

Chapter 12. Geomapping

Bar charts, scatterplots, ring charts, and even force-directed graphs... *Yeah, that's all okay*, you're thinking, *but get to the maps already!*

JSON, Meet GeoJSON

You've already met JSON. Now meet GeoJSON, the JSON-based standard for encoding geodata for web applications. GeoJSON actually is not a totally different format, but just a very specific use of JSON.

Before you can generate a geographic map, you need to acquire the path data (the outlines) for the shapes you want to display. We'll start with a common example, mapping US state boundaries. I've included a file *us-states.json* with the sample code. This file is taken directly from one of the D3 examples, and we owe Mike Bostock a word of thanks for generating this nice, clean file of state boundaries.

Opening up *us-states.json*, you'll see it looks something like this (reformatted and greatly abbreviated here):

```
{
  "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],[-85.058981,32.13674] ...
        ]]
      }
    },
    {
      "type": "Feature",
      "id": "02",
      "properties": { "name": "Alaska" },
      "geometry": {
        "type": "MultiPolygon",
        "coordinates": [[[[[-131.602021,55.117982],
          [-131.569159,55.28229],[-131.355558,55.183705],
          [-131.38842,55.01392],[-131.645836,55.035827],
          [-131.602021,55.117982]]],[[-131.832052,55.42469],
          [-131.645836,55.304197],[-131.749898,55.128935],
          [-131.832052,55.189182], ...
        ]]]
      }
    }
  ]
}
```

```
    }  
    ...
```

In typical GeoJSON style, we see, first of all, that this is all one giant object. (Curly brackets, remember?) That object has a `type` of `FeatureCollection`, followed by `features`, which is an array of individual `Feature` objects. Each one of these `Features` represents a US state. You can see each state's name under `properties`.

But the real meat in any GeoJSON file is under `geometry`. This is where the feature's `type` is specified, followed by the many `coordinates` that constitute the feature's boundary. Within the `coordinates` are sets of longitude/latitude pairs, each one as a small, two-value array. This is the information that cartographers dedicate their lives to compiling and refining. We owe generations of explorers and researchers our gratitude for creating these sequences of tiny, yet extremely powerful numbers.

It's important to note that *longitude* is always listed first. So despite the cultural bias toward lat/lon, GeoJSON is a lon/lat world.

Also, in case your cartographic skills are a bit rusty, here's how you can always remember which is which:

- Longitude is long. Therefore, longitudinal lines run vertically, as though hanging down from above.
- Latitude is fatitude. Therefore, latitudinal lines run horizontally, as though wrapping around the Earth's waist.

Longitude and latitude together constitute an enormous grid that encircles the whole globe. Conveniently for us, lon/lat can be easily converted to x/y values for screen display. In bar charts, we map data values to display values—numbers to rectangle heights. In geomapping, we also map data values to display values—lon/lat becomes x/y. Thinking in terms of x/y also makes it easier to get used to the uncomfortable order of longitude first, latitude second.

[Get Lat+Lon](#) is a great resource by Michal Migurski for double-checking coordinate values. Keep it open in a browser tab whenever you're working on geomaps. You will reference it often.

Paths

We've got our geodata. Now get ready to rock.

First, we define our first *path generator*:

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

`d3.geo.path()` is a total lifesaver of a function. It does all the dirty work of translating that mess of GeoJSON coordinates into even messier messes of SVG path codes. All hail `d3.geo.path()`!

Now we *could* paste all that GeoJSON directly into our HTML file, but ugh, so many coordinates and curly brackets—what a mess! It's cleaner and more common to keep the geodata in a separate file and load it in using `d3.json()`:

```
d3.json("us-states.json", function(json) {  
  
    svg.selectAll("path")  
      .data(json.features)  
      .enter()
```

```
.append("path")
.attr("d", path);

});
```

`d3.json()` takes two arguments. First, it takes a string pointing to the path of the file to load in. Second, it takes a callback function that is fired when the JSON file has been loaded and parsed. (See [“Handling Data Loading Errors”](#) for details on the callback function.) `d3.json()`, just like `d3.csv()`, is *asynchronous*, meaning it won't prevent the rest of your code from running while the browser waits for that external file to load. For example, code placed *after* the callback function might be executed *before* the contents of the callback itself:

```
d3.json("someFile.json", function(json) {
    //Put things here that depend on the JSON loading
});

//Only put things here that can operate independently of the JSON
console.log("I like cats.");
```

So as a rule, when loading external datafiles, put the code that depends on that data within the callback function. (Or put the code into other custom functions, and then call those functions from within the callback.)

