Elijah Meeks

D3.js IN ACTION

MEAP

MANNING

MEAP Edition
Manning Early Access Program
D3.js in Action
Version 6

For more information on this and other Manning titles go to
www.manning.com

Welcome

Thank you for purchasing the MEAP for *D3.js in Action*. To get the most benefit from this book, you'll want to have some established skills in programming, with experience in HTML5 and basic knowledge about CSS and DOM, or be transitioning from dealing with data in R/Python/SQL with the desire to build more sophisticated shareable applications.

When I first started experimenting with D3 two years ago, it was out of necessity. Flash was dead, and I needed an information visualization library that was feature rich and, hopefully, long lived. D3 proved to be more than that, with robust capabilities not only for data visualization, but for creating complex applications on the web.

Since that time, there have been numerous examples, some really great introductions to the library, and a few cookbooks to help new users learn the basics and get specific tasks done with D3. In contrast, *D3.js in Action* is my attempt to produce an exhaustive, highly informative, deep dive into the library, one that covers the fundamental structures of how D3 works with data to produce the stunning products we're all so impressed by. D3 does a lot, and not just with graphics, and I've tried to explain in detail the whole library.

Along with all that, I also wanted to focus on two specific spheres that D3 handles really well: networks and maps. As a result, this book spends a full chapter on each, dealing with the variety of features and functions in D3 that let you create the most amazing network visualizations and online interactive mapping applications. This is along with chapters that focus on general charts and D3-specific layouts more broadly.

Finally, throughout *D3.js in Action* you'll see an approach that embraces the functionality available in modern browsers.

It's a big book for a big library, and I hope you find it as useful to read as I did to write it. Please be sure to post any questions, comments, or suggestions you have about the book in the Author Online forum. Your feedback is essential in developing the best book possible.

— Elijah Meeks

brief contents

1 An introduction to D3.js

1.1 What is D3?

Data-Driven Documents is a brand name. It's what "D3" stands for. But it's also a class of applications that have been offered on the web in one form or another for years. Whether as interactive dashboards, rich internet applications, or dynamically driven content, we've been building and dealing with data-driven documents for quite some time. So in one sense, the D3.js library is an iterative step in a chain of various technologies used for data-driven documents, but in another sense, it is a radical one.

D3.js comes out of a need for sophisticated data visualization on the web, but does more than that because of its robust design. Coming out of the data visualization program at Stanford Computer Science, Mike Bostock worked with Jeff Heer and Vadim Ogievetsky to create Protovis, which like D3.js is a JavaScript library designed for information visualization but which was designed to provide compatibility with older browsers. Bostock also developed Polymaps, another JavaScript library which provided vector and tile mapping capability in a lightweight form. These earlier endeavors would inform the creation of D3.js, which focused on modern standards and modern browsers. As Bostock describes it, "This avoids proprietary representation and affords extraordinary flexibility, exposing the full capabilities of web standards such as CSS3, HTML5 and SVG." This is the radical nature of D3.js. It will not run on Internet Explorer 6, and while that cost may be too much to bear for certain developers, the wide adoption of standards on modern browsers has finally afforded the capacity to break with the past. In that regard, D3 is a sign of the new capabilities that let web developers deliver dynamic and interactive content seamlessly in the browser.

The iterative nature of D3.js comes from its resemblance to earlier methods of deploying rich interactivity to the web, such as Flash using ActionScript3. As I or any other former Flash developer can tell you, the period after which Steve Jobs condemned Flash--but before the maturation of JavaScript engines and browsers--was a difficult one. You simply could not build the kind of high-performance rich Internet applications in the browser like you could in a

compiled Flash runtime. The performance of Flash more than outweighed its proprietary nature and the sometimes seemingly willful obtuseness of Adobe. Flash for animation and RIAs is still common on the Internet, and especially for internal webapps, for this very reason. D3.js provides the same performance, but integrated into web standards and the document object model at the core of HTML.

With that in mind, let's take a look at the basic principles of data visualization and how D3 works. We'll do this by establishing the basic principles of how D3 selections and data-binding work, followed by understanding how it interacts with SVG and the DOM, then we'll look at data types that you'll commonly encounter. Finally, we'll use D3 to create some simple DOM and SVG elements.

1.1.1 Data Visualization is More Than Just Data Visualization

You may think of data visualization as limited to pie charts, line charts, and the variety of charting methods popularized by Tufte and deployed in research. It's much more than that. One of the core strengths of d3.js is that it allows for the creation of vector graphics for traditional charting, but also the creation of geospatial and network visualizations as well as traditional HTML elements like tables, lists, and paragraphs. This broad-based approach to data visualization, where a map or a network graph or a table is just another kind of representation of data, is the core of the D3.js library's appeal for application development. By requiring a break with the practice of supporting long-obsolete browsers, D3.js affords developers the capacity not only to make richly interactive applications, but applications that are styled and served just like traditional web content. This makes them more portable, more amenable to the growing linked data web, and more easily maintained by large teams.

The decision on Bostock's part to deal broadly with data, and create a library capable of presenting maps as easily as charts as easily as networks as easily as ordered lists, also means that a developer need not split their effort trying to understand the abstractions and syntax of one library for maps, and another for dynamic text content, and another for data visualization. Instead, the code for running an interactive force-directed network layout is not only very close to pure JavaScript, it's also very similar to representing dynamic points of interest on a D3.js map. Not only are the methods the same, the very data could be the same, formulated in one way for lists and paragraphs and spans while formulated in another way for geospatial representation. The class of data-driven document is already a broad one on its face, and becomes even more all-encompassing when one also treats images and text as data.

1.1.2 D3 is About Selecting and Binding

Throughout this chapter, we'll see code snippets that you can run in your browser to make changes to the graphical appearance of elements on your web site. At the end of the chapter is a simple application written in D3 that explains the basics of the code you're running in JavaScript. But before that we'll explore the basic principles of web development using D3, and you'll just see this pattern of code over and over again:

```
d3.selectAll("circle.a").style("fill", "red").attr("cx", 100)
```

This will make every circle on your page with the class of “a” turn red and move it so that its center is 100 pixels to the right of the left side of your <svg> canvas. Likewise, this code:

```
d3.selectAll("div").style("background", "red").attr("class", "b")
```

Will make every div on your web page turn red and change its class to “b”. But before you can change your circles and divs, you’ll need to make them, and before you do that, it’s best for you to understand what’s happening in this pattern.

The first part of that line of code, `d3.selectAll()`, is part of the core functionality necessary for understanding D3: selections. Selections can be made with `d3.select()`, which selects a single element, but you’ll use `d3.selectAll()`, which can be used to select multiple elements, more often. Selections are a group of one or more web page elements that may be associated with a set of data, like so:

```
d3.selectAll("div.market").data([1,5,11,3])
```

Which would bind the elements in the array [1,5,11,3] to elements with the class of “market”. This association is known in D3 as **binding data** and so a selection should be thought of as a set of web page elements and a corresponding, associated set of data. Sometimes there are more data elements than DOM elements, or vice verse, in which case there are functions in D3 designed to create or remove elements that you can use to generate content. Selections and data-binding will be explained in detail in chapter 2. Selections might not include any data binding, and won’t for most of the examples in this chapter, but it’s this inclusion that allows for the powerful information visualization techniques that D3 affords. A selection can be made on any elements in a web page, which includes items in a list, circles, or even regions on a map of Africa. Just as the elements can take a number of shapes, the data associated with those elements (where applicable) can take many forms.

Imagine you have a set of data such as the price and size of a few houses, and a set of web page elements, whether graphics or traditional elements, that you want to represent that data, whether with text or through size and color A selection, then, is the group of all of them together, upon which you perform some actions such as moving them, changing their color, or updating the values in the data. You can, and will, work with the data and the web page elements separately, but the real power of D3 comes from leveraging selections to use data and web page elements together.

1.1.3 D3 is about deriving the appearance of web page elements from bound data

Once you have a selection, you can then use D3 to modify the appearance of web page elements to reflect differences in the data. You may want to make the length of a line equal to the value of the data, or change the color to particular color that corresponds to a class of data. You might want to hide or show elements as they correspond to a user’s navigation of a dataset.

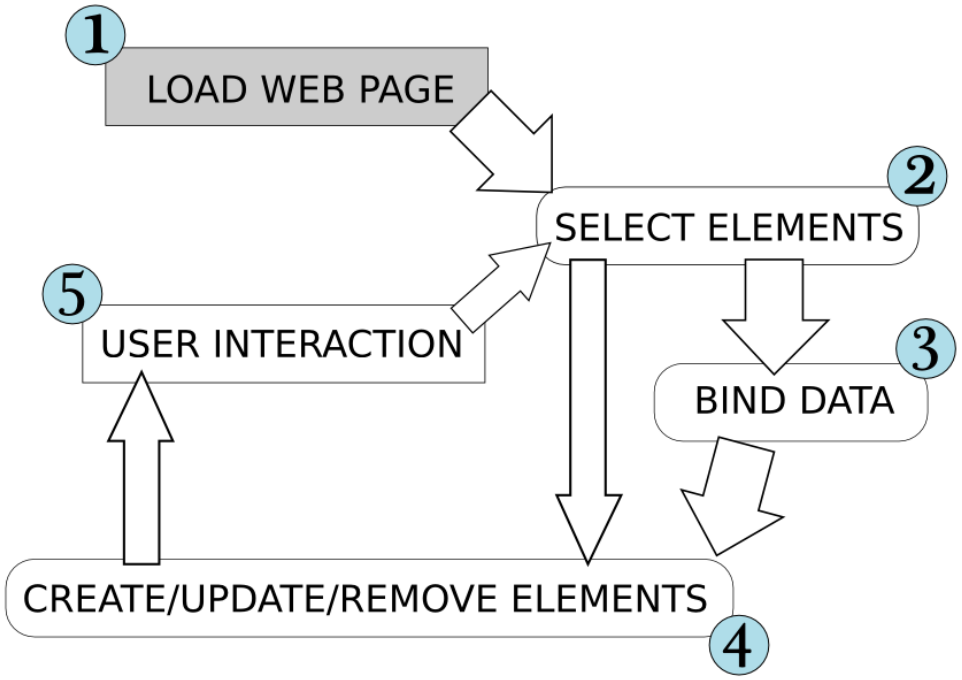


Figure 1.1 A page utilizing D3 is typically built in such a way that the page loads with styles, data, and content as defined in traditional HTML development (1) with its initial display using D3 selections of HTML elements (2) either using data binding (3) or without it to modify the structure and appearance of the page (4) . The changes in structure prompt user interaction (5) which causes new selections with and without data-binding to further alter the page. Step 1 is shown differently because it only happens once (when you load the page) whereas every other step may happen multiple times, depending on user interaction.

This is done by using selections to reference the data bound to an element in a selection. D3 is built to iterate through the elements in your selection and perform the same action using the bound data, which results in different graphical effects. While the action you perform is the same, the effect is different because it is based on the variation in the data. We'll first see data binding in process at the end of the chapter, and in much more detail throughout this book.

1.1.4 Web page elements can now be divs, countries, flowcharts

We've grown accustomed to thinking of web pages as consisting of text elements with containers for pictures or videos or embedded applications. But as you grow more familiar with D3, you'll begin to recognize that every element on the page can be treated with the same high-level abstractions. The most basic element on a web page, a that represents a rectangle into which you can drop paragraphs and lists and tables, can be selected and

modified in the same way you could select and modify a country on a web map, or individual circles and lines that make up a complex data visualization.

This doesn't always hold true. To gain access to the ability to select items on a web page, you have to ensure that they are built in a manner that makes them a part of the traditional structure of a web page. You cannot select items in a Java applet, or in a Flash runtime, nor could you select the labels on an embedded Google map, but if you create these elements so that they exist as elements in your web page, then you give yourself tremendous flexibility. To get a taste of this, look at Chapter 7, where we build robust mapping applications in D3, and you'll see the `d3.select()` syntax being used to update the appearance of a mapping application in the same manner as it's being used here and elsewhere to create and move circles or elements.

1.2 Leveraging HTML5

We've come a long way from the days when animated gifs and frames were the pinnacle of dynamic content on the web. In figure 1.2 below, we can see why gifs never caught on for robust data visualization on the web. GIFs, like the infoviz libraries designed to use VML or canvas, are still necessary for earlier browsers, but D3 is designed for modern browsers that don't need the helper libraries necessary for backward compatibility. This means that D3 development isn't for everyone, but if your audience can be assumed to have access to a modern web browser, it also brings with it a significant reduction in the cost necessary not only to code for older browsers but to learn and keep updated on the various libraries that support backward compatibility with those older browsers.

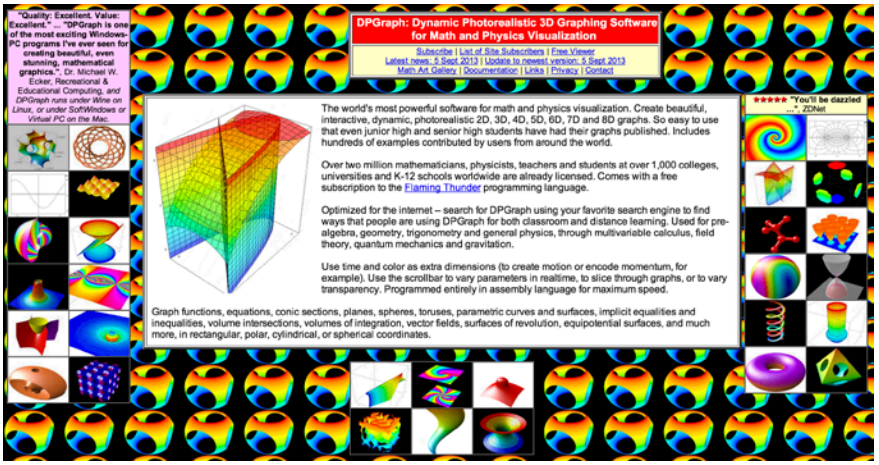


Figure 1.2 Before GIFs were weaponized to share cute animal behavior, they were your only hope for animated data visualization on the web. Few examples from the 90s like dpgraph.com still exist, but there are more than enough GIFs on this single page to remind us of the dangers of GIFs.

A modern browser typically means one that can not only display SVG graphics and obey CSS3 rules, but that does so with great performance. Along with CSS (Cascading Style Sheets) and Scalable Vector Graphics (SVG), we can break down HTML5 into the Document Object Model (DOM) and JavaScript. The following sections will treat with each of them and include code you can run to see how D3 leverages their functionality to create interactive and dynamic web content.

1.2.1 The DOM

A web page is structured according to the Document Object Model, or DOM. You need a passing familiarity with the DOM to do web development, and it's beyond what we can cover in this book. Instead, we'll take a quick look at DOM elements and structure in a simple web page in your browser and touch on the basics of the DOM. To get started, you'll need a web server that you can access from the computer that you're using to code. With that in place, you can download the D3 library from d3js.org (d3.js or d3.min.js for the minified version) and place that in the directory where you'll make your web page. We'll create a simple page called d3ia.html in the text editor with the following contents:

```
<!doctype html>
<html>
<script>d3.v3.min.js</script>#a
<body> #b
<div id="someDiv" style="width:200px;height:100px;border:black 1px solid;">
#c
<input id="someCheckbox" type="checkbox" /> #d
</div>
</body>
<html>
```

#a a child element of <html>
#b a child element of <html>
#c a child element of <body>
**#d a child element of **

Basic HTML like the kind seen here follows the DOM. It defines a set of nested elements, starting with an `<html>` element with all its child elements and their child elements and so on. In the above example, the `<script>` and `<body>` elements are children of the `<html>` element and the `<div>` element is a child of the `<body>` element. The `<script>` element loads the D3 library here, or it can have inline JavaScript code, whereas any content in the `<body>` element shows up onscreen when you navigate to this page.

UTF-8 and D3.js

D3 utilizes UTF-8 characters in its code, which means that you can do one of three things to make sure you don't have any errors:

You can either set your document to utf-8:

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

Or you can set the charset of the script to utf-8:

```
<script charset="utf-8" src="d3.js"></script>
```

Or you can use the minified script, which shouldn't have any utf-8 characters in it:

```
<script src="d3.min.js"></script>
```

There are three categories of information about each element that determine its behavior and appearance: styles, attributes, and properties. **Styles** can determine transparency, color, size, borders and so on. **Attributes** typically refer to classes, IDs, and interactive behavior, though some attributes can also determine appearance, depending on which type of element you're dealing with. **Properties** typically refer to states, such as the "checked" property of a checkbox, which is true if the box is checked and false if the box is unchecked. D3 has three corresponding functions to modify these values. If we wanted to modify the HTML elements above, we could do so using D3 functions that abstract this process:

```
d3.select("#someDiv").style("border", "5px darkgray dashed")
d3.select("#someDiv").attr("id", "newID")
d3.select("#someCheckbox").property("checked", true)
```

Like many D3 functions of this kind, if you don't signify a new value, then the function will return the existing value. You'll see this in action throughout this book, and later in the chapter as we write more code, but for now just remember that these three functions allow you to change how an element appears and interacts.

The DOM also determines the order of drawing of elements onscreen, with child elements drawn after and inside parent elements. While there is some control over drawing elements above or below each other with traditional HTML using `z-index`, this isn't available for SVG elements (though it might be implemented at some point using the `render-order` attribute).

Examining the DOM in the Console

Navigate to d3ia.html and we can get some exposure to how D3 works. The page isn't very impressive, with just a single black-outlined rectangle. You could modify the look and feel of this web page by updating d3ia.html, but you'll find that it is very easy to modify the page by using your web browser's developer console. You'll find this useful for testing changes to classes or elements before implementing them in your code. Open up the developer console and you'll have two very useful screens shown in figures 1.3 and 1.4, which we'll go back to over and over.

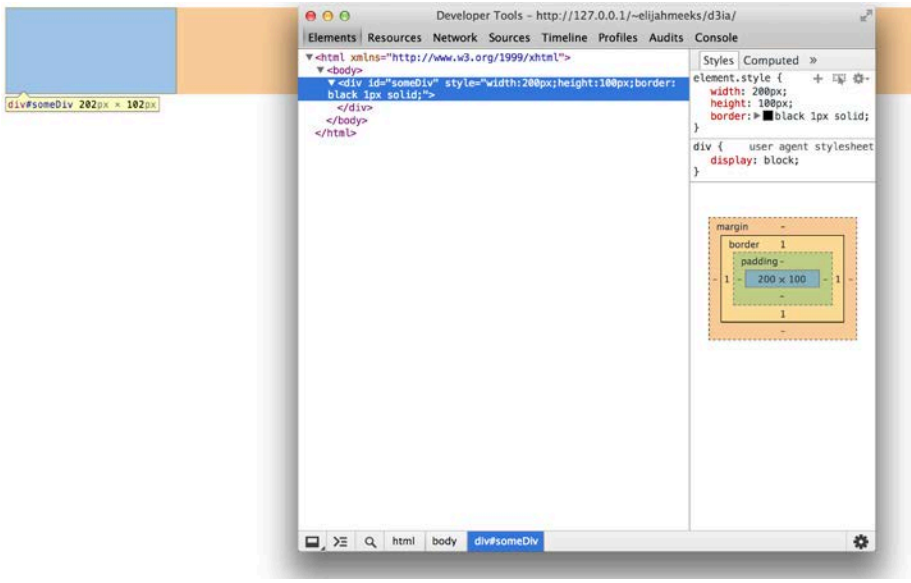


Figure 1.3 The developer tools in Chrome places the JavaScript console on the rightmost tab labeled 'Console' with the element inspector available using the hourglass on the bottom left or by browsing the DOM in the 'Elements' tab.

The element inspector allows you to look at the elements that make up your web page by navigating through the DOM (represented as nested text where each child element is shown indented). Or by selecting an element on screen graphically, typically represented as a magnifying glass or cursor icon.

