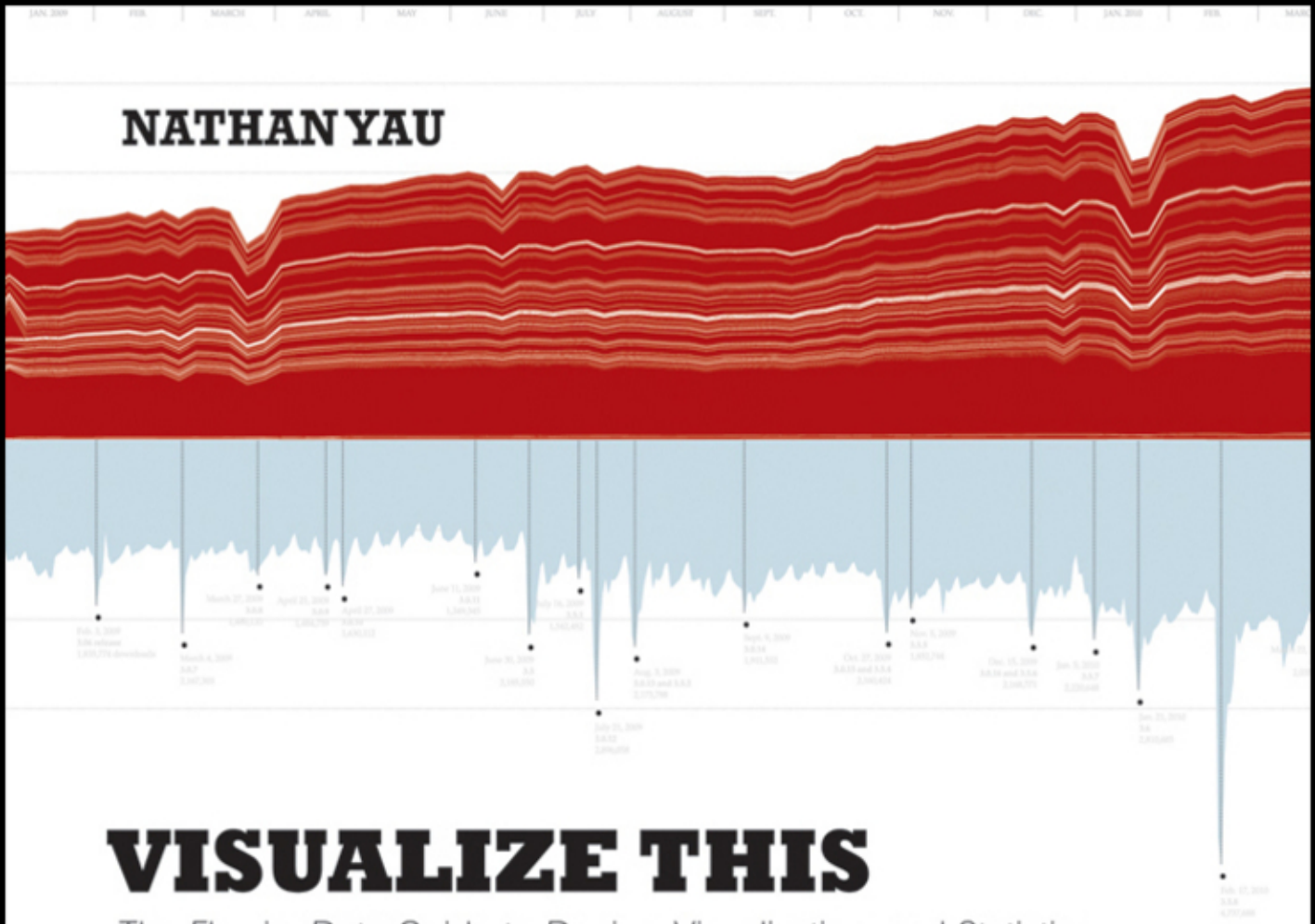
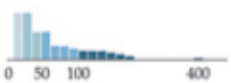
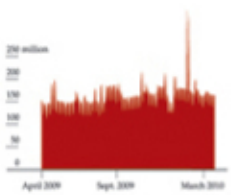
NATHAN YAU

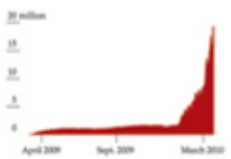


VISUALIZE THIS

The FlowingData Guide to Design, Visualization, and Statistics







Visualizing Spatial Relationships

Maps are a subcategory of visualization that have the added benefit of being incredibly intuitive. Even as a kid, I could read them. I remember sitting in the passenger seat of my dad's car and sounding off directions as I read the fantastically big unfolded map laid out in front of me. An Australian lady with a robotic yet calming voice spits out directions from a small box on the dash nowadays.

In any case, maps are a great way to understand your data. They are scaled down versions of the physical world, and they're everywhere. In this chapter you dive into several spatial datasets, looking for patterns over space and time. You create some basic maps in R and then jump to more advanced mapping with Python and SVG. Finally, you round it up with interactive and animated maps in ActionScript and Flash.

What to Look For

You read maps much the same way that you read statistical graphics. When you look at specific locations on a map, you still look for clustering in specific regions or for example, compare one region to the rest of a country. The difference is that instead of x- and y-coordinates, you deal with latitude and longitude. The coordinates on a map actually relate to each other in the same way that one city relates to another. Point A and Point B are a specific number of miles away, and it takes an estimated time to get there. In contrast, the distance on a dot plot is abstract and (usually) has no units.

This difference brings with it a lot of subtleties to maps and cartography. There's a reason *The New York Times* has a group of people in its graphics department who exclusively design maps. You need to make sure all your locations are placed correctly, colors make sense, labels don't obscure locations, and that the right projection is used.

This chapter covers only a handful of the basics. These can actually take you pretty far in terms of finding stories in your data, but keep in mind there's a whole other level of awesome that you can strive for.

Things can get especially interesting when you introduce time. One map represents a slice in time, but you can represent multiple slices in time with several maps. You can also animate changes to, say, watch growth (or decline) of a business across a geographic region. Bursts in specific areas become obvious, and if the map is interactive, readers can easily focus in on their area to see how things have changed. You don't get the same effect with bar graphs or dot plots, but with maps, the data can become instantly personal.

Specific Locations

A list of locations is the easiest type of spatial data you'll come across. You have the latitude and longitude for a bunch places, and you want to map them. Maybe you want to show where events, such as crime, occurred, or you want to find areas where points are clustered. This is straightforward to do, and there are a lot of ways to do it.

On the web, the most common way to map points is via Google or Microsoft Maps. Using their mapping APIs, you can have an interactive map that you can zoom and pan in no time with just a few lines of JavaScript. Tons of tutorials and excellent documentation are online on how to make use of these APIs, so I'll leave that to you.

However, there is a downside. You can only customize the maps so much, and in the end you'll almost always end up with something that still looks like a Google or Microsoft map. I'm not saying they're ugly, but when you're developing an application or designing a graphic that fits into a publication, it's often more fitting to have a map that matches your design scheme. There are sometimes ways to get around these barriers, but it's not worth the effort if you can just do the same thing but better, with a different tool.

NOTE

Google and Microsoft provide super straightforward tutorials that start with their mapping APIs, so be sure to check those out if you're interested in taking advantage of some basic mapping functionality.

Find Latitude and Longitude

Before you do any mapping, consider the available data and the data that you actually need. If you don't have the data you need, then there's nothing to visualize, right? In most practical applications, you need latitude and longitude to map points, and most datasets don't come like that. Instead, you most likely will have a list of addresses.

As much as you might want to, you can't just plug in street names and postal codes and expect a pretty map. You have to get latitude and longitude first, and for that, turn to *geocoding*. Basically, you take an address, give it to a service, the service queries its database for matching addresses, and then you get latitude and longitude for where the service thinks your address is located in the world.

As for which service to use, well, there are many. If you have only a few locations to geocode, it's easy to just go to a website and manually enter them. Geocoder.us is a good free option for that. If you don't need your locations to be exact, you can try Pierre Gorissen's *Google Maps Latitude Longitude Popup*. It's a simple Google Maps interface that spits out latitude and longitude for anywhere you click on the map.

If, however, you have a lot of locations to geocode, then you should do it programmatically. You don't need to waste your time copying and pasting. Google, Yahoo!, Geocoder.us, and Mediawiki all provide APIs for gecoding; and Geopy, a geocoding toolbox for Python, wraps them all up into one package.

USEFUL GEOCODING TOOLS

- Geocoder.us, `http://geocoder.us`—Provides a straightforward interface to copy and paste location to get latitude and longitude. Also provides an API.
- Latitude Longitude Popup, `www.gorissen.info/Pierre/maps/`—Google Maps' mashup. Click a location on the map, and it gives you latitude and longitude.
- Geopy, `http://code.google.com/p/geopy/`—Geocoding toolbox for Python. Wraps up multiple geocoding APIs into a single package.

Visit the Geopy project page for instructions on how to install the package. There are also lots of straightforward examples on how to start. The following example assumes you have already installed the package on your computer.

NOTE

Visit `http://code.google.com/apis/maps/signup.html` to sign up for a free API key for the Google Maps API. It's straightforward and takes only a couple of minutes.

After you install Geopy, download location data at `http://book.flowingdata.com/ch08/geocode/costcos-limited.csv`. This is a CSV file that contains the address of every Costco warehouse in the United States, but it doesn't have latitude or longitude coordinates. That's up to you.

Open a new file and save it as `geocode-locations.py`. As usual, import the packages that you need for the rest of the script.

```
from geopy import geocoders
import csv
```

You also need an API key for each service you want to use. For the purposes of this example, you only need one from Google.

Store your API key in a variable named `g_api_key`, and then use it when you instantiate the geocoder.

```
g_api_key = 'INSERT_YOUR_API_KEY_HERE'
g = geocoders.Google(g_api_key)
```

Load the `costcos-limited.csv` data file, and then loop. For each row, you piece together the full address and then plug it in for geocoding.

```
costcos = csv.reader(open('costcos-limited.csv'), delimiter=',')
next(costcos) # Skip header

# Print header
```

```
print "Address,City,State,Zip Code,Latitude,Longitude"
for row in costcos:

    full_addy = row[1] + "," + row[2] + "," + row[3] + "," + row[4]
    place, (lat, lng) = list(g.geocode(full_addy, exactly_one=False))[0]
    print full_addy + "," + str(lat) + "," + str(lng)
```

That's it. Run the Python script, and save the output as costcos-geocoded.csv. Here's what the first few lines of the data looks like:

```
Address,City,State,Zip Code,Latitude,Longitude
1205 N. Memorial Parkway,Huntsville,Alabama,35801-5930,34.7430949,-86
.6009553
3650 Galleria Circle,Hoover,Alabama,35244-2346,33.377649,-86.81242
8251 Eastchase Parkway,Montgomery,Alabama,36117,32.363889,-86.150884
5225 Commercial Boulevard,Juneau,Alaska,99801-7210,58.3592,-134.483
330 West Dimond Blvd,Anchorage,Alaska,99515-1950,61.143266,-149.884217
...
```

Pretty cool. By some stroke of luck, latitude and longitude coordinates are found for every address. That usually doesn't happen. If you do run into that problem, you should add error checking at the second to last line of the preceding script.

```
    try:
        place, (lat, lng) = list(g.geocode(full_addy, exactly_one=False))
[0]
        print full_addy + "," + str(lat) + "," + str(lng)
    except:
        print full_addy + ",NULL,NULL"
```

This tries to find the latitude and longitude coordinates, and if it fails, it prints the row with the address and null coordinate values. Run the Python script and save the output as a file, and you can go back and look for the nulls. You can either try a different service for the missing addresses via Geopy, or you can just manually enter the addresses in Geocoder.us.

Just Points

Now that you have points with latitude and longitude, you can map them. The straightforward route is to do the computer equivalent of putting pushpins in a paper map on a billboard. As shown in the framework in Figure 8-1, you place a marker for each location on the map.

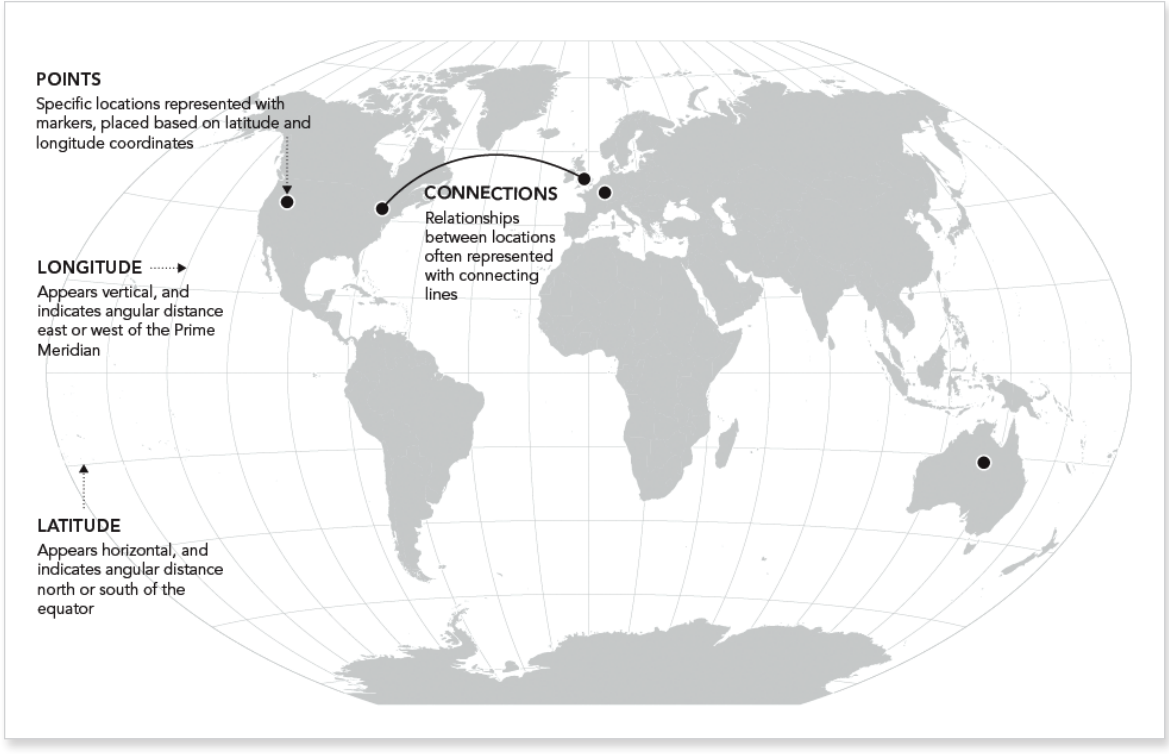


FIGURE 8-1 Mapping points framework

Although a simple concept, you can see features in the data such as clustering, spread, and outliers.

MAP WITH DOTS

R, although limited in mapping functionality, makes placing dots on a map easy. The `maps` package does most of the work. Go ahead and install it via the Package Installer, or use `install.packages()` in the console. When installed, load it into the workspace.

```
library(maps)
```

Next step: Load the data. Feel free to use the Costco locations that you just geocoded, or for convenience, I've put the processed dataset online, so you can load it directly from the URL.

```
costcos <-
  read.csv("http://book.flowingdata.com/ch08/geocode/costcos-geocoded
  .csv", sep=",")
```

Now on to mapping. When you create your maps, it's useful to think of them as layers (regardless of the software in use). The bottom layer is usually the base map that shows geographical boundaries, and then you place data layers on top of that. In this case the bottom layer is a map of the United States, and the second layer is Costco locations. Here's how to make the first layer, as shown in Figure 8-2.

```
map(database="state")
```

FIGURE 8-2 Plain map of the United States

The second layer, or Costco's, are then mapped with the `symbols()` function. This is the same function you used to make the bubble plots in Chapter 6, "Visualizing Relationships," and you use it in the same way, except you pass latitude and longitude instead of x- and y-coordinates. Also set *add* to *TRUE* to indicate that you want symbols to be added to the map rather than creating a new plot.

```
symbols(costcos$Longitude, costcos$Latitude,
  circles=rep(1, length(costcos$Longitude)), inches=0.05, add=TRUE)
```

Figure 8-3 shows the results. All the circles are the same size because you set *circles* to an array of ones with the length equal to the number of locations. You also set *inches* to 0.05, which sizes the circles to that number. If you want smaller markers, all you need to do is decrease that value.