Back to the example. Finally, we bind the GeoJSON features to new path elements, creating one new path for each feature:

```
svg.selectAll("path")
  .data(json.features)
  .enter()
  .append("path")
  .attr("d", path);
```

Notice that last line, in which `d` (the path data attribute) is referred to our path generator, which magically takes the bound geodata and calculates all that crazy SVG code. The result is [Figure 12-1](#).



Figure 12-1. Our first view of GeoJSON data

A map! That was so easy! Check it out in `01_paths.html`. The rest is just customization.

You can find lots more detail on paths and path generator options [on the wiki](#).

Projections

As an astute observer, you noticed that our map isn't quite showing us the entire United States. To correct this, we need to modify the *projection* being used.

What is a projection? Well, as an astute observer, you have also noticed that the globe is round, not flat. Round things are three-dimensional, and don't take well to being represented on two-dimensional surfaces. A *projection* is an algorithm of compromise; it is the method by which 3D space is "projected" onto a 2D plane.

We define D3 projections using a familiar structure:

```
var projection = d3.geo.albersUsa()  
                .translate([w/2, h/2]);
```

D3 has several built-in projections. Albers USA is a composite projection that nicely tucks Alaska and Hawaii beneath the Southwest. (You'll see in a second.) `albersUsa` is actually the default projection for `d3.path.geo()`, but now that we've specified it explicitly, we can set several custom options, such as a translation value. You can see we're translating the projection to the center of the image (half of its width and half of its height).

The only other change we have to make is to tell the path generator explicitly that it should reference our customized projection when generating all those paths:

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

That gives us [Figure 12-2](#). Getting there! See [02_projection.html](#) for the working code.



Figure 12-2. The same GeoJSON data, but now with a centered projection

We can also add a `scale()` method to our projection in order to shrink things down a bit and achieve the result shown in [Figure 12-3](#):

```
var projection = d3.geo.albersUsa()  
                .translate([w/2, h/2])  
                .scale([500]);
```

The default scale value is 1,000. Anything smaller will shrink the map; anything larger will expand it.



Figure 12-3. The USA, scaled and centered within the image

Cool! See that working code in *o3_scaled.html*.

By adding a single `style()` statement, we could set the path's fills to something less severe, like the blue shown in [Figure 12-4](#).

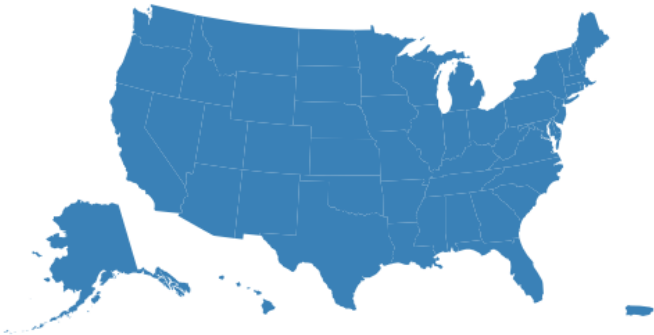


Figure 12-4. Now more blue-ish than black

See *o4_fill.html* for that. You could use the same technique to set stroke color and width, too.

Map projections are extremely powerful algorithms, and different projections are useful for different purposes and different parts of the world (near the poles, versus the equator, for example).

Thanks primarily to the contributions of [Jason Davies](#), D3's geo projections plug-ins now support essentially every obscure projection you could imagine. Reference [the complete visual reference of D3 projections on the wiki](#). You might also find the [projection comparison demo](#) useful.