The other screen you'll want to use quite often is the console (Figure 1.x), which allows you to write and run JavaScript code right on your web page:

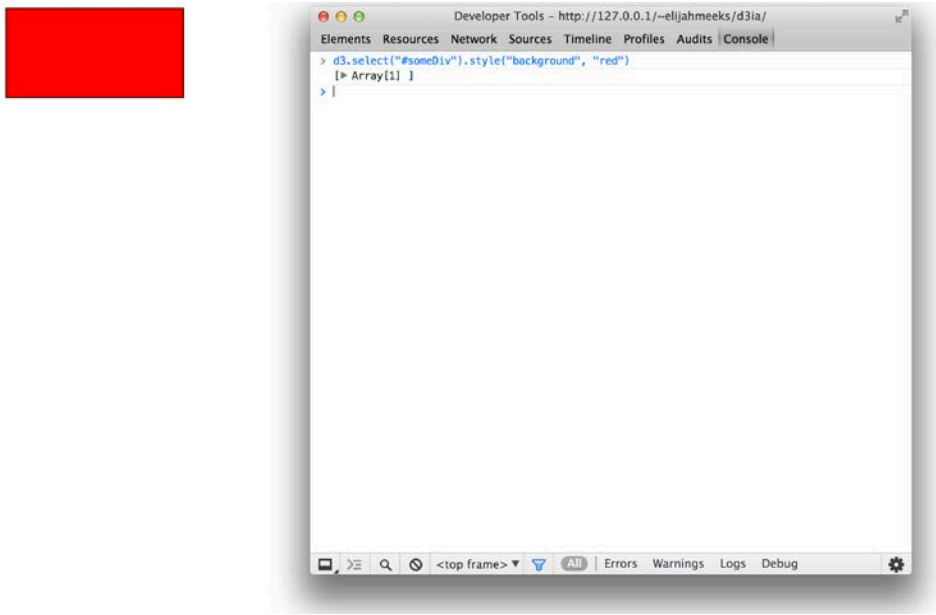


Figure 1.4 You can run JavaScript code in the console, and also call global variables or declare new ones as necessary. Any code you write in the console and changes made to the web page are lost as soon as you reload the page.

The examples in this book use Google Chrome and its developer console, but you could use Safari's developer tools or Firebug in Firefox or whatever developer console you're most comfortable with to follow along. You can see and manipulate DOM elements such as the or <body> above by clicking on the element inspector or looking at the DOM as represented in HTML. You can click on one of these elements and change its appearance by modifying it in the console.

You can even delete elements in the console. Give it a try: select the div either in the DOM or visually and press delete. Now your webpage is very lonely. Hit refresh so that your page reloads the HTML and your div comes back. You can adjust the size and color of your div by adding new styles or changing the existing one, so you can increase the width of the border and making it dashed by changing the border style to "black 5px dashed". Or you can add content to the div in the form of other elements or simply text by right-clicking on the element and selecting "edit as HTML" as shown in figure 1.5 and 1.6.

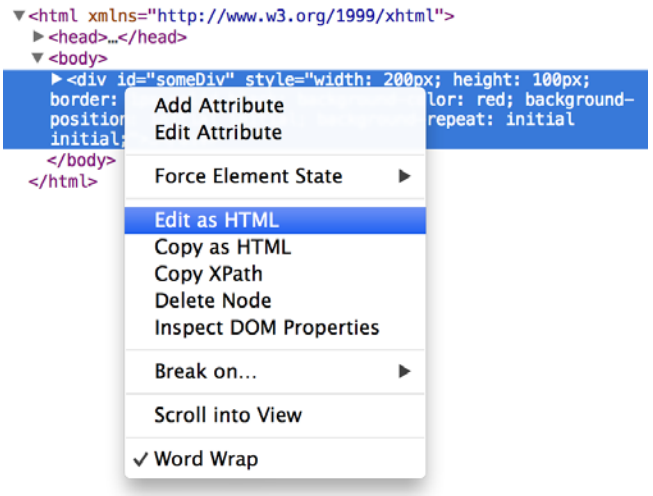


Figure 1.5 Rather than adding or modifying individual styles and attributes, you have the ability to rewrite the HTML code as you would in a text editor. As with any changes, these only last until you reload the page.

And then writing in between the opening and closing HTML whatever you'd like:

```
background-position: initial initial; ba
repeat: initial initial;">
    Here's some text to put in my div
```

Figure 1.X Changing the content of a DOM element is as simple as adding text between the opening and ending brackets of the element.

Any changes you make, regardless of whether they're well-structured or not, will be reflected on the web page. In figure 1.7 we see the results of modifying the HTML, which is rendered immediately on our page.

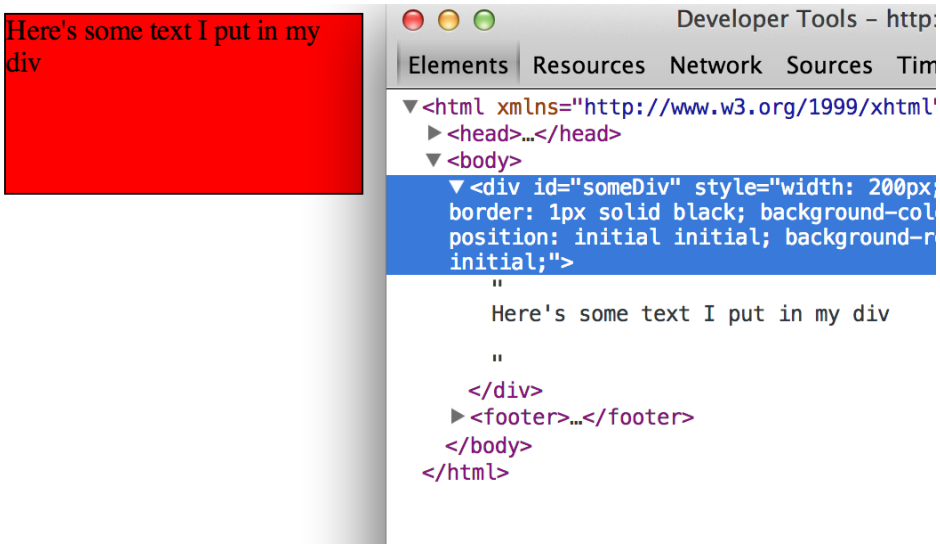


Figure 1.7 The page is updated as soon as you finish making your changes. Writing HTML manually in this way is really only useful for planning how you might want to dynamically update the content.

In this way, you could slowly and painstakingly create a web page in the console. We're not going to do that. Instead, we're going to use D3 to create elements on the fly with size, position, shape and content based on your data.

1.2.2 Coding in the Console

You'll do a lot of your coding in the IDE of your choice, but one of the great things about web development is that you can test JavaScript code changes by using your console. Later we'll focus on writing JavaScript, but for now, just to demonstrate how the console works, copy the following code into your console and hit enter and you should see the effect show in figure 1.8.

```
d3.select("div").style("background","lightblue").html("Something else maybe")
```

Figure 1.8 The D3 select syntax modifies style using the .style() function and traditional HTML content using the .html() function.

Now, if all D3 could do was select HTML elements and change their style and content like this, then it wouldn't be much of a library. To do more, we have to move away from traditional HTML and focus on a special type of elements in the DOM: Scalable Vector Graphics.

1.2.3 SVG

A major value of HTML5 is the integrated support for SVG (scalable vector graphics). These allow for simple mathematical representation of images that scale and are amenable to animation and interaction. Part of D3's attractiveness is that it provides an abstraction layer for drawing SVG. This is because SVG drawing can be a little confusing. SVG drawing instructions for complex shapes, known as `<path>` elements, is written a bit like the old LOGO programming language. You start at a point on a canvas and draw a line from that point to another, and if you want it to curve you can give the SVG drawing code coordinates on which take make that curve. So if you want to draw the line on the right, you would create a <path> element in an <svg> canvas element in your web page, and you would set the "d" attribute of that <path> element to equal the text on the left:

```
M50,485.7142857142857L58.33333333333332,473
.8095238095238C66.66666666666666,461.904761
90476187,83.33333333333331,438.095238095238
1,99.99999999999999,433.3333333333333C116.6
6666666666666,428.57142857142856,133.333333
33333334,442.85714285714283,150,445.2380952
380952C166.66666666666669,447.6190476190476
,183.33333333333331,438.0952380952381,199.9
9999999999997,421.4285714285714C216.6666666
6666663,404.7619047619047,233.3333333333333
1,380.9523809523809,250,383.3333333333333C2
66.6666666666667,385.71428571428567,283.333
33333333333337,414.2857142857143,300,430.95238
095238096C316.66666666666674,447.6190476190
476,333.33333333333337,452.3809523809524,35
0,450C366.6666666666667,447.6190476190476,3
83.3333333333333,438.0952380952381,400,442.
85714285714283C416.66666666666663,447.61904
76190476,433.3333333333333,466.666666666666
63,449.99999999999994,469.047619047619C466.
66666666666663,471.42857142857144,483.33333
33333333333,457.1428571428571,491.666666666666
63,450L500,442.8571428571429
```

But you would almost never want to create SVG by manually writing drawing instructions like this. Instead, you'll want to use D3 to do the drawing with a variety of helper functions, or rely on other SVG elements that represent simple shapes (known as geometric or graphical primitives) using more readable attributes. We'll update d3ia.html to look like Listing 1.1 to include the necessary elements to display SVG, as well as some examples of the various shapes you might use:

Listing 1.1 A sample web page with SVG elements

```
<!doctype html>
<html>
<script src="d3.v3.min.js" type="text/JavaScript">
</script>
<body>
<div id="infovizDiv">
<svg style="width:500px;height:500px;border:1px lightgray solid;">
  <path d="M 10,60 40,30 50,50 60,30 70,80"
style="fill:black;stroke:gray;stroke-width:4px;" />
  <polygon style="fill:gray;" points="80,400 120,400 160,440 120,480 60,460"
/>
<g>
<line x1="200" y1="100" x2="450" y2="225"
style="stroke:black;stroke-width:2px;"/>
<circle cy="100" cx="200" r="30"/>
<rect x="410" y="200" width="100" height="50"
style="fill:pink;stroke:black;stroke-width:1px;" />
</g>
</svg>
</div>
</body>
</html>
```

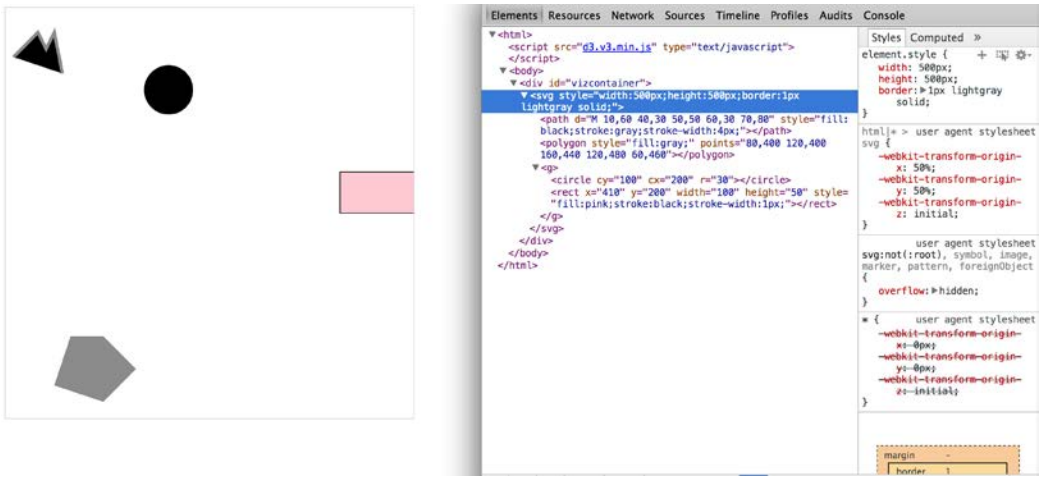


Figure 1.9 Inspecting the DOM of a web page with an SVG canvas reveals the nested graphical elements as well as the style and attributes that determine their position. Notice that the circle and rectangle exist as child elements of a group.

You can inspect the elements just like you would the traditional elements we looked at earlier, as you can see in figure 1.9, and you can manipulate these elements using traditional

JavaScript selectors like `document.getElementById` or with D3, removing them or changing the style like so:

```
d3.select("circle").remove()
d3.select("rect").style("fill", "purple")
 #a deletes the circle
 #b changes the rectangle color to purple
```

Now, refresh your page and let's take a look at the new elements. You're already familiar with divs, and it's always useful to put an SVG canvas in a div so you can access the parent container for layout and styling. Let's take a look at each of the elements we've added:

<SVG>

This is your actual canvas on which everything is drawn. The top-left corner is 0,0 and the canvas will clip anything drawn beyond its defined height and width of 500,500 (the rectangle in our example). An <svg> element can be styled with CSS to have different borders and backgrounds. The <svg> element can also be dynamically resized using the viewBox attribute, which is more complex and beyond the scope of the overview here.

You can use CSS (which we'll touch on later in this chapter) to style your SVG canvas or use D3 to add inline styles like so:

```
d3.select("svg").style("background", "darkgray") #a
 #a infoviz is always cooler on a dark background
```

REMEMBER: The x-axis is drawn left to right, but the y-axis is drawn top to bottom, so you'll see that the circle is set 200 pixels to the right and 100 pixels down.

We'll look at a few of the SVG elements below. Each of which will include some examples that you can paste into the <svg> element in your HTML to see on your web page.

<canvas>

There's a second mode of drawing available with HTML5 using <canvas> elements to draw bitmaps. We won't be going into in detail here but will see this method used in Chapter 8. Canvas creates static graphics drawn in a manner similar to SVG that can then be saved as images. There are four main reasons to use canvas:

Compatibility - which we won't worry about because if you're using D3, then you're coding for a modern browser.

Creating static images - You can draw your data visualization with canvas to save views as snapshots for thumbnail and gallery views.

Large amounts of data - SVG creates individual elements in the DOM and while this is great for attaching events and styling, it can overwhelm a browser and cause significant slowdown.

WebGL - Canvas allows you to use webGL to draw, so that you can create 3D objects. There are also ways to create 3D objects like globes and polyhedrons using SVG, which we'll get into a bit in Chapter 7 as we examine geospatial information visualization.

<CIRCLE> <RECT> <LINE> <POLYGON>

SVG provides a set of common shapes, each of which has **attributes** that determine their size and position in such a way that is easier to deal with than the generic "d" attribute we saw above. These attributes vary depending on the element that you're dealing with, so that the `<rect>` has "x" and "y" attributes that determine the shape's top-left corner, as well as "height" and "width" attributes that determine its overall form. In comparison, the `<circle>` element has a "cx" and a "cy" attributes that determine the center of the circle, and an "r" attribute that determines the radius of the circle. The `<line>` element has "x1" and "y1" attributes that determine the starting point of the line and "x2" and "y2" attributes that determine its end point. There are other simple shapes that are similar to these, such as the `<ellipsis>` and there are more complex shapes, like the `<polygon>` that has a "points" attribute that holds a set of comma-separated XY coordinates in clockwise order that determines the area bounded by the polygon.

Infoviz Vocabulary: Geometric Primitive

Accomplished artists can draw anything with vector graphics, but you're probably not looking at D3 because you're an artist. Instead, you're dealing with graphics with more pragmatic goals in mind. From that perspective, it's important to understand the concept of geometric primitives (also known as graphical primitives). Geometric primitives are simple shapes such as points, lines, circles and rectangles. These simple shapes, which can be combined to make more complex graphics, are particularly useful for visually displaying information.

Primitives are also useful for understanding complex information visualization that you see out in the world. Dendrograms like that seen in Figure 1.20 are far less intimidating when you realize they are just circles and lines. Interactive timelines are easier to understand and create when you think of them as collections of rectangles and points. Even geographic data, which primarily comes in the form of polygons, points and lines, is less confusing when you break it down into its most basic graphical structures.

Each of these attributes can be hand-edited in HTML to adjust their size, form and position. Open up your element inspector and click on the `<rect>`. Change its "width" to 25 and its "height" to 25 as shown in Figure 1.10.

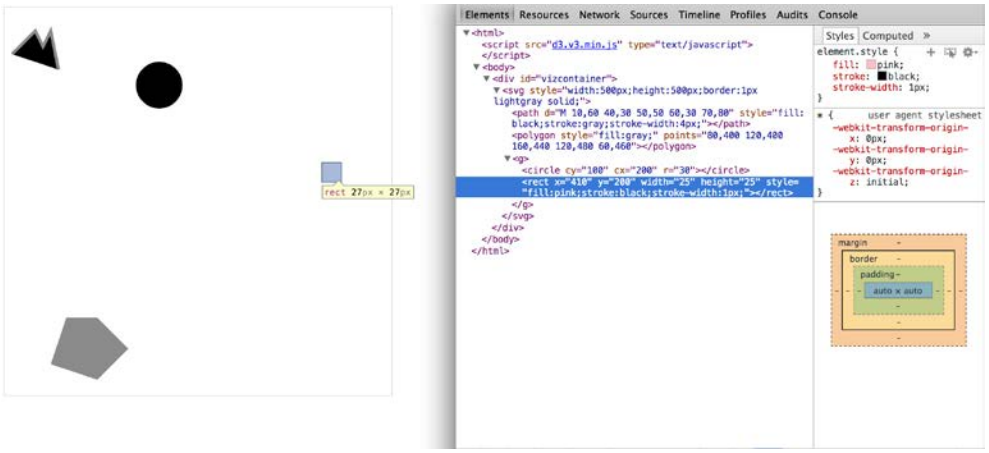


Figure 1.10 Modifying the height and width attributes of a <rect> element will change the appearance of that element. Inspecting the element also shows how the stroke adds to the computed size of the element.

Now you've learned why there is no SVG <square>. The color, stroke and transparency of any shape can be changed by adjusting the style of the shapes, with "fill" determining the color of the area of the shape while "stroke" and "stroke-width" and "stroke-dasharray" determine its outline.

Notice, though, that the inspected element has a measurement of 27px x 27px. That's because the 1px stroke is drawn on the outside of the shape. That makes sense, once you know the rule, but if you change the stroke-width to "2px" it will still be 27px x 27px. That's because the stroke is drawn evenly over the inside and outside border. This may not seem too big a deal, but it's something to remember when you're trying to line up your shapes later on.

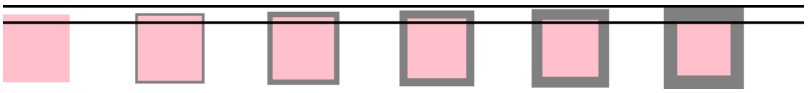


Figure 1.11 The same 25x25 <rect> with no, 1px, 2px, 3px, 4px and 5px stroke. Though these are being drawn on a retina screen using half-pixels, the second and third report the same width and height (27px x 27px) as do the fourth and fifth (29px x 29px).

Go ahead and change the style parameters of the rectangle to:

```
"fill:purple;stroke-width:5px;stroke:cornflowerblue;".
```

Congratulations, you've now successfully visualized the complex and ambiguous phenomenon known as "ugly".

<TEXT>

SVG provides the capacity to write text as well as shapes. SVG text, though, does not have the formatting support found in HTML elements, and so it's primarily used for labels. If you do want to do some basic formatting, you can nest <tspan> elements within <text> elements.