FIGURE 8-3 Map of Costco locations

As before, you can change the colors of both the map and the circles so that the locations stand out and boundary lines sit in the background, as shown in Figure 8-4. Now change the dots to a nice Costco red and the state boundaries to a light gray.

```
map(database="state", col="#cccccc")
symbols(costcos$Longitude, costcos$Latitude, bg="#e2373f", fg="#ffffff",
  lwd=0.5, circles=rep(1, length(costcos$Longitude)),
  inches=0.05, add=TRUE)
```

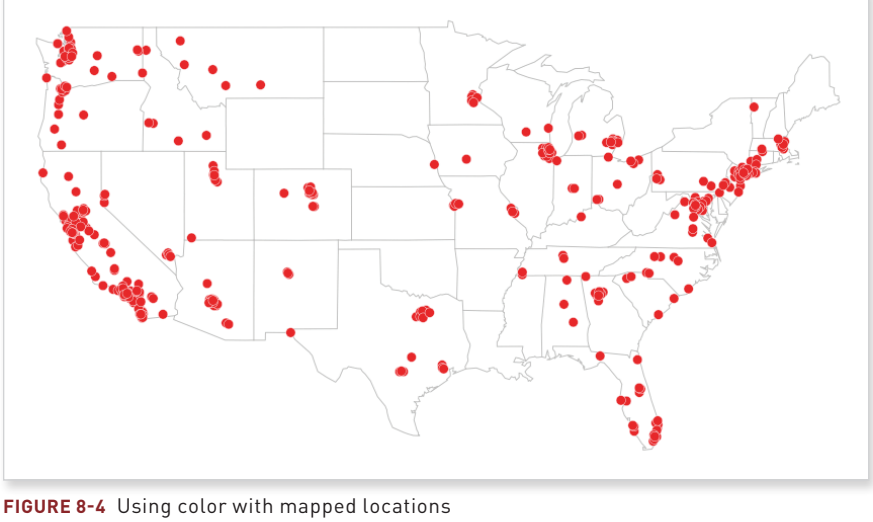


FIGURE 8-4 Using color with mapped locations

In Figure 8-3, the unfilled circles and the map were all the same color and line width, so everything blended together, but with the right colors, you can make the data sit front and center.

It's not bad for a few lines of code. Costco has clearly focused on opening locations on the coasts with clusters in southern and northern California, northwest Washington, and in the northeast of the country.

However, there is a glaring omission here. Well, two of them actually. Where are Alaska and Hawaii? They're part of the United States, too, but are nowhere to be found even though you use the "state" database with `map()`. The two states are actually in the "world" database, so if you want to see Costco locations in Alaska in Hawaii, you need to map the entire world, as shown in Figure 8-5.

```
map(database="world", col="#cccccc")
symbols(costcos$Longitude, costcos$Latitude, bg="#e2373f", fg="#ffffff",
   lwd=0.3, circles=rep(1, length(costcos$Longitude)),
   inches=0.03, add=TRUE)
```

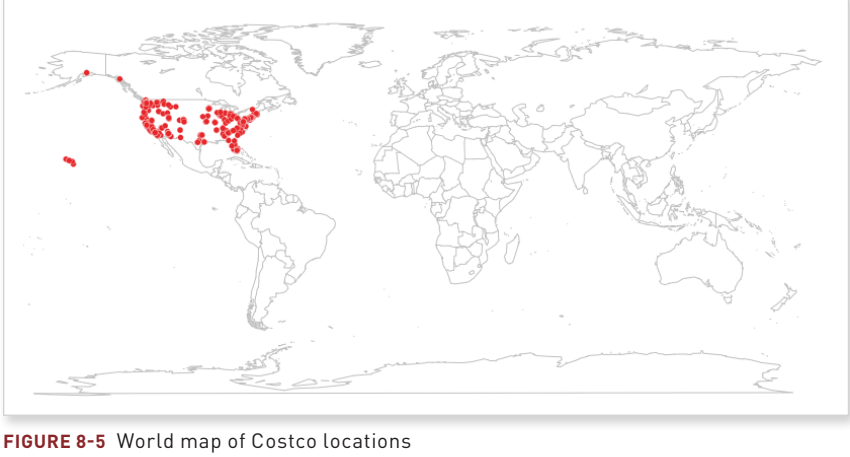


FIGURE 8-5 World map of Costco locations

It's a waste of space, I know. There are options that you can mess around with, which you can find in the documentation, but you can edit the rest in Illustrator to zoom in on the United States or remove the other countries from view.

TIP

With R, when in doubt, always jump to the documentation for the function or package you're stuck on by preceding the name with a question mark.

Taking the map in the opposite direction, say you want to only map Costco locations for a few states. You can do that with the *region* argument.

```
map(database="state", region=c("California", "Nevada", "Oregon",
  "Washington"), col="#cccccc")
symbols(costcos$Longitude, costcos$Latitude, bg="#e2373f", fg="#ffffff",
  lwd=0.5, circles=rep(1, length(costcos$Longitude)), inches=0.05,
  add=TRUE)
```

As shown in Figure 8-6, you create a bottom layer with California, Nevada, Oregon, and Mexico. Then you create the data layer on top of that. Some dots are not in any of those states, but they're in the plotting region, so they still appear. Again, it's trivial to remove those in your favorite vector editing software.

MAP WITH LINES

In some cases it could be useful to connect the dots on your map if the order of the points have any relevance. With online location services such as Foursquare growing in popularity, location traces aren't all that rare. An easy way to draw lines is, well, with the `lines()` function. To demonstrate, look at locations I traveled during my seven days and nights as a spy for the fake government of Fakesville. Start with loading the data (as usual) and drawing a base world map.

```
faketrace <-
    read.csv("http://book.flowingdata.com/ch08/points/fake-trace.txt",
      sep="\t")
map(database="world", col="#cccccc")
```

Take a look at the data frame by entering **faketrace** in your R console. You see that it's just two columns for latitude and longitude and eight data points. You can assume that the points are already in the order that I traveled during those long seven nights.

```
   latitude    longitude
1  46.31658     3.515625
2  61.27023    69.609375
3  34.30714   105.468750
4 -26.11599   122.695313
5 -30.14513    22.851563
6 -35.17381   -63.632813
7  21.28937   -99.492188
8  36.17336  -115.180664
```

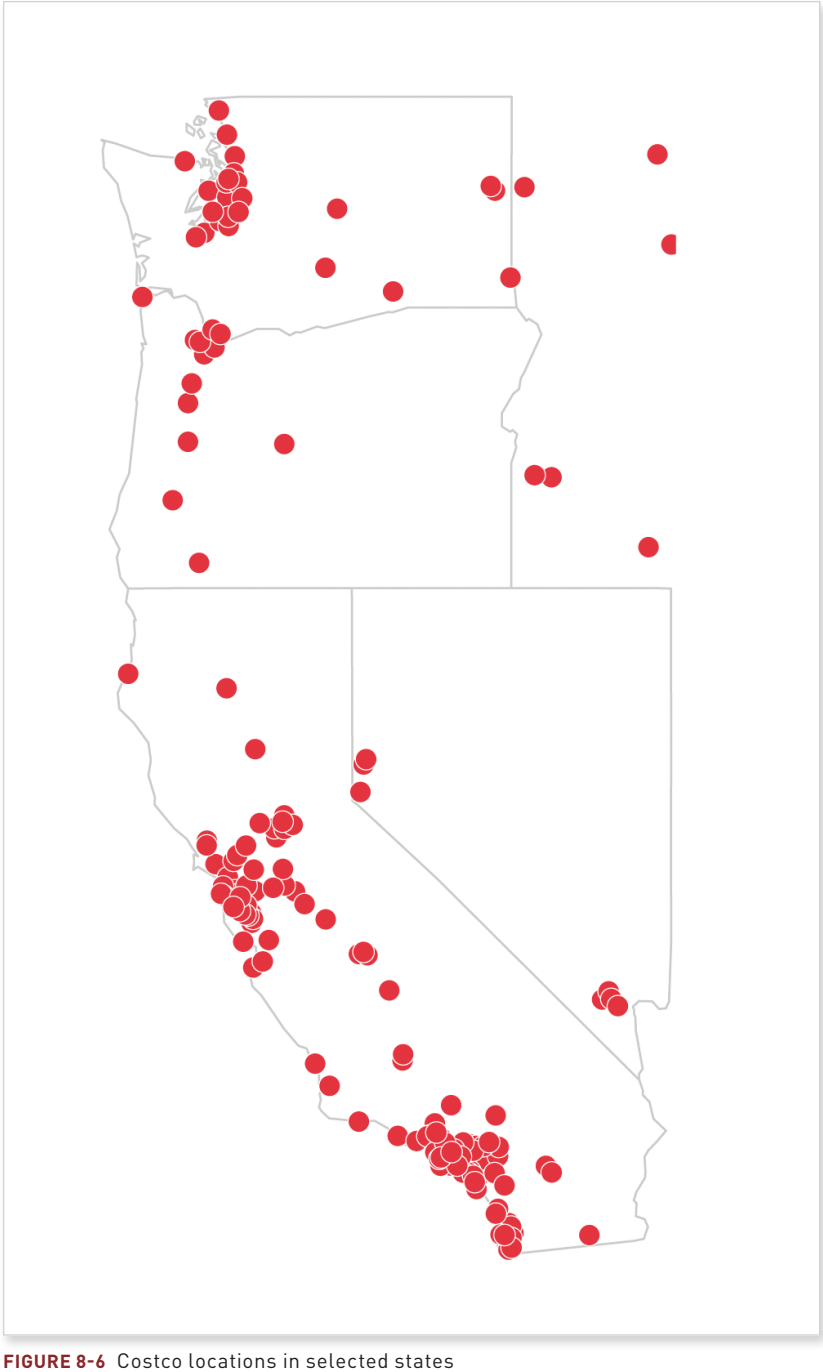


FIGURE 8-6 Costco locations in selected states

Draw the lines by simply plugging in the two columns into `lines()`. Also specify color (*col*) and line width (*lwd*).

```
lines(faketrace$longitude, faketrace$latitude, col="#bb4cd4", lwd=2)
```

Now also add dots, exactly like you just did with the Costco locations, for the graphic in Figure 8-7.

```
symbols(faketrace$longitude, faketrace$latitude, lwd=1, bg="#bb4cd4",
fg="#ffffff", circles=rep(1, length(faketrace$longitude)), inches=0.05,
add=TRUE)
```

FIGURE 8-7 Drawing a location trace

After those seven days and nights as a spy for the Fakesville government, I decided it wasn't for me. It's just not as glamorous as James Bond makes it out to be. However, I did make connections in all the countries I visited. It could be interesting to draw lines from my location to all the others, as shown in Figure 8-8.

```
map(database="world", col="#cccccc")
for (i in 2:length(faketrace$longitude)-1) {
        lngs <- c(faketrace$longitude[8], faketrace$longitude[i])
        lats <- c(faketrace$latitude[8], faketrace$latitude[i])
        lines(lngs, lats, col="#bb4cd4", lwd=2)
}
```

FIGURE 8-8 Drawing worldwide connections

After you create the base map, you can loop through each point and draw a line from the last point in the data frame to every other location. This isn't incredibly informative, but maybe you can find a good use for it. The point here is that you can draw a map and then use R's other graphics functions to draw whatever you want using latitude and longitude coordinates.

By the way, I wasn't actually a spy for Fakesville. I was just kidding about that.

Scaled Points

Switching gears back to real data and a more interesting topic than my fake spy escapades, more often than not, you don't just have locations. You also have another value attached to locations such as sales for a business or city population. You can still map with points, but you can take the principles of the bubble plot and use it on a map.

I don't have to explain how bubbles should be sized by area and not radius again, right? Okay, cool.

MAP WITH BUBBLES

In this example, look at adolescent fertility rate as reported by the United Nations Human Development Report—that is, the number of births per 1,000 women aged 15 to 19 in 2008. The geo-coordinates were

provided by GeoCommons. You want to size bubbles in proportion to these rates.

The code is almost the same as when you mapped Costco locations, but remember you just passed a vector of ones for circle size in the `symbols()` function. Instead, we use the `sqrt()` of the rates to indicate size.

```
fertility <-
    read.csv("http://book.flowingdata.com/ch08/points/adol-fertility.csv")
map('world', fill = FALSE, col = "#cccccc")
symbols(fertility$longitude, fertility$latitude,
    circles=sqrt(fertility$ad_fert_rate), add=TRUE,
    inches=0.15, bg="#93ceef", fg="#ffffff")
```

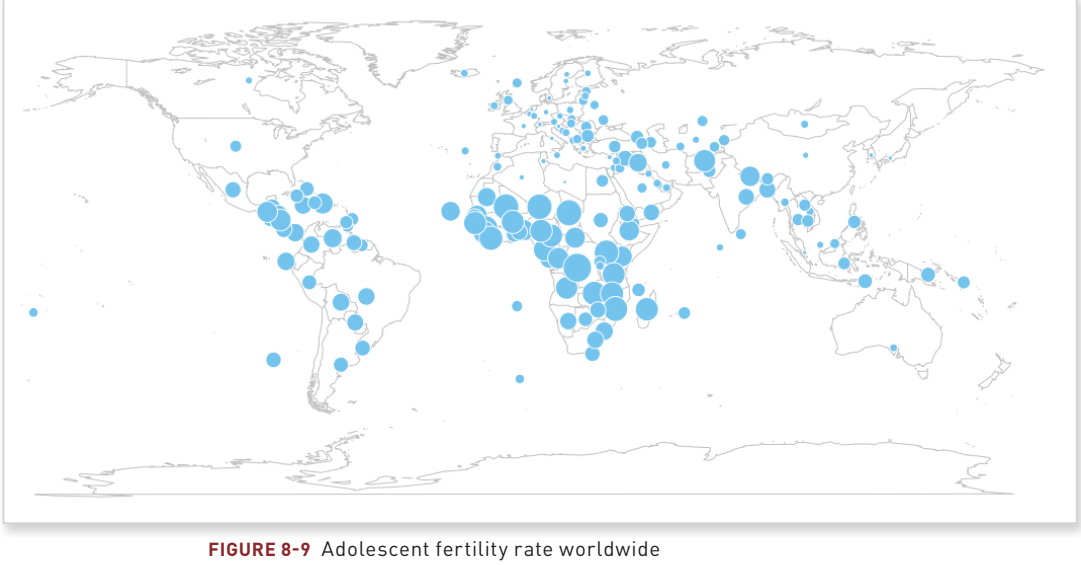


FIGURE 8-9 Adolescent fertility rate worldwide

Figure 8-9 shows the output. Immediately, you should see that African countries tend to have the highest adolescent fertility rates, whereas European countries have relatively lower rates. From the graphic alone, it's not clear what value each circle represents because there is no legend. A quick look with `summary()` in R can tell you more.

```
summary(fertility$ad_fert_rate)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
   3.20   16.20   39.00   52.89   78.20  201.40    1.00
```

That's fine for us, an audience of one, but you need to explain more if you want others, who haven't looked at the data, to understand the graphic. You can add annotation to highlight countries with the highest and lowest fertility rates, point out the country where most readers will be from (in this case, the United States), and provide a lead-in to set up readers for what they're going to look at. Figure 8-10 shows these changes.