Choropleth

Choro-what? This word, which can be difficult to pronounce, refers to a geomap with areas filled in with different values (light or dark) or colors to reflect associated data values. In the United States, so-called “red state, blue state” choropleth maps showing the Republican and Democratic leanings of each state are ubiquitous, especially around election time. But choropleths can be generated from any values, not just political ones.

These maps are also some of the most requested uses of D3. Although choropleths can be fantastically useful, keep in mind that they have some inherent perceptual limitations. Because they use *area* to encode values, large areas with low density (such as the state of Nevada) might be overrepresented visually. A standard choropleth does not represent per-capita values fairly—Nevada is too big, and Delaware far too

small. But they *do* retain the geography of a place, and—as maps—they look really, really cool. So let's dive in. (You can follow along with *o5_choropleth.html*.)

First, I'll set up a scale that can take data values as input, and will return colors. This is the heart of choropleth-style mapping:

```
var color = d3.scale.quantize()  
    .range([ "rgb(237,248,233)", "rgb(186,228,179)",  
            "rgb(116,196,118)", "rgb(49,163,84)", "rgb(0,109,44)" ] );
```

A *quantize* scale functions as a linear scale, but it outputs values from within a discrete range. These output values could be numbers, colors (as we've done here), or anything else you like. This is useful for sorting values into “buckets.” In this case, we're using five buckets, but there could be as many as you like.

Notice I've specified an output range, but not an input domain. (I'm waiting until our data is loaded in to do that.) These particular colors are taken from the `colorbrewer.js` file included in [the D3 GitHub repository](#)—a collection of perceptually optimized colors, selected by Cynthia Brewer, and based on her research.

Next, we need to load in some data. I've provided a file *us-ag-productivity-2004.csv*, which looks like this:

```
state,value  
Alabama,1.1791  
Arkansas,1.3705  
Arizona,1.3847  
California,1.7979  
Colorado,1.0325  
Connecticut,1.3209  
Delaware,1.4345  
...
```

This data, provided by the US Department of Agriculture, reports agricultural productivity by state during the year 2004. The units are relative to an arbitrary baseline of the productivity of the state of Alabama in 1996 (1.0), so greater values are more productive, and smaller values less so. (Find lots of open government datasets at <http://data.gov>.) I expect this data will give us a nice map of states' agricultural productivity.

To load in the data, we use `d3.csv()`:

```
d3.csv("us-ag-productivity-2004.csv", function(data) { ...
```

Then, in the callback function, I want to set the `color` quantize scale's input domain (before I forget!):

```
    color.domain([  
        d3.min(data, function(d) { return d.value; } ),  
        d3.max(data, function(d) { return d.value; } )  
    ] );
```

This uses `d3.min()` and `d3.max()` to calculate and return the smallest and largest data values, so the scale's domain is dynamically calculated.

Next, we load in the JSON geodata, as before. But what's new here is I want to *merge* the agricultural data *into* the GeoJSON. Why? Because we can only bind one set of data to elements at a time. We definitely need

the GeoJSON, from which the paths are generated, but we also need the new agricultural data. So if we can smush them into a single, monster array, then we can bind them to the new path elements all at the same time. (There are several approaches to this step; what follows is my preferred method.)

```
d3.json("us-states.json", function(json) {

  //Merge the ag. data and GeoJSON
  //Loop through once for each ag. data value
  for (var i = 0; i < data.length; i++) {

    //Grab state name
    var dataState = data[i].state;

    //Grab data value, and convert from string to float
    var dataValue = parseFloat(data[i].value);

    //Find the corresponding state inside the GeoJSON
    for (var j = 0; j < json.features.length; j++) {

      var jsonState = json.features[j].properties.name;

      if (dataState == jsonState) {

        //Copy the data value into the JSON
        json.features[j].properties.value = dataValue;

        //Stop looking through the JSON
        break;

      }

    }

  }

})
```

Read through that closely. Basically, for each state, we are finding the GeoJSON element with the same name (e.g., “Colorado”). Then we take the state’s data value and tuck it in under `json.features[j].properties.value`, ensuring it will be bound to the element and available later, when we need it.