<G>

The <g> or group element is distinct from the above SVG elements in that it has no graphical representation and does not exist as a bounded space. Instead, it is a logical grouping of elements. You'll want to use <g> elements extensively when creating graphical objects that are made up of several shapes and text. For instance, if you wanted to have a circle with a label above it and move the label and the circle at the same time, then you would place it inside a <g> element:

```
<g>
<circle r="2"/>
<text>This circle's Label</text>
</g>
```

Moving a <g> around your canvas requires you to adjust the "transform" attribute of the <g> element. The "transform" attribute is more intimidating than the various x/y attributes of shapes, since it accepts a structured description in text of how you want to transform a shape. One of those structures is "translate()" which will accept a pair of coordinates that will move the element to the x and y position defined by the values in "translate(x,y)". So if you want to move a <g> element 100 pixels to the right and 50 pixels down, then you need to set its "transform" attribute to transform="translate(100,50)". The transform attribute also accepts a "scale()" setting, so that you can change the rendered scale of the shape. We can see these settings in action by modifying the example above with the results shown in figure 1.12.

```
<g>
<circle r="2"/>
<text>This circle's Label</text>
</g>
<g transform="translate(100,50)">
<circle r="2" />
<text>This circle's Label</text>
</g>
<g transform="translate(100,50) scale(2.5)">
<circle r="2"/>
<text>This circle's Label</text>
</g>
```

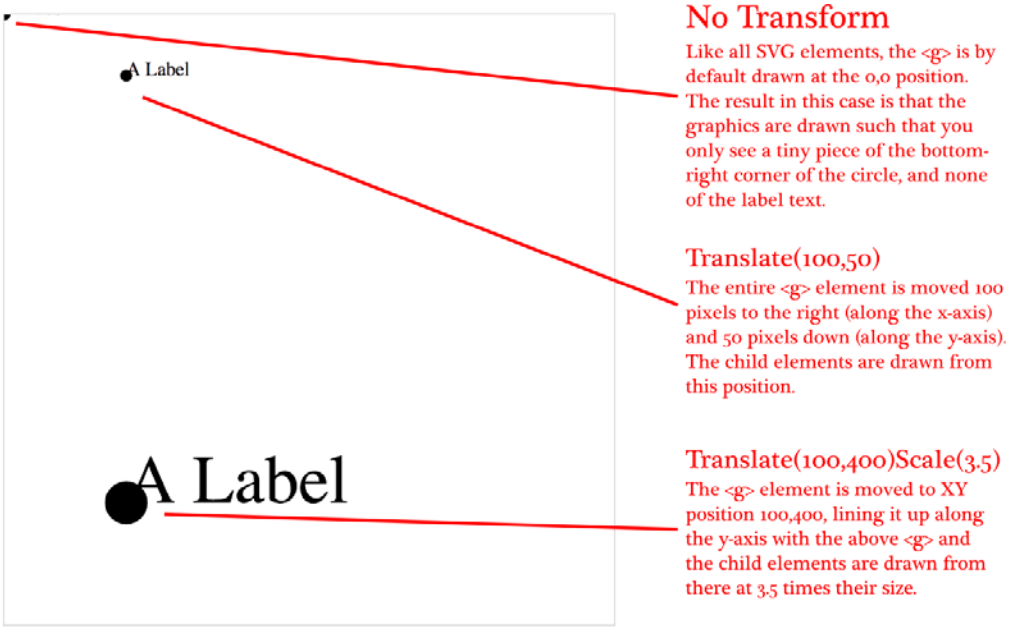


Figure 1.12 All SVG elements can be affected by the Transform attribute, but this is particularly salient when working with <g> elements, which require this approach to adjust their position. The child elements are drawn from using the position of their parent <g> as their relative 0,0 position. The scale() setting in the transform attribute then affects the scale of any of the size and position attributes of the child elements.

<PATH>

A path is an area determined by its "d" attribute. Paths can be open or closed, meaning the last point connects the first if closed and does not if open. The open or closed nature of a path is determined by the absence or presence of the letter "Z" at the end of the text string in the "d" attribute. It can still be filled either way. You can see the difference in figure 1.13.

```
<path style="fill:none;stroke:gray;stroke-width:4px;" d="M 10,60 40,30 50,50
60,30 70,80" transform="translate(0,0)" />
  <path style="fill:black;stroke:gray;stroke-width:4px;" d="M 10,60 40,30
50,50 60,30 70,80" transform="translate(0,100)" />
  <path style="fill:none;stroke:gray;stroke-width:4px;" d="M 10,60 40,30
50,50 60,30 70,80Z" transform="translate(0,200)" />
  <path style="fill:black;stroke:gray;stroke-width:4px;" d="M 10,60 40,30
50,50 60,30 70,80Z" transform="translate(0,300)" />
```

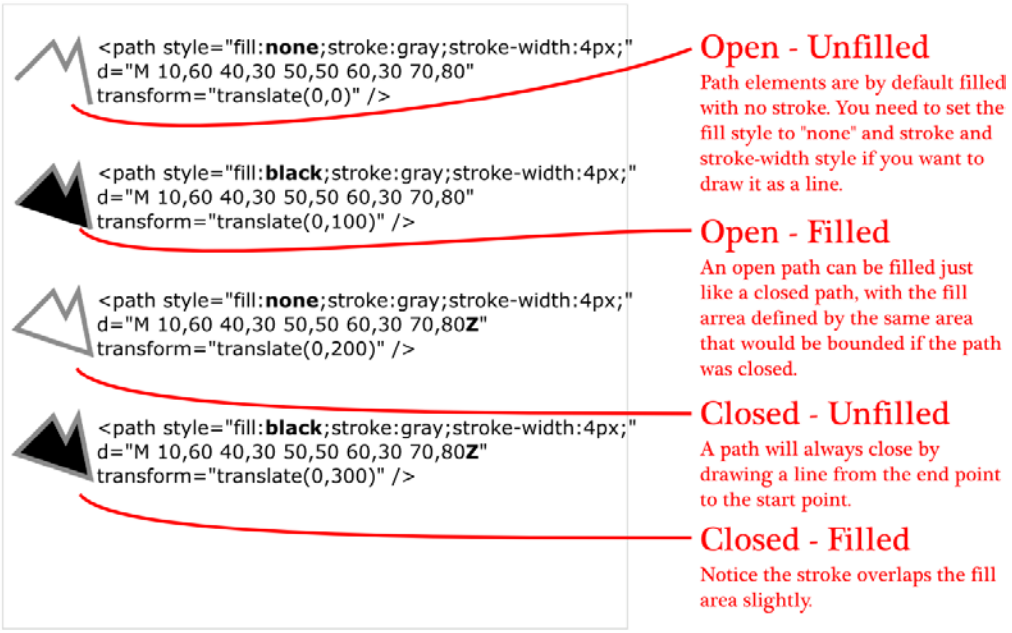


Figure 1.13 Each path shown here uses the same coordinates in its “d” attribute, with the only differences between them being the presence or absence of “Z” at the end of the text string defining the “d” attribute, the settings for fill and stroke, and the position via the “transform” attribute.

While there are times when you may want to write that "d" attribute yourself, it’s more likely that your experience crafting SVG will come in one of three ways: using geometric primitives like circles, rectangles or polygons; drawing SVG using a vector graphics editor like Adobe Illustrator or Inkscape; or drawing SVGs parametrically using hand-written constructors or built-in constructors in D3. Most of this book will focus on using D3 to create SVG, but don’t overlook the possibility of creating SVG in another and then manipulating it using D3 like we’ll do using d3.html in Chapter 3.

1.2.4 CSS

Cascading Style Sheets (CSS) are used to style the elements in the DOM. A stylesheet can exist as a separate .css file that you include in your HTML page or can be embedded directly in the HTML page. Stylesheets refer to an ID or Class or type of element and determine the appearance of that element. The terminology used to define the style is known as a CSS selector and is the same type of selector used in the `d3.select()` syntax. You can set inline styles (that are applied to only a single element) by using `d3.select("#someElement").style("opacity", .5)` to set the opacity of an element to 50%. Let’s update our d3ia.html to include a stylesheet as seen in Listing 1.2.

Listing 1.2 A sample web page with a stylesheet

```
<!doctype html>
<html>
<script src="d3.v3.min.js" type="text/JavaScript">
</script>
<style>
.inactive, .tentative {
  stroke: darkgray;
  stroke-width: 2px;
  stroke-dasharray: 5 5;
}

.tentative {
  opacity: .5;
}

.active {
  stroke: black;
  stroke-width: 4px;
  stroke-dasharray: 1;
}

circle {
  fill: red;
}

rect {
  fill: darkgray;
}
</style>
<body>
<div id="infovizDiv">
<svg style="width:500px;height:500px;border:1px lightgray solid;">
  <path d="M 10,60 40,30 50,50 60,30 70,80" />
  <polygon style="fill:gray;" />
<g>
<circle class="active tentative" cy="100" cx="200" r="30"/>
<rect class="active" x="410" y="200" width="100" height="50" />
</g>
</svg>
</div>
</body>
</html>
```

The results stack on each other, so when we examine the rectangle element, as shown in figure 1.14, we can see that its style is set by the reference to `rect` in the stylesheet as well as the reference to it having the class attribute of `active`.

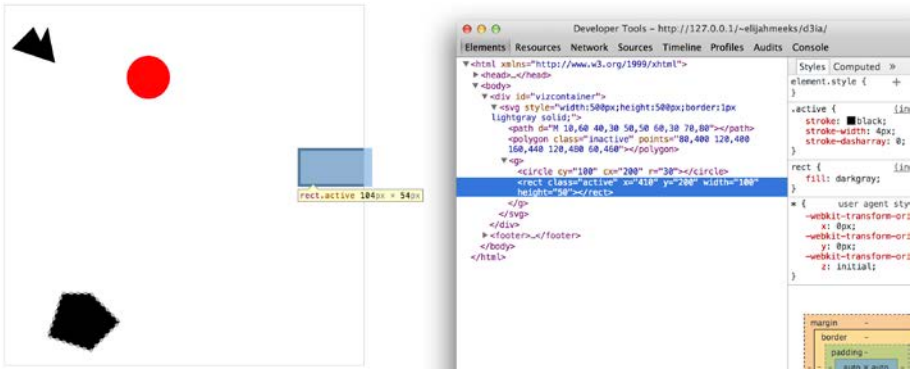


Figure 1.14 Examining an SVG rectangle in the console shows that it inherits its fill style from the CSS style applied to <rect> types and its stroke style from the .active class.

Stylesheets can also refer to a state of the element, so that with :hover you can change the way an element looks when the user mouses over that element. There are other complex CSS selectors that you can find out about in more detail in a book devoted to that subject. For this book, we'll focus mostly on using CSS classes and IDs for selection and to change style. The most useful way to do this is to have CSS classes associated with particular stylistic changes and then change the class of an element. You can change the class of an element, which is an attribute of an element, by selecting and modifying the class attribute. The circle shown in figure 1.15 is affected by two overlapping classes: .active and .tentative.

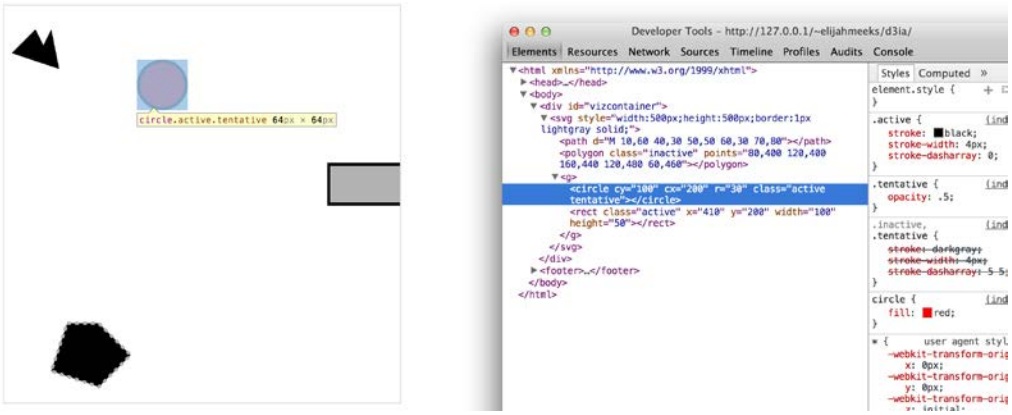


Figure 1.15 The SVG circle has its fill value set by its type in the stylesheet, with its opacity set by its membership in the .tentative class and its stroke set by its membership in the .active class. Notice that the stroke settings from the .tentative class are overwritten by the stroke settings in the later declared .active class.

Here we have a couple overlapping classes defined, with tentative, active, and inactive all applying different style changes to our shape. There are times when an element is just in one of these classes, in which case you could overwrite the class attribute entirely:

```
d3.select("circle").attr("class", "tentative");
```

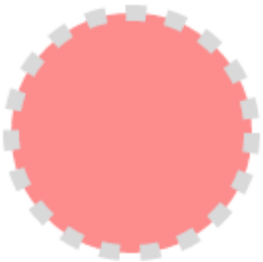


Figure 1.16 An SVG circle with fill style determined by its type and its opacity and stroke settings determined by its membership in the tentative class.

The results, as we see in Figure 1.16, are what we would expect. But this overwrites the entire class attribute to the value you set. But elements can have multiple classes and there are times when something is both active and tentative or inactive and tentative, so let's reload the page and take advantage of the helper function `d3.classed()`, which allows you to add or remove a class from the classes in an element:

```
d3.select("circle").classed("active", true);
```

By using `.classed()`, we don't overwrite the existing attribute, but rather append or remove the named class from the list. Here we can see the results of two classes with conflicting styles defined. The active style overwrites the tentative style because it occurs later in the stylesheet. Another rule to remember is that more specific rules overwrite more general rules. There's more to CSS, but there are other books for that.

By defining style in your stylesheet and changing appearance based on class membership, you create code that is more maintainable and readable. You'll need to use inline styles to set the graphical appearance of a set of elements to a variety of different values, such as changing the fill color to correspond to a color ramp based on the data bound to that set of elements. We'll see that functionality in action later as we deal with bound data. But as a general rule setting inline styles should only be used when you can't use traditional classes and states defined in a stylesheet.

1.2.5 JavaScript

D3, like many information visualization libraries in JavaScript, provides functions to abstract the process of creating and modifying. On top of that, it provides mechanisms to link data and web page elements in a way that makes the drawing and updating of these SVG elements reusable and maintainable. But these mechanisms are also applicable to more traditional HTML elements like paragraphs and divs.

As a result, a web application written in D3 can accomplish much of the user interface functionality that users expect without relying on libraries like jQuery. This is because the latest version of JavaScript has built-in functionality that once required the custom. If you read the solutions on StackOverflow, you may think that being a JavaScript developer requires being a jQuery developer, but look more closely and you'll see that unless you're developing for an audience using a browser from the Bush administration, which by necessity means it won't support the key features of D3, or you need to use a plugin that requires jQuery, then you might just as easily write the same functionality in plain JavaScript.

When it comes to writing JavaScript, there are two subjects that you should be familiarize yourself with: method chaining and arrays:

METHOD CHAINING

D3 examples, like many examples written in JavaScript, use method chaining extensively. Method chaining, also known as function chaining, is facilitated by returning the method itself with the successful completion of functions associated with a method. One way to think of method chaining is to think of how we talk and refer to each other. Imagine you were talking to someone at a party, and you asked about another guest:

"What's her name?"
"Her name is Lindsay."
"Where does she work?"
"She works at Tesla."
"Where does she live?"
"She lives in Cupertino."
"Does she have any children?"
"Yes, she has a daughter."
"What's her name?"

Do you think the answer to that last question would be "Lindsay"? Of course not, we'd expect the answer to refer to Lindsay's daughter, even though all the previous questions referred to Lindsay. Method chaining is like that. Method chaining is used a lot in D3 examples, which means you'll see something like this written on one line or formatted (but functionally identical) to something written on multiple lines:

```
d3.selectAll("div").data(someData).enter().append("div").html("Wow").append("
span").html("Even More Wow").style("font-weight", "900")
```

It's the same as what's below. The only change is in the use of line breaks, which JavaScript ignores:

```
d3.selectAll("div") #a
.data(someData) #b
.enter() #c
.append("div") #d
.html("Wow") #e
.append("span") #f
.html("Even More Wow") #g
.style("font-weight", "900"); #h
```

#a Returns the Function 1, a selection
#b Sets the data on Function 1 and returns Function 1
#c Returns Function 2, the selection.enter() function
#d Sets .append() behavior on function 2 and returns function 3, a selection
#e Sets .html() for function 3 and returns function 3
#f Sets the append() behavior on function 3 and returns function 4
#g Sets the html() for function 4 and returns function 4
#h Sets the font-weight style on function 4 & returns function 4

You could write each line separately, declaring the different variables as you go, and achieve the same effect. It might also make method chaining make a little more sense to you if you haven't been exposed to it before:

```
var function1 = d3.selectAll("div");
function1.data(someData);
var function2 = function1.enter();
var function3 = function2.append("div");
function3.html("Wow");
var function4 = function3.append("span");
function4.html("Even More Wow");
function4.style("font-weight", "900");
```

We can see this running the code in our console. This is the first time we've used the `.data()` function, which along with `.select()` is at the core of developing with D3. When you use `.data()` you're binding each element in your selection to each item in an array. If you have more items in your array than elements in your selection, then you can use the `.enter()` function to define what to do with each extra element. In the above function, we're selecting all the elements in the `<body>` and the `.enter()` function tells D3 to `.append()` a new div when there are more elements in the array than elements in the selection. Given that our d3ia.html page already has one div, this means if we bind an array with more than one value, D3 will append, or add, a div for each value in the array beyond the first.

There's a corresponding `.exit()` function that defines how to respond when there are fewer values in an array than there are elements in a selection. For now, we're just going to run the code as it appears in the examples, and in later chapters we'll get into much more detail on the way selections and binding work.

With this example, we're not doing anything with the data in the array and are only creating elements based on the size of the array (one `<div>` for each element in the array). For this to work, you need to give someData a value. With that in place, you can run your code:

```
var someData = ["filler", "filler", "filler", "filler"];
d3.select("body").selectAll("div").data(someData).enter().append("div").html(
"Wow").append("span").html("Even More Wow").style("font-weight", "900");
```

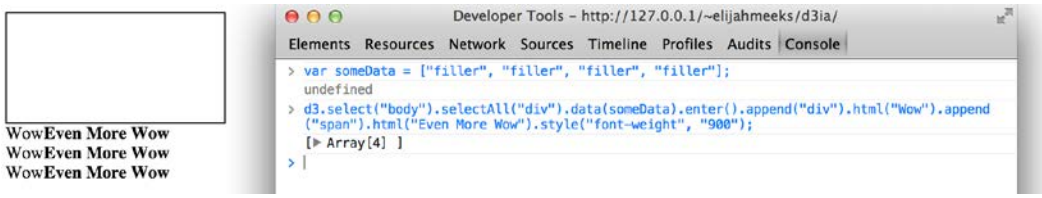


Figure 1.17 By binding an array of four values to a selection of elements on the page, we can see that the .enter() function created three new elements to reflect the size mismatch between the data array and the selection.

The result, as seen in figure 1.17, is the addition of three lines of text. It might surprise you that there are three lines, given that the array has four values, but that's because while the data was bound to the existing `<div>` element on the page, the actions defined to change the contents were only applied to the `.enter()` function, which means that they were only applied to the newly created `<div>` elements that were "entering" the DOM for the first time.

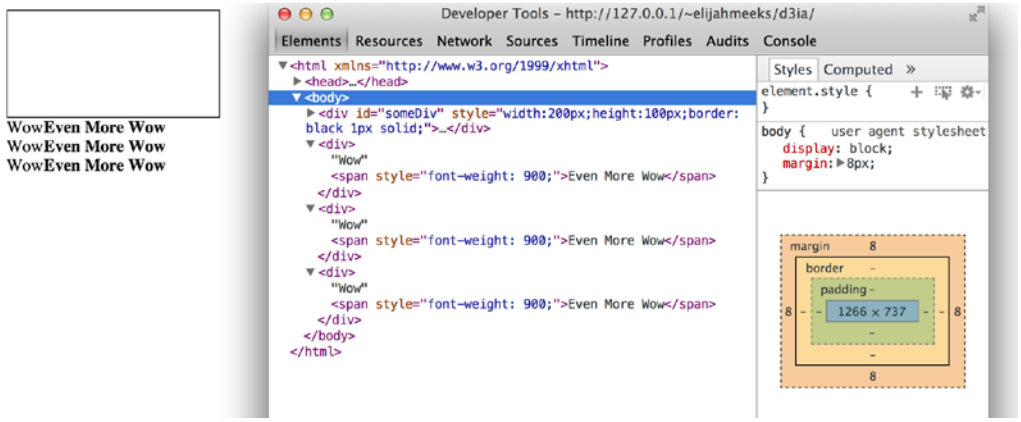


Figure 1.18 Inspecting the DOM shows that the new elements have been created with unformatted content followed by the child element with style and content set by your code.

