@heydt.org and mike@seamlessthingies.tech and on Twitter at @mikeheydt.
Learning D3.js on your own can be a daunting task. There are literally thousands of examples online with differing degrees of effective, or ineffective, explanation.

This book uses examples that take you right from the beginning, with the basic concepts of D3.js, using practical examples that progressively build on each other both within a specific chapter and also with reference to previous chapters.

We will focus on the examples created for this book as well as those found online that are excellent but could use some additional explanation. Each example will explain how the example works either line by line or by comparison with other examples and concepts learned earlier in the book.

What this book covers

Chapter 1, Getting Started with D3.js, introduces you to D3.js and building a simple application using several tools to help with its creation.

Chapter 2, Selections and Data Binding, teaches you how to use D3.js selections to create DOM elements based on data.

Chapter 3, Creating Visuals with SVG, introduces you to Scalable Vector Graphics and how to use them to render various shapes that are commonly used in D3.js visualizations.

Chapter 4, Creating a Bar Graph, demonstrates how to create a bar graph from given data.

Chapter 5, Using Data and Scales, shows you how to load data from external sources in different formats and convert it into information suitable for visualization.
Chapter 6, Creating Scatter and Bubble Plots, demonstrates how to load, scale, and plot multidimensional data in a manner that makes patterns clear to users.

Chapter 7, Creating Animated Visuals, teaches you to use animations in your D3.js applications to demonstrate how data changes over time.

Chapter 8, Adding User Interactivity, shows you how to allow users to interact with your visualizations using the mouse.

Chapter 9, Complex Shapes Using Paths, shows you how to use many of the built-in tools in D3.js to automatically generate complex paths with a few simple statements.

Chapter 10, Using Layouts to Visualize Series and Hierarchical Data, focuses on creating complex graphs that utilize the layout objects of D3.js. This includes a multitude of graphs in different categories, including stacked, packed, clustered, flow-based, hierarchical, and radial.

Chapter 11, Visualizing Information Networks, dives into demonstrating how you can use D3.js to visualize network data such as is found in social networks.

Chapter 12, Creating Maps with GeoJSON and TopoJSON, teaches you how to create maps and highlight regions on them using two forms of geographic data: Geo and TopoJSON.

Chapter 13, Combining D3.js and AngularJS, discusses how you can integrate multiple D3.js visualizations using Angular.js to create reactive visualizations.
Creating Maps with GeoJSON and TopoJSON

D3.js provides extensive capabilities for creating maps and to facilitate you in presenting data as part of the map or as an overlay. The functions for mapping within D3.js leverage a data format known as GeoJSON, a form of JSON that encodes geographic information.

Another common type of data for maps in D3.js is TopoJSON. TopoJSON is a more compressed form of GeoJSON. Both these formats are used to represent the cartographic information required to create a map, and D3.js processes this data and performs its usual magic of converting this information into SVG paths that visualize the map.

This chapter will start with a brief overview of GeoJSON and TopoJSON. This will give you the foundation to understand how maps are represented and rendered with D3.js. We will then jump into many examples using both data formats for rendering maps of various types, coloring the geometries within the map based upon data, and for overlaying information at specific locations on those maps.

The specific topics that we will cover in this chapter include:

- A brief overview of TopoJSON and GeoJSON
- Drawing a map of the United States with GeoJSON
- Using TopoJSON to draw the countries of the world
- Styling the geometries that comprise a map
- Panning and zooming of a map
Creating Maps with GeoJSON and TopoJSON

- Interaction with a globe
- Highlighting the boundaries of geometries on mouseover events
- Adding symbols to a map at specific locations
- Rendering maps of regions based upon data (using a choropleth)

Introducing TopoJSON and GeoJSON

Almost every map example in D3.js will use either GeoJSON or TopoJSON. GeoJSON is an open, standard, JSON-based format for representing basic geographical features as well as the non-spatial properties for those features (such as the name of a city or a landmark).

The core geometries in GeoJSON are points, line strings, and polygons. The basic description of a GeoJSON entity uses the following syntax:

```json
{
   "type": name of the type of geometry (point, line string, ...)
   "coordinates": one or more tuple of latitude / longitude
}
```

Let's take a look at the four basic types of geometry types available in GeoJSON. A **point** represents a position in two-dimensional space, and consists of a pair of one latitude and longitude. A point is normally used to specify the location of a feature on a map (such as a building):

<table>
<thead>
<tr>
<th>Example</th>
<th>Representative GeoJSON</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Point Example" /></td>
<td><code>{ &quot;type&quot;: &quot;Point&quot;,   &quot;coordinates&quot;: [30, 10] }</code></td>
</tr>
</tbody>
</table>

**LineString** describes a sequence of points which have a line drawn between them, starting at the first, through all intermediate points, and ending at the last coordinate. The name conjures up visions of stretching a string caught between all the points. These shapes are normally used to represent items such as, roads or rivers:
A **polygon** is a closed shape normally consisting of three or more points, where the last point is the same as the first and forms a closed shape. The JSON representation is shown as follows; note that the coordinates are an array of arrays of tuples:

```json
Example Representative GeoJSON