Lastly, we create the paths just as before, only we make our `style()` value dynamic:

```
svg.selectAll("path")
  .data(json.features)
  .enter()
  .append("path")
  .attr("d", path)
  .style("fill", function(d) {
    //Get data value
    var value = d.properties.value;
```

```

        if (value) {
            //If value exists...
            return color(value);
        } else {
            //If value is undefined...
            return "#ccc";
        }
    });

```

Instead of "steelblue" for everyone, now each state path gets a different fill value. The trick is that we don't have data for *every* state. The dataset we're using doesn't have information for Alaska, the District of Columbia, Hawaii, or Puerto Rico (which, although not a state, is still included in the GeoJSON and appears in the projection).

So to accommodate those exceptions, we include a little logic: an `if ()` statement that checks to see whether or not the data value has been defined. If it exists, then we return `color (value)`, meaning we pass the data value to our quantize scale, which returns a color. For undefined values, we set a default of light gray (`#ccc`).

Beautiful! Just look at the result in [Figure 12-5](#). Check out the final code and try it yourself in `o5_choropleth.html`.

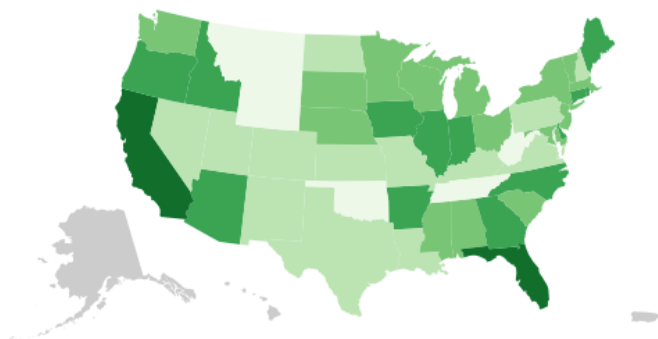


Figure 12-5. A choropleth map showing agricultural productivity by state

Adding Points

Wouldn't it be nice to put some cities on this map, for context? Maybe it would be interesting or useful to see how many large, urban areas there are in the most (or least) agriculturally productive states. Again, let's start by finding the data.

Fortunately, the US Census has us covered, once again. (Your tax dollars at work!) Here's the start of a raw CSV dataset from the Census that shows ["Annual Estimates of the Resident Population for Incorporated Places Over 50,000"](#).

```

table with row headers in column A and column headers in rows 3 through 4,,,,,,
'''
"Table 1. Annual Estimates of the Resident Population for Incorporated Places
Over 50,000, Ranked by July 1, 2011 Population: April 1, 2010 to July 1, 2011"
,,,,,,,,
Rank,Geographic Area,, "April 1, 2010",, Population Estimate (as of July 1),,
,,, Place, State, Census, Estimates Base, 2010, 2011, , , ,

```



```

1,New York city,New York,"8,175,133","8,175,133","8,186,443","8,244,910",,,,
2,Los Angeles city,California,"3,792,621","3,792,625","3,795,761","3,819,702"
,,,,
3,Chicago city,Illinois,"2,695,598","2,695,598","2,698,283","2,707,120",,,,
4,Houston city,Texas,"2,099,451","2,099,430","2,108,278","2,145,146",,,,
5,Philadelphia city,Pennsylvania,"1,526,006","1,526,006","1,528,074","1,536,471"
,,,,
6,Phoenix city,Arizona,"1,445,632","1,445,656","1,448,531","1,469,471",,,,
7,San Antonio city,Texas,"1,327,407","1,327,606","1,334,431","1,359,758",,,,
8,San Diego city,California,"1,307,402","1,307,406","1,311,516","1,326,179",,,,
9,Dallas city,Texas,"1,197,816","1,197,816","1,201,715","1,223,229",,,,
10,San Jose city,California,"945,942","952,612","955,091","967,487",,,,
...