When we inspect the DOM, as shown in figure 1.18, we can see that the method chaining operated in the manner described above. A `<div>` was added, its HTML content was set to "Wow" and a `<span>` element with a different style was appended to the `<div>` and its HTML content was set to "Even More Wow". There's a lot more we can do, but first we need to examine the array object we're binding, and focus on JavaScript arrays and array functions.

ARRAYS AND ARRAY FUNCTIONS

D3 is all about arrays and so it's important to understand the structure of arrays and the options available to you to prepare those arrays for binding to data. Your array might just be an array of string or number literals, such as this:

```
someNumbers = [17, 82, 9, 500, 40]
someColors = ["blue", "red", "chartreuse", "orange"]
```

Or it may be an array of JSON objects, which will become more common as you want to do more interesting things with D3:

```
somePeople = [{name: "Peter", age: 27}, {name: "Sulayman", age: 24}, {name:
"K.C.", age: 49}]
```

In either case, one example of an extremely useful array function is `.filter()`, which returns an array wherein the elements in that array satisfy a test you provide. For instance, here's how to create an array out of `someNumbers` that had values greater than 40:

```
someNumbers.filter(function(el) {return el >= 40})
```

Likewise, here's how you could create an array out of someColors with words shorter than five letters:

```
someColors.filter(function(el) {return el.length < 5})
```

The function `.filter()` is a method of an array and accepts a function that iterates through the array with the variable you've named. In the function above, I named that variable "el" and the function runs a test on each value by testing on "el". When that test evaluates true, the element is kept in our new array.

The result of this `.filter()` function, which you can see in figure 1.19, returns either the element or nothing depending on if it satisfies the test, building a new array only made up of the elements that do.

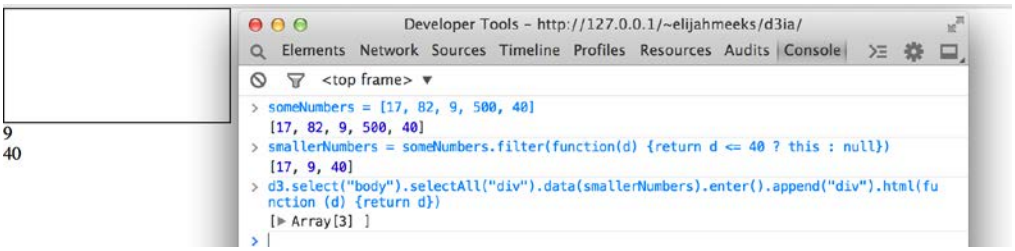


Figure 1.19 Running JavaScript in the console allows you to test your code. Here we've created a new array called smallerNumbers that consists of only three values, which we can then use as our data in a selection to update and create new elements.

```
smallerNumbers = someNumbers.filter(function(el) {return el <= 40 ? this :
null})

d3.select("body").selectAll("div").data(smallerNumbers).enter().append("div")
.html(function (d) {return d})
```

The resulting code creates two new divs from your three value array smallerNumbers (remember that one div already exists and so the `.enter()` function does not trigger though data is bound to that existing div) and the contents of the div are the values in your array. This is done through an anonymous function (sometimes referred to in D3 examples as an "accessor") in your `.html()` function and is another key aspect of utilizing D3. Any anonymous function called when setting the `.style()`, `.attr()`, `.property()`, `.html()` or other function of a selection can provide you with the data bound to that selection. As you explore examples, you'll see this function deployed over and over again:

```
.style("background", function(d,i) {return d})
.attr("cx", function(d,i) {return i})
.html(function(d,i) {return d})
```

In every case, the first variable (typically represented with the letter "d" but you can declare it as whatever you want) will contain the data value bound to that element, and the second variable will return the array position of the value bound to that element. This may seem a bit strange, but you'll get used to it as you see it utilized in a variety of ways in the upcoming chapters.

There are, of course, many other array functions, and much more you can do with JavaScript than was covered here, but that's the subject of several other books. For us, it's time to move on to understanding the kinds of data you'll work with.

1.3 Data Standards

Standardization of methods of displaying data has been fed by and feeds into standardization of methods of formatting that data. Data can be formatted in a variety of manners for a variety of purposes, but it tends to fall within a few recognizable classes: Tabular data, nested data, network data, geographic data, raw data and objects.

1.3.1 Tabular Data

Tabular data refers to data appearing in columns and rows typically found in a spreadsheet or a table in a database. While you invariably end up creating arrays of objects in D3, it is often more efficient and simply easier to pull in data in tabular format. Tabular data is delimited with a particular character, and that delimiter determines its format. So you can have a CSV, which stands for Comma-Separated Values, where the delimiter is a comma, or tab-delimited values, or a semicolon or a pipe symbol acting as the delimiter. For instance, you may have a spreadsheet of user information that includes age and salary. If you export it in a delimited form, it will look like table 1.1.

name,age,salary	name age salary	name|age|salary
Sal,34,50000	Sal 34 50000	Sal|34|50000
Nan,22,75000	Nan 22 75000	Nan|22|75000

Table 1.1 Delimited data can take different forms expressing the same data. Here we see a dataset storing name, age, and a salary of two people using commas, spaces or the bar symbol to delimit the different fields.

D3 provides three different functions to deal with pulling in tabular data: `d3.csv()`, `d3.tsv()` and `d3.dsv()`. The only difference between them is `d3.csv()` is built for comma-delimited files, `d3.tsv()` is built for tab-delimited files and `d3.dsv()` allows you to declare the delimiter. We'll see them in action throughout the book.

1.3.2 Nested Data

Data that appears in a nested manner, wherein objects exist as children of objects recursively, is very common. Many of the most intuitive layouts in D3 are based on nested data, which can be represented as trees, such as the one in figure 1.20, or packed in circles or boxes. Data is not often output in such a format, and requires a bit of scripting to organize it as such, but the flexibility of representation of such a format is worth the effort. We will see hierarchical data in detail in Chapter 5 as we look at various popular D3 layouts.

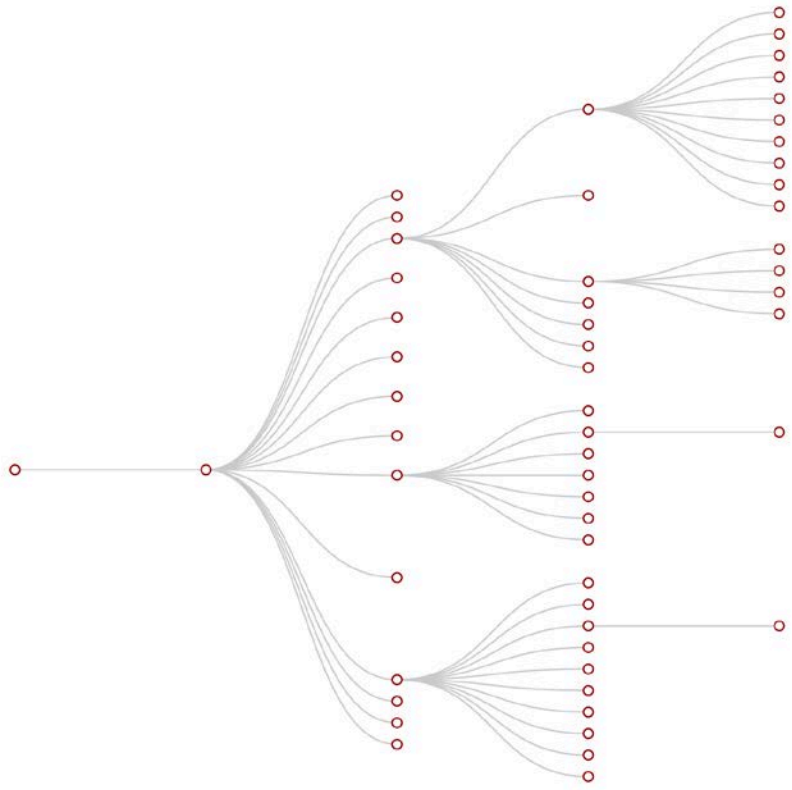


Figure 1.20 Nested data represents parent/child relationships of objects, typically with each object having an array of child objects, and is represented in a number of forms, such as this dendrogram. Notice that each object can only have one parent.

1.3.3 Network Data

Networks are everywhere. Whether they are the raw output of our social networking streams or transportation networks or a flowchart, networks are a powerful method of delivering to users an understanding of complex systems. Networks are often represented as node-link diagrams like that seen in figure 1.21. Like geographic data, there are many standards for network data, but this text will focus only on two forms: node/edge lists and connected arrays. As with geodata, network data in many forms can be easily transformed into these data types by using a freely available network analysis tool like Gephi (available at gephi.org). We will examine network data and network data standards as we deal with network visualization in Chapter 6.

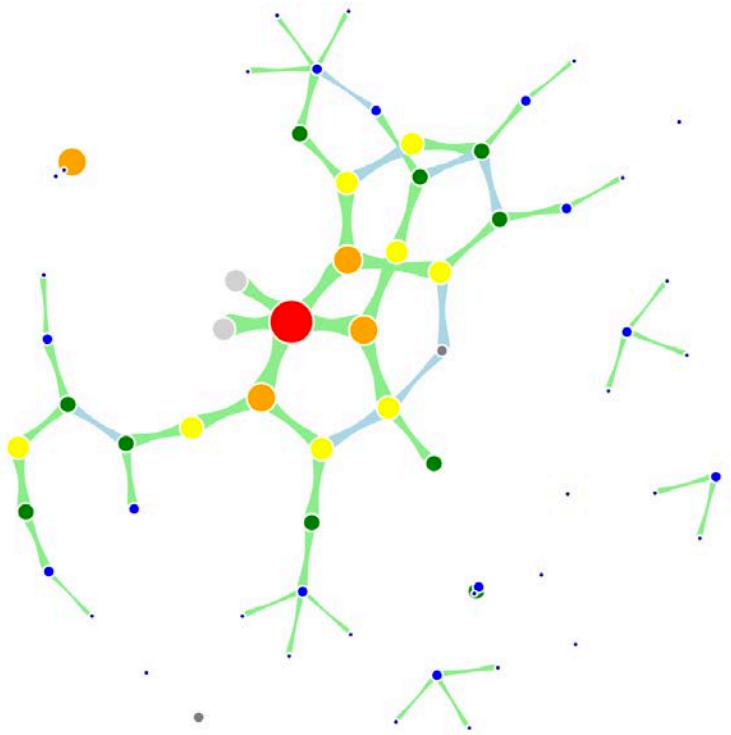


Figure 1.21 Network data consists of objects and the connections between them. The objects are typically referred to as nodes or vertices, while the connections are referred to as edges or links. Networks are often represented using force-directed algorithms such as the example here, that arrange the network in such a way to pull connected nodes toward each other.

1.3.4 Geographic Data

Geographic data consists of data that refers to locations either as points or shapes, and is used to create the variety of online maps seen on the web today, such as the map of the United States in figure 1.22. The incredible popularity of web mapping means that you can get access to a massive amount of publicly accessible geodata for any project. There are a few standards but the focus in this book will be on two: the GeoJSON and topoJSON standards. While geodata may come in many forms, the use of readily available geographic information systems (GIS) tools like Quantum GIS allow developers to transform it into this format for ready delivery to the web. We'll look at geographic data closely in Chapter 7.

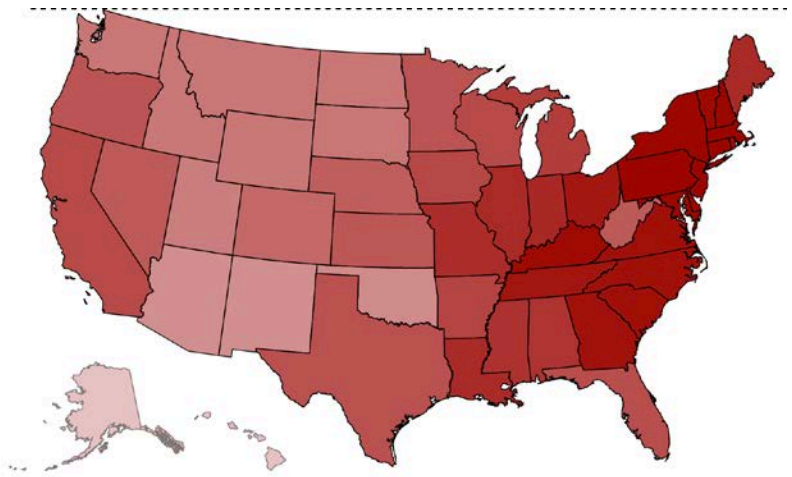


Figure 1.22 Geographic data stores the spatial geometry of objects, such as states. Each of the states in this image is represented as a separate feature with an array of values indicating their shape. Geographic data can also consist of points, such as for cities, or lines, such as for roads.

1.3.5 Raw Data

As we'll deal with in chapter two, everything is data, and this includes images or blocks of text. An important thing to remember with D3 is that while information visualization typically is thought of as using shapes encoded by color and size to represent data, sometimes the best way to represent it is with linear narrative text, an image or a video. If you plan to develop applications for an audience that needs to understand complex systems but consider the manipulation of text or images to be somehow separate from the representation of numerical or categorical data as shapes, then you arbitrarily reduce your capability to communicate. The layouts and formatting that come along with dealing with text and images, typically thought of as tied to older modes of web publication, are possible in D3, and we'll deal with that throughout this book, but especially in Chapter 8.

1.3.6 Objects

There are two types of data points that you'll be using with D3: Literals and Objects. A literal, such as a string literal like "Apple" or "beer" or a number literal like 64273 or 5.44, is pretty straightforward. A JavaScript object, expressed using JavaScript Object Notation (JSON), is not so straightforward but something that you need to understand if you plan to do sophisticated data visualization.

Let's say you have a dataset that consists of individuals from an insurance database, where you need to know how old someone is, whether they're employed, their name, and their children, if any. A JSON object that represents each individual in such a database would be expressed as follows:

```
{name: "Charlie", age: 55, employed: true, childrenNames: ["Ruth", "Charlie
Jr."]}
```

Each object is surround by { } and has attributes that have a string, number, array, boolean or object as their value. You can assign an object to variable and access its attributes by referring to them, like so:

```
var person = {name: "Charlie", age: 55, employed: true, childrenNames:
["Ruth", "Charlie Jr."]};

person.name // Charlie
person["name"] // Charlie
person.name = "Charles" // Sets name to Charles
person["name"] = "Charles" // Sets name to Charles
person.age < 65 // true
person.childrenNames // ["Ruth", "Charlie Jr."]
person.childrenNames[0] // "Ruth"
```

Objects can be stored in arrays and associated with elements using d3.select() syntax. But objects can also be iterated through like arrays using a for loop:

```
for (x in person) {console.log(x); console.log(person[x]);}
```

The x in the loop represents each attribute in the person object. Each x will be one of the attributes such as "name", "age", etc. This allows you to iterate through the attributes using `person[x]` to show the value of that attribute of the object.

If your data is stored as JSON, then you can import it using `d3.json()`, which is demonstrated many times in later chapters. But remember that whenever we use `d3.csv()`, D3 imports the data as an array of JSON objects. We'll look at objects more extensively as we use them later.

1.4 Infoviz Standards Expressed in D3

The chances are good that you're interested in D3 because of data or information visualization. Information visualization has never been so popular as it is today. The wealth of maps and charts and complex representations of systems and datasets is not just present in the workplace but in our entertainment and everyday lives. With this popularity comes a growing library of classes and sub-classes of representation of data and information using visual means, as well as aesthetic rules to promote legibility and comprehension. Your audience, whether it is the general public, or academics, or decision makers, has grown accustomed to what we once would have thought of as incredibly abstract and complicated representations of trends in data. This is why libraries like D3 aren't just popular among data scientists but also among journalists, artists, scholars, IT professionals and even fan communities.

But the wealth of options can seem overwhelming and the relative ease of modifying a dataset to appear in a streamgraph or treemap or even a simple histogram tends to promote the idea that information visualization is more about style than substance. Fortunately, there are well-established rules for what charts and methods to use for different types of data from

different systems. While I can't possibly cover every aspect of this in the book, I'll touch on things that will be useful to consider as we create more complicated information visualizations. While some developers are using D3 to revolutionize the use of color and layout, most simply want to create visual representations of data that support practical concerns. Because D3 is being developed in this mature information visualization environment, it contains numerous helper functions to let developers worry about interface and design rather than color and axes.

Still, to properly deploy information visualization, you should be familiar with what to do and what not to do. The best way to do so is to review the work of established designers and information visualization practitioners. This requires you to have a firm understanding not only of your data but of your audience. While there is an entire library of works that deal with these issues, here are a few that I've found useful and can get you oriented on the basics:

Edward Tufte	***The Visual Display of Quantitative Information***
	Envisioning Information
Isabel Meirelles	***Designing for Information***
Christian Swinehart	***Pattern Recognition***

These are by no means the only or most applicable texts for learning data visualization, but I've found them very useful, especially for getting started. The best advice is to pare down and establish fundamental, even basic, data visualization practices that clearly represent the trends that are salient to your audience. When in doubt, simplify--it is often better to present a histogram than a streamgraph, or a hierarchical network layout (like a dendrogram) than a force-directed one. The more visually complex methods of displaying data tend to inspire more excitement, but can also promote an audience seeing what they want to see or focusing on the aesthetics of the graphics rather than the data.

Infoviz Tip: Kill Your Darlings

One of the best pieces of advice when it comes to working in information visualization comes from the practice of writing: "Kill your darlings". Just as writers may become enamored of certain scenes or characters, we become enamored of a particularly elegant or sophisticated-looking graphics. Our love of a cool chart or animation can distract us from the goal of communicating the structure and patterns in the data. So, remember to save your harshest criticism for your most beloved pieces, because you may find, much to your chagrin, that they're not as useful and informative as you think they are.

One thing to keep in mind while reading up on data visualization is that the literature is often focused on static charts. With D3 you'll be making interactive and dynamic visualizations and not just static ones. You'll make a dynamic (or animated) data visualization before you finish this chapter, and using D3 to make a chart interactive is incredibly simple. Just a few interactive touches can not only make a visualization more readable but significantly more

engaging, and a user who feels like they're exploring rather than reading, even if only with a few mouseover events or a simple click to zoom, will find the content of the visualization more compelling than that in a static page. But this added complexity requires an investment in learning principles of interface design and user experience. We'll get into this in more detail in Chapter 9 and Chapter 12.

1.5 Your First D3 App

Throughout this chapter, you've seen various lines of code and hopefully the effect of those lines of code on the growing d3ia.html sample page we've been building. But I've avoided explaining the code in too much detail so that we could concentrate on the principles at work in D3. By now, you've probably realized that it's simple enough to build an application from scratch that uses D3 to create and modify elements. Let's put it all together and see how it works. First, let's start with a clean HTML page that doesn't have any styles defined or existing divs.

```
<!doctype html>
<html>
<head>
<script>d3.v3.min.js</script>
</head>
<body>
</body>
<html>
```

1.5.1 Hello World With Divs

With that in place, you can use D3 as an abstraction layer for adding traditional content to the page. While you can write JavaScript inside the your .html file or in its own .js file. For now, let's just put some code in the console and see how it works. Later, we'll focus on the various commands and get into them in more detail as they're applied to layouts and interfaces. We can get started with a very simple piece of code that uses D3 to write to your web page:

```
d3.select("body").append("div")
.style("border", "1px black solid")
.html("hello world");
```

We can adjust the element on the page and give it some interactivity with the inclusion of the .on() function:

```
d3.select("div")
.style("background-color", "pink")
.style("font-size", "24px")
.attr("id", "newDiv")
.attr("class", "d3div")
.on("click", function() {console.log("You clicked a div")})
```

The `.on()` method allows you to create an event listener for the currently selected element or set of elements. It accepts the variety of events that can happen to an element, such as

"click", "mouseover", "mouseout" and so on. If you click on your div, you'll notice that it gives a response in your console as shown in figure 1.23.