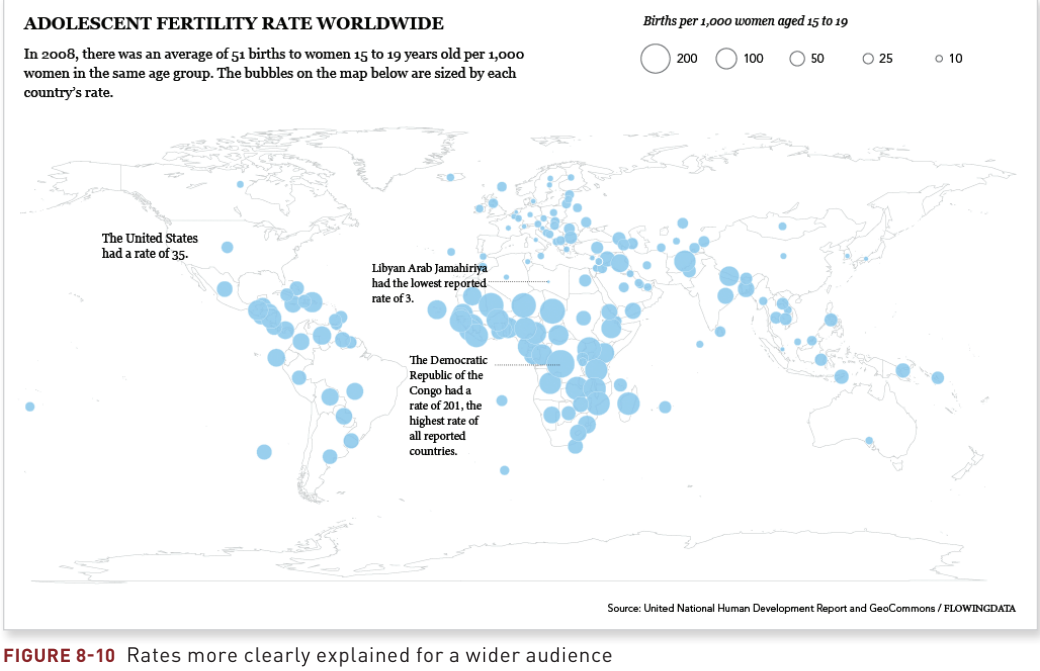


FIGURE 8-10 Rates more clearly explained for a wider audience

Regions

Mapping points can take you only so far because they represent only single locations. Counties, states, countries, and continents are entire regions with boundaries, and geographic data is usually aggregated in this way. For example, it's much easier to find health data for a state or a country than it is for individual patients or hospitals. This is usually done for privacy, whereas other times aggregated data is just easier to distribute. In

any case, this is usually how you're going to use your spatial data, so now learn to visualize it.

Color by Data

Choropleth maps are the most common way to map regional data. Based on some metric, regions are colored following a color scale that you define, as shown in Figure 8-11. The areas and location are already defined, so your job is to decide the appropriate color scales to use.

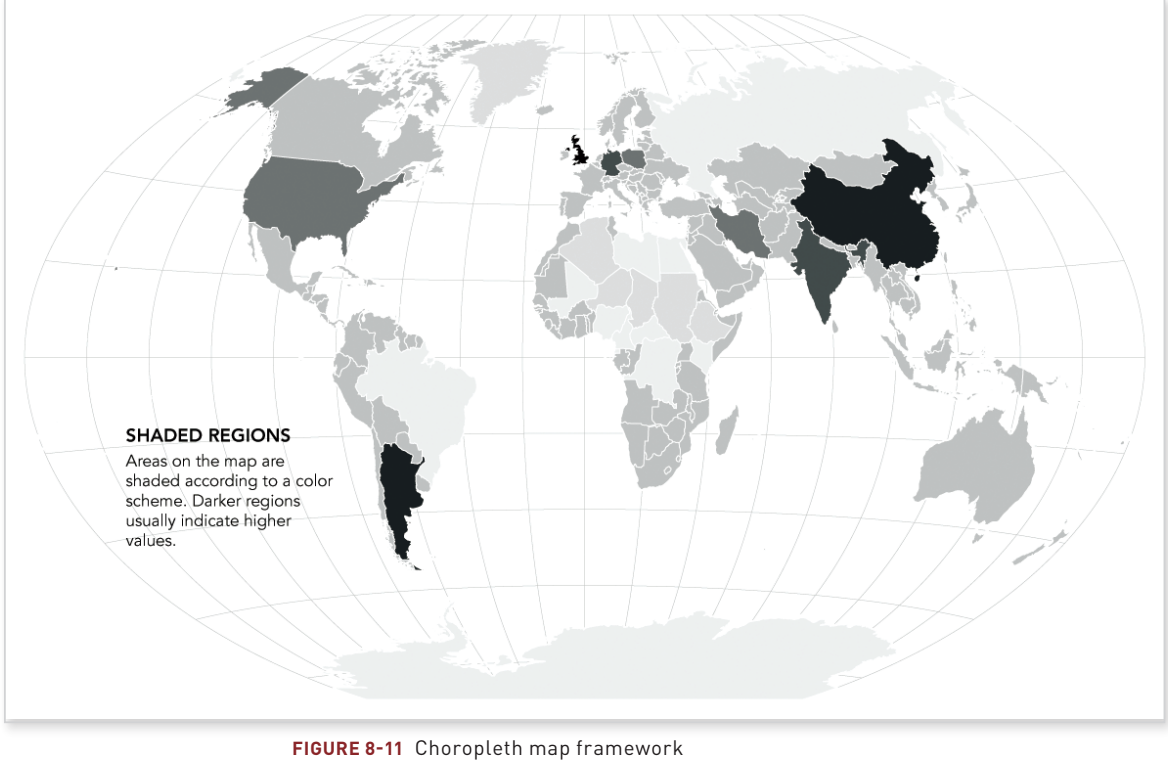


FIGURE 8-11 Choropleth map framework

As touched on in a previous chapter, Cynthia Brewer's ColorBrewer is a great way to pick your colors, or at least a place to start to design a color palette. If you have continuous data, you might want a similarly continuous color scale that goes from light to dark, but all with the same hue (or multiple similar hues), as shown in Figure 8-12.

A diverging color scheme, as shown in Figure 8-13, might be good if your data has a two-sided quality to it, such as good and bad or above and below a threshold.

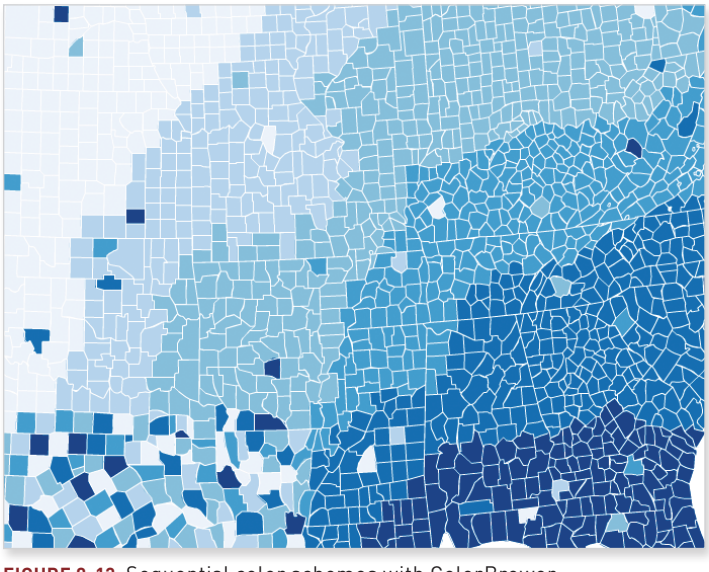


FIGURE 8-12 Sequential color schemes with ColorBrewer

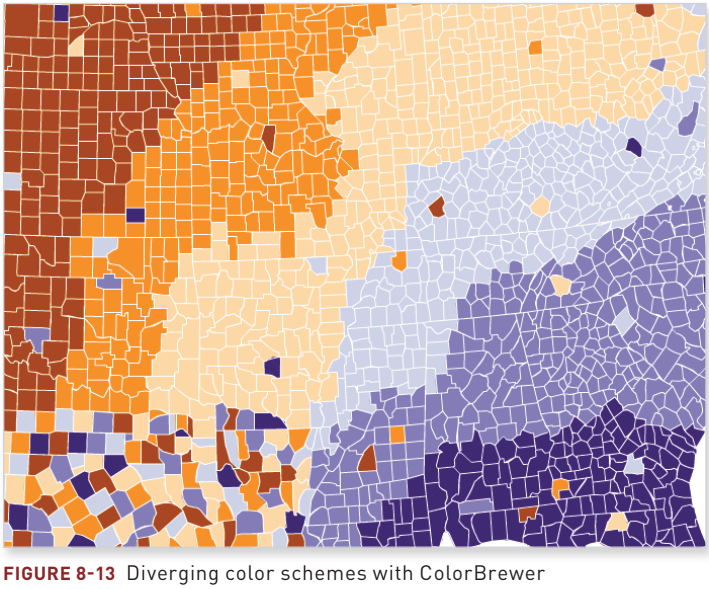


FIGURE 8-13 Diverging color schemes with ColorBrewer

Finally, if your data is qualitative with classes or categories, then you might want a unique color for each (Figure 8-14).

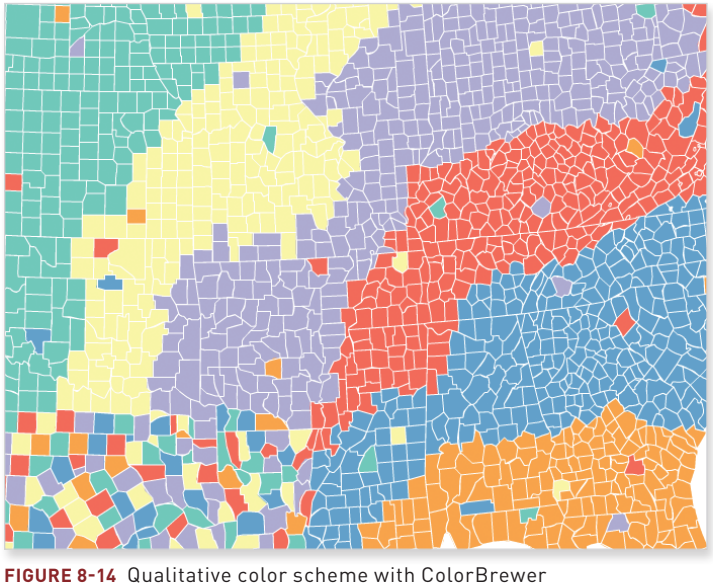


FIGURE 8-14 Qualitative color scheme with ColorBrewer

When you have your color scheme, you have two more things to do. The first is to decide how the colors you picked match up to the data range, and the second is to assign colors to each region based on your choice. You can do both with Python and Scalable Vector Graphics (SVG) in the following examples.

MAP COUNTIES

The U.S. Bureau of Labor Statistics provides county-level unemployment data every month. You can download the most recent rates or go back several years from its site. However, the data browser it provides is kind of outdated and roundabout, so for the sake of simplicity (and in case the BLS site changes), you can download the data at `http://book.flowingdata.com/ch08/regions/unemployment-aug2010.txt`. There are six columns. The first is a code specific to the Bureau of Labor Statistics. The next two together are a unique id specifying county. The fourth and fifth columns are the county name and month the rate is an estimate of, respectively. The last column

is the estimated percentage of people in the county who are unemployed. For the purposes of this example, you care only about the county id (that is, FIPS codes) and the rate.

Now for the map. In previous examples, you generated base maps in R, but now you can use Python and SVG to do this. The former is to process the data, and the latter is for the map itself. You don't need to start completely from scratch, though. You can get a blank map from Wikimedia Commons found here: `http://commons.wikimedia.org/wiki/File:USA_Counties_with_FIPS_and_names.svg`, as shown in Figure 8-15. The page links to the map in four sizes in PNG format and then one in SVG. You want the SVG one. Download the SVG file and save it as `counties.svg`, in the same directory that you save the unemployment data.

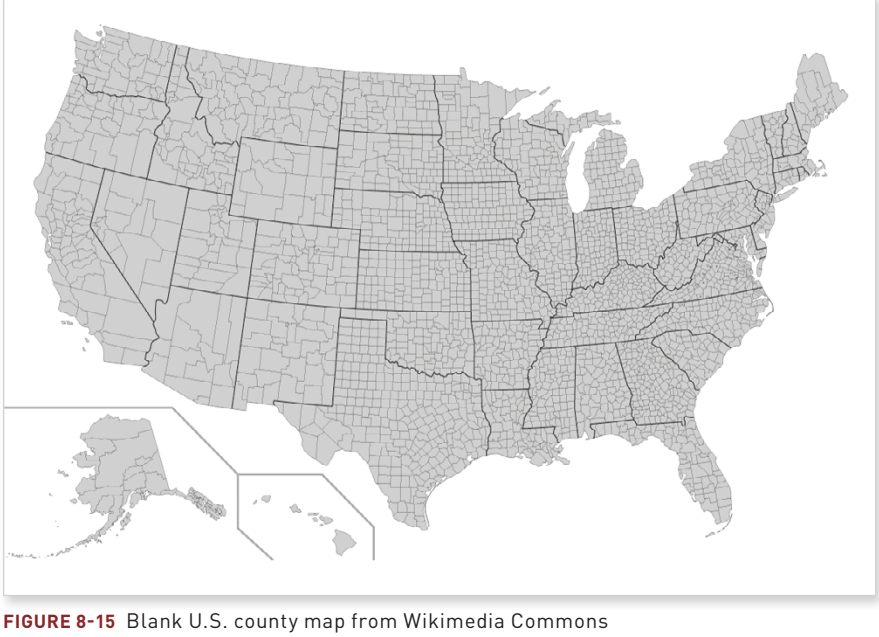


FIGURE 8-15 Blank U.S. county map from Wikimedia Commons

The important thing here, if you're not familiar with SVG, is that it's actually an XML file. It's text with tags, and you can edit it in a text editor like you would an HTML file. The browser or image viewer reads the XML, and the XML tells the browser what to show, such as the colors to use and shapes to draw.

TIP

SVG files are XML files, which are easy to change in a text editor. This also means that you can parse the SVG code to make changes programmatically.

To drive the point home, open the map SVG file in a text editor to see what you're dealing with. It's mostly SVG declarations and boiler plate stuff, which you don't totally care about right now.

Scroll down some more to start to see some <path> tags, as shown in Figure 8-16. All those numbers in a single tag specify the boundaries of a county. You're not going to touch those. You're interested in changing the fill color of each county to match the corresponding unemployment rate. To do that, you need to edit the *style* in the path.

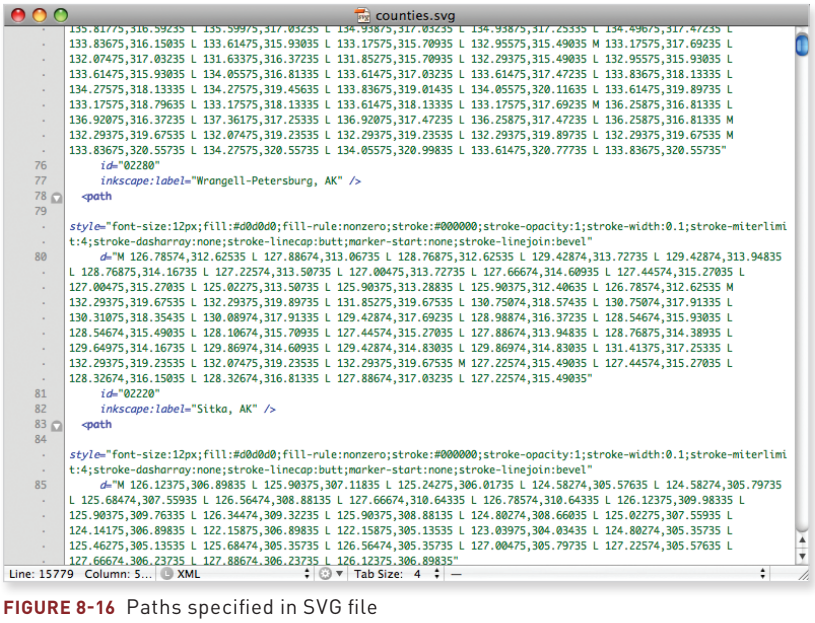


FIGURE 8-16 Paths specified in SVG file

Notice how each <path> starts with *style*? Those who have written CSS can immediately recognize this. There is a *fill* attribute followed by a hexadecimal color, so if you change that in the SVG file, you change the color of the output image. You could edit each one manually, but there are more than 3,000 counties. That would take way too long. Instead, come back to your old friend Beautiful Soup, the Python package that makes parsing XML and HTML relatively easy.