```

This is quite messy, and I don't even need all this data. So I'll open the CSV up in my favorite spreadsheet program and clean it up a bit, removing unneeded columns. (You could use LibreOffice Calc, Apple Numbers, or Microsoft Excel.) I'm also interested in only the largest 50 cities, so I'll delete all the others. Exporting back to CSV, I now have this:

```

rank,place,population
1,New York city,8175133
2,Los Angeles city,3792621
3,Chicago city,2695598
4,Houston city,2099451
5,Philadelphia city,1526006
6,Phoenix city,1445632
7,San Antonio city,1327407
8,San Diego city,1307402
9,Dallas city,1197816
10,San Jose city,945942
...

```

This information is useful, but to place it on the map, I'm going to need the latitude and longitude coordinates for each of these places. Looking this up manually would take *forever*. Fortunately, we can use a *geocoding* service to speed things up. Geocoding is the process of taking place names, looking them up on a map (or in a database, really), and returning precise lat/lon coordinates. "Precise" might be a bit of an overstatement—the geocoder does the best job it can, but it will sometimes be forced to make assumptions given vague data. For example, if you specify "Paris," it will probably assume you mean Paris, France and not Paris, Texas. It's good practice to eyeball the geocoder's output once you get it on the map, and manually adjust any erroneous coordinates (using teczo.com/squares as a reference).

I'll head over to [my favorite batch geocoder](#), paste in just the place names, and click Start! A few minutes later, the geocoder spits out some more comma-separated values, which includes lat/lon pairs. I bring those back into my spreadsheet, and save out a new, unified CSV with coordinates:

```

rank,place,population,lat,lon
1,New York city,8175133,40.71455,-74.007124
2,Los Angeles city,3792621,34.05349,-118.245323
3,Chicago city,2695598,45.37399,-92.888759

```

```

4,Houston city,2099451,41.337462,-75.733627
5,Philadelphia city,1526006,37.15477,-94.486114
6,Phoenix city,1445632,32.46764,-85.000823
7,San Antonio city,1327407,37.706576,-122.440612
8,San Diego city,1307402,37.707815,-122.466624
9,Dallas city,1197816,40.636,-91.168309
10,San Jose city,945942,41.209716,-112.003047
...

```

That was unbelievably easy. Ten years ago that step would have taken us hours of research and tedious data entry, not seconds of mindless copying-and-pasting. Now you see why we're experiencing an explosion of online mapping.

Our data is ready, and we already know how to load it in:

```

d3.csv("us-cities.csv", function(data) {
    //Do something...
});

```

In the callback function, we can specify how to create a new `circle` element for each city, and then *position each circle* according to the corresponding city's geo-coordinates:

```

svg.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", "yellow")
    .style("opacity", 0.75);

```

The magic here is in those `attr()` statements that set the `cx` and `cy` values. You see, we can access the raw latitude and longitude values as `d.lat` and `d.lon`. But what we really need for positioning these circles are *x/y screen coordinates*, not *geo-coordinates*.

So we bring back our magical friend `projection()`, which is basically just a two-dimensional scale method. With D3 scales, we put in one number, and get back another. With projections, we put in two numbers, and get back two. (The other main difference is that the behind-the-scenes math for projections is much more complex than the simple normalization of scales.)

The map projection takes a two-value array as input, with *longitude* first (remember, it's lon/lat, not lat/lon, in GeoJSON-ville). Then the projection returns a two-value array with x/y screen values. So, for `cx`, we use `[0]` to grab the *first* of those values, which is *x*. For `cy`, we use `[1]` to grab the *second* of those values, which is *y*. Make sense?

The resulting map in [Figure 12-6](#) is suh-weeeet! Check out the code in `o6_points.html`.