Figure 1.23 Using console.log() you can easily test to see if an event is properly firing. Here we create a and assign an onclick event handler using the .on() syntax. When we click on that element and fire the event, the action is noted in the console.

1.5.2 Hello World with Circles

You probably didn't pick up this book just to learn how to add divs to a webpage, though, and likely want to deal graphics like lines and circles. To append shapes to a page with D3, remember that you need to have an SVG canvas element somewhere in your page's DOM. You could either add this SVG canvas to the page as you were writing the HTML. You could append it using the D3 syntax we've learned:

```
d3.select("body").append("svg");
```

But for now let's adjust our d3ia.html page to start with an SVG canvas:

```
<!doctype html>
<html>
<head>
<script>d3.v3.min.js</script>
</head>
<body>
<div id="vizcontainer">
<svg style="width:500px;height:500px;border:1px lightgray solid;" />
</div>
</body>
<html>
```

Once you have an SVG canvas on your page, you can append various shapes to it using the same select() and append() syntax we've been using in 1.5.1 for elements:

```
d3.select("svg").append("line").attr("x1", 20).attr("y1",
20).attr("x2",400).attr("y2",400).style("stroke", "black").style("stroke-
width","2px");

d3.select("svg").append("text").attr("x",20).attr("y",20).text("HELLO");

d3.select("svg").append("circle").attr("r",
20).attr("cx",20).attr("cy",20).style("fill","red");

d3.select("svg").append("circle").attr("r",
100).attr("cx",400).attr("cy",400).style("fill","lightblue");

d3.select("svg").append("text").attr("x",400).attr("y",400).text("WORLD");
```

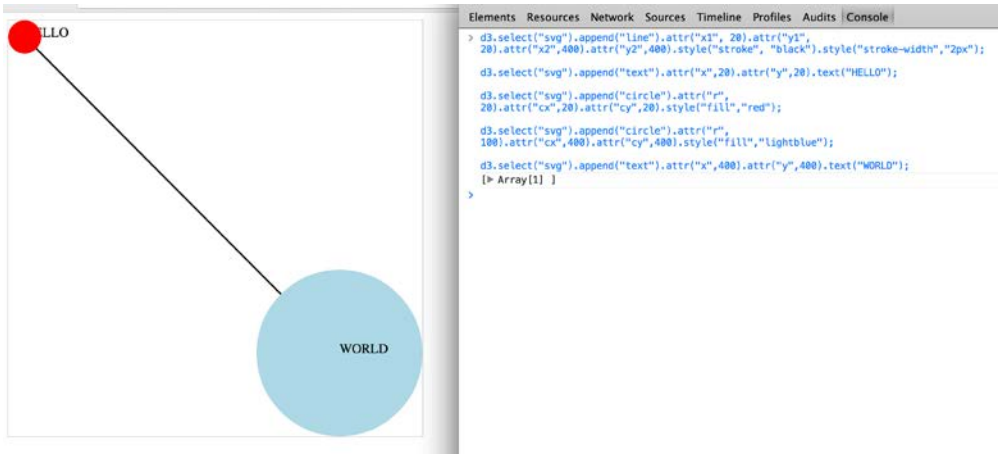


Figure 1.24 The results of running the above code in the console is the creation of two circles, a line, and two text elements. The order in which these elements are drawn results in the first label covered by the circle drawn later.

Notice that your circles are drawn over the line and the text is drawn above or below the circle depending on the order that you run your commands as you can see in figure 1.24. Recall that this is because the draw order of SVG is based on its DOM order. There are some methods we'll learn later on to adjust that order.

1.5.3 A Conversation with D3

There are so many examples of writing Hello World with languages that I thought we should give the world a chance to respond. Let's add the same big circle and little circle from before but this time, when we add text, notice the inclusion of the .style("opacity") setting that makes our text invisible. We've also given each of them a .attr("id") setting so that the text

near the small circle has an "id" attribute with the value of "a" and the text near the large circle has an "id" attribute with the value of "b".

```
d3.select("svg").append("circle").attr("r",
20).attr("cx",20).attr("cy",20).style("fill","red");

d3.select("svg").append("text").attr("id", "a").attr("x",20).style("opacity",
0).attr("y",20).text("HELLO WORLD");

d3.select("svg").append("circle").attr("r",
100).attr("cx",400).attr("cy",400).style("fill","lightblue");

d3.select("svg").append("text").attr("id",
"b").attr("x",400).attr("y",400).style("opacity", 0).text("Uh, hi.");
```

Two circles, no line, and no text. So now we make the text appear using the `.transition()` method with the `.delay()` method and we should have an end state like the one seen in figure 1.25.

```
d3.select("#a").transition().delay(1000).style("opacity", 1);
d3.select("#b").transition().delay(3000).style("opacity", .75);
```

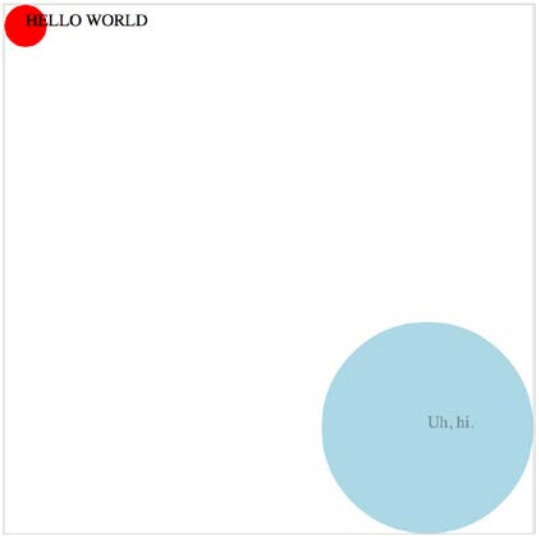


Figure 1.25 Transition behavior when associated with a delay will result in a pause before the application of the attribute or style.

Congratulations, you've made your first dynamic data visualization. The `.transition()` method indicates that you don't want your change to be instantaneous, and by chaining it with the `.delay()` method we indicate how many milliseconds we want to wait before implementing the style or attribute changes that appear in the chain after that .delay() setting.

We'll get a bit more ambitious later on but before we finish here, let's look at the another `.transition()` setting. Along with being able to set a `.delay()` before which the new style or attribute is applied, you can set a .duration() over which the change is applied. The results in your browser should move the shapes in the direction of the arrows in figure 1.26.

```
d3.selectAll("circle").transition().duration(2000).attr("cy", 200);
```

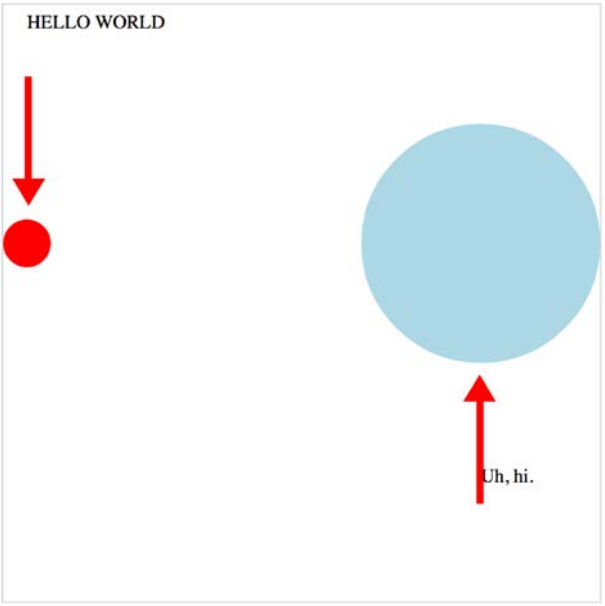


Figure 1.26 Transition behavior when associated with position will make the shape graphically move to its new position over the course of the assigned duration. Since we used the same "y" position for both circles, this results in the first circle moving down and the second circle moving up the the "y" position we've set, which is between the two circles.

The `.duration()` method, as you can see, adjusts the setting over the course of the amount of time (again, in milliseconds) that you set it for.

That covers the basics of how D3 works and how it's designed, and these fundamental concepts will come back again and again throughout the following chapters, where we'll learn about more complicated variations on representing and manipulating data.

1.6 Summary

In this chapter you've received a brief overview as to what D3 is and how it is particularly well-suited for developers building web applications for the modern browser. To do so, I've highlighted the the standardizations and advances that allow this all to happen. Data

standards, practices for dealing with data, and the improved performance of JavaScript and HTML5 all factor into making this possible.

D3.js is just another JavaScript library, one of thousands, but it is also indicative of a change in our expectations of what a web page can do. While you may initially explore D3 as a method for building one-off data visualizations of a highly-processed static dataset, it has built in so much more power and functionality than that leverages the functionality of modern web standards to allow you to create interactive data-driven documents. These documents will improve your ability to communicate with every audience with reusable and robust methods.

D3 itself has many strengths, some of which are simply its being positioned as a library for modern web development, but D3 application development brings with it more than that: As you begin to understand the processes and functions that D3 uses to manipulate, compute, and represent data, even for one-off sites, you'll find that these methods will translate immediately to your work in all aspects of web development, whether dealing with traditional layout and design issues or in more complex interactive applications.

2

Information Visualization Data Flow

2.1 Working With Data

Toy examples and on-line demos will sometimes present data in the format of a JavaScript-defined array, the same way we did in Chapter 1. But in the real world, your data is going to come from an API or a database and you're going to need to load it, format it, and transform it before you start creating web elements based on that data. Along with this process of getting data into a suitable form, this chapter is going to touch on the basic structures that you'll use over and over in D3: loading data from an external source, formatting that data, and creating graphical representations of that data.

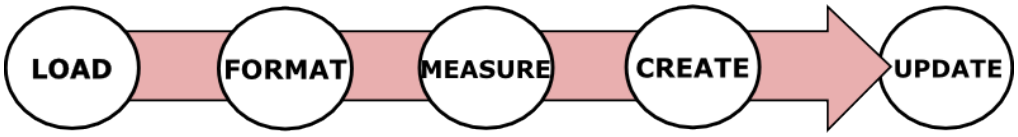


Figure 2.1 The data visualization process that we'll explore in this chapter assumes you begin with a set of data and want to create (and probably update) an interactive or dynamic data visualization.

We'll deal with two small datasets in this chapter. One consists of a few cities and their geographic location and population. The other is just a few madeup tweets with some information about who made them and who reacted to them. This is the kind of data you're often presented with, and tasked with finding out which tweets have more of an impact than others, or which cities are more susceptible to natural disasters than others. In this chapter, we'll learn how D3 lets you measure data in a number of ways, and use that to create some simple charts.

Out in the real world, you're going to deal with much larger datasets with hundreds of cities and thousands of tweets, but you'll use all the same principles outlined in this chapter. This chapter doesn't teach you how to create very complex data visualizations, but it does explain in

detail some of the most important core processes in D3 that you'll need to understand in order to do so.

2.1.1 Loading Data

As we touched on in Chapter 1, your data is typically going to be formatted in different but standardized ways. Regardless of the actual source of the data, it will likely be formatted into single document data files in XML, CSV, or JSON format. D3 provides several functions for importing and dealing with this data. There is one core difference between these formats in how they model data because JSON and XML provide the capacity to encode nested relationships in a way that delimited formats like CSV do not. Another difference is that d3.csv() and d3.json() will produce an array of JSON objects, whereas d3.xml() will create an XML document that needs to be accessed in a different manner.

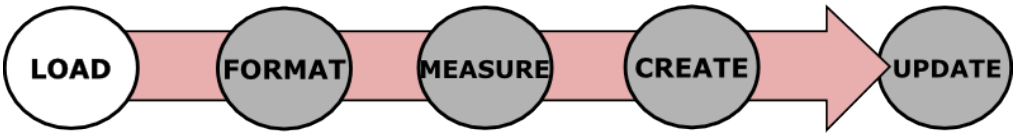


Figure 2.2 The first step to creating a data visualization is getting the data.

FILE FORMATS

There are five D3 functions for loading data that correspond to the five types of files you'll likely encounter: `d3.text()`, `d3.xml()`, `d3.json()`, `d3.csv()`, and `d3.html()`. We'll spend most of our time working with d3.csv() and d3.json(). We'll see d3.html() a bit in the next chapter, where it will be used to create complex DOM elements that are written as prototypes. As for the other `d3.xml()` and `d3.text()`, you might find them more useful depending on how you typically deal with data. You might be more comfortable with XML rather than JSON, in which case you can rely on d3.xml() and format your data functions accordingly. If you prefer working with text strings, then you can use d3.text() to pull in the data and process it using another library or code.

Both d3.csv() and d3.json() use the same format when calling the function, declaring the path to the file being loaded and defining the callback function like so:

```
d3.csv("cities.csv",function(error,data) {console.log(error,data)});
```

The error variable is optional, and if you only declare a single variable with the callback function, it will be the data:

```
d3.csv("cities.csv",function(d) {console.log(d)});
```

It's within the callback function where we first get access to the data, and may want to declare the data as a global variable so that you can use it elsewhere. To get started, we need a data file, so for this chapter we'll be working with two data files, a CSV that contains data about cities, and a JSON file that contains data about tweets:

Listing 2.1: cities.csv file contents

```
"label","population","country","x","y"
"San Francisco", 750000,"USA",122,-37
"Fresno", 500000,"USA",119,-36
"Lahore",12500000,"Pakistan",74,31
"Karachi",13000000,"Pakistan",67,24
"Rome",2500000,"Italy",12,41
"Naples",1000000,"Italy",14,40
"Rio",12300000,"Brazil",-43,-22
"Sao Paolo",12300000,"Brazil",-46,-23
```

Listing 2.2: tweets.json file contents

```
{
"tweets": [
{"user": "Al", "content": "I really love seafood.", "timestamp": " Mon Dec 23
2013 21:30 GMT-0800 (PST)", "retweets": ["Raj","Pris","Roy"], "favorites":
["Sam"]},
{"user": "Al", "content": "I take that back, this doesn't taste so good.",
"timestamp": "Mon Dec 23 2013 21:55 GMT-0800 (PST)", "retweets": ["Roy"],
"favorites": []},
{"user": "Al", "content": "From now on, I'm only eating cheese sandwiches.",
"timestamp": "Mon Dec 23 2013 22:22 GMT-0800 (PST)", "retweets":
[],"favorites": ["Roy","Sam"]},
{"user": "Roy", "content": "Great workout!", "timestamp": " Mon Dec 23 2013
7:20 GMT-0800 (PST)", "retweets": [],"favorites": []},
{"user": "Roy", "content": "Spectacular oatmeal!", "timestamp": " Mon Dec 23
2013 7:23 GMT-0800 (PST)", "retweets: [],"favorites": []},
{"user": "Roy", "content": "Amazing traffic!", "timestamp": " Mon Dec 23 2013
7:47  GMT-0800 (PST)", "retweets": [],"favorites": []},
{"user": "Roy", "content": "Just got a ticket for texting and driving!",
"timestamp": " Mon Dec 23 2013 8:05 GMT-0800 (PST)", "retweets":
[],"favorites": ["Sam", "Sally", "Pris"]},
{"user": "Pris", "content": "Going to have some boiled eggs.", "timestamp": "
Mon Dec 23 2013 18:23 GMT-0800 (PST)", "retweets": [],"favorites":
["Sally"]},
{"user": "Pris", "content": "Maybe practice some gymnastics.", "timestamp": "
Mon Dec 23 2013 19:47  GMT-0800 (PST)", "retweets": [],"favorites":
["Sally"]},
{"user": "Sam", "content": "@Roy Let's get lunch", "timestamp": " Mon Dec 23
2013 11:05 GMT-0800 (PST)", "retweets": ["Pris"], "favorites": ["Sally",
"Pris"]}
]
}
```

With these two files available, we can access the data by using the appropriate function to load them:

```
d3.csv("cities.csv",function(data) {console.log(data)});
d3.json("tweets.json",function(data) {console.log(data)}); #a
```

#a Prints Object {tweets: Array[10]} in the console

In both cases, the results of loading the data file is an array of JSON objects, though in the case of tweets.json, this array is found at `data.tweets`, while in the case of cities.csv, this

array is data. Basically, d3.json() allows you to load a JSON-formatted file, which can have objects and attributes in a way that a loaded CSV cannot. When you load a CSV, it returns an array of objects, which in this case is initialized as "data". When you load a JSON file, it could return an object with several name/value pairs. In this case, the object that is initialized as "data" has a name/value pair of "tweets": [Array of Data]. That's why you need to refer to data.tweets after you've loaded tweets.json, but refer to data when you load cities.csv. I set up the structure of tweets.json to highlight this distinction.

Both d3.csv and d3.json are asynchronous, and will return after the request to open the file and not after finishing processing the file. This means that loading a file, which is typically an operation that will take more time than most other functions, will not be complete by the time other functions are called. If you call functions that require the loaded data before it's loaded, then they'll fail. There are two ways to get around this asynchronous behavior. The first is to nest the functions operating on the data within the data loading function:

```
d3.csv("somefiles.csv", function(data) {doSomethingWithData(data});
```

Or, you can use a helper library like queue.js to trigger events upon completion of the loading of one or more files. We'll see queue.js in action in later chapters. Note that d3.csv() has a method .parse() that can be used on a block of text rather than an external file. If you need more direct control over getting data, you should review the documentation for d3.xhr(), which allows for more fine-grained control of sending and receiving data.

2.1.2 Formatting Data

Once you have a dataset loaded, you'll need to define some methods so that the attributes of the data directly relate to settings for color, size, and position graphical elements. If you want to display the cities in the CSV, you probably want to use circles and size those circles based on population, and place them according to their geographic coordinates. We have long-established conventions for representing cities on maps graphically, but the same cannot be said about tweets. What graphical symbol would we use to represent a single tweet, how we size it, and where we place it are all open questions. To answer these questions, you need to understand the forms of data you'll encounter when doing data visualization. While there are numerous data types defined in the sense of programming languages and ontologies, broadly speaking it's good to think of them as either quantitative, categorical, geometric, temporal, topological or raw.

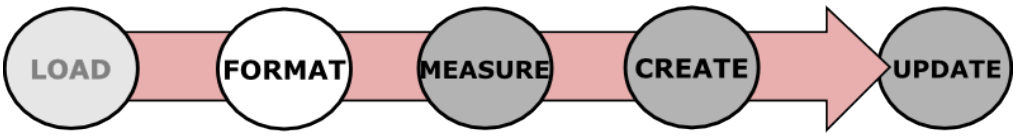


Figure 2.3 After loading data, you need to make sure that it's formatted in such a way that it can be used by various JavaScript functions to create graphics.

QUANTITATIVE

Numerical or quantitative data is the most commonly seen and assumed type when it comes to data visualization. Quantitative data can be effectively represented with size, position, or color. You'll typically need to normalize quantitative data, such as defining scales using the d3.scale() as explained in 2.1.3 Scales or by transforming your quantitative data into categorical data using techniques like quantiles, which group numeric values.

For one of our datasets, we have readily accessible quantitative data: the population figures in the `cities.csv` table. For the tweets dataset, though, it seems like we don't have any quantitative data available, which is why we'll spend some time in Section 2.1.3 looking at how to transform data.

CATEGORICAL

Categorical data exists as discrete groups into which the data falls, typically represented by text, such as nationality or gender. Categorical data is often represented using shape or color, mapping the categories to distinct colors or shapes so as to try to identify the pattern of the groups of elements positioned according to other attributes.

The tweets data has categorical data in the form of the user data, which we can recognize by intuitively thinking of coloring the tweets by the user that made them. There are also methods to derive categorical data that we'll touch on below.

TOPOLOGICAL

Topological data describes the relationship of one piece of data with another, which can also be another form of location data. The genealogical connection between two people or the distance of a shop from a train station each represent a way of defining relationships between objects. Topological attributes can be represented with text referring to unique ID values or with pointers to the other objects. You will find yourself creating topological data, especially in the form of nested hierarchies, which we'll see below.

For the cities data, it seems like we don't have topological data, though you could easily produce it by designating one city, such as San Francisco, to be your frame-of-reference and then create a Distance to San Francisco measure that would give you simple topological data if you needed it. The tweets data has its topological component in the favorites and retweets arrays, which provide the basis for a simple social network.