Open a blank file in the same directory as your SVG map and unemployment data. Save it as `colorize_svg.py`. You need to import the CSV data file

and parse the SVG file with Beautiful Soup, so start by importing the necessary packages.

```
import csv
from BeautifulSoup import BeautifulSoup
```

Then open the CSV file and store it so that you can iterate through the rows using `csv.reader()`. Note that the "r" in the `open()` function just means that you want to open the file to read its contents, as opposed to writing new rows to it.

```
reader = csv.reader(open('unemployment-aug2010.txt', 'r'), delimiter=",")
```

Now also load the blank SVG county map.

```
svg = open('counties.svg', 'r').read()
```

Cool, you loaded everything you need to create a choropleth map. The challenge at this point is that you need to somehow link the data to the SVG. What is the commonality between the two? I'll give you a hint. It has to do with each county's unique id, and I mentioned it earlier. If you guessed FIPS codes, then you are correct!

Each path in the SVG file has a unique id, which happens to be the combined FIPS state and county FIPS code. Each row in the unemployment data has the state and county FIPS codes, too, but they're separate. For example, the state FIPS code for Autauga County, Alabama, is 01, and its county FIPS code is 001. The path id in the SVG are those two combined: 01001.

TIP

Paths in SVG files, geographic ones in particular, usually have a unique id. It's not always FIPS code, but the same rules apply.

You need to store the unemployment data so that you can retrieve each county's rate by FIPS code, as we iterate through each path. If you start to become confused, stay with me; it'll be clearer with actual code. But the main point here is that the FIPS codes are the common bond between your SVG and CSV, and you can use that to your advantage.

To store the unemployment data so that it's easily accessible by FIPS code later, use a construct in Python called a dictionary. It enables you to store and retrieve values by a keyword. In this case, your keyword is a combined state and county FIPS code, as shown in the following code.

```
unemployment = {}
min_value = 100; max_value = 0
for row in reader:
```

```
    try:
        full_fips = row[1] + row[2]
        rate = float( row[8].strip() )
        unemployment[full_fips] = rate
    except:
        pass
```

Next parse the SVG file with BeautifulSoup. Most tags have an opening and closing tag, but there are a couple of self-closing tags in there, which you need to specify. Then use the `findAll()` function to retrieve all the paths in the map.

```
soup = BeautifulSoup(svg, selfClosingTags=['defs','sodipodi:namedview'])
paths = soup.findAll('path')
```

Then store the colors, which I got from ColorBrewer, in a Python list. This is a sequential color scheme with multiple hues ranging from purple to red.

```
colors = ["#F1EEF6", "#D4B9DA", "#C994C7", "#DF65B0", "#DD1C77", "#980043"]
```

You're getting close to the climax. Like I said earlier, you're going to change the style attribute for each path in the SVG. You're just interested in fill color, but to make things easier, you can replace the entire style instead of parsing to replace only the color. I changed the hexadecimal value after `stroke` to `#ffffff`, which is white. This changes the borders to white instead of the current gray.

```
path_style = 'font-size:12px;fill-rule:nonzero;stroke:#ffffff;stroke-
opacity:1;stroke-width:0.1;stroke-miterlimit:4;stroke-
dasharray:none;stroke-linecap:butt;marker-start:none;stroke-
linejoin:bevel;fill:'
```

I also moved `fill` to the end and left the value blank because that's the part that depends on each county's unemployment rate.

Finally, you're ready to change some colors! You can iterate through each path (except for state boundary lines and the separator for Hawaii and Alaska) and color accordingly. If the unemployment rate is greater than 10, use a darker shade, and anything less than 2 has the lightest shade.

```
for p in paths:

    if p['id'] not in ["State_Lines", "separator"]:
        # pass
```

```
try:
    rate = unemployment[p['id']]
except:
    continue

if rate > 10:
    color_class = 5
elif rate > 8:
    color_class = 4
elif rate > 6:
    color_class = 3
elif rate > 4:
    color_class = 2
elif rate > 2:
    color_class = 1
else:
    color_class = 0

color = colors[color_class]
p['style'] = path_style + color
```

The last step is to print out the SVG file with `prettify()`. The function converts your soup to a string that your browser can interpret.

```
print soup.prettify()
```

Now all that's left to do is run the Python script and save the output as a new SVG file named, say, `colored_map.svg` (Figure 8-17).

▶ You can grab the script in its entirety here: `http://book.flowingdata.com/ch08/regions/colorize_svg.py.txt`

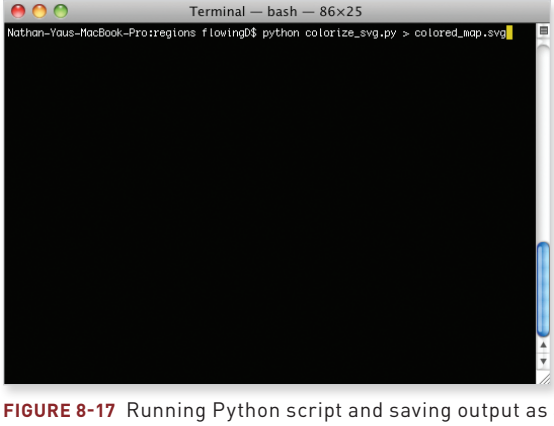


FIGURE 8-17 Running Python script and saving output as a new SVG file

Open your brand spanking new choropleth map in Illustrator or a modern browser such as Firefox, Safari, or Chrome to see the fruits of your labor, as shown in Figure 8-18. It's easy to see now where in the country there were higher unemployment rates during August 2010. Obviously a lot of the west coast and much of the southeast had higher rates, as did Alaska and Michigan. There are a lot of counties in middle America with relatively lower unemployment rates.

With the hard part of this exercise done, you can customize your map to your heart's content. You can edit the SVG file in Illustrator, change border colors and sizes, and add annotation to make it a complete graphic for a larger audience. (Hint: It still needs a legend.)

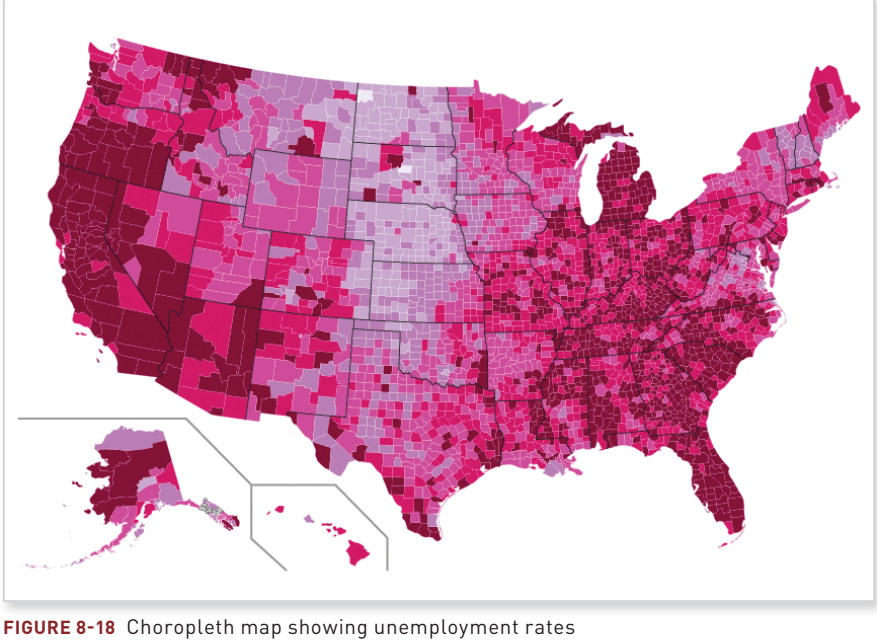


FIGURE 8-18 Choropleth map showing unemployment rates

The best part is that the code is reusable, and you can apply it to other datasets that use the FIPS code. Or even with this same dataset, you can mess around with color scheme to design a map that fits with the theme of your data.

Depending on your data, you can also change the thresholds for how to color each region. The examples so far used equal thresholds where regions

were colored with six shades, and every 2 percentage points was a new class. Every county with an unemployment rate greater than 10 percent was one class; then counties with rates between 8 and 10, then 6 and 8, and so forth. Another common way to define thresholds is by quartiles, where you use four colors, and each color represents a quarter of the regions.

For example, the lower, middle, and upper quartiles for these unemployment rates are 6.9, 8.7, and 10.8 percent, respectively. This means that a quarter of the counties have rates below 6.9 percent, another quarter between 6.9 and 8.7, one between 8.7 and 10.8, and the last quarter is greater than 10.8 percent. To do this, change the colors list in your script to something like the following. It's a purple color scheme, with one shade per quarter.

```
colors = ["#f2f0f7", "#cbc9e2", "#9e9ac8", "#6a51a3"]
```

Then modify the color conditions in the `for` loop, using the preceding quartiles.

```
        if rate > 10.8:
            color_class = 3
        elif rate > 8.7:
            color_class = 2
        elif rate > 6.9:
            color_class = 1
        else:
            color_class = 0
```

Run the script and save like before, and you get Figure 8-19. Notice how there are more counties colored lightly.

To increase the usability of your code, you can calculate quartiles programmatically instead of hard-coding them. This is straightforward in Python. You store a list of your values, sort them from least to greatest, and find the values at the one-quarter, one-half, and three-quarters marks. More concretely, as it pertains to this example, you can modify the first loop in `colorize_svg.py` to store just unemployment rates.

```
unemployment = {}
rates_only = [] # To calculate quartiles
min_value = 100; max_value = 0; past_header = False
for row in reader:
    if not past_header:
```

```
            past_header = True
            continue

        try:
            full_fips = row[1] + row[2]
            rate = float( row[5].strip() )
            unemployment[full_fips] = rate
            rates_only.append(rate)
        except:
            pass
```

Then you can sort the array, and find the quartiles.

```
# Quartiles
rates_only.sort()
q1_index = int( 0.25 * len(rates_only) )
q1 = rates_only[q1_index]     # 6.9

q2_index = int( 0.5 * len(rates_only) )
q2 = rates_only[q2_index]     # 8.7

q3_index = int( 0.75 * len(rates_only) )
q3 = rates_only[q3_index]     # 10.8
```

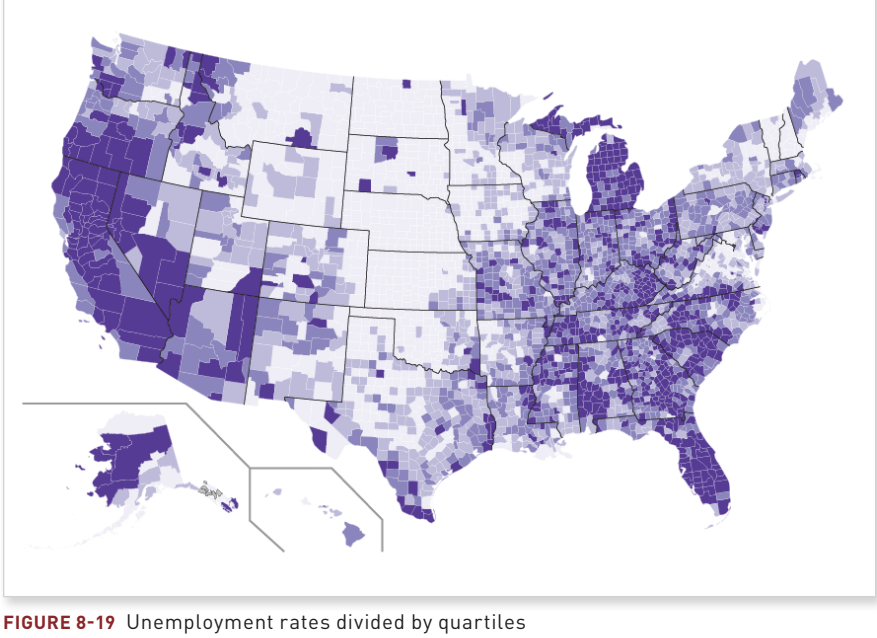


FIGURE 8-19 Unemployment rates divided by quartiles

Instead of hard-coding the values 6.9, 8.7, and 10.8 in your code, you can replace those values with *q1*, *q2*, and *q3*, respectively. The advantage of calculating the values programmatically is that you can reuse the code with a different dataset just by changing the CSV file.

Which color scale you choose depends on that data you have and what message you want to convey. For this particular dataset, I prefer the linear scale because it represents the distribution better and highlights the relatively high unemployment rates across the country. Working from Figure 8-18, you can add a legend, a title, and a lead-in paragraph for a more finalized graphic, as shown in Figure 8-20.

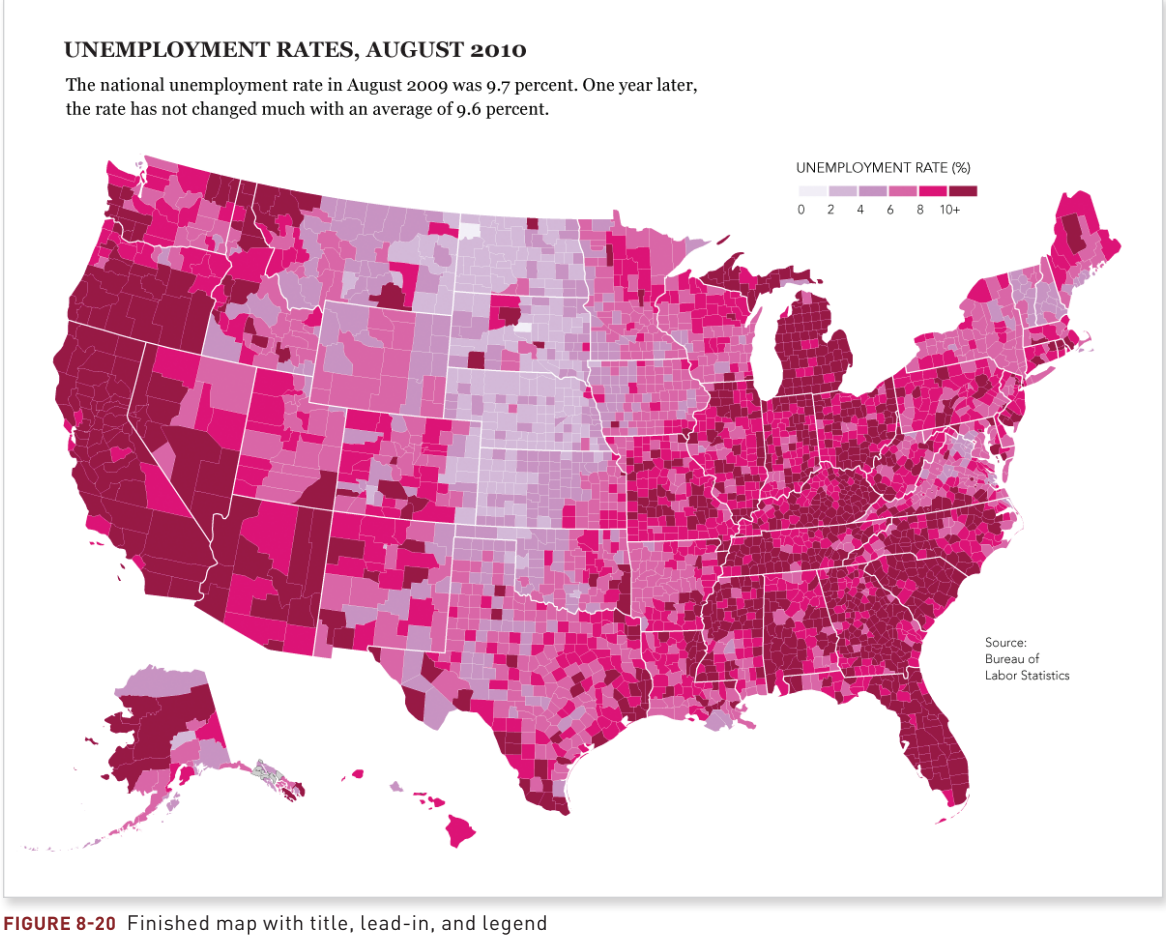


FIGURE 8-20 Finished map with title, lead-in, and legend

TIP

World Bank is one of the most complete resources for country-specific demographic data. I usually go here first.