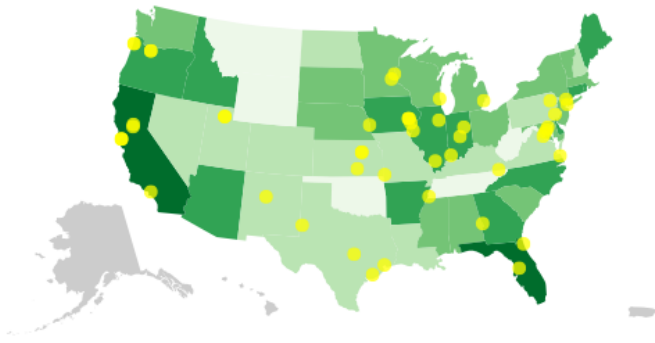


Figure 12-6. The top 50 largest US cities, represented as cute little yellow dots

Yet these dots are all the same size. Let's link the population data to circle size. Instead of a static area, we'll reference the population value:

```
.attr("r", function(d) {  
    return Math.sqrt(parseInt(d.population) * 0.00004);  
})
```

Here we grab `d.population`, wrap it in `parseInt()` to convert it from a string to an integer, scale that value down by an arbitrary amount, and finally take the square root (to convert from an area value to a radius value). See the code in *07_points_sized.html*.

As you can see in [Figure 12-7](#), now the largest cities really stand out. The differences in city size are significant; this representation might be better suited to a log scale, especially if even smaller cities are included. Instead of multiplying by 0.00004, you could more properly use a custom D3 scale function. (I'll leave that to you.)

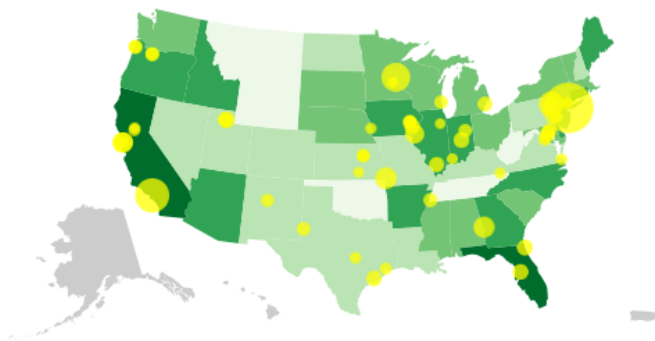


Figure 12-7. Cities as dots, with area set by population

The point here is to see that we've successfully loaded and visualized two different datasets on a map. (Make that three, counting the geocoded coordinates we merged in!)

Acquiring and Parsing Geodata

If we only ever wanted to make maps of the United States, then we would already have all the GeoJSON we'd ever need. But the world is much bigger than that, and there are lots of other areas to map. So allow me to digress for a moment and address how to acquire and parse geodata for the area of your choosing. The end goal is to produce a GeoJSON file, like *us-states.json*, that can be used with D3.

Find Shapefiles

So-called *shapefiles* predate the current explosion of online mapping and visualization. These are documents that essentially contain the same kind of information you could store in GeoJSON—boundaries of geographic areas, and points within those areas—but their contents are not plain text and, therefore, are very hard to read. The shapefile format grew out of the community of geographers, cartographers, and scientists using Geographic Information Systems (GIS) software. If you have easy access to expensive GIS software, then shapefiles are your best friend. But that's a small group of people, and web browsers can't make heads or tails of shapefiles, in any case.

If you can't find nice GeoJSON describing your area of interest, then hunt down a shapefile. Government websites are good sources, especially if you're looking for data on a specific country or region. My favorite two sources are:

[Natural Earth](#)

A massive collection of geographic data for both cultural/political and natural features made available in the public domain. Mapping countries is highly political, and Natural Earth posts detailed notes explaining their design decisions.

[The United States Census](#)

Boundary data for every US state, plus counties, roadways, water features, and so much more, also in the public domain.

Choose a Resolution

Before you download, check the *resolution* of the data. All shapefiles are vector data (not bitmap), so by resolution I don't mean pixels—I mean the level of *cartographic detail* or granularity.

Natural Earth's datasets come in three different resolutions, listed here from most detailed to least:

- 1:10,000,000
- 1:50,000,000
- 1:110,000,000

That is, in the highest resolution data, one unit of measurement in the file corresponds to 10 million such units in the actual, physical world. Or, to flip that around, every 10 million units in real life is simplified into one. So 10 million inches (158 miles) would translate to a single inch in data terms.