GEOMETRIC

Geometric data is most commonly associated with the boundaries and tracks of geographic data, such as countries, rivers, cities and roads. Geometric data might also be the SVG code to draw a particular icon that you want to use, the text for a class of shape, or a numerical value indicating the size of the shape. Geometric data is, not surprisingly, most often represented using shape and size, but can also be transformed like other data, such as into quantitative data by measuring area and perimeter.

The cities data has obvious geometric data in the form of traditional latitude and longitude coordinates that allow the points to be placed on a map. The tweets data, on the other hand, has no readily accessible geometric data.

TEMPORAL

Dates and time can be represented using numbers for days, years, or months, or with specific date time encoding for more complex calculations. The most common format is ISO-8601, and if your data comes formatted that way as a string, it's pretty easy to turn it into a date datatype in JavaScript as you'll see in 2.1.4. You'll find yourself dealing with dates and times more than you'd ever want to. Fortunately, both the built-in functions in JavaScript as well as a few helper functions in D3 are available to handle what can be tricky data to measure and represent.

While there is no temporal data for the cities dataset, keep in mind that temporal data for common entities like cities and countries is often available. In situations where you can easily expand your dataset like this, you need to ask yourself if it makes sense given the scope of your project. In contrast, the tweets data has a string that conforms to RFC2822 (supported by JavaScript for representing dates along with ISO-8601) and will easily be turned into a date datatype in JavaScript.

RAW

Raw or free or unstructured data is typically thought of as text and image content. Raw data can be transformed by measuring it or leveraging sophisticated text and image analysis to derive attributes more suited to data visualization. In its unaltered form, raw data is used in the content fields of graphical elements, such as in labels or snippets.

The city names will provide convenient labels for that dataset, but how would you label the individual tweets? One way is to use the entire content of the tweet as a label, as we do in Chapter 5, but when dealing with raw data, the most difficult and important task is coming up with ways of summarizing and measuring it effectively.

2.1.3 Transforming Data

As you deal with different forms of data, you'll find it necessary to change data from one type to another to better represent it. There are many ways to transform data. Here we'll look at casting, normalizing (or scaling), binning (or grouping), and nesting data.

CASTING: CHANGING DATA TYPES

The act of casting data refers to turning one data type into another from the perspective of your programming language, which in this case is JavaScript. When you load data, it will often be in a string format, even if it's actually a date, integer, floating point number, or array. The date string in the tweets data, for instance, needs to be changed from a string into an actual date datatype if we want to work with the date methods available in JavaScript. With that in mind, you should familiarize yourself with the basic JavaScript functions that allow you to transform data. Here are a few:

```
parseInt(“77”); #A
parseFloat("3.14"); #B
Date.parse("Sun, 22 Dec 2013 08:00:00 GMT"); #C
text = "alpha,beta,gamma"; text.split(","); #D
```

#a casts the string 77 into the number 77 with no decimal places
#b casts the string 3.14 into the number 3.14 with decimal places

#c casts an ISO-8601 or RFC2822 compliant string to a date datatype
#d splits the comma-delimited string into an array, which isn't strictly speaking a casting operation but changes the type of data

NOTE JavaScript defaults to type conversion when using the `==` test while using `===` and the like will force no type conversion, so you'll find your code will often work fine without casting. However, this will come back to haunt in situations where it does not default to the type you expect, such as when you are trying to sort an array and JavaScript is sorting your numbers alphabetically.

SCALES AND SCALING

Numerical data rarely corresponds directly to the position and size of graphical elements on screen. You can use `d3.scale()` functions to normalize your data for presentation on a screen (among other things). The first scale we'll look at is `d3.scale().linear()`, which makes a direct relationship between one range of numbers and another. Scales have a domain and a range setting that accept arrays with the domain determining the ramp of values being transformed and the range referring to the ramp to which those values are being transformed. For example, if we take the smallest population figure in cities.csv and the largest population figure, we can create a ramp that scales from the smallest to the largest so that we can display the difference between them easily on a 500px canvas. In figure 2.4 and the code that follows, you can see that the same linear rate of change from 500,000 to 13,000,000 is going to be mapped to a linear rate of change from 0 to 500.

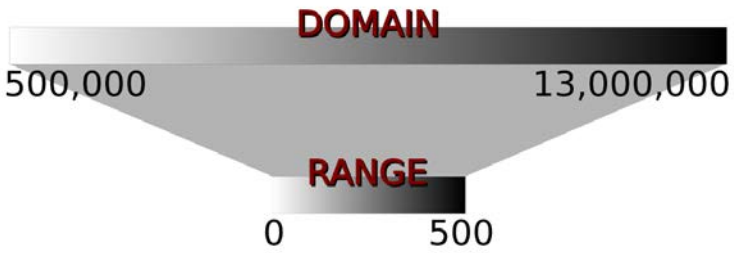


Figure 2.4 Scales in D3 map one set of values (the domain) to another set of values (the range) in a relationship determined by the type of scale you create.

We create this ramp by instantiating a new scale object and setting its domain and range values like so:

```
var newRamp = d3.scale.linear().domain([814,37120]).range([0, 500]);
newRamp(1000000); #A
newRamp(9000000); #B
newRamp.invert(313); #C
```

#a returns 20, allowing you to place a country with 10,000,000 population at 20px
#b returns 340
#c the invert function reverses the transformation, in this case returning 8325000

This can also be used to create a color ramp by referencing CSS color names, RGB colors, or hex colors in the range field. The effect in that case is a linear mapping of a band of colors to the band of values defined in the domain, as shown in figure 2.5.

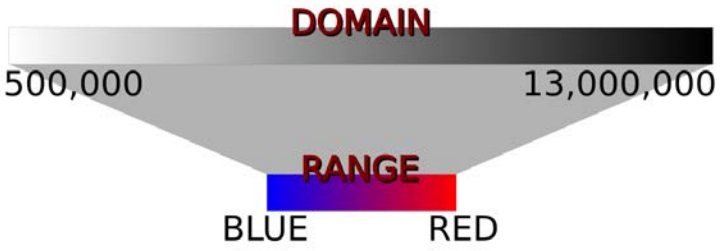


Figure 2.5 Scales can also be used to map numerical values to color bands, to make it easier to denote values using a color scale.

The code to create a ramp like this is fundamentally the same, except for the reference to colors in the range array:

```
var newRamp = d3.scale.linear().domain([500000,13000000]).range(["blue",
"red"]);
newRamp(1000000); #A
newRamp(9000000); #B
newRamp.invert("#ad0052"); #C
```

#a returns "#0a005f", allowing you to draw a city with 1,000,000 population as dark purple
#b returns "#ad0052"
#c the invert function only works with a numeric range, so inverting in this case returns NaN

You can also use `d3.scale.logarithmic()`, `d3.scale.pow()`, `d3.scale.ordinal()` and other less common scales to map data where these scales are more appropriate to your data set. We'll see these in action later on in the book as we deal with just those kinds of datasets. Finally, if that's not enough scales, there's `d3.time.scale()` that provides a linear scale that's designed to deal with date datatypes, which we'll see later in this chapter.

BINNING: CATEGORIZING DATA

It's useful to sort quantitative data into categories, placing the values into a range or "bin" to allow you to group them together. One method to do so is to use quantiles, which simply splits the array into equal-sized parts. The quantile scale in D3 is, not surprisingly, called `d3.scale.quantile()` and it has the same settings as other scales. The number of parts and their labels are determined by the `.range()` setting. Unlike other scales, there is no error if there is a mismatch between the number of `.domain()` values and the number of `.range()` values in a quantile scale because it is automatically sorting and binning the values in the domain into, typically, a smaller number of values in the range.

The scale will sort the array of numbers in its `.domain()` from smallest to largest and automatically split the values at the appropriate point to create the necessary categories. Any

number passed into the quantile scale function will return one of the set categories based on these break points.

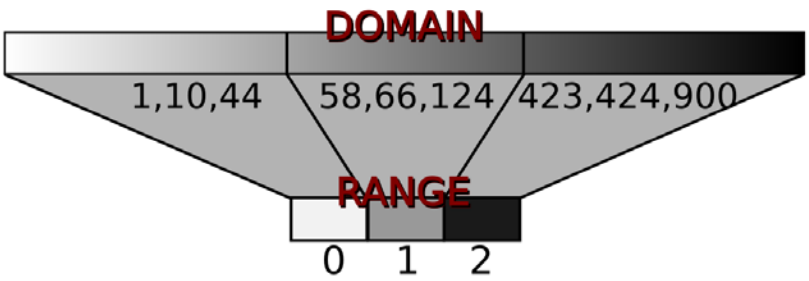


Figure 2.6 Quantile scales take a range of values and reassign them into a set of equally sized bins.

```
var sampleArray = [423,124,66,424,58,10,900,44,1];
var qScale = d3.scale.quantile().domain(sampleArray).range([0,1,2]);
qScale(423); #A
qScale(20); #B
qScale(10000); #C
```

#a returns 2
#b returns 0
#c returns 2

Notice that the range values are fixed, and can accept text that may correspond to a particular CSS class or color or other arbitrary value.

```
var qScaleName =
d3.scale.quantile().domain(sampleArray).range(["small","medium","large"]);
qScaleName (68); #A
qScaleName (20); #B
qScaleName (10000); #C
```

#a returns “medium”
#b returns “small”
#c returns “large”

NESTING

Hierarchical representations of data is incredibly useful, and not limited to data wherein hierarchical relationships are of the more traditional or explicit kind, such as a dataset of parents and their children. We’ll get into hierarchical data and representation in more detail in chapter 4 & 5, but in this chapter we’ll use the D3 nesting function, which you can probably guess is called `d3.nest()`.

The basic concept behind nesting is that shared attributes of data can be used to sort them into discrete categories and subcategories. For instance, if you want to group tweets by the user who made them, then you would use nesting.

```
d3.json("tweets.json",function(data) {
tweetData = data.tweets;
```

```
var nestedTweets = d3.nest()
.key(function(el) {return el.user})
.entries(tweetData);
})
```

The result of this nesting function is to combine the tweets into arrays under new objects labeled by the unique “user” attribute values.

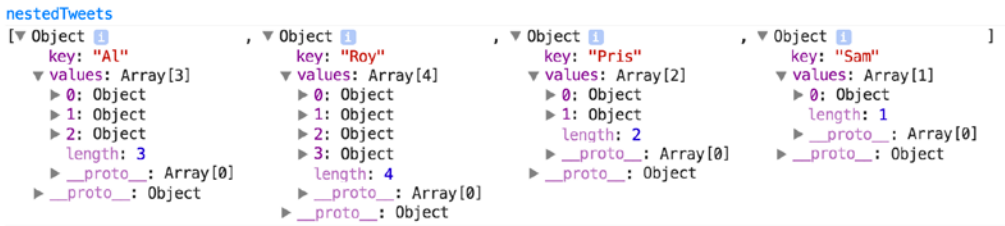


Figure 2.7 Objects nested into a new array.

Now that we have our data loaded and transformed into types that are accessible, we can focus on better understanding the patterns of that data by measuring it.

2.1.4 Measuring Data

After loading your data array, one of the first things you should do is measure and sort it. It’s particularly important to understand the distribution of values of particular attributes, as well as the minimum and maximum values and the names of the attributes. D3 provides a set of array functions that can facilitate understanding your data better.

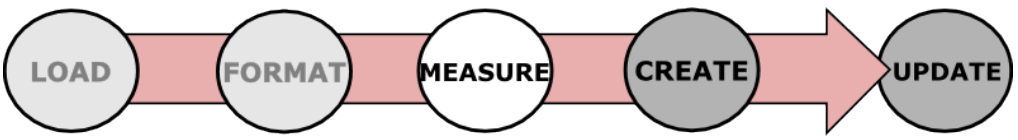


Figure 2.8 After formatting your data, you’ll need to measure it to ensure that the graphics you create are appropriately sized and positioned based on the parameters of the dataset.

Since you’re always dealing with arrays filled with data that you’ll want to size and position based on the relative value of an attribute compared to the distribution of the values in the array, you should familiarize yourself with the ways that D3 allows you to determine the distributions of values in an array. We’ll work with an array of numbers first before we see these functions in operation with more complex and more data-rich JSON object arrays.

```
var testArray =  [88,10000,1,75,12,35];
```

Nearly all the D3 measuring functions follow the same pattern, which requires you to designate the array and an accessor function for the value that you want to measure. In our

case, since we're working an array of numbers and not an array of objects, the accessor only needs to point at the element itself.

```
d3.min(testArray, function (el) {return el});     #A
d3.max(testArray, function (el) {return el});     #B
d3.mean(testArray, function (el) {return el});    #C
```

#a returns the minimum value in the array, which is 1
#b returns the maximum value in the array, which is 10000
#c returns the average of values in the array, which is 1701.8333333333335

If you're dealing with a more complex JSON object array, then you'll need to designate the attribute you want to measure. For instance, if we're working with the array of JSON objects from cities.csv, we might want to derive the minimum, maximum and average population:

```
d3.csv("cities.csv", function(data) {
d3.min(data, function (el) {return el.population}); #A
d3.max(data, function (el) {return el.population }); #B
d3.mean(data, function (el) {return el.population });#C
});
```

#a returns the minimum value of the "population" attribute of each object in the array, which is 500000
#b returns the maximum value of the "population" attribute of each object in the array, which is 1300000
#c returns the average value of the "population" attribute of each object in the array, which is 61.6

Finally, since dealing with minimum and maximum values is a common occurrence, `d3.extent()` conveniently returns `d3.min()` and `d3.max()` in a two-piece array.

```
d3.extent(testObjectArray, function (el) {return el.population}); #A
```

#a returns [1,199]

You can also measure non-numerical data like text by using the JavaScript .length() function for strings and arrays. When dealing with topological data, you need more robust mechanisms to measure network structure like centrality and clustering. When dealing with geometric data, you can calculate the area and perimeter of shapes mathematically, which can become rather difficult with complex shapes.

Now that we've managed to load our data, format it and measure it, we can finally get to the process of creating data visualizations. This requires us to step back into the use of selections and the functions that come with them, which we'll examine in much more detail in the next section.

2.2 Data Binding

We touched on data binding a bit in Chapter 1, but here we'll go into it in much more detail, explaining how selections work with data binding to create and change elements. Our first example will use the data from cities.csv, and after that we'll see the process using this data as well as simple numerical arrays and later we'll do more interesting things with the tweets data.

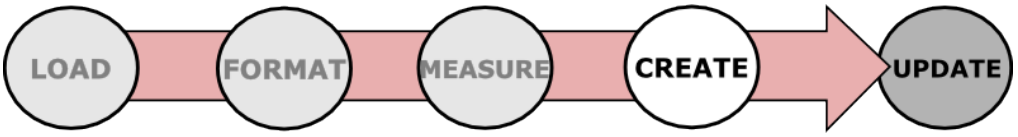


Figure 2.9 To create graphics in D3, you use selections that bind data to DOM elements.

2.2.1 Selections and Binding

Any changes to the structure and appearance of your web page with D3 are going to be made using selections. Remember that a selection consists of one or more elements in the DOM as well as the data, if any, associated with them. The creation or deletion of elements is made using selections, as are changes to style and content. We've seen how you can use d3.select() to change a DOM element, now we'll focus on creating and removing elements based on data. For this example, we'll be using cities.csv as our data source and so we'll need to load cities.csv and trigger our data visualization function in the callback to create a set of new elements on the page:

```
d3.csv("cities.csv",function(error,data) {dataViz(data)});
function dataViz(incomingData) {
d3.select("body").selectAll("div.cities") #A
.data(incomingData) #B
.enter() #C
.append("div") #D
.attr("class","cities") #E
.html(function(d,i) {return d.label}) #F
     }
```

#a this is an empty selection because there are no elements in <body> with class of "cities"
#b here we bind the data to our selection
#c the .enter() function defines how to respond when there is more data than DOM elements in a selection
#d the .append() function is used to create an element in the current selection
#e set the class of each newly created element
**#f set the content of the created **

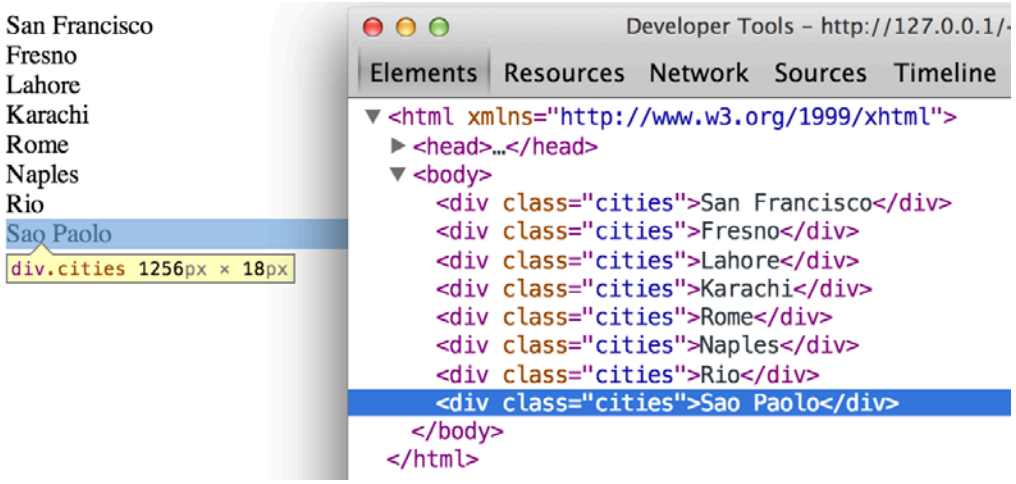


Figure 2.10 The result of our selection binding the cities.csv data to our web page is the creation of eight new divs, each of which is classed with “cities” and with content drawn from our data.

The selection and binding procedure shown here will be a common pattern throughout the rest of this book. By default, newly created elements will be added to the body, which means that you’ll need to use a sub-selection when working with D3. A sub-selection simply means that you first select one element and then select the elements underneath it, which we’ll see in more detail later. First, let’s take a look at each individual part of this basic example:

D3.SELECTALL()

The first part of any selection is the use of d3.select() or d3.selectAll() with some kind of CSS identifier that corresponds to a part of the DOM. It will often be the case that there are no elements that match the identifier, which is referred to as an empty selection, because you want to create new elements on the page using the .enter() function. You can make a selection on a selection to designate how to create and modify child elements of a specific DOM element. Note that a sub-selection will not automatically generate a parent, which needs to be created using `.append()` or already in existence.

.DATA()

Here we associate an array with the DOM elements we’ve selected. Each city in our dataset will be associated with a DOM element in the selection, and that associated data will be stored in a “__data__” attribute of the element. You could access these values manually using JavaScript like so:

```
document.getElementsByClassName("cities")[0].__data__ #A
```

#a returns a pointer to the object representing San Francisco

But we’ll be working with those values in a more sophisticated way using D3, which we’ll get into below.

.ENTER()

When binding data to selections there will either be more, less, or the same number of DOM elements as there are data values. In the case where we have more data values than DOM elements in the selection, then we trigger the `.enter()` function, which allows us to define some behavior to perform for every value for which there is not a corresponding DOM element in the selection. In our case, `.enter()` will fire four times, since there are no DOM elements that correspond to "div.cities" and we have four values in our `incomingData` array. When there are fewer data elements, then .exit() behavior is triggered, and when there are equal data values and DOM elements in a selection, then neither .exit() nor .enter() is fired.

.APPEND()

You will almost always want to add elements to the DOM when there are more data values than DOM elements. The `.append()` function allows us to add more elements and define which elements to add. In our example, we're adding "div" elements, but later in this chapter we'll use this to add SVG shapes, and in later chapters we'll be adding tables and buttons and any other element type supported in HTML. The `.insert()` function is a sister function to `.append()`, but `.insert()` gives you some control over where in the DOM you'll be adding the new element. You can also perform an append or insert directly on a selection, which will add one DOM element of the kind you specify for each DOM element in your selection.