MAP COUNTRIES

The process to color counties in the previous example isn't exclusive to these regions. You can use the same steps to color states or countries. All you need is an SVG file with unique ids for each region you want to color (which are easily accessible on Wikipedia) and data with ids to match. Now try this out with open data from the World Bank.

Look at percentages of urban populations with access to an improved water source, by country, in 2008. You can download the Excel file from the World Bank data site here: `http://data.worldbank.org/indicator/SH.H2O.SAFE.UR.ZS/countries`. For convenience, you can also download the stripped down data as a CSV file here: Full URL is: `http://book.flowingdata.com/ch08/worldmap/water-source1.txt`. There are some countries with missing data, which is common with country-level data. I've removed those rows from the CSV file.

There are seven columns. The first is the country name; the second is a country code (could this be your unique id?); and the last five columns are percentages for 1990 to 2008.

For the base map, again go to Wikipedia. You can find a lot of versions when you search for the SVG world map, but use the one found here: `http://en.wikipedia.org/wiki/File:BlankMap-World6.svg`. Download the full resolution SVG file, and save it in the same directory as your data. As shown in Figure 8-21, it's a blank world map, colored gray with white borders.

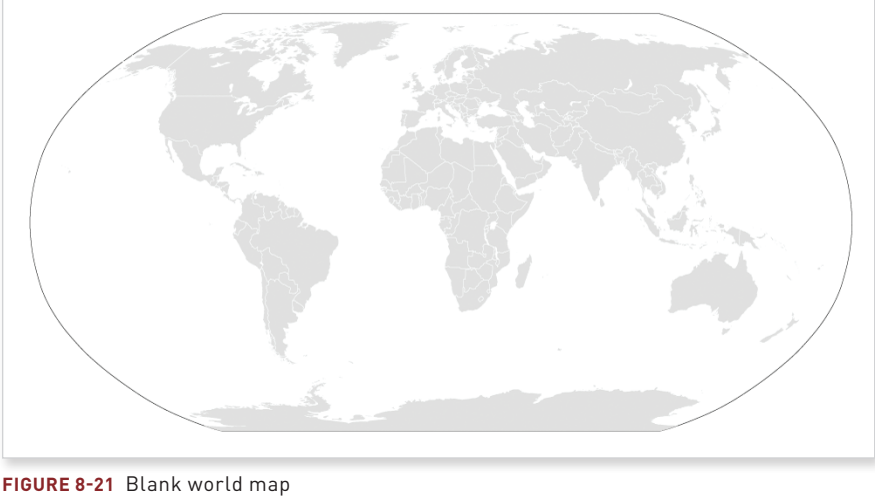


FIGURE 8-21 Blank world map

Open the SVG file in a text editor. It is of course all text formatted as XML, but it's formatted slightly differently than your counties example. Paths don't have useful ids and the *style* attribute is not used. The paths do, however, have classes that look like country codes. They have only two letters, though. The country codes used in the World Bank data have three letters.

According to World Bank documentation, it uses ISO 3166-1 alpha 3 codes. The SVG file from Wikipedia, on the other hand, uses ISO 3166-1 alpha 2 codes. The names are horrible, I know, but don't worry; you don't have to remember that. All you need to know is that Wikipedia provides a conversion chart at `http://en.wikipedia.org/wiki/ISO_3166-1`. I copied and pasted the table into Excel and then saved the important bits as a text file. It has one column for the alpha 2 and another for the alpha 3. Download it here: `http://book.flowingdata.com/ch08/worldmap/country-codes.txt`. Use this chart to switch between the two codes.

As for styling each country, take a slightly different route to do that, too. Instead of changing attributes directly in the path tags, use CSS *outside* of the paths to color the regions. Now jump right in.

Create a file named `generate_css.py` in the same directory as the SVG and CSV files. Again, import the CSV package to load the data in the CSV files with the country codes and water access percentages.

```
import csv
codereader = csv.reader(open('country-codes.txt', 'r'), delimiter="\t")
waterreader = csv.reader(open('water-source1.txt', 'r'), delimiter="\t")
```

Then store the country codes so that it's easy to switch from alpha 3 to alpha 2.

```
alpha3to2 = {}
i = 0
next(codereader)
for row in codereader:

    alpha3to2[row[1]] = row[0]
```

This stores the codes in a Python dictionary where alpha 3 is the key and alpha 2 is the value.

Now like in your previous example, iterate through each row of the water data and assign a color based on the value for the current country.

```
i = 0
next(waterreader)
for row in waterreader:

    if row[1] in alpha3to2 and row[6]:
        alpha2 = alpha3to2[row[1]].lower()
        pct = int(row[6])
        if pct == 100:
            fill = "#08589E"
        elif pct > 90:
            fill = "#08589E"
        elif pct > 80:
            fill = "#4EB3D3"
        elif pct > 70:
            fill = "#7BCCC4"
        elif pct > 60:
            fill = "#A8DDB5"
        elif pct > 50:
            fill = "#CCEBC5"
        else:
            fill = "#EFF3FF"
        print '.' + alpha2 + ' { fill: ' + fill + ' }'

    i += 1
```

This part of the script executes the following steps:

1. Skip the header of the CSV.
2. It starts the loop to iterate over water data.
3. If there is a corresponding alpha 2 code to the alpha 3 from the CSV, and there is data available for the country in 2008, it finds the matching alpha 2.
4. Based on the percentage, an appropriate fill color is chosen.
5. A line of CSS is printed for each row of data.

Run `generate_css.py` and save the output as **`style.css`**. The first few rows of the CSS will look like this:

```
.af { fill: #7BCCC4 }
.al { fill: #08589E }
```

```
.dz { fill: #4EB3D3 }
.ad { fill: #08589E }
.ao { fill: #CCEBC5 }
.ag { fill: #08589E }
.ar { fill: #08589E }
.am { fill: #08589E }
.aw { fill: #08589E }
.au { fill: #08589E }
...
```

This is standard CSS. The first row, for example, changes the fill color of all paths with class `.af` to #7BCCC4.

Open `style.css` in your text editor and copy all the contents. Then open the SVG map and paste the contents at approximately line 135, below the brackets for *.oceanxx*. You just created a choropleth map of the world colored by the percentage of population with access to an improved water source, as shown in Figure 8-22. The darkest blue indicates 100 percent, and the lightest shades of green indicate lower percentages. Countries that are still gray indicate countries where data was not available.

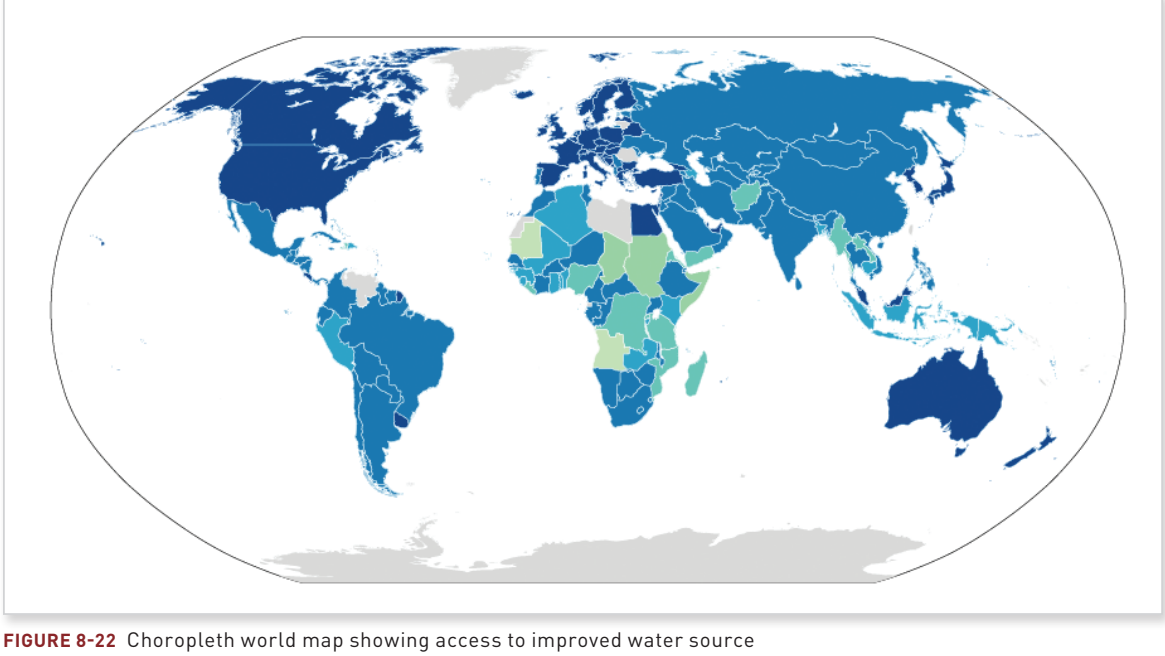


FIGURE 8-22 Choropleth world map showing access to improved water source

The best part is that you can now download almost any dataset from the World Bank (and there are a lot of them) and create a choropleth map fairly quickly just by changing a few lines of code. To spruce up the graphic in Figure 8-22, again, you can open the SVG file in Illustrator and edit away. Mainly, the map needs a title and a legend to indicate what each shade means, as shown in Figure 8-23.

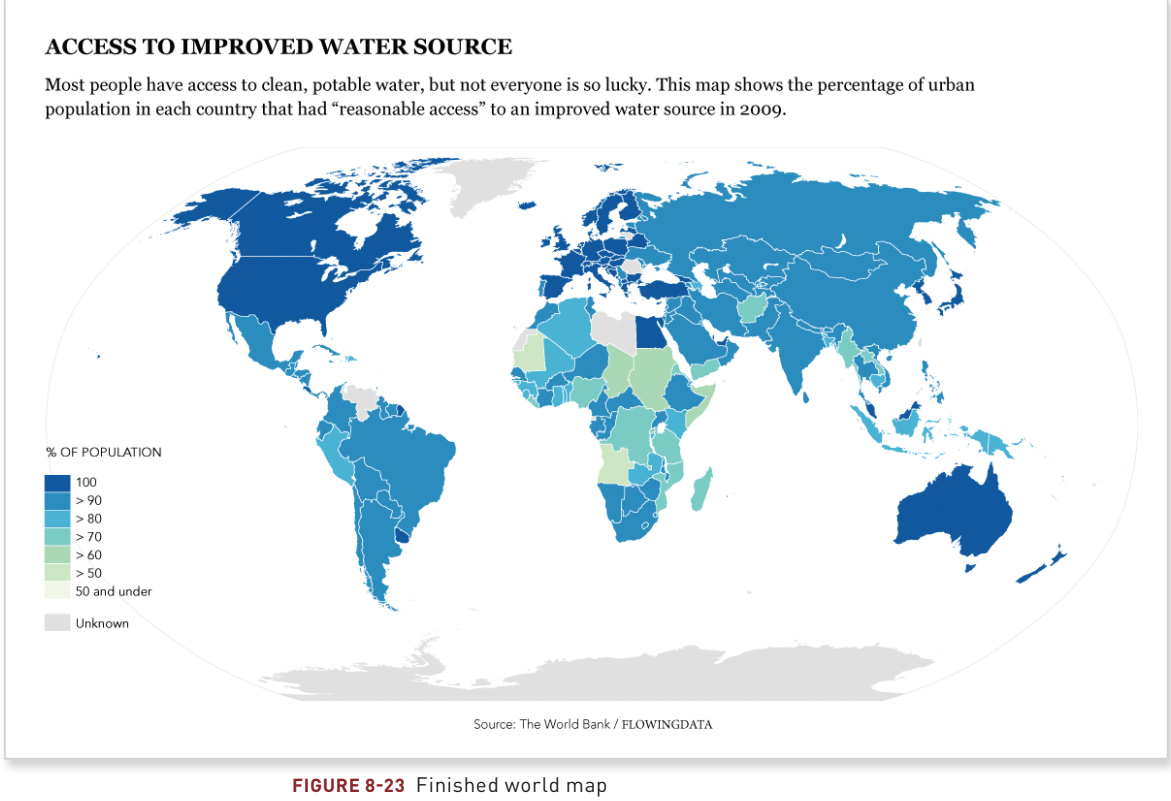


FIGURE 8-23 Finished world map

Over Space and Time

The examples so far enable you to visualize a lot of data types, whether it be qualitative or quantitative. You can vary colors, categories, and symbols to fit the story you're trying to tell; annotate your maps to highlight specific regions or features; and aggregate to zoom in on counties or countries.

But wait, there's more! You don't have to stop there. If you incorporate another dimension of data, you can see changes over both time and space.

In Chapter 4, "Visualizing Patterns over Time," you visualized time more abstractly with lines and plots, which is useful, but when location is attached to your data, it can be more intuitive to see the patterns and changes with maps. It's easier to see clustering or groups of regions that are near in physical distance.

The best part is that you can incorporate what you've already learned to visualize your data over space and time.

Small Multiples

You saw this technique in Chapter 6, "Visualizing Relationships," to visualize relationships across categories, and it can be applied to spatial data, too, as shown in Figure 8-24. Instead of small bar graphs, you can use small maps, one map for each slice of time. Line them up left to right or stack them top to bottom, and it's easy for your eyes to follow the changes.

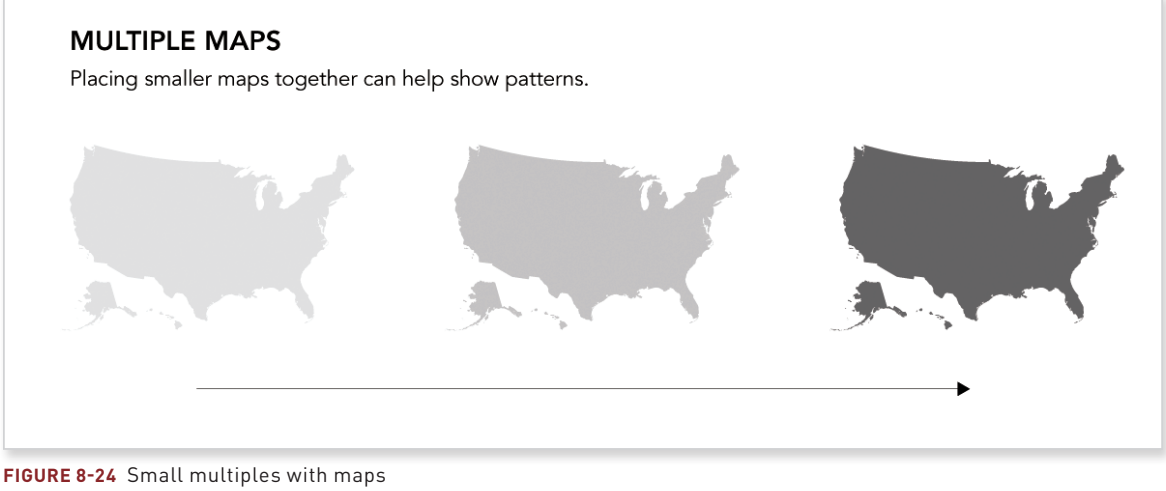


FIGURE 8-24 Small multiples with maps

For example, in late 2009, I designed a graphic that showed unemployment rates by county (Figure 8-25). I actually used a variation of the code you just saw in the previous section, but I applied it to several slices of time.

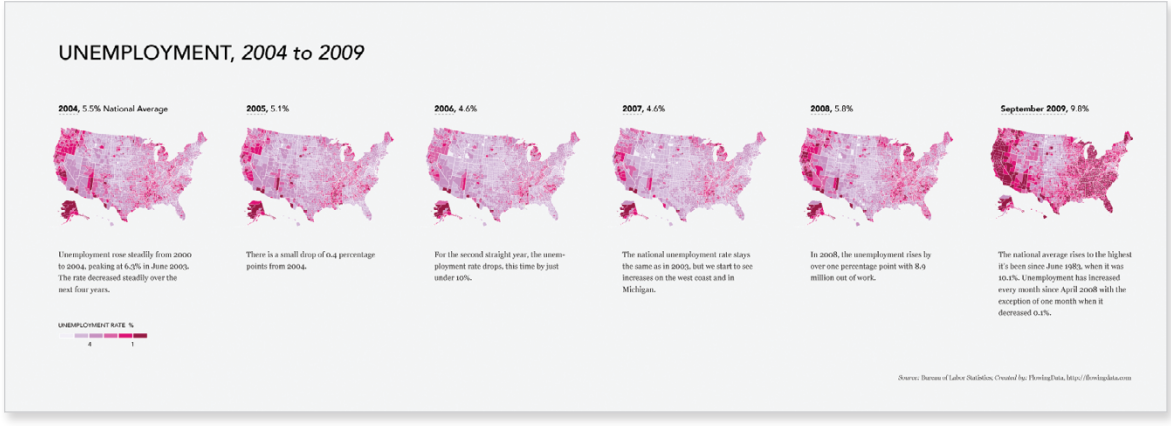


FIGURE 8-25 Unemployment rates from 2004 to 2009

It's easy to see the changes, or lack thereof, by year, from 2004 through 2006, as shown in Figure 8-26. The national average actually went down during that time.