```

```
    "type": "LineString",
    "coordinates": [
        [30, 10], [10, 30],
        [40, 40]
    ]
```

```
    "type": "Polygon",
    "coordinates":
    [
        [[30, 10], [40, 40],
        [20, 40], [10, 20],
        [30, 10]]
    ]
```

The purpose of an array of arrays of tuples is to allow multiple polygons to be defined, which exclude each other, thereby allowing the exclusions of one or more polygonal regions within one another:

```
Example Representative GeoJSON

```

```
    "type": "Polygon",
    "coordinates":
    [
        [[35, 10], [45, 45],
        [15, 40], [10, 20],
        [35, 10]],
        [[20, 30], [35, 35],
        [30, 20], [20, 30]]
    ]
```

```
It is possible to define multi-part geometries where a particular geometry type is reused, and where the coordinates describe multiple instances of the type of geometry. These types are the previous types prefaced with *Multi*—*MultiPoint*, *MultiLineString*, and *MultiPolygon*. Each is demonstrated as follows:

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
<th>Representative GeoJSON</th>
</tr>
</thead>
</table>
| MultiPoint   | ![Example](image1.png) | `{  
    "type": "MultiPoint",
    "coordinates":
    [[10, 40], [40, 30],
    [20, 20], [30, 10]]
  }` |
| MultiLineString | ![Example](image2.png) | `{  
    "type": "MultiLineString",
    "coordinates":
    [
      [[10, 10], [20, 20],
        [10, 40]],
      [[40, 40], [30, 30],
        [40, 20], [30, 10]]
    ]
  }` |
| MultiPolygon | ![Example](image3.png) | `{  
    "type": "MultiPolygon",
    "coordinates": [  
      [[40, 40], [20, 45], [45, 30], [40, 40]],
      [[20, 35], [10, 30], [10, 10], [30, 5], [45, 20],
        [20, 35]],
      [[30, 20], [20, 15], [20, 25], [30, 20]]
    ]
  }` |
These basic geometries can be wrapped within a feature. A feature contains a geometry and also a set of properties. As an example, the following defines a feature which consists of a point geometry, and which has a single property, name, which can be used to describe a name for that feature:

```
{
  "type": "Feature",
  "geometry": {
    "type": "Point",
    "coordinates": [46.862633, -114.011593]
  },
  "properties": {
    "name": "Missoula"
  }
}
```

We can go up one more level in the hierarchy, and define what is known as a feature collection:

```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "geometry": {
        "type": "Point",
        "coordinates": [102.0, 0.5]
      },
      "properties": {
        "prop0": "value0"
      }
    },
    {
      "type": "Feature",
      "geometry": {
        "type": "LineString",
        "coordinates": [
          [102.0, 0.0], [103.0, 1.0], [104.0, 0.0], [105.0, 1.0]
        ]
      },
      "properties": {
        "prop0": "value0",
        "prop1": 0.0
      }
    },
    {
      "type": "Feature",
      "geometry": {
        "type": "Polygon",
        "coordinates": [
          [100.0, 0.0], [101.0, 0.0], [101.0, 1.0],
          [100.0, 1.0], [100.0, 0.0]
        ]
      },
      "properties": {
        "prop0": "value0",
        "prop1": "this": "that"
      }
    }
  ]
}
```
By combining geometries, features, and feature collections, it is possible to describe very complex shapes such as maps.

But one of the problems with GeoJSON is that it is very verbose, and particular geometries and features cannot be reused. If the same geometry is required in multiple locations, it must be completely specified a second time.

To help fix this situation, TopoJSON was created. TopoJSON provides additional constructs for the encoding of topology and reuse. Instead of discretely describing each geometry, TopoJSON allows you to define geometries, and then stitch them together using concepts known as arcs.

Arcs allows TopoJSON to eliminate redundancy, and to provide a much more compact representation as compared to GeoJSON. It is stated that TopoJSON can commonly provide 80 percent compression over GeoJSON. With every millisecond of the download time of a web page being important, this can be significant for user experience when using large sets of geometry.

A full explanation of TopoJSON is a bit beyond the scope of this book, but to briefly demonstrate it, we can look at the following and briefly examine its content:

```json
{
    "type": "Topology",
    "objects": {
        "example": {
            "type": "GeometryCollection",
            "geometries": [
                { "type": "Point",
                  "properties": {
                    "prop0": "value0"
                  },
                },
                { "type": "LineString",
                  "properties": {
                    "prop0": "value0",
                    "prop1": 0
                  },
                  "arcs": [0]
                },
                { "type": "Polygon",
                  "properties": {
                    "prop0": "value0",
                    "prop1": {
                      "this": "that"
                    }
                  }
                }
            ]
        }
    }
}
```
This TopoJSON object has three properties: type, objects, and arcs. The value of type is always "topology". The objects property consists of a geometry collection similar to those in GeoJSON, with the difference that instead of specifying coordinates, the object can, instead, specify one or more arcs.

Arcs are the big difference in TopoJSON versus GeoJSON, and represent the means of reuse. The arcs property provides an array of arrays of positions, where a position is essentially a coordinate.

These arcs are referenced by geometries of 0-based array semantics. Hence, the LineString geometry in the preceding code is referencing the first arc in the topology object by specifying arcs[0].

The polygon object is referencing an arc with value -2. A negative arc value specifies that the one's complement of the arc that should be utilized. This essentially infers that the positions in the arc should be reversed. Therefore, -2 instructs to get the reversed position of the second arc. This is one of the strategies that TopoJSON uses to reuse and compress data.

There are other options, such as transforms and bounding boxes, and other rules. For a more detailed specification, please see https://github.com/mbostock/topojson-specification.

An important thing to note about TopoJSON is that D3.js itself only uses GeoJSON data. To use data in the TopoJSON format, you will need to use the TopoJSON plugin available at https://github.com/mbostock/topojson. This plugin will convert TopoJSON into GeoJSON that can be used by D3.js functions, thereby affording the capabilities of TopoJSON to your D3.js application.
Creating a map of the United States

Our first examples will examine creating a map of the United States. We will start with an example that loads the data and gets the map rendered, and then we will examine styling the map to make it more visible, followed by examples of modifying the projection used to render the content more effectively.

Creating our first map of the United States with GeoJSON

Our first map will render the United States. We will use a GeoJSON data file, `us-states.json`, available at https://gist.githubusercontent.com/d3byex/65a128a9a499f7f0b37d/raw/176771c2f08dbd3431009ae27bef9b2f2fb56e36/us-states.json. The following are the first few lines of this file, and demonstrate how the shapes of the states are organized within the file:

```json
{
    "type": "FeatureCollection",
    "features": [
        {
            "type": "Feature",
            "id": "01",
            "properties": {
                "name": "Alabama"
            },
            "geometry": {
                "type": "Polygon",
                "coordinates": [
                    [ -87.359296, 35.00118 ], [ -85.606675, 34.984749 ],
                    [ -85.431413, 34.124869 ], [ -85.184951, 32.859696 ],
                    [ -85.069935, 32.580372 ], [ -84.960397, 32.421541 ],
                    [ -85.004212, 32.322956 ], [ -84.889196, 32.262709 ]
                ]
            }
        }
    ]
}
```

FeatureCollection at the top level consists of an array of features, each element of which is a state (or territory) as well as Washington D.C. Each state is a feature, has a single property Name, and a polygon geometry representing the outline of the state expressed in latitude and longitude tuples.

The code for the example is available at the following link:

[bl.ock](http://goo.gl/dzKsVd)
On opening the URL, you will see the following map:

The code required to take this data and render a map is sublimely simple (by design). It begins by creating the main SVG element:

```javascript
var width = 950, height = 500;
var svg = d3.select('body')
    .append('svg')
    .attr({'width': width,
           'height': height});
```

GeoJSON is simply JSON and can be loaded with `d3.json()`:

```javascript
var url = 'https://gist.githubusercontent.com/d3byex/65a128a9a499f7f0b37d/raw/176771c2f08dbd3431009ae27bef9b2f2fb56e36/us-states.json';
d3.json(url, function (error, data) {
    var path = d3.geo.path();
    svg.selectAll('path')
        .data(data.features)
        .enter()
        .append('path')
        .attr('d', path);});
```

```javascript
d3.json("/data/us-states.json", function (error, data) {
```
Creating Maps with GeoJSON and TopoJSON

Once we have the data, we can then create a `d3.geo.path()`. This object has the smarts for taking the features in the GeoJSON and converting them into an SVG path. The code then adds a path to the main SVG element, binds the data, and sets the `d` property of the path to our `d3.geo.path()` object.

Wow, with just a few lines of code, we have drawn a map of the United States!

### Styling the map of the United States

Overall, this image is dark, and the borders between the states are not particularly visible. We can change this by providing a style for the fill and stroke values used to render the map.

The code for this example is available at the following link:

```
```

When opening this URL, you will see the following map:
The only change to the previous example is to set the fill to transparent, and the borders to black:

```javascript
svg.selectAll('path')
  .data(data.features)
  .enter()
  .append('path')
  .attr('d', path)
  .style({ fill: 'none', stroke: 'black' });
```

Using the albersUsa projection

You may have a few questions about the map in the previous two examples. First, how is the map scaled to the size of the SVG element? Second, can I change this scale? And why are Alaska and Hawaii drawn down where Mexico would normally be?

These are related to some underlying assumptions about a projection. A projection is a way of taking geographic data, which is 2D data (latitude and longitude), but which is really on a three dimensional sphere (the earth), and rendering it onto a 2D surface with specific dimensions (your computer screen or viewport in the browser).

In this example, D3.js made some implicit assumptions on these factors. To help exemplify these assumptions, suppose we change the SVG element to be of size 500 x 250. When running this, we get the following output:

![Map of the United States](image)

The code that creates this is available at the following location. The only change from the previous example is that the height and width of the SVG element have each been halved:
Creating Maps with GeoJSON and TopoJSON

The result is that the actual rendering is the same size, and we have clipped the lower and rightmost three-quarters of the map due to the smaller container.

Why is this? It is because, by default, D3.js uses a projection known as an **albersUsa** projection, which has a number of assumptions that come with it:

- The dimensions of the resulting map are 1024 x 728
- The map is centered at half of the width and height (512, 364)
- The projection also places Alaska and Hawaii in the lower-left side of the map (aha!)

To change these assumptions, we can create our own **albersUsa** projection using a `d3.geo.albersUsa()` projection object. This object can be used to specify both a translation and scaling of the rendering of the results.

The following example creates an **albersUsa** projection and centers the map:

With the following result:
The code creates a d3.geo.albersUsa projection, and tells it to center the map of the United States at \([width/2, height/2]\):

```javascript
var projection = d3.geo.albersUsa()
  .translate([width / 2, height / 2]);
```

The projection object then needs to be assigned to the d3.geo.path() object using its .projection() function:

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

We have translated the center of the map, but the scale is still the same size. To change the scale, we use the projection’s .scale() function. The following example sets the scale to the width, telling D3.js that the width of the map should not be 1024, but the value of `width` and `height`:

```javascript
var projection = d3.geo.albersUsa()
  .translate([width / 2, height / 2])
  .scale([width]);
```

Note that we only pass a single value to scale. The projection scales along the width, and then automatically and proportionately along the height.
Creating Maps with GeoJSON and TopoJSON

Creating a flat map of the world

The albersUsa projection is one of many D3.js supplied projection objects. You can see the full list of these projections at https://github.com/mbostock/d3/wiki/Geo-Projections.

We don't have space to demonstrate all of these in this book, but a few are worth the effort to demonstrate a couple of TopoJSON concepts. Specifically, we will demonstrate the rendering of a map of the countries of the world, sourced from TopoJSON, and projected onto both flat and spherical surfaces.

For data in these examples, will use the world-110m.json data file provided with the TopoJSON data library source code available at https://gist.githubusercontent.com/d3byex/65a128a9a499f7f0b37d/raw/176771c2f08dbd3431009ae27bef9b2f2fb56e36/world-110m.json.

This data represents country data with features, specified at a 110-meter resolution.

Loading and rendering with TopoJSON

Now let's examine loading and rendering of TopoJSON. The following example demonstrates the process:

```
var path = d3.geo.path();
var countries = topojson.feature(world, world.objects.countries).features;
```

The code does not vary much from the previous example. The change comes after the data is loaded:

```
var path = d3.geo.path();
var countries = topojson.feature(world, world.objects.countries).features;
```

The example still uses a d3.geo.path() object, but this object cannot directly be given the TopoJSON. What needs to be done is to first extract the portion of this data that represents the countries, which is done by calling the topojson.feature() function.
The `topojson` variable is globally declared in the `topojson.js` file. Its .`feature()` function, when given a TopoJSON object (in this case, `world`), and a GeometryCollection (in this case, `world.objects.countries`), returns a GeoJSON feature that can be used by a path.

The selection to render the map then binds to this result, giving us the following map:

![Map](image)

Whoops! That's not what we expected (but as we will see, it is exactly what we coded). Why is everything globed together? It is because we are still using the default projection, a `d3.geo.albersUsa()` projection.

**Creating a map of the world using a Mercator projection**

To fix this, we simply need to create a Mercator projection object, and apply it to the path. This is a well known projection that renders the map of the globe in a rectangular area.

The process is demonstrated in the following example:

```javascript
block(12.7): http://goo.gl/IWQPte
```
The only difference in this code is the setup of the path to use a Mercator projection object:

```javascript
var projection = d3.geo.mercator()
  .scale((width + 1) / 2 / Math.PI)
  .translate([width / 2, height / 2]);
var path = d3.geo.path().projection(projection);
```

We need to give the projection object a little information about the width and height of our rendering, and the resulting map is now the following, which looks a lot more like the familiar world map:
Creating spherical maps with orthographic projection

Now let's change our projection to an orthographic projection. This projection maps data on to a simulated sphere. This is demonstrated by the following example:

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

The preceding example code gives us this beautiful rendering of the planet:

![Planet Rendering](image)

If you examine this closely, you will notice that it is not quite perfect. Notice that Australia seems to be colliding with Africa and Madagascar, and New Zealand is seen in the South Atlantic ocean.
Creating Maps with GeoJSON and TopoJSON

This is because this projection renders through all 360 degrees of the globe, and we are essentially seeing through a clear globe to the backside of the land masses on the far side.

To fix this, we can use the `.clipAngle()` function of the Mercator projection. The parameter is the number of degrees around the center point to which the landmasses should be rendered.

The following example demonstrates this in action:

```
var projection = d3.geo.orthographic()
  .clipAngle(90);
```

This changes one line of code:

```
var projection = d3.geo.orthographic()
  .clipAngle(90);
```

And gives us the following result:
It may not be apparent in the image provided in the book, but this image of the globe on the web page is fairly small. We can change the scaling of the rendering using the `.scale()` function of the projection. The default value for scale is 150, and the corresponding values will make the rendering larger or smaller.

The following example makes the globe twice as large along with setting the center of the globe to not be clipped by the SVG container:

```javascript
var projection = d3.geo.orthographic()
  .scale(300)
  .clipAngle(90)
  .translate([width / 2, height / 2]);
```

This orthographic projection, by default, centers the view on the globe at latitude and longitude (0,0). If we want to center on another location, we need to `.rotate()` the projection by a number of degrees of latitude and longitude.

The following example rotates the globe to show the United States prominently:

```javascript
var projection = d3.geo.orthographic()
  .scale(300)
  .clipAngle(90)
  .translate([width / 2, height / 2])
  .rotate([90, -40]);
```
Creating Maps with GeoJSON and TopoJSON

This change in the projection gives us the following result:

Spicing up a globe
Although this globe is quite impressive for the amount of code used to create it, it feels a little dull. Let's differentiate the countries a little more, and also add the lines of latitude and longitude.

Coloring the countries on a globe
We can color the countries on the globe using a `d3.scale.category20()` color scale. But we can't simply rotate through the colors, as there will be cases where adjacent countries will be filled with the same color.
To avoid this, we will take advantage of another function of TopoJSON, `topojson.neighbors()`. This function will return, given a set of geometries (like the countries), a data structure that identifies which geometries are adjacent to each other. We can then utilize this data to prevent the potential problem with colors.

The process is demonstrated in the following example:

```
block (12.12): http://goo.gl/9UimER
```

The projection in this example remains the same. The remainder of the code is changed.

We start by using the same projection as the last example so that code is not repeated here. The following creates the data structure of the colors, the countries, and the neighbors:

```javascript
var color = d3.scale.category20();
var countries = topojson.feature(world,
    world.objects.countries).features;
var neighbors = topojson.neighbors(
    world.objects.countries.geometries);
```

The creation of the globe then uses the following statement:

```javascript
var color = d3.scale.category20();
svg.selectAll('.country')
    .data(countries)
    .enter()
    .append('path')
    .attr('d', path)
    .style('fill', function (d, i) {
        return color(d.color = d3.max(neighbors[i],
            function (n) {
                return countries[n].color;
            } + 1 | 0);
    });
```
Our resulting globe is the following:

Pretty nice! But it's still lacking in the lines of longitude or latitude, and you can't really tell what the extents of the globe are. Let's fix that now by adding the lines of latitude and longitude.

You'll be really surprised at how easy it is to add the latitudes and longitudes. In D3.js, these are referred to as graticules. We create them by instantiating a `d3.geo.graticules()` object, and then by appending a separate path prior to the path for the countries.

This is demonstrated in the following example:

```
block (12.13): http://goo.gl/5eJ0ai
```
The only code added to the previous example is the following:

```javascript
var graticule = d3.geo.graticule();
svg.append('path')
  .datum(graticule)
  .attr('d', path)
  .style({
    fill: 'none',
    stroke: '#777',
    'stroke-width': '.5px',
    'stroke-opacity': 0.5
  });
```

The change in code results in the following:

Voila! And as they say, easy-peasy!
Adding interactivity to maps

What good is a map if the user is not able to pan and zoom around the map to change the focus, and take a closer look at things? Fortunately, because of D3.js, this becomes very simple to implement. We will look at three different examples of interactivity and maps:

- Panning and zooming a world map
- Highlighting country borders on mouseover
- Rotating a globe with the mouse

Panning and zooming a world map

To demonstrate panning and zooming of a world map, we will make a few modifications to our world Mercator projection example. These modifications will be for using the mouse wheel to zoom in and out, and to be able to drag the map to move it to another center.

A possible image with this version of the map code could look like the following, which is centered just east of Brazil, and brought up several factors of zoom:
There are a couple of considerations that we should take into account when panning and zooming a map:

- We can only zoom in and out between two extents so that we do not zoom out too far as to lose sight of the map, or too close as to get lost in a single country
- We can only drag the map to a certain extent to ensure that it is constrained and not dragged off some edge

The example is available at the following location:

```
http://goo.gl/jjouGK
```

Much of the code is reused from the Mercator projection example, and also adds the code to uniquely color the countries.

The creation of the main SVG element differs to allow for drag and zoom. This starts with creating a zoom behavior, and assigning it to the main SVG element. Additionally, since we need to zoom the client elements, we add a group to facilitate this action:

```javascript
var zoom = d3.behavior.zoom()
    .scaleExtent([1, 5])
    .on('zoom', moveAndZoom);

var svg = d3.select('body')
    .append('svg')
    .attr({
        width: width,
        height: height
    })
    .call(zoom);
var mainGroup = svg.append('g');
```

The rest of the main part of the code loads the data and renders the map, and is identical to the previous examples.
The `moveAndZoom` function, which will be called on any drag and zoom events, is
given as follows:

```javascript
function moveAndZoom() {
  var t = d3.event.translate;
  var s = d3.event.scale;

  var x = Math.min(
    (width / height) * (s - 1),
    Math.max(width * (1 - s), t[0]));

  var h = height / 4;
  var y = Math.min(
    h * (s - 1) + h * s,
    Math.max(height * (1 - s) - h * s, t[1]));

  mainGroup.attr('transform', 'translate(' + x + ',' + y + ')
    scale(' + s + ')');
}
```

From these values, we need to adjust the SVG translate on the map based upon the
current mouse position, while taking into account the scale level. We also do not
want this to translate the map in any direction such that there is padding between
the map and the boundaries; this is handled by combined calls to `Math.min` and
`Math.max`.

Congratulations, you now have a fully pan and scan map!

Note that as you zoom in, the boundaries on the countries are fairly
ragged. This is due to the 110-meter resolution of the data. To have
more accurate graphics, use the files with the finer details. Even better,
dynamically change to higher resolution data depending upon the
zoom level.

### Highlighting country borders on mouse hover

Now let's add another interactivity effect to our map: highlighting the border
of a country which has the mouse currently over its geometry. This will help us
accentuate the country the user is currently examining. A quick demonstration of
this is the following, where Peru has a thin white border:
The example is available at the following location:

```
bl.ock (12.15): http://goo.gl/DTtJ2A
```

This is implemented with a few modifications to the previous example. The modifications start with the creation of the top-level group element:

```javascript
mainGroup.style(
  {
    stroke: 'white',
    'stroke-width': 2,
    'stroke-opacity': 0.0
  });
```

This code informs D3.js that all SVG elements contained within the group will have a 2-pixel white border, which is initially transparent. When we hover the mouse, we will make this visible on the appropriate geometry.

Now we need to hook up mouse event handlers on each of the path elements that represent countries. On the mouseover event, we make the stroke-opacity opaque, and set it back to transparent when the mouse exits:

```javascript
mainGroup.selectAll('path')
  .on('mouseover', function () {
    d3.select(this).style('stroke-opacity', 1.0);
  });
mainGroup.selectAll('path')
  .on('mouseout', function () {
    d3.select(this).style('stroke-opacity', 0.0);
  });
```
There is one more small change that we will want to make whenever the zoom level changes. As the zoom level goes up, the country borders get disproportionately thick. To prevent this, we can add the following statement to the end of the moveAndZoom function:

```javascript
    g.style("stroke-width", ((1 / s) * 2) + "px");
```

This is stating that the border of a country should always stay at what is visually 2px thick, no matter what the zoom level.

**Rotating a globe using the mouse**

Interactivity can also be applied to other projections. We will examine rotating an orthographic globe using the mouse. The example is available at the following location:

```
    block (12.16): http://goo.gl/cpH0LN
```

To save a little space, we won’t show an image here, as it looks the same as the earlier example in the chapter, except that it rotates following the mouse. That, and the rotation effect is lost in a print medium.

But the way this works is very simple. The technique involves creating two scales, one for longitude and the other for latitude. Longitude is calculated as mapping the mouse position from 0 to the width of the graphic to -180 and 180 degrees of longitude. The latitude is a mapping of the vertical mouse position to 90 and -90 degrees:

```javascript
    var scaleLongitude = d3.scale.linear()
        .domain([0, width])
        .range([-180, 180]);

    var scaleLatitude = d3.scale.linear()
        .domain([0, height])
        .range([90, -90]);
```
When the mouse is moved over the SVG element, we capture it and scale the mouse position into a corresponding latitude and longitude; we then set the rotation of the projection:

```javascript
svg.on('mousemove', function() {
    var p = d3.mouse(this);
    projection.rotate([scaleLongitude(p[0]),
                      scaleLatitude(p[1])]);
    svg.selectAll('path').attr('d', path);
});
```

It's a pretty cool little trick of mathematics and scales that allows us to be able to see every position on the entire globe.

**Annotating a map**

Our final examples of working with maps will demonstrate making annotations to a map. The first two will demonstrate placing labels and markers on a map, and the third will demonstrate the use of gradient colors to color regions all the way down to a state level.

All of these techniques would normally involve some fairly complex math if we had to do it on our own, but thankfully, D3.js again comes to help us solve this with just a few statements.

**Labelling states using centroids**

The maps of the United States we've created up to this point feel a little lacking in content, as they have not had the names of the states placed over their geometries. It would be very helpful to many reading a map to have the names visible. The example is available at the following location:

[bl.ock (12.17): http://goo.gl/3vChcR]
Creating Maps with GeoJSON and TopoJSON

The result of the example is the following:

This is actually fairly easy to implement, with only the addition of one statement to our United States Mercator projection example. The following code is placed immediately after the `selectAll()` statement that creates the boundaries for all the states:

```javascript
svg.selectAll('text')
    .data(data.features)
    .enter()
    .append('text')
    .text(function(d) { return d.properties.name; })
    .attr(
        x: function(d) { return path.centroid(d)[0]; },
        y: function(d) { return path.centroid(d)[1]; },
        'text-anchor': 'middle',
        'font-size': '6pt'
    );
```
This statement creates a text element for each geometric feature in the data file, and sets the text to be the value of the name property of the geometry object.

The position of the text uses a function of the path that calculates the centroid of the geometry. The centroid is the mathematical center of the geometry, and can be calculated using the .centroid() function of a path.

For most states, especially rectangular ones, this works well. For others with irregular shapes, take Michigan for example, the placement is perhaps not optimal for aesthetics. There are various ways to fix this, but those are beyond the scope of this book (a hint: it involves adding additional data to represent location offsets for each geometry).

**Placing symbols at specific geographic locations**

The last example with maps that we will look at will be to place SVG elements on the map at specific coordinates. Specifically, we will place circles at the position of the 50 most populous cities, and size the circle relative to the population.

The data we will use is in us-cities.csv, which is available at https://gist.githubusercontent.com/d3byex/65a128a9a499f7f0b37d/raw/176771c2f08dadb3431009ae27bef9b2f2fb56e36/us-cities.csv. The data is straightforward; the following are the first few lines:

```
name,population,latitude,longitude
New York,8491079,40.6643,-73.9385
Los Angeles,3792621,34.0194,-118.4108
Chicago,2695598,41.8376,-87.6818
```

The example is available at the following location:

[ block (12.18): http://goo.gl/Y9MN5q ]
Creating Maps with GeoJSON and TopoJSON

The resulting visualization is the following:

The preceding example leverages the United States Mercator examples code. This example does, however, need to load two data files. To facilitate this, we will use a library called queue created by Mike Bostock to load these files asynchronously, and when both are complete, execute the ready() function. You can get this library and documentation at https://github.com/mbostock/queue:

```javascript
queue()
  .defer(d3.json, usDataUrl)
  .defer(d3.csv, citiesDataUrl)
  .await(function (error, states, cities) {
```

[248]
The map is then rendered as in the earlier examples. Then we need to place the circles. To do this, we will need to convert the latitude and longitude values to $X$ and $Y$ pixel locations. We can do this in D3.js using the projection object:

```javascript
svg.selectAll('circle')
  .data(cities)
  .enter()
  .append('circle')
  .each(function(d) {
    var location = projection([d.longitude, d.latitude]);
    d3.select(this).attr({
      cx: location[0],
      cy: location[1],
      r: Math.sqrt(+d.population * 0.00004)
    });
  });
```

For each circle that is created, this code calls the projection function passing it the latitude and longitude for each city. The return value is the $x$ and $y$ location of the pixel representing that location. So we just set the center of the circle to this result, and assign the circle a radius that is a scale value of the population.

### Creating a choropleth

Our last map example is for creating a choropleth. A choropleth is a map with areas filled in with different colors to reflect the underlying data values—not just differing colors to represent different geographic boundaries. These are quite common types of visuals, and they commonly show a difference in opinion amongst the populations in adjacent regions, or how economic factors differ along neighbors.

The example is available at the following location:

```bash
block (12.19): http://goo.gl/ZeTh4o
```
The resulting visualization is the following:

![Map of the United States with varying shades indicating unemployment rates](image)

This choropleth represents the unemployment rate in the US counties for the year 2008. The shade of blue varies from darker, representing lower unemployment, to lighter and higher unemployment.

The data for unemployment is available at [https://gist.githubusercontent.com/d3byex/65a128a9a499f7f0b37d/raw/176771c2f08db3431009ae27bef9b2f2fb56e36/unemployment.tsv](https://gist.githubusercontent.com/d3byex/65a128a9a499f7f0b37d/raw/176771c2f08db3431009ae27bef9b2f2fb56e36/unemployment.tsv). The first few lines are the following:

<table>
<thead>
<tr>
<th>id</th>
<th>rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>.097</td>
</tr>
<tr>
<td>1003</td>
<td>.091</td>
</tr>
<tr>
<td>1005</td>
<td>.134</td>
</tr>
<tr>
<td>1007</td>
<td>.121</td>
</tr>
<tr>
<td>1009</td>
<td>.099</td>
</tr>
<tr>
<td>1011</td>
<td>.164</td>
</tr>
</tbody>
</table>

The data consists of a pair of a county identifier and the respective unemployment rate. The county ID will be matched to county IDs in the us.json file available at [https://gist.githubusercontent.com/d3byex/65a128a9a499f7f0b37d/raw/176771c2f08db3431009ae27bef9b2f2fb56e36/us.json](https://gist.githubusercontent.com/d3byex/65a128a9a499f7f0b37d/raw/176771c2f08db3431009ae27bef9b2f2fb56e36/us.json).
This file consists of TopoJSON describing the shape of all of the counties in the US, each with the same county ID in the unemployment file. A snippet of this file is the following, which shows for country 1001 the arcs that should be used to render it:

```json
{
  "type": "Polygon",
  "id": 1001,
  "arcs": [ [ -8063, 8094, 8095, -8084, -7911 ] ]
}
```

Our goal is to quantize the unemployment rates, and then fill each geometry with a color mapped to that quantile. It's actually easier to do than it may seem.

In this example, we will map our unemployment rates into ten quantiles. The color used for each will be specified using a style with a specific name. These are declared as follows:

```html
<style>
  .q0-9 { fill:rgb(247,251,255); }
  .q1-9 { fill:rgb(222,235,247); }
  .q2-9 { fill:rgb(198,219,239); }
  .q3-9 { fill:rgb(158,202,225); }
  .q4-9 { fill:rgb(107,174,214); }
  .q5-9 { fill:rgb(66,146,198); }
  .q6-9 { fill:rgb(33,113,181); }
  .q7-9 { fill:rgb(8,81,156); }
  .q8-9 { fill:rgb(8,48,107); }
</style>
```

The data is loaded using the `queue()` function:

```javascript
queue()
  .defer(d3.json, usDataUrl)
  .defer(d3.tsv, unempDataUrl, function(d) {
    rateById.set(d.id, +d.rate);
  })
  .await(function(error, us) {
```

This code uses an alternate form of `.defer()` for the unemployment data, which calls a function for each data item that is loaded (another cool thing about queue). This builds a `d3.map()` object (like a dictionary object) that maps the county ID to its unemployment rate, and we use this map later during rendering.
Creating Maps with GeoJSON and TopoJSON

The county data is rendered first. To do this, we need to create a quantile scale which maps the domain from 0 to 0.15. This will be used to map the unemployment levels to one of the styles. The range is then configured to generate the names of the nine styles:

```javascript
var quantize = d3.scale.quantize()
    .domain([0, .15])
    .range(d3.range(9).map(function(i) {
        return 'q' + i + '-9';
    }));
```

Next, the code creates the albersUsa projection and an associated path:

```javascript
var projection = d3.geo.albersUsa()
    .scale(1280)
    .translate([width / 2, height / 2]);

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

The next step is to create a group to hold the shaded counties. Then, to this group, we will add a path for each county by binding it to the counties features:

```javascript
svg.append('g')
    .attr('class", "counties"
    .selectAll("path")
    .data(topojson.feature(us, us.objects.counties).features)
    .enter()
    .append("path")
    .attr("class", function(d) {
        return quantize(rateById.get(d.id));
    })
    .attr("d", path);
```

Finally, we overlay the outlines of the states using a white stroke for the borders to help us differentiate the state borders:

```javascript
svg.append('path')
    .datum(topojson.mesh(us, us.objects.states)
    .attr({'
        'class': 'states',
        fill: 'none',
        stroke: '#fff',
        'stroke-linejoin': 'round',
        'd': path
    });
```
This particular piece of code also uses the `topojson.mesh` function to extract the `MultiPolygon` (GeoJSON) data for all of the states from the TopoJSON object.

And that's all! We've created a choropleth, and used a coding pattern that can be reused easily with other types of data.

**Summary**

We started this chapter by looking briefly at GeoJSON and TopoJSON. If you do anything with maps in D3.js, you will be using one or both of these. We covered it just enough to give an understanding of its structure, and how it is used to define data that can be rendered as a map.

From there, we dove into creating several maps and covered many of the concepts that you will use in their creation. These included loading the data, creating projections, and rendering the geometries within the data.

We examined two projections, Mercator and orthographic, to give an idea of how these present data. Along the way, we also looked at how to style elements on the map, filling geometries with color, and highlighting geometries on mouseover.

Then we examined how to annotate our maps with labels as well as color elements based upon data (choropleths), and to place symbols on the map at specific geographic positions, with a size that is based upon the data.

At this point in the book, we have been pretty thorough in covering much of the core of D3.js, at least enough to make you very dangerous with it. But we have also only ever created stand-alone visualizations, ones that do not interact with other visualizations.

In the next chapter, the final one of this book, we will look at combining multiple D3.js visualizations using AngularJS, and where those visuals also react to the user manipulating other content on their page.
Where to buy this book

You can buy D3.js By Example from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.