.ATTR()

We're already familiar with changing styles and attributes using D3 syntax. The only thing to note here is that each of the functions you define here will be applied to each new element added to the page. In our example, each of our four new elements will be created with class="cities". Remember that even though our selection referenced "div.cities", you still have to manually declare that you are creating elements and also manually set their class to "cities".

.HTML()

For traditional DOM elements, you set the content with a .html() function. Below, we'll see how to set content based on the data bound to the particular DOM element.

2.2.2 Accessing Data with Inline Functions

If you've run the code above, you'll see that each element was set with different content derived from the data array that you bound to the selection. This is done using an inline anonymous function in your selection that automatically provides access to two variables that are critical to representing data graphically: the data value itself and the array position of the data. In most examples you'll see these represented as "**d**" for data and "**i**" for array index but of course they could be declared using any available variable name.

The best way to see this in action is to use your data to create a simple data visualization. We'll keep working with `d3ia.html`, which we created in Chapter 1, and which is a simple HTML page with minimal DOM elements and styles. A histogram or bar chart is one of the most simple and effective ways of expressing numerical data broken down by category. To get

started, we'll avoid the more complex datasets for now and start with a simple array of numbers:

```
[15, 50, 22, 8, 100, 10]
```

If we bind this array to a selection, we can use the values to determine the height of the rectangles (our bars in a bar chart). We need to set a width, the value of which we would base on the space available for the chart and which we'll start by setting to 10px:

```
d3.select("svg")
.selectAll("rect")
.data([15, 50, 22, 8, 100, 10])
.enter()
.append("rect")
.attr("width", 10) #A
.attr("height", function(d) {return d}) #B
```

#a Set the width of the rectangles to a fixed value
#b Set the height to be equal to the value of the data associated with each element

When we used the label values of our array above to create content with labels in 2.2.1, we pointed to the object's "**label**" attribute. Here, since we're dealing with an array of number literals, we use the inline function to point directly at the value in the array to determine the height of our rectangles. The result, seen in figure 2.x, is not nearly as interesting as you might expect.

Figure 2.11: The default settings for any shape in SVG is to be filled in black with no stroke, which makes it hard to tell when the drawing of the shapes overlap each other.

That's because all the rectangles are overlapping each other--they have the same default x and y positions. It's easier to see if our rectangles have a different outline, known as stroke, than their fill. Making them transparent by adjusting their opacity style will also highlight this situation:

```
d3.select("svg")
.selectAll("rect")
.enter()
.append("rect")
.data([15, 50, 22, 8, 100, 10])
.attr("width", 10)
.attr("height", function(d) {return d})
.style("fill", "blue")
.style("stroke", "red")
.style("stroke-width", "1px")
.style("opacity", .25)
```

Figure 2.12: By changing the fill, stroke, and opacity settings, we can see the overlapping rectangles being drawn.

You might wonder what practical use the second variable in the inline function, typically represented as "**i**", can provide. One use of the array position of a data value is that it can be used to place visual elements. If we set the x position of each rectangle based on the "**i**" value (multiplied by the width of the rectangle) then we get a step closer to an actual bar chart:

```
d3.select("svg")
.selectAll("rect")
.data([15, 50, 22, 8, 100, 10])
.enter()
.append("rect")
.attr("width", 10)
.attr("height", function(d) {return d})
.style("fill", "blue")
.style("stroke", "red")
.style("stroke-width", "1px")
.style("opacity", .25)
.attr("x", function(d,i) {return i * 10})
```

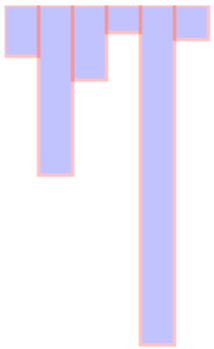


Figure 2.13: SVG rectangles are drawn from top to bottom.

But our histogram seems to be drawn from top to bottom because SVG draws rectangles down and to the right from the 0,0 point that you specify. To adjust this, we need to move each rectangle so that its y position corresponds to a position that is offset based on its height. We know that the tallest rectangle will be 100, and remember that y is measured based on position from the top left of the canvas, so if we set the "**y**" attribute of each rectangle to be equal to its length minus 100, then the histogram draws in the manner we'd expect.

```
d3.select("svg")
.selectAll("rect")
.data([15, 50, 22, 8, 100, 10])
.enter()
.append("rect")
.attr("width", 10)
.attr("height", function(d) {return d})
.style("fill", "blue")
.style("stroke", "red")
.style("stroke-width", "1px")
.style("opacity", .25)
.attr("x", function(d,i) {return i * 10})
.attr("y", function(d) {return 100 - d})
```

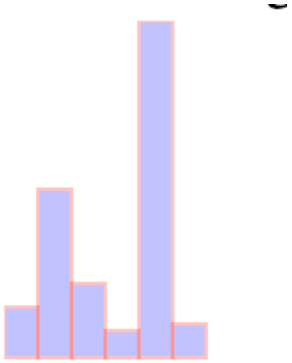


Figure 2.14: By setting the y-position of the rectangle to be the desired y-position minus the height of the rectangle, this provides the effect of drawing it bottom to top from that y-position.

2.2.3 Integrating Scales

This way of building a chart works fine if you're dealing with an array of values that correspond directly to the height of the created rectangles relative to the height and width of your <svg> element. But if you have real data, then it tends to have widely divergent values that do not correspond directly to the size of the shape you want to draw. The above code doesn't deal with an array of values like this:

```
[14, 68, 24500, 430, 19, 1000, 5555]

d3.select("svg")
.selectAll("rect")
.data([14, 68, 24500, 430, 19, 1000, 5555])
.enter()
.append("rect")
.attr("width", 10)
.attr("height", function(d) {return d})
.style("fill", "blue")
.style("stroke", "red")
.style("stroke-width", "1px")
.style("opacity", .25)
.attr("x", function(d,i) {return i * 10})
.attr("y", function(d) {return 100 - d})
```

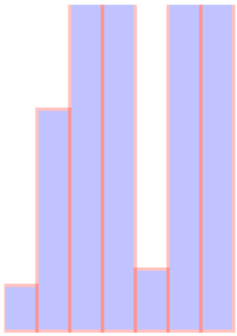


Figure 2.15: SVG shapes will continue to be drawn off-screen.

And it works just as well if you set a y-offset equal to the maximum value:

```
d3.select(“svg”)
.selectAll(“rect”)
.data([14, 68, 24500, 430, 19, 1000, 5555])
.enter()
.append(“rect”)
.attr(“width”, 10)
.attr(“height”, function(d) {return d})
.style(“fill”, “blue”)
.style(“stroke”, “red”)
.style(“stroke-width”, “1px”)
.style(“opacity”, .25)
.attr(“x”, function(d,i) {return i * 10})
.attr("y", function(d) {return 24500 - d})
```

In this case there’s no need to bother with a screenshot, it’s just a single bar running vertically across your canvas. In this case, it’s best to use D3’s scaling functions to normalize the values for display. We’ll use the relatively straightforward `d3.scale.linear()` for this bar chart. Remember, there are two primary functions in a D3 scale, both of which expect arrays and which must have arrays of the same length to get the right results: `.domain()` and `.range()`. The array in `.domain()` indicates the series of values being mapped to `.range()` which will make more sense in practice. First we make a scale for the y-axis:

```
var yScale = d3.scale.linear().domain([0,24500]).range([0,100]);
yScale(0); #A
yScale(100); #B
yScale(24000); #C
```

#a returns 0
#b returns 0.40816326530612246
#c returns 97.95918367346938

As you can see, yScale now allows us to map the values in a way suitable for display. If we then use yScale determining the height and y-position of the rectangles, we end up with a bar chart that's a bit more legible:

```
var yScale = d3.scale.linear().domain([0,24500]).range([0,100]);

d3.select("svg")
.selectAll("rect")
.data([14, 68, 24500, 430, 19, 1000, 5555])
.enter()
.append("rect")
.attr("width", 10)
.attr("height", function(d) {return yScale(d)})
.style("fill", "blue")
.style("stroke", "red")
.style("stroke-width", "1px").style("opacity", .25).attr("x", function(d,i)
{return i * 10}).attr("y", function(d) {return 100 - yScale(d)});
```

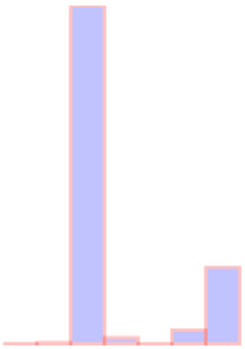


Figure 2.16: A bar chart drawn using a linear scale.

But when we deal with such widely diverging values, it often makes more sense to use a polylinear scale. A polylinear scale is simply a linear scale with multiple points in the domain and range. Let's suppose that for our dataset, we're particularly interested in values between 1 and 100, while recognizing that sometimes we get interesting values between 100 and 1000, and occasionally we get outliers that can be quite large. You could express this in a polylinear scale as follows:

```
var yScale =
d3.scale.linear().domain([0,100,1000,24500]).range([0,50,75,100]);
```

The same draw code from above produces a different chart with this scale:

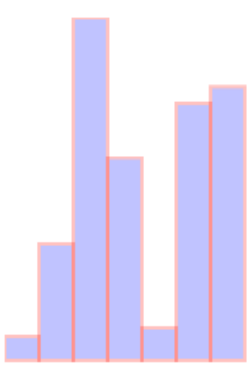


Figure 2.17: The same bar chart from 2.x drawn with a polylinear scale.

It might instead be the case that there is a cutoff value, after which it isn't so important to express how large a datapoint is. For instance, let's say these datapoints represent the number of responses for a survey, and it's deemed a success if there are more than 500 responses. You might only want to show the range of the data values between 0 and 500, while emphasizing the variation at the 0 to 100 level with a scale like this:

```
var yScale = d3.scale.linear().domain([0,100,500]).range([0,50,100]);
```

You might think that's enough to draw a new chart that caps the bars at a maximum height of 100 if the datapoint has a value over 500. This is not the default behavior for scales in D3, though. Here's what would happen running the draw code with that scale:

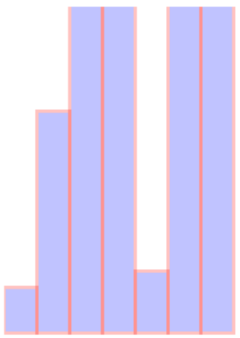


Figure 2.18: A bar chart drawn with a linear scale where the maximum value in the domain is lower than the maximum value in the dataset.

Notice the rectangles are still being drawn above the canvas, as evidenced by the lack of a border on the top of the four rectangles with values over 500. You can confirm this is happening by putting a value greater than 500 into the scale function you've created:

```
yScale(1000);
 #a returns 162.5
```

By default, a D3 scale will continue to extrapolate values greater than the maximum domain value and less than the minimum domain value. If you want it to set all such values to equal the maximum (for greater) or minimum (for lesser) range value, then you'll need to use the `.clamp()` function:

```
var yScale =
d3.scale.linear().domain([0,100,500]).range([0,50,100]).clamp(true);
```

Running the draw code now produces rectangles that have a maximum value of 100 for height and position:

Figure 2.19: A bar chart drawn with values in the dataset greater than the maximum value of the domain of the scale, but with the clamp() function set to true.

You can confirm this by plugging a value into yScale() that's greater than 500:

```
yScale(1000);
 #a returns 100
```

Scale functions are key to determining position, size, and color of elements in data visualization. As you'll see later on in this chapter and throughout the book, this is just the basics of using scales in D3.

2.3 Data Presentation Style, Attributes, and Content

We'll move on by working with the cities and tweets data to create a second bar chart combining the techniques we've learned in this chapter and Chapter 1. After that, we'll deal

with the more complicated methods necessary to represent the tweets data in a simple data visualization. Along the way, we'll learn how to set styles and attributes based on the data bound to the elements, as well as explore how D3 creates, removes, and changes elements based on changes in the data.

2.3.1 Visualization from loaded data

A bar chart based on the cities.csv data is pretty straightforward, requiring only a scale based on the maximum population value, which we can determine using `d3.max()`. This bar chart will end up showing us the distribution of population sizes of the cities in our dataset.

Listing 2.3 loading data from CSV, casting it, measuring it, and displaying it on a bar chart.

```
d3.csv("cities.csv",function(error,data) {dataViz(data)});
function dataViz(incomingData) {

maxPopulation = d3.max(incomingData,
function(el) {return parseInt(el.population)}
); #A
var yScale = d3.scale.linear().domain([0,maxPopulation]).range([0,460]);

d3.select("svg")
.selectAll("rect")
.data(incomingData)
.enter()
.append("rect")
.attr("width", 50)
.attr("height", function(d) {return yScale(parseInt(d.population))})
.attr("x", function(d,i) {return i * 60})
.attr("y", function(d) {return 480 - yScale(parseInt(d.population))})
.style("fill", "blue")
.style("stroke", "red")
.style("stroke-width", "1px")
.style("opacity", .25)
}
```

#a notice the need to transform the population value into an integer

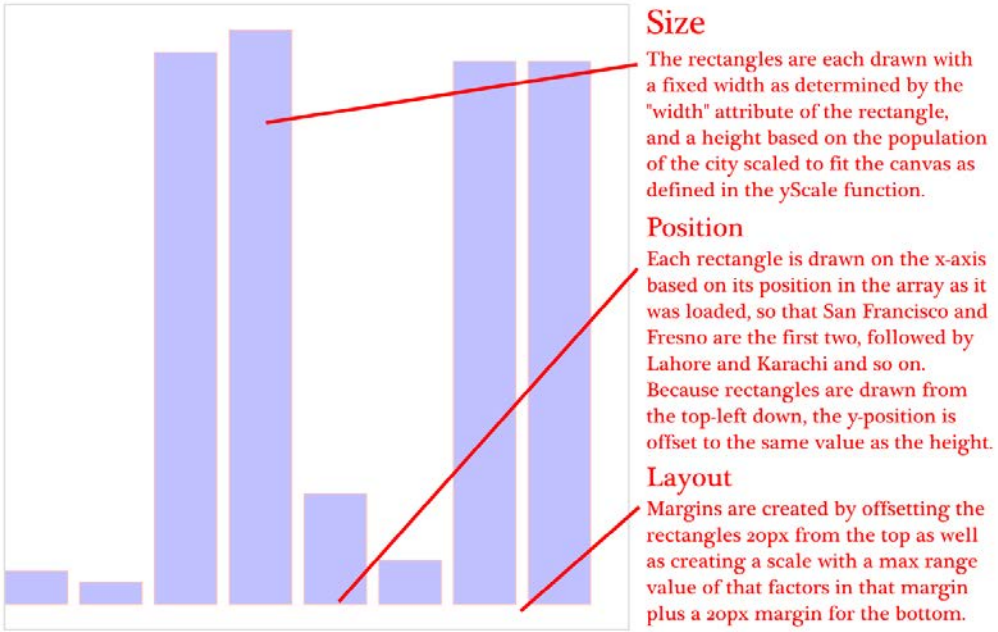


Figure 2.20: Here we can see the cities.csv data drawn as a bar chart using the maximum value of the population attribute in the domain setting of the scale.

Making a bar chart out of the Twitter data requires a bit more transformation. We'll use `d3.nest()` to gather the tweets under the person making them, and then rely on the length of that array to provide us with a bar chart of number of tweets.

Listing 2.4 Loading data, nesting it, measuring the nested arrays, and representing that measure on a bar chart.

```
d3.json("tweets.json",function(error,data) {dataViz(data.tweets)}); #A
function dataViz(incomingData) {

var nestedTweets = d3.nest()
.key(function (el) {return el.user})
.entries(incomingData);

nestedTweets.forEach(function (el) {
el.numTweets = el.values.length #B
})

var maxTweets = d3.max(nestedTweets, function(el) {return el.numTweets});

var yScale = d3.scale.linear().domain([0,maxTweets]).range([0,100]);

d3.select("svg")
```

```
.selectAll("rect")
.data(nestedTweets)
.enter()
.append("rect")
.attr("width", 50)
.attr("height", function(d) {return yScale(d.numTweets)})
.attr("x", function(d,i) {return i * 60})
.attr("y", function(d) {return 100 - yScale(d.numTweets)})
.style("fill", "blue")
.style("stroke", "red")
.style("stroke-width", "1px").style("opacity", .25)
}
```

#a notice that we need to specify data.tweets, where our data array is located
#b create a new attribute based on the number of tweets

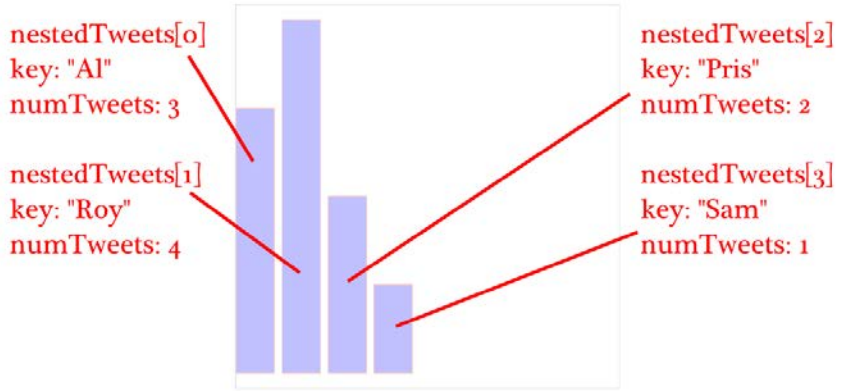


Figure 2.21: By nesting data and counting the objects that are nested, we can create a simple bar chart out of basic hierarchical data.

2.3.2 Setting Channels

So far, we've only used the height of a rectangle to correspond to a point of data, and in cases where we are dealing with one piece of quantitative data, then that's all we need. That's why bar charts are so popular in spreadsheet applications. But most of the time you'll be using multivariate data, such as census data for counties or medical data for patients. When you look at your medical history, it does not come back as a single score between 0 and 100, instead it consists of multiple measures that explain different aspects of your health. In cases where you're dealing with multivariate data, you need to develop techniques that let you represent multiple data points in the same shape. The technical term for the visual way that a shape can express data is a channel, and depending on the data you're working with, different channels are better suited to express data graphically.

Infoviz Terms: Channels

The representation of data using graphics requires you to consider the best visual methods to represent the types of data you're working with. Each graphical object, as well as the whole display, can be broken down into component channels that relay information visually. These channels, such as height, width, area, color, position, and shape, have been found to be particularly well-suited to represent different classes of information. For instance, when we represent magnitude by changing the size of a circle, if you create a direct correspondence between radius and magnitude, then your readers will be confused, since we tend to recognize the area of a circle rather than its radius. Channels also exist at multiple levels, and some techniques use hue, saturation, and value to represent three different pieces of information, rather than just using color more generically.

The important thing to remember here is to avoid using too many channels, and focus on using the channels most suitable to your data. If you are not varying shape, for instance if you're using a bar chart where all the shapes are rectangles, then you can leverage color for category and value (lightness) to represent magnitude.