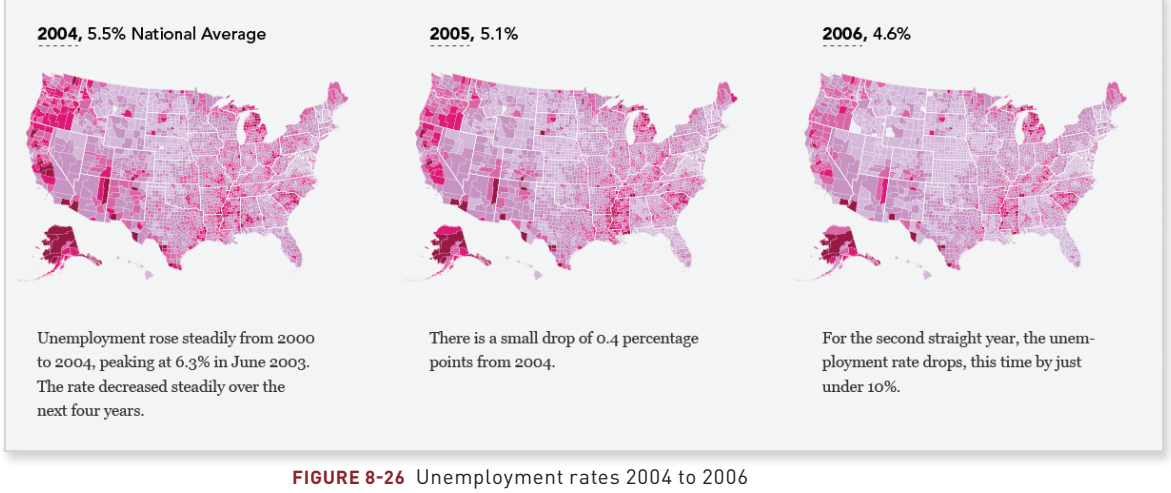


FIGURE 8-26 Unemployment rates 2004 to 2006

Then 2008 hits (Figure 8-27), and you start to see some of the increases in the unemployment rate, especially in California, Oregon, and Michigan, and some counties in the southeast.

Fast forward to 2009, and there is a clear difference, as shown in Figure 8-28. The national average increased 4 percentage points and the county colors become very dark.

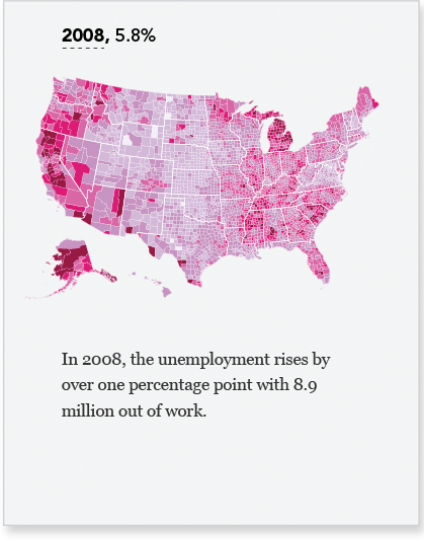


FIGURE 8-27 Unemployment rates in 2008

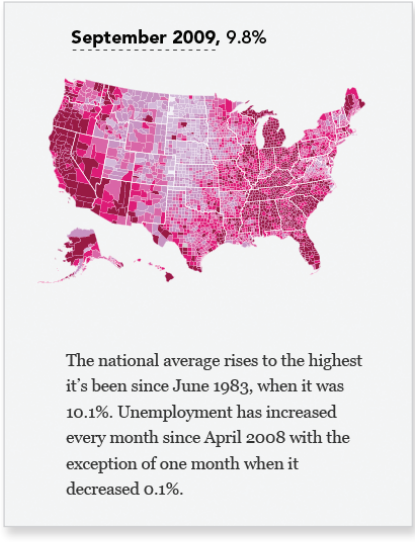


FIGURE 8-28 Unemployment rates during September 2009

This was one of the most popular graphics I posted on FlowingData because it's easy to see that dramatic change after several years of relative standstill. I also used the OpenZoom Viewer, which enables you to zoom in on high-resolution images, so you can focus on your own area to see how it changed.

I could have also visualized the data as a time series plot, where each line represented a county; however, there are more than 3,000 U.S. counties. The plot would have felt cluttered, and unless it was interactive, you would not be able to tell which line represented which county.

▶ When high-resolution images are too big to display on a single monitor, it can be useful to put the image in OpenZoom Viewer (`http://openzoom.org`) so that you can see the picture and then zoom in on the details.

Take the Difference

You don't always need to create multiple maps to show changes. Sometimes it makes more sense to visualize actual differences in a single map. It saves space, and it highlights changes instead of single slices in time, as shown in Figure 8-29.

FIGURE 8-29 Focusing on change

If you were to download urban population counts from the World Bank, you'd have similar data to the previous example using access to improved water. Each row is a country, and each column is a year. However, the urban population data is raw counts for an estimated number of people in the country living in urban areas. A choropleth map of these counts would inevitably highlight larger countries because they of course have larger populations in general. Two maps to show the difference in urban population between 2005 and 2009 wouldn't be useful unless you changed the values to proportions. To do that, you'd have to download population data for 2005 and 2009 in all countries and then do some simple math. It's not all that hard to do that, but it's an extra step. Plus, if the changes are subtle, they'll be hard to see across multiple maps.

Instead, you can take the difference and show it in a single map. You can easily calculate this in Excel or modify the previous Python script, and then make a single map, as shown in Figure 8-30.

It's easy to see which countries changed the most and which ones changed the least when you visualize the differences. In contrast, Figure 8-31 shows the proportion of each country's total population that lived in an urban area in 2005.

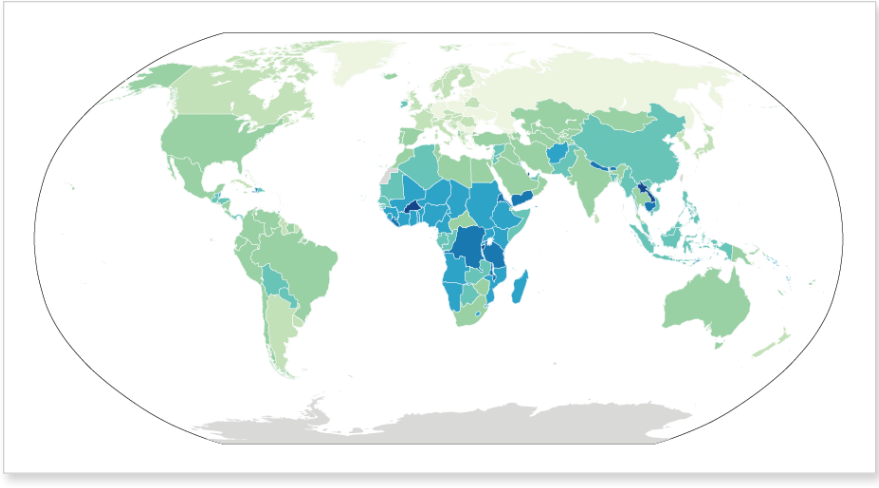


FIGURE 8-30 Change in urban population from 2005 to 2009

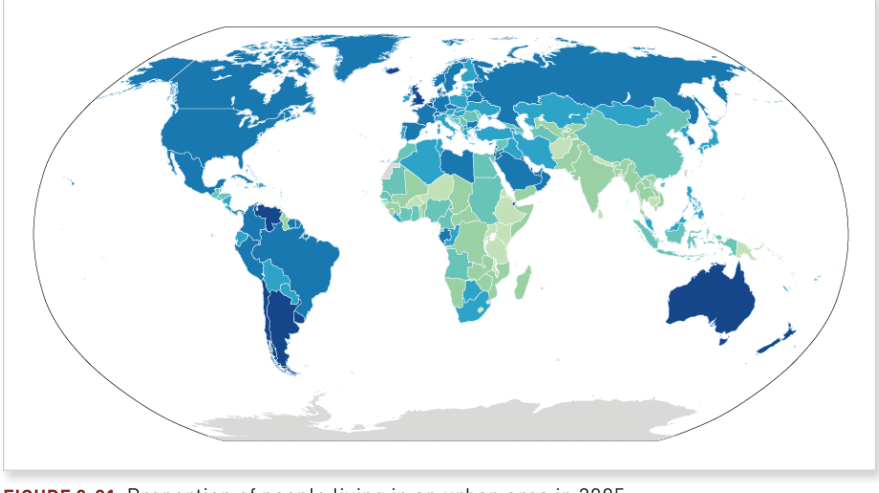


FIGURE 8-31 Proportion of people living in an urban area in 2005

Figure 8-32 shows the same data for 2009. It looks similar to Figure 8-31, and you can barely notice a difference.

For this particular example, it's clear that the single map is more informative. You have to do a lot less work mentally to decipher the changes. It's obvious that although many countries in Africa have a relatively lower percentage of their population living in urban areas compared to the rest of the world, they have also changed the most in recent years.

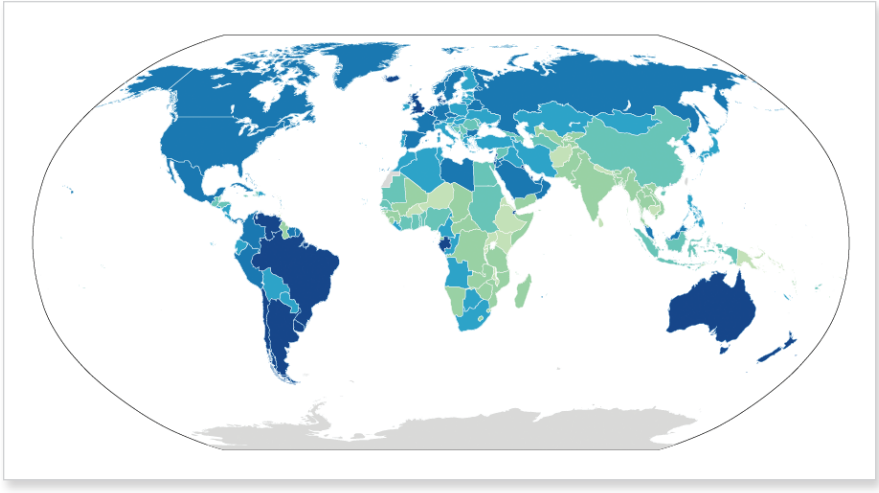


FIGURE 8-32 Proportion of people living in an urban area in 2009

Remember to add a legend, source, and title if your graphic is for a wider audience, as shown in Figure 8-33.

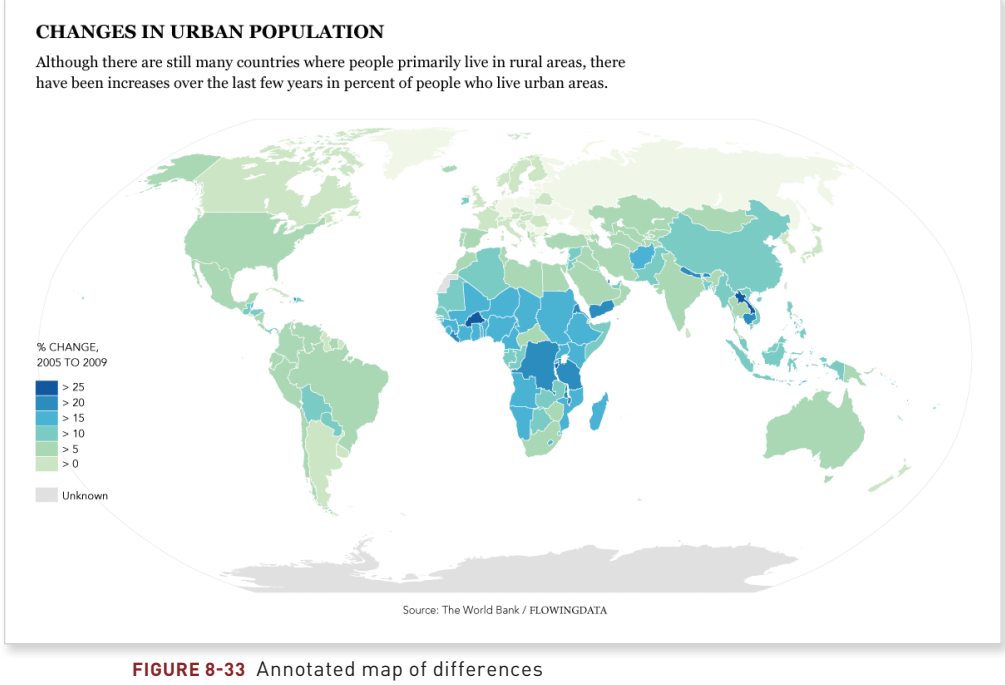


FIGURE 8-33 Annotated map of differences

Animation

One of the more obvious ways to visualize changes over space and time is to animate your data. Instead of showing slices in time with individual maps, you can show the changes as they happen on a single interactive map. This keeps the intuitiveness of the map, while allowing readers to explore the data on their own.

A few years ago, I designed a map that shows the growth of Walmart across the United States, as shown in Figure 8-34. The animation starts with the fist store that opened in 1962 in Rogers, Arkansas, and then moves through 2010. For each new store that opened up, another dot appears on the map. The growth is slow at first, and then Walmarts spread across the country almost like a virus. It keeps growing and growing, with bursts in areas where the company makes large acquisitions. Before you know it, Walmart is everywhere.

▶ View the Walmart map in its entirety at `http://datafl.ws/197`.

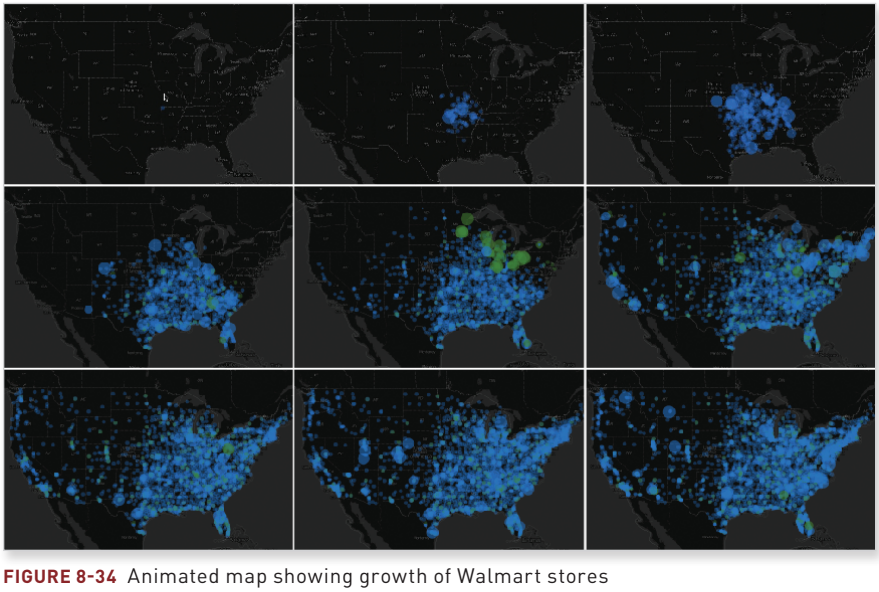


FIGURE 8-34 Animated map showing growth of Walmart stores

At the time, I was just trying to learn Flash and ActionScript, but the map was shared across the web and has been viewed millions of times. I later created a similar map showing the growth of Target (Figure 8-35), and it was equally well spread.