These resolution ratios can be written more simply as:

- 1:10m
- 1:50m
- 1:110m

For a low-detail ("zoomed out") map of the world, the 1:110m resolution could work fine. But to show detailed outlines of a specific state, 1:10m will be better. If you're mapping a very small ("zoomed in") area, such as a specific city or even neighborhood, then you'll need much higher resolution data. (Try local state or city government websites for that information.)

Different sources will offer different resolutions. Many of the US Census shapefiles are available in these three scales:

- 1:500,000 (1:500k)

- 1:5,000,000 (1:5m)
- 1:20,000,000 (1:20m)

Choose a resolution, and download the file. Typically you'll get a compressed ZIP file that contains several other files. As an example, I'm going to download Natural Earth's 1:110m (low detail) ocean file from here: <http://www.naturalearthdata.com/downloads/110m-physical-vectors/110m-ocean/>.

Uncompressed, it contains these files:

```
ne_110m_ocean.dbf
ne_110m_ocean.prj
ne_110m_ocean.README.html
ne_110m_ocean.shp
ne_110m_ocean.shx
ne_110m_ocean.VERSION.txt
```

Wow, those are some funky extensions. We're primarily interested in the file ending with *.shp*, but don't delete the rest just yet.

Simplify the Shapes

Ideally, you can find shapefiles in exactly the resolution you need. But what if you can only find a super-high-resolution file, say at 1:100k? The size of the file might be huge. And now that you're a JavaScript programmer, you're really concerned about efficiency, remember? So no sending multimegabyte geodata to the browser.

Fortunately, you can *simplify* those shapefiles, in effect converting them into lower-detail versions of themselves. The process of simplification is illustrated beautifully in [this interactive piece by Mike Bostock](#).

[MapShaper](#), by Matt Bloch, is an excellent, easy-to-use interactive tool for this purpose. It lets you upload your own shapefiles, and then drag sliders around to preview different levels of simplification. Typically, the trade-off is between good detail and small files.

If you use MapShaper, choose the "Shapefile – polygons" option when exporting. This will generate both a *.shp* and a *.shx* file for you. Download both files, and rename them so the filenames match the names of the original *.shp* and *.shx* files. Then, before converting to GeoJSON, make a copy of the original *.dbf* file, and put it in the same folder as the simplified shapefiles. This important step will ensure you don't lose any of the important metadata that's stored in the *.dbf*, like country IDs and other path identifiers.

Other options include Mike Migurski's [Bloch](#) with a Python implementation of Matt Bloch's simplification algorithms, and a `d3.simplify` plug-in (used for Mike's demo mentioned earlier). The dream is one day we could use JavaScript to perform line simplification directly, and then export that simplified JSON to use in projects. The tools are evolving quickly, so watch this space! (Literally while I was writing this paragraph, Mr. Bostock posted [a demo](#) for a new project for geometry simplification, [TopoJSON](#). That's how fast things are changing! By the time you read this, the TopoJSON command-line tool may be your best bet, as it can load shapefiles, perform simplification, *and* convert your data to JSON. As you'd expect, TopoJSON is designed to play well with D3, although it outputs a new TopoJSON format, which is similar to, but more efficient than GeoJSON.)

Convert to GeoJSON

If you don't already have the right software installed, this can be the hairiest part of the process. Our end goal is to get a terminal command called `ogr2ogr` running on your Mac, Unix, or Windows system. The problem is that `ogr2ogr` depends on several other frameworks, libraries, and so on, so for it to work, they all have to be in place.

I won't cover the intricacies of the installation here, but I'll point you in the right direction.

First, you need [the Geospatial Data Abstraction Library](#), or GDAL. The `ogr2ogr` utility is part of this package.

You also need [GEOS](#), which cleverly stands for Geometry Engine, Open Source.