▶ You can watch the growth of Target stores at http://datafl.ws/198.

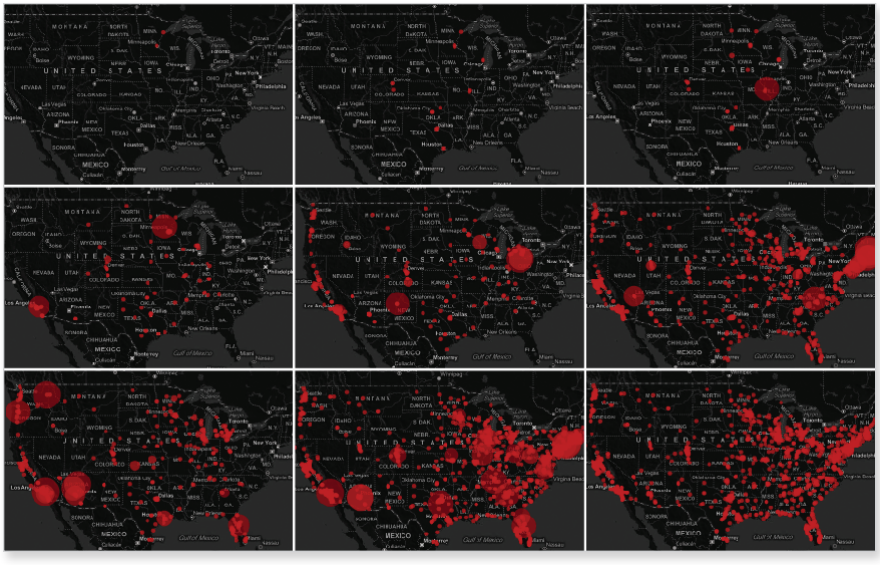


FIGURE 8-35 Animated map showing growth of Target stores

People have been so interested for two main reasons. The first is that the animated map enables you to see patterns that you wouldn't see with a time series plot. A regular plot would show only the number of store openings per year, which is fine if that's the story you want to tell, but the animated maps show growth that's more organic, especially with the Walmart one.

The second reason is that the map is immediately understandable to a general audience. When the animation starts, you know what you're seeing. I'm not saying there isn't value in visualization that takes time to interpret; it's often the opposite. However, there's a low time threshold for the web, so because the map is intuitive (and that people can zoom in on their own local areas) certainly helped the eager sharing.

▶ Download Modest Maps at http://modestmaps.com.

CREATE AN ANIMATED GROWTH MAP

In this example, you create the Walmart growth map in ActionScript. You use Modest Maps, an ActionScript mapping library to provide interaction and the base map. The rest you code yourself. Download the complete source code at http://book.flowingdata.com/ch08/Openings_src.zip. Instead

of going through every line and file, you'll look at just the important bits in this section.

As in Chapter 5, "Visualizing Proportions," when you create a stacked area chart with ActionScript and the Flare visualization toolkit, I highly recommend you use Adobe Flex Builder. It makes ActionScript a lot easier and keeps your code organized. You can of course still code everything in a standard text editor, but Flex Builder wraps up the editor, debugging, and compiling into one package. This example assumes you do have Flex Builder, but you are of course welcome to grab an ActionScript 3 compiler from the Adobe site.

To begin, open Flex Builder 3, and right-click the left sidebar, which shows the current list of projects. Select Import, as shown in Figure 8-36.

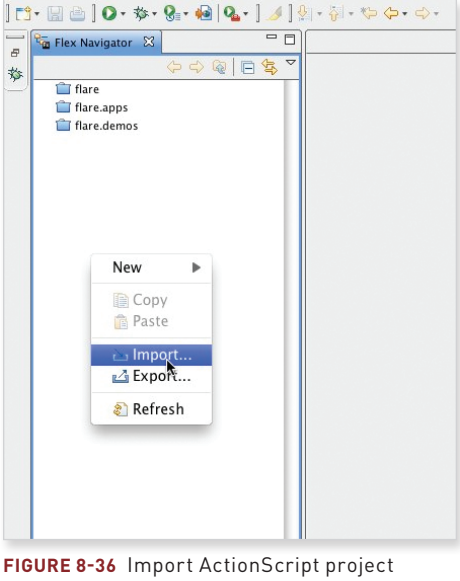


FIGURE 8-36 Import ActionScript project

Select Existing Projects Into Workspace, as shown in Figure 8-37.

Then, as shown in Figure 8-38, browse to the directory in which you saved the code. The Openings project should appear after selecting the root directory.

NOTE

Adobe Flex Builder was recently changed to Adobe Flash Builder. There are small differences between the two, but you can use either.

▶ Download the growth map code in its entirety at `http://book.flowingdata.com/ch08/Openings_src.zip` to follow along in this example.

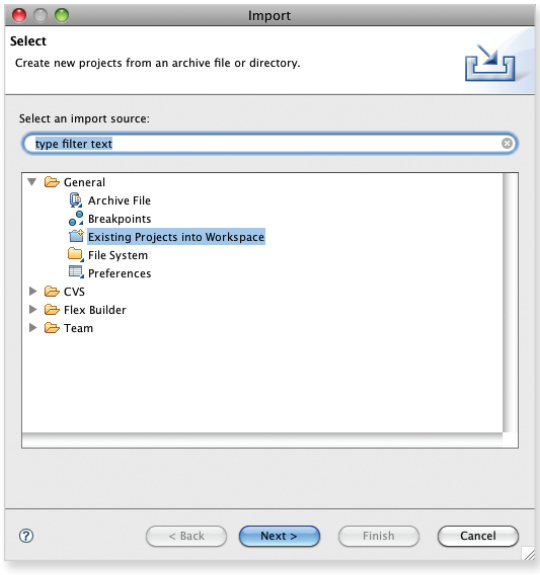


FIGURE 8-37 Existing project

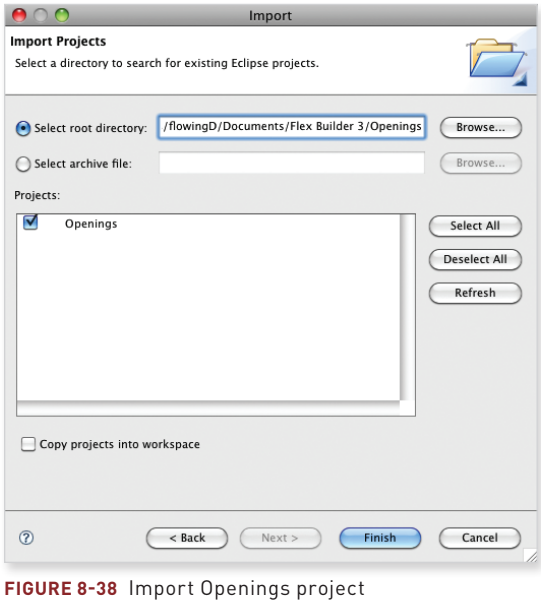


FIGURE 8-38 Import Openings project

Your workspace in Flex Builder should look similar to Figure 8-39.

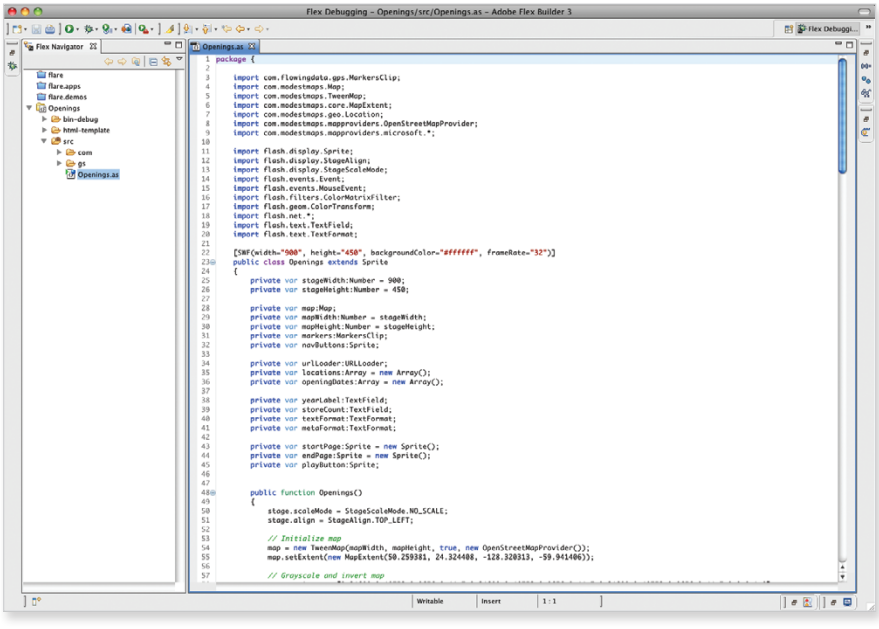


FIGURE 8-39 Workspace after importing project

All of the code is in the src folder. This includes Modest Maps in the com folder and TweenFilterLite in the gs folder, which help with transitions.

With the Openings project imported, you're ready to start building the map. Do this in two parts. In the first part create an interactive base map. In the second add the markers.

Add the Interactive Base Map

In Openings.as, the first lines of code import the necessary packages.

```
import com.modestmaps.Map;
import com.modestmaps.TweenMap;
import com.modestmaps.core.MapExtent;
import com.modestmaps.geo.Location;
import com.modestmaps.mapproviders.OpenStreetMapProvider;

import flash.display.Sprite;
import flash.display.StageAlign;
import flash.display.StageScaleMode;
```

```
import flash.events.Event;
import flash.events.MouseEvent;
import flash.filters.ColorMatrixFilter;
import flash.geom.ColorTransform;
import flash.text.TextField;
import flash.net.*;
```

The first section imports classes from the Modest Maps package, whereas the second section imports display objects and event classes provided by Flash. The name of each class isn't important right now. That becomes clear as you use them. However, the naming pattern for the first section matches the directory structure, starting with `com`, then `modestmaps`, and ending with `Map`. This is how you import classes most of the time when you write your own ActionScript.

Above `public class Openings extends Sprite`, several variables—width, height, background color, and frame rate—of the compiled Flash file are initialized.

```
[SWF(width="900", height="450", backgroundColor="#ffffff",
frameRate="32")]
```

Then after the class declaration, you need to specify some variables and initialize a Map object.

```
private var stageWidth:Number = 900;
private var stageHeight:Number = 450;
private var map:Map;
private var mapWidth:Number = stageWidth;
private var mapHeight:Number = stageHeight;
```

In between the brackets of the `Openings()` function, you can now create your first interactive map with Modest Maps.

```
stage.scaleMode = StageScaleMode.NO_SCALE;
stage.align = StageAlign.TOP_LEFT;

// Initialize map
map = new TweenMap(mapWidth, mapHeight, true, new
OpenStreetMapProvider());
map.setExtent(new MapExtent(50.259381, 24.324408, -128.320313,
-59.941406));
addChild(map);
```

Like in Illustrator, you can think of the full interactive as a bunch of layers. In ActionScript and Flash, the first layer is the stage. You set it to not scale objects when you zoom in on it, and you align the stage in the top left. Next you initialize the map with the *mapWidth* and *mapHeight* that you specified in the variables, turn on interaction, and use map tiles from OpenStreetMap. By setting the map extent to the preceding code, you frame the map around the United States.

The coordinates in `MapExtent()` are latitude and longitude which set the bounding box for what areas of the world to show. The first and third numbers are latitude and longitude for the top left corner, and the second and fourth numbers are latitude and longitude for the bottom right.

Finally, add the map (with `addChild()`) to the stage. Figure 8-40 shows the result when you compile the code without adding any filters to the map. You can either press the Play button in the top left of Flex Builder, or from the main menu, you can select Run ➪ Run Openings.

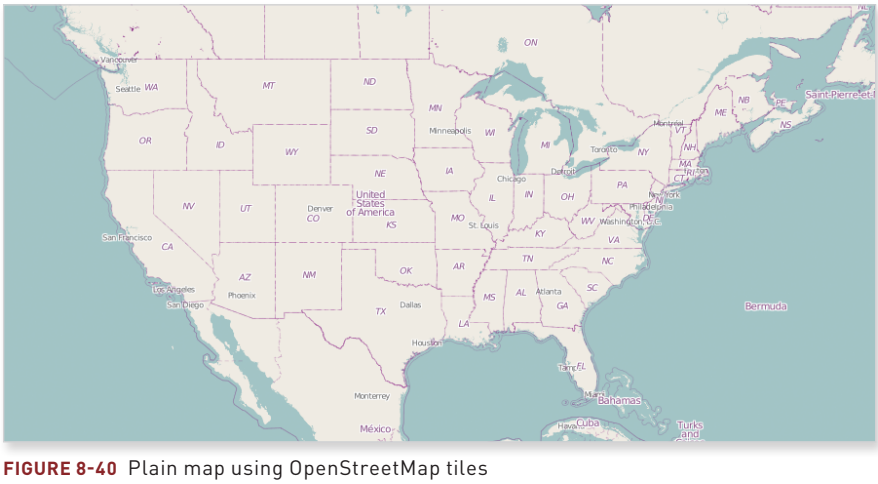


FIGURE 8-40 Plain map using OpenStreetMap tiles

When you run Openings, the result should pop up in your default browser. There's nothing on it yet, but you can click-and-drag, which is kind of cool. Also if you prefer a different set of map tiles, you can use the Microsoft road map (Figure 8-41) or Yahoo! hybrid map (Figure 8-42).

FIGURE 8-41 Plain map with Microsoft road map

► You can also use your own tiles if you want. There's a good tutorial on the Modest Maps site.

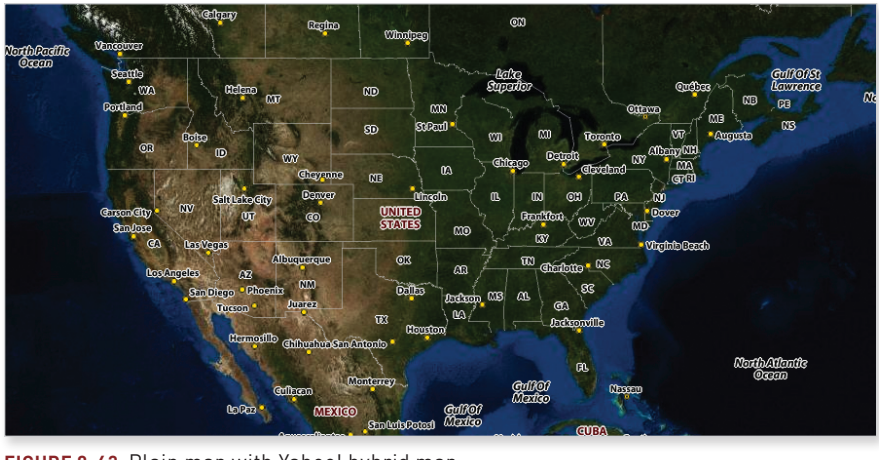


FIGURE 8-42 Plain map with Yahoo! hybrid map

► See the Adobe reference for more on how to use color matrices to customize objects in ActionScript at `http://livedocs.adobe.com/flash/9.0/ActionScriptLangRefV3/flash/filters/ColorMatrixFilter.html`.

You can also experiment with the colors of the map by applying filters. You could for example, change the map to grayscale by placing the following under the code you just wrote. The mat array is of length 20 and takes values from 0 to 1. Each value represents how much red, green, blue and alpha each pixel gets.

```
var mat:Array = [0.24688,0.48752,0.0656,0,44.7,0.24688,0.48752,
    0.0656,0,44.7,0.24688,0.48752,0.0656,0,44.7,0,0,0,1,0];
var colorMat:ColorMatrixFilter = new ColorMatrixFilter(mat);
map.grid.filters = [colorMat];
```

As shown in Figure 8-43, the map is all gray, which can be useful to highlight the data that you plan to overlay on top of the map. The map serves as background instead of battling for attention.

FIGURE 8-43 Grayscale map after applying filter

You can also invert the colors with a color transform.

```
map.grid.transform.colorTransform =
    new ColorTransform(-1,-1,-1,1,255,255,255,0);
```

This turns white to black and black to white, as shown in Figure 8-44.