If you have a Windows or Unix/Linux machine, you can now have fun downloading the source and installing it by typing funny commands like **build**, **make**, and **seriously why isn't this working omg please please please install this time or i will freak out**.

I don't remember the exact command names, but they are something like that. (On a serious note, if you get stuck on this step, be aware that there are literally entire other O'Reilly books on how to download and install software packages like this.)

If you're on a Mac, and you happen to have both Xcode *and* Homebrew installed, then simply type **brew install gdal** into the terminal, and you're done! (If you don't have either of these amazing tools, they might be worth getting. Both tools are free, but could take some time and effort on your part to install. Xcode is a massive [download from the App Store](#). Once you have Xcode, Homebrew can, in theory, be installed with [a simple terminal command](#). In my experience, some troubleshooting was required to get it working.)

To Mac users without Xcode or Homebrew: you are very lucky that some kind soul has precompiled a friendly GUI installer, which installs GDAL, GEOS, and several other tools whose names you don't really need to know. Look for the newest version of the "[GDAL Complete](#)" package. Review the GDAL ReadMe file closely. After installation, you won't automatically be able to type **ogr2ogr** in a terminal window. You'll need to add the GDAL programs to your shell path. The easiest way to do that is: Open a new terminal window. Type **nano .bash_profile**. Paste in `export PATH=/Library/Frameworks/GDAL.framework/Programs:$PATH` and then Control-X and Control-y to save. Type **exit** to end the session, then open a new terminal window and type **ogr2ogr** to see if it worked.

No matter what system you're on, once all the tools are installed, open up a new terminal window and navigate to whatever directory contains all the shapefiles (for example, **cd ~/ocean_shapes/**.) Then use the form:

```
ogr2ogr -f "GeoJSON" output.json filename.shp
```

This tells `ogr2ogr` to take `filename`, which should be a `.shp` file, convert it to GeoJSON, and finally save it as a file called `output.json`.

For my sample ocean data file, using `ogr2ogr` looks like the following:

```
ogr2ogr -f "GeoJSON" output.json ne_110m_ocean.shp
```

Type that in, and hopefully you see nothing at all.

So anticlimactic! I know, after the hours you spent hacking the command line to get ol' `ogr` installed, you were expecting some gradiose finale, as though you saved the princess in Super Mario 3, yet again. (I actually

have never done that, but I imagine it is pretty amazing.)

But no, hopefully nothing happened at all. Except there should be a new file in the same directory called *output.json*.

Here's the start of mine:

```
{
  "type": "FeatureCollection",
  "features": [ { "type": "Feature", "properties":
    { "scalerank": 0, "featurecla": "Ocean" },
  "geometry": { "type": "Polygon", "coordinates":
    [ [ [ 49.110290527343778, 41.28228759765625 ],
      [ 48.584472656250085, 41.80889892578125 ],
      [ 47.492492675781335, 42.9866943359375 ],
      [ 47.590881347656278, 43.660278320312528 ],
      [ 46.682128906250028, 44.609313964843807 ],
      [ 47.675903320312585, 45.641479492187557 ],
      [ 48.645507812500085, 45.806274414062557 ]
    ]
    ...
  }
```

Hey, finally, this is starting to look familiar!

Excitedly, now we copy our new GeoJSON into our D3 directory. I rename mine *oceans.json*, copy an earlier HTML document, and in the D3 code, simply change the reference from `us-states.json` to `oceans.json`, and then get the result shown in [Figure 12-8](#).



Figure 12-8. GeoJSON showing, um, the world's oceans?

Blaaa! What is that?! Whatever it is, you can see it in *o8_oceans.html*.

In my haste, I forgot to update the projection! Let's make one tiny change, from `albersUsa` to `mercator` (see [Figure 12-9](#)).



Figure 12-9. GeoJSON of the world's oceans, now properly projected

See the result in *o9_mercator.html*—oceanic GeoJSON paths, downloaded, parsed, and visualized.

© 2013, O'Reilly Media, Inc.

- [Terms of Service](#)
- [Privacy Policy](#)
- Interested in [sponsoring content?](#)