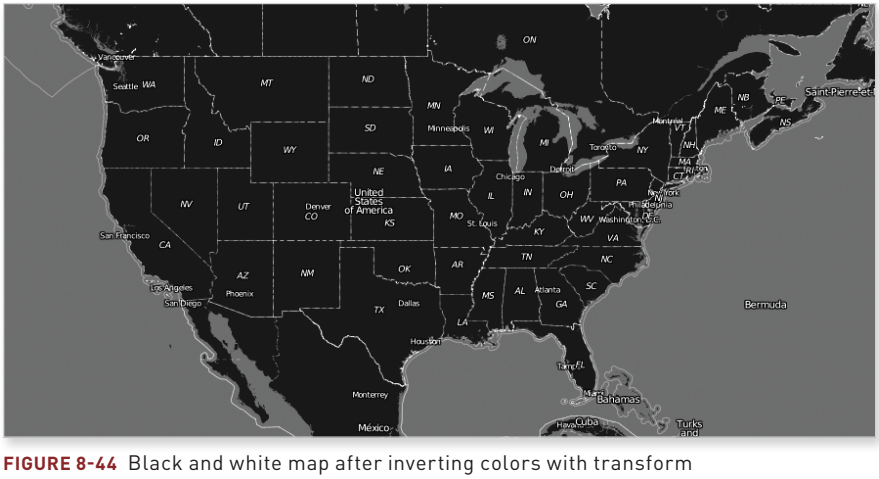


FIGURE 8-44 Black and white map after inverting colors with transform

To create zooming buttons, first write a function to make buttons. You'd think that there would be a quick default way to do this by now, but it still takes a handful of code to get the job done. The function definition of `makeButton()` is at the bottom of the Openings class.

```
public function makeButton(clip:Sprite, name:String, labelText:String,
action:Function):Sprite
{
    var button:Sprite = new Sprite();
    button.name = name;
    clip.addChild(button);

    var label:TextField = new TextField();
    label.name = 'label';
    label.selectable = false;
    label.textColor = 0xffffff;
    label.text = labelText;
    label.width = label.textWidth + 4;
    label.height = label.textHeight + 3;
    button.addChild(label);

    button.graphics.moveTo(0, 0);
    button.graphics.beginFill(0xFDBB30, 1);
    button.graphics.drawRect(0, 0, label.width, label.height);
    button.graphics.endFill();

    button.addEventListener(MouseEvent.CLICK, action);
    button.useHandCursor = true;
    button.mouseChildren = false;
    button.buttonMode = true;

    return button;
}
```

Then create another function that makes use of the function and draws the buttons you want. The following code creates two buttons using `makeButton()`—one for zooming in and the other for zooming out. It puts them at the bottom left of your map.

```
// Draw navigation buttons
private function drawNavigation():void
{
    // Navigation buttons (zooming)
```

```
    var buttons:Array = new Array();
    navButtons = new Sprite();
    addChild(navButtons);
    buttons.push(makeButton(navButtons, 'plus', '+', map.zoomIn));
    buttons.push(makeButton(navButtons, 'minus', '–', map.zoomOut));
    var nextX:Number = 0;
    for(var i:Number = 0; i < buttons.length; i++) {
        var currButton:Sprite = buttons[i];
        Sprite(buttons[i]).scaleX = 3;
        Sprite(buttons[i]).scaleY = 3;
        Sprite(buttons[i]).x = nextX;
        nextX += 3*Sprite(buttons[i]).getChildByName('label').width;
    }
    navButtons.x = 2; navButtons.y = map.height-navButtons.height-2;
}
```

However, because it's a function, the code won't execute until you call it. In the `Openings()` function, also known as the constructor, under the filters, add `drawNavigation()`. Now you can zoom in to locations of interest, as shown in Figure 8-45.

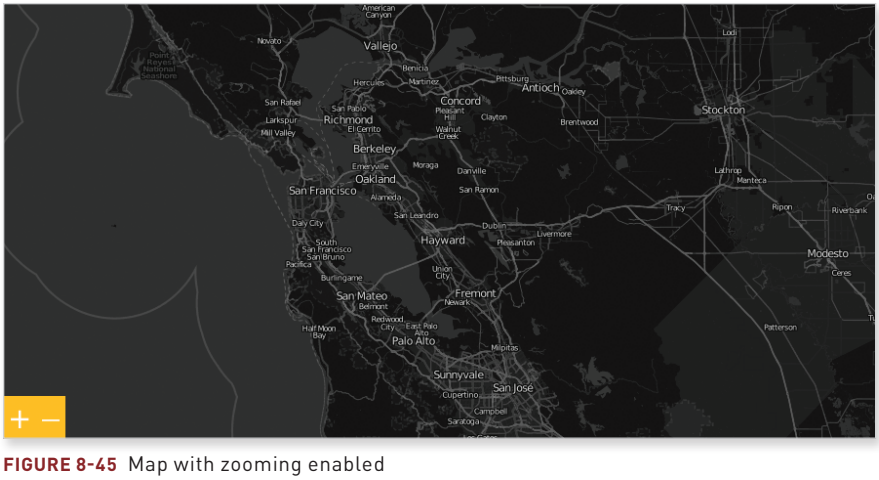


FIGURE 8-45 Map with zooming enabled

That's all you need for the base map. You pick your tiles, set your variables, and enable interaction.

Add the Markers

The next steps are to load the Walmart location data and create markers for each store opening. In the constructor, the following code loads an XML file from a URL. When the file finishes loading, a function named `onLoadLocations()` is called.

```
    var urlRequest:URLRequest =
        new URLRequest('http://projects.flowingdata.com/walmart/walmarts_
new.xml');
    urlLoader = new URLLoader();
    urlLoader.addEventListener(Event.COMPLETE, onLoadLocations);
    urlLoader.load(urlRequest);
```

The obvious next step is to create the `onLoadLocations()` function. It reads the XML file and stores the data in arrays for easier use later. Before you do that though, you need to initialize a few more variables after *navButtons*.

```
private var urlLoader:URLLoader;
private var locations:Array = new Array();
private var openingDates:Array = new Array();
```

These variables are used in `onLoadLocations()`. Latitude and longitude are stored in *locations*, and opening dates, in year format, are stored in *openingDates*.

```
private function onLoadLocations(e:Event):void {
    var xml:XML = new XML(e.target.data);
    for each(var w:* in xml.walmart) {
        locations.push(new Location(w.latitude, w.longitude));
        openingDates.push(String(w.opening_date));
    }
    markers = new MarkersClip(map, locations, openingDates);
    map.addChild(markers);
}
```

The next step is to create the MarkersClip class. Following the same directory structure discussed earlier, there is a directory named flowingdata in the com directory. A gps directory is in the flowingdata directory. Finally, in com ➪ flowingdata ➪ gps is the MarkersClip class. This is the container that will hold all the Walmart markers, or rather, the data layer of your interactive map.

As before, you need to import the classes that you will use. Usually, you add these as you need them in the code, but for the sake of simplicity, you can add all of them at once.

```
import com.modestmaps.Map;
import com.modestmaps.events.MapEvent;

import flash.display.Sprite;
import flash.events.TimerEvent;
import flash.geom.Point;
import flash.utils.Timer;
```

The first two are from Modest Maps, whereas the last four are native classes. Then you set variables right before the `MarkersClip()` function. Again, you would add these as you need them, but you can add them all now to get to the meat of this class—the functions.

```
protected var map:Map;           // Base map
public var markers:Array;       // Holder for markers
public var isStationary:Boolean;

public var locations:Array;
private var openingDates:Array;

private var storesPerYear:Array = new Array();
private var spyIndex:Number = 0;  // Stores per year index
private var currentYearCount:Number = 0; //  Stores shown so far
private var currentRate:Number;  // Number of stores to show
private var totalTime:Number = 90000;  // Approx. 1.5 minutes
private var timePerYear:Number;
public var currentYear:Number = 1962;  // Start with initial year

private var xpoints:Array = new Array(); // Transformed longitude
private var ypoints:Array = new Array(); // Transformed latitude

public var markerIndex:Number = 0;
private var starting:Point;
private var pause:Boolean = false;
public var scaleZoom:Boolean = false;
```

In the `MarkersClip()` constructor, store the variables that will be passed to the class and compute a few things such as time per year and coordinates for stores. You can think of this as the setup.

The storesPerYear variable stores how many stores opened during a given year. For example, one store opened the first year, and no stores opened the next. When you use this code with your own data, you need to update *storesPerYear* appropriately. You could also write a function that computes stores or location openings per year to increase the reusability of your code. A hard-coded array is specified in this example for the sake of simplicity.

```
this.map = map;

this.x = map.getWidth() / 2;
this.y = map.getHeight() / 2;

this.locations = locations;
setPoints();
setMarkers();

this.openingDates = openingDates;

var tempIndex:int = 0;

storesPerYear = [1,0,1,1,0,2,5,5,5,15,17,19,25,19,27,
    39,34,43,54,150,63,87,99,110,121,142,125,131,178,
    163,138,156,107,129,53,60,66,80,105,106,114,96,
    130,118,37];
timePerYear = totalTime / storesPerYear.length;
```

There are two other functions in the MarkersClip class: `setPoints()` and `setMarkers()`. The first one translates latitude and longitude coordinates to x- and y-coordinates, and the second function places the markers on the map without actually showing them. Following is the definition for `setPoints()`. It uses a built-in function provided by Modest Maps to calculate x and y and then stores the new coordinates in *xpoints* and *ypoints*.

```
public function setPoints():void {
    if (locations == null) {
        return;
    }
```

```
        var p:Point;
        for (var i:int = 0; i < locations.length; i++) {
            p = map.locationPoint(locations[i], this);
            xpoints[i] = p.x;
            ypoints[i] = p.y;
        }
    }
```

The second function, `setMarkers()`, uses the points that `setPoints()` stores and places markers accordingly.

```
    protected function setMarkers():void
    {
        markers = new Array();
        for (var i:int = 0; i < locations.length; i++)
        {
            var marker:Marker = new Marker();
            addChild(marker);
            marker.x = xpoints[i]; marker.y = ypoints[i];
            markers.push(marker);
        }
    }
```

The function also uses a custom Marker class, which you can find in com ➪ flowingdata ➪ gps ➪ Marker.as, assuming you have downloaded the complete source code. It's basically a holder, and when you call its `play()` function, it "lights up."

Now you have location and markers loaded on the map. However, if you compiled the code now and played the file, you would still see a blank map. The next step is to cycle through the markers to make them light up at the right time.

The `playNextStore()` function simply calls `play()` of the next marker and then gets ready to play the one after that. The `startAnimation()` and `onNextYear()` functions use timers to incrementally display each store.

```
    private function playNextStore(e:TimerEvent):void
    {
        Marker(markers[markerIndex]).play();
        markerIndex++;
    }
```

If you were to compile and run the animation now, you'd get dots, but it doesn't work with the map's zoom and pan, as shown in Figure 8-46. As you drag the map back and forth or zoom in and out, the bubbles for each store are stationary.

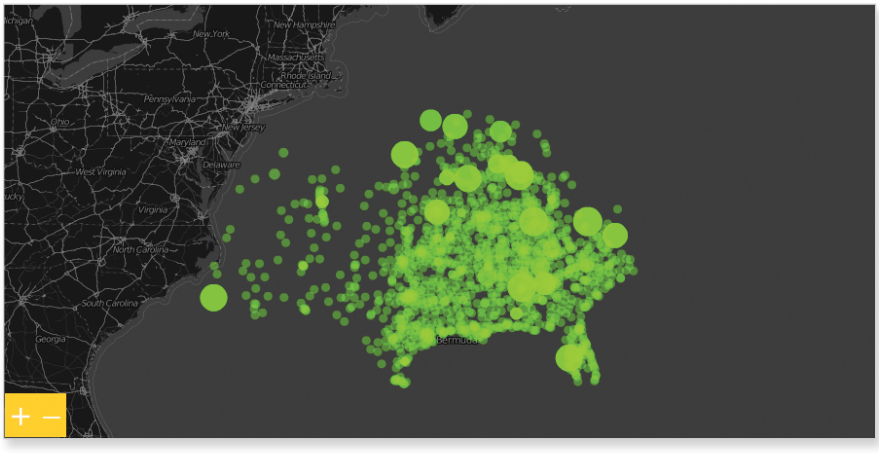


FIGURE 8-46 Growth map with incorrect pan and zoom

Listeners are added in the constructor so that the dots move whenever the map moves. Whenever a MapEvent is triggered by Modest Maps, a corresponding function defined in MarkersClip.as is called. For example in the first line below, `onMapStartZooming()` is called when a user clicks on the map's zoom button.

```
this.map.addEventListener(MapEvent.START_ZOOMING,
  onMapStartZooming);
this.map.addEventListener(MapEvent.STOP_ZOOMING,
  onMapStopZooming);
this.map.addEventListener(MapEvent.ZOOMED_BY, onMapZoomedBy);
this.map.addEventListener(MapEvent.START_PANNING,
  onMapStartPanning);
this.map.addEventListener(MapEvent.STOP_PANNING,
  onMapStopPanning);
this.map.addEventListener(MapEvent.PANNED, onMapPanned);
```

This gives you the final map, as shown in Figure 8-47.

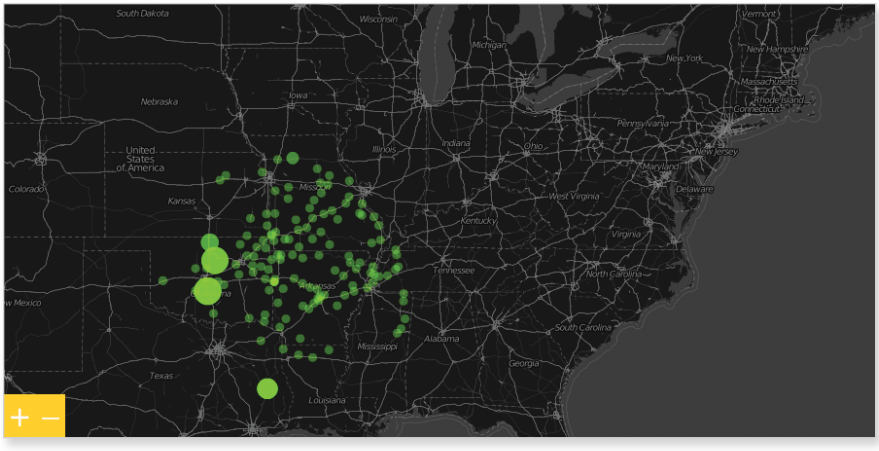


FIGURE 8-47 Fully interactive growth map showing Wal-Mart openings

The story with Walmart store openings is the organic growth. The company started in a single location and slowly spread outward. Obviously, this isn't always the case. For example, Target's growth doesn't look so calculated. Costco's growth is less dramatic because there are fewer locations, but its strategy seems to be growth on the coasts and then a move inward.

In any case, it's a fun and interesting way to view your data. The growth maps seem to spur people's imaginations, and they can wonder about the spread of McDonald's or Starbucks. Now that you have the code, it's a lot easier to implement. The hard part is finding the data.

Wrapping Up

Maps are a tricky visualization type because in addition to your own data, you have to handle the dimension of geography. However, because of how intuitive they are, maps can also be rewarding, both in how you can present data to others and how you can explore your data deeper than you could with a statistical plot.

As seen from the examples in this chapter, there are a lot of possibilities for what you can do with spatial data. With just a few basic skills, you can visualize a lot of datasets and tell all sorts of interesting stories. This is

just the tip of the iceberg. I mean, people go to college and beyond to earn degrees in cartography and geography, so you can imagine what else is out there. You can play with cartograms, which size geographic regions according to a metric; add more interaction in Flash; or combine maps with graphs for more detailed and exploratory views of your data.

Online maps have become especially prevalent, and their popularity is only going to grow as browsers and tools advance. For the growth map example, ActionScript and Flash were used, but it could have also been implemented in JavaScript. Which tool you use depends on the purpose. If it doesn't matter what tool you use, then go with the one you're more comfortable with. The main thing, regardless of software, is the logic. The syntax might change, but you do the same with your data, and you look for the same flow in your storytelling.