Going back to the `tweets.json` data, it may seem like there's not much data available to put on a chart, but depending on what factors we want to measure and display, we can actually take a couple different approaches. Let's imagine we want to measure the impact factor of tweets, treating tweets that are favorited or retweeted as more important than tweets that are not. This time instead of a bar chart, we create a scatterplot to do so, and instead of just using array position to place it along the x-axis, let's use time, because there's good evidence indicating that tweets made at certain times are more likely to be favorited or retweeted. We'll place each tweet along the y-axis using a scale based on the maximum impact factor of our set of tweets. From this point on, I'll focus on just the `dataViz()` function, because you should be familiar now with getting your data in and sending it to such a function.

```
function dataViz(incomingData) {

incomingData.forEach(function (el) {
el.impact = el.favorites.length + el.retweets.length #A
el.tweetTime = new Date(el.timestamp); #B
})

var maxImpact = d3.max(incomingData, function(el) {return el.impact});
var startEnd = d3.extent(incomingData, function(el) {return el.tweetTime});
#C
var timeRamp = d3.time.scale().domain(startEnd).range([20,480]); #D
var yScale = d3.scale.linear().domain([0,maxImpact]).range([0,460]);
var radiusScale = d3.scale.linear().domain([0,maxImpact]).range([1,20]);
var colorScale =
d3.scale.linear().domain([0,maxImpact]).range(["white","#990000"]); #E

d3.select("svg")
.selectAll("circle")
.data(incomingData)
.enter()
.append("circle")
```

```
.attr("r", function(d) {return radiusScale(d.impact)}) #F
.attr("cx", function(d,i) {return timeRamp(d.tweetTime)})
.attr("cy", function(d) {return 480 - yScale(d.impact)})
.style("fill", function(d) {return colorScale(d.impact)})
.style("stroke", "black")
.style("stroke-width", "1px")
}
```

#a create a simple impact score by totaling the number of favorites and retweets
#b transform the ISO-8906 compliant string into a date datatype
#c extent will return the earliest and latest times for a scale
#d startEnd is an array
#e build a scale that maps impact to a ramp from white to dark red
#f Size, color and vertical position will all be based on impact

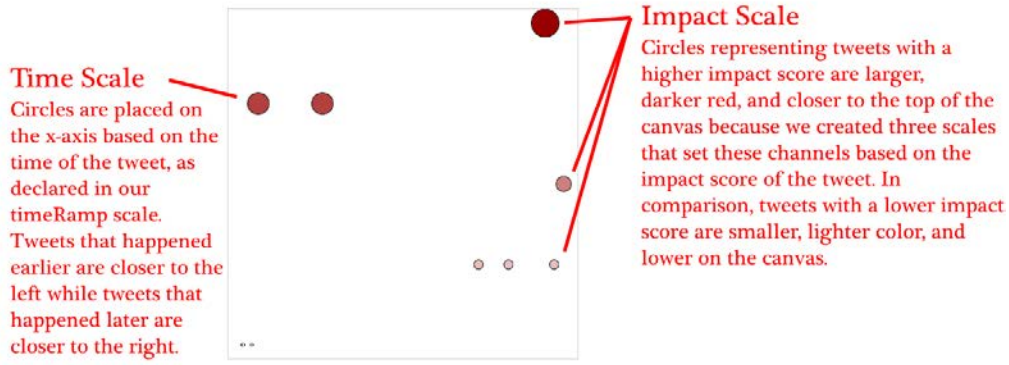


Figure 2.22: Tweets are represented as circles sized by the total number of favorites and retweets, and are placed on the canvas along the x-axis based on the time of the tweet and along the y-axis according to the same impact factor used to size the circles. Two tweets with the same impact factor that were made at nearly the same time can be seen overlapping at the bottom left.

2.3.3 Enter, Update and Exit

We've used the `.enter()` behavior of a selection many times already, now let's take a closer look at it and its counterpart, `.exit()`. Both of these functions operate when there's a mismatch between the number of data values bound to a selection and the number of DOM elements in the selection. If there are more data values than DOM elements, then `.enter()` fires, whereas if there are fewer data values than DOM elements, then `.exit()` fires, as in figure 2.x. You use selection.enter() to define how you want to create new elements based on the data you're working with, and you use selection.exit() to define how you want to remove existing elements in a selection when the data that corresponds to them has been deleted. Updating data, as you'll see below, is accomplished through re-applying the functions you used to create the graphical elements based on your data.

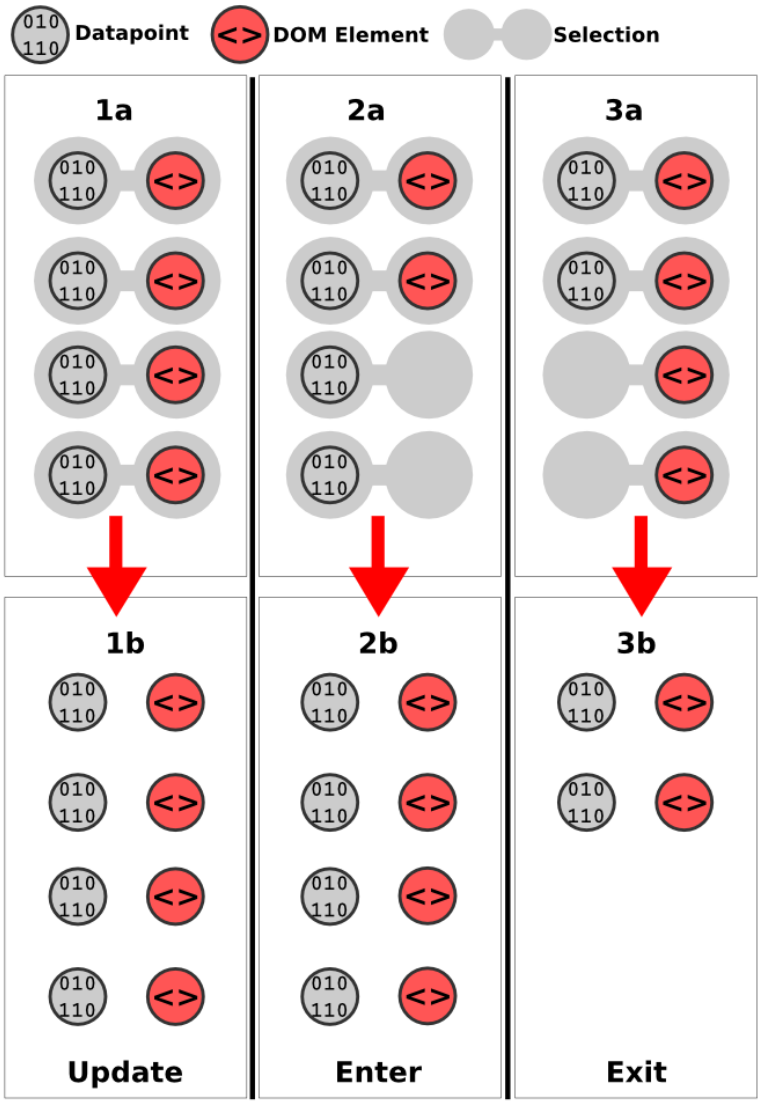


Figure 2.23: Selections where the number of DOM elements and number of values in an array don't match will either fire an .enter() event or an .exit() event, depending on whether there are more or fewer data values than DOM elements.

Each `.enter()` or `.exit()` event can include actions taken on child elements. This is mostly useful with `.enter()` events where you leverage the `.append()` function to add new elements.

If you declare this new appended element as a variable, and if that element is amenable to child elements, like a <g> element is, then you can include any number of child elements. For example, let's say we want to show a bar chart based our newly measured impact score, and we want the bars on the bar chart to have labels.

To do so, we'll need to append <g> elements to the <svg> canvas in our initial selection and not shapes. Because the data is bound to these elements, we can use the same syntax when we add child elements. Since we're using <g> elements, we need to set the position using the "**transform**" attribute. We add child elements to the `.append()` function, and to do so we need to declare it as a variable `tweetG`. This allows tweetG to stand in for d3.select("svg").selectAll("g") so we don't have to retype it throughout the example. This code uses all the same scales to determine size and position as the previous example.

```
var tweetG = d3.select("svg")
.selectAll("g")
.data(incomingData)
.enter()
.append("g")
.attr("transform", function(d) {
return "translate(" +
timeRamp(d.tweetTime) + "," + (480 - yScale(d.impact)) #A
+ ")"
})

tweetG.append("circle")
.attr("r", function(d) {return radiusScale(d.impact)})
.style("fill", "#990000")
.style("stroke", "black")
.style("stroke-width", "1px")

tweetG.append("text")
.text(function(d) {return d.user + "-" + d.tweetTime.getHours()}); #B
```

#a <g> requires a "transform" which takes a constructed string
#b use .getHours() to make the label a bit more legible

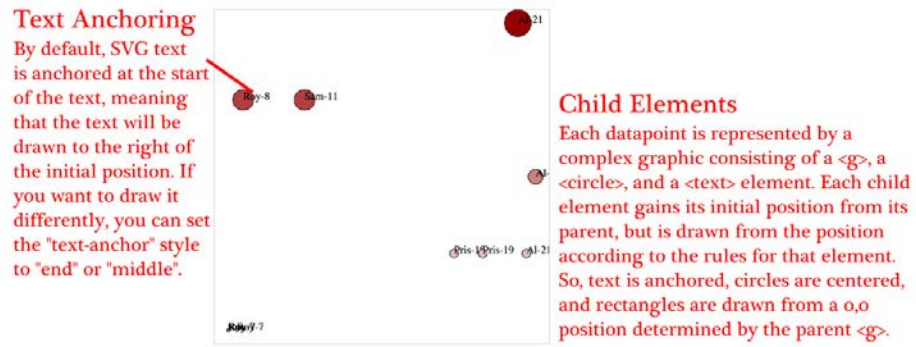


Figure 2.24: Each tweet is a <g> element with a circle and a label appended to it. We can see that the various tweets by Roy at 7AM happen so close to each other that they're difficult to label.

The labels are illegible in the bottom left, but they're not much better for the rest. Later on, we'll learn how to make better labels. You'll notice that the inline functions such as the `.text(function(d) {return d.user + "-" + d.tweetTime.getHours()})` that sets the label to be the name of the person making the tweet, followed by a dash, followed by the hour of the tweet, all refer to the same data elements, because the child elements inherit their parent's data functions. If one of your data elements is an array, you might think you could bind it to a selection on the child element and you'd be right. We'll see that in the next chapter and later in the book.

Corresponding to the `.append()` function is the `.remove()` function available with `.exit()`. To see `.exit()` in action, we need to have some elements in the DOM, which could already exist based on what you put in your HTML or which could have been added with D3. Let's stick with the state that the code above creates, which provides us with ample opportunity to test the `.exit()` function. Remember that DOM element style and attribute is not updated if you make a change to the array unless you call the necessary `.style()` and `.attr()` functions. So if we bind any array to the existing <g> elements in your DOM, then we can use `.exit()` to remove them:

```
d3.selectAll("g").data([1,2,3,4]).exit().remove();
```

The code above deleted all but four of our <g> elements, because there are only four values in our array. In most of the explanations of how D3's .enter() and .exit() behavior perform, you won't see this kind of binding of an entirely different array to a selection. Instead, you'll see a re-binding of the initial data array after it has been filtered to represent a change via user interaction or other behavior. You'll see an example like this next, and throughout the book. But it's important to understand the difference between your data, your selection, and your DOM elements. The data that is bound to our DOM elements has been overwritten, so our data-rich objects from tweets.csv have now been replaced with boring numbers, but the visual representation has only changed insofar as it has been reduced in number to reflect the size of the array we've bound. D3 doesn't follow the convention that when the data has changed that the corresponding display is updated, you need to build that functionality yourself, but because it does not follow that convention, it gives you greater flexibility that we'll explore in later chapters.

UPDATING

We can see this by updating the <text> elements in each g to reflect the newly bound data:

```
d3.selectAll("g").select("text").text(function(d) {return d});
```

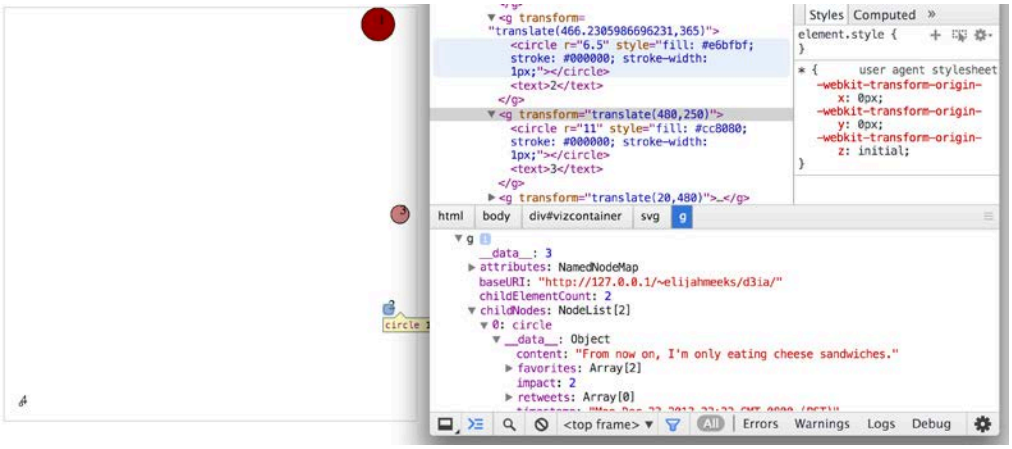


Figure 2.25: Only four <g> elements remain, corresponding to the four data values in the new array, with their <text> labels reset to match the new values in the array. However, when we inspect the <g> element, we can see that its __data__ property, where D3 stores the bound data, is different than that of its <circle> child element, which still has the JSON object we bound when we first created the visualization.

It is critical to notice that in this example we had to `.selectAll()` the parent elements and then sub-select the child elements to re-initialize the data binding for the child elements. Whenever you bind new data to a selection that utilizes child elements, you will need to follow this pattern. You can see, since we didn't update the <circle> elements, that they still have the old data bound to each element:

```
d3.selectAll("g").each(function(d) {console.log(d)}); #A
d3.selectAll("text").each(function(d) {console.log(d)}); #B
d3.selectAll("circle").each(function(d) {console.log(d)}); #C
```

#a returns values from the newly bound array.
#b returns values from the newly bound array, because we used a sub-select
#c returns values from the old tweetData array, because we have not specified overwriting with a sub-select

Using the `.exit()` function by binding a new array of completely different values like this isn't quite what it's intended for. Instead, it's meant to update the page based on the removal of elements from the array that's been bound to the selection. But if you plan to do this, you need to specify how the `.data()` function binds data to your selected elements. By default, `.data()` binds based on the array position of the data value. This means, in our example above, that the first four elements in our selection are maintained and bound to the new data while the rest are subject to the `.exit()` function. In general, though, we don't want to rely on array position as our binding key. Rather, we should use something meaningful, such as the value of the data object itself. The key requires a string or number, so if you pass a JSON object without using JSON.stringify, it will treat all objects as "[object object]" and only return

one unique value. To manually set the binding key, you will use the second setting in the `.data()` function and use the inline syntax typical in D3:

```
function dataViz(incomingData) {

incomingData.forEach(function (el) {
el.impact = el.favorites.length + el.retweets.length
el.tweetTime = new Date(el.timestamp);
})

var maxImpact = d3.max(incomingData, function(el) {return el.impact});
var startEnd = d3.extent(incomingData, function(el) {return el.tweetTime});
var timeRamp = d3.time.scale().domain(startEnd).range([50,450]);
var yScale = d3.scale.linear().domain([0,maxImpact]).range([0,20]);

d3.select("svg")
.selectAll("circle")
.data(incomingData, function(d) {return JSON.stringify(d)}) #A
.enter()
.append("circle")
.attr("r", function(d) {return yScale(d.impact)})
.attr("cx", function(d,i) {return timeRamp(d.tweetTime)})
.attr("cy", function(d) {return 100 - yScale(d.impact)})
.style("fill", "#990000")
.style("stroke", "black")
.style("stroke-width", "1px")
}
```

#a we could use any unique attribute as the key, but using the entire object works if you don't have a unique value, though you have to stringify it first

The visual results are the same as our earlier scatterplot with the same settings, but now if we filter the array we used for the data, and bind that to the selection, then we can define some useful `.exit()` behavior:

```
var filteredData = incomingData.filter(
function(el) {return el.impact > 0}
);
d3.selectAll("circle")
.data(filteredData, function(d) {return JSON.stringify(d)})
.exit()
.remove();
```

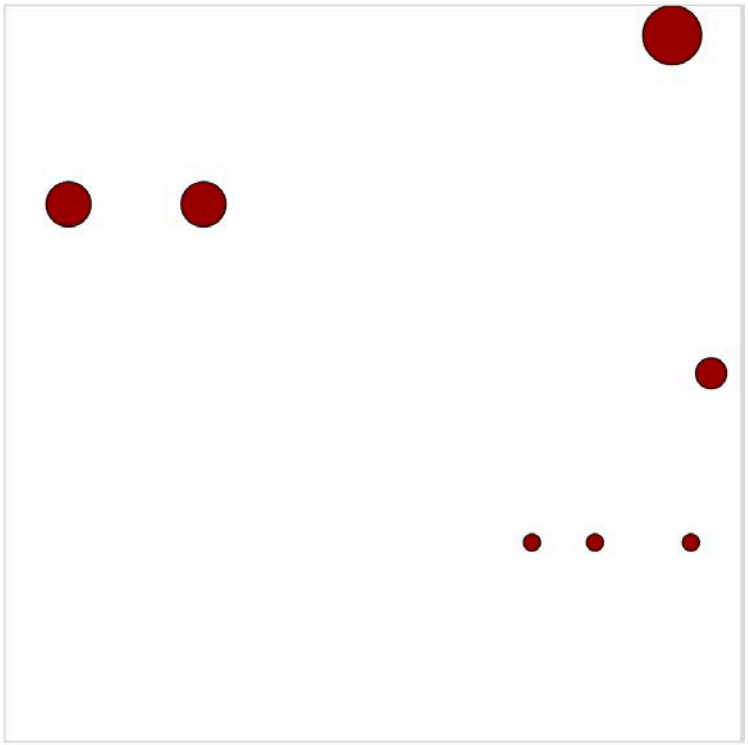


Figure 2.26: All elements corresponding to tweets that were not favorited and not retweeted were removed.

Note that using the stringified object will not work if you are changing the data in the object, because then it will no longer correspond with the original binding string. If you plan to do significant changing and updating, then you will need a unique ID of some sort for your objects to use as your binding key.

2.4 Summary

In this chapter we looked closely at the core elements for building data visualizations using D3. This included:

- Loading data from external files in CSV and JSON format.
- Formatting and transforming data using D3 scales and built-in JavaScript functions.
- Measuring data in order to build graphically useful visualizations.
- Binding data to create graphics based on the attributes of the data.
- Using sub-selections to create complex graphical objects made of multiple shapes using the <g> element.
- Understanding how to create, change, and move elements using enter, exit, and

selections.

Almost all the code you will write using D3 is some kind of variation or elaboration on the material covered in this chapter. In the next chapter we'll focus on the design details necessary for a successful D3 project, while exploring how D3 implements interaction, animation, and the use of pregenerated content.

3

Data-Driven Design and Interaction

Data visualization frameworks have long existed in a form that separates them from the rest of web development. Flash or Java apps are dropped into a web page and the only design necessary for them is to have a big enough or to account for the fact that it may be resized. D3 changes that, and gives you the opportunity to integrate the design of your data visualization with the design of your more traditional web elements.

Content you generate with D3 can and should be styled with all the same CSS settings as traditional HTML content so that you can easily maintain those styles and have a consistent look and feel. This can be done by using the same stylesheet classes for what you create with D3 as you do with your traditional page elements when possible and following thoughtful use of color and interactivity with the graphics you create using D3.

This chapter deals with design broadly speaking, and as such it will touch not only on graphical design but interaction design, project architecture and integrating pre-generated content. The focus here is on highlighting the connections from D3 to other methods of development, whether in identifying libraries typically used alongside D3 or in integrating HTML and SVG resources created using other tools. I cannot cover all of the principles of design (which is not one field but many) and instead focus on how to use particular D3 functionality to follow the best practices established by design professionals.

3.1 Project Architecture

If you're just creating a single webpage with an interesting information visualization on it, then you don't need to think too much about where all your files are going to live. But if you're building an application that is going to provide multiple points of interaction and different states, then you should identify the resources that you'll need and plan your project accordingly.

3.1.1 Data

Your data will tend to come in one of two forms: either dynamically delivered via server/API or in static files. Even if you're pulling data dynamically from a server or API, it's possible that you'll have static files as well. A good example of this is seen when building maps, where the base data layer (such as a map of countries) is from a static file and the dynamic data layer (such as the places where tweets are made) comes from a server. For this chapter, we'll use a simple CSV representing some statistics for the 2010 World Cup:

worldcup.csv

```
"team","region","win","loss","draw","points","gf","ga","cs","yc","rc"
"Netherlands","UEFA",6,0,1,18,12,6,2,23,1
"Spain","UEFA",6,0,1,18,8,2,5,8,0
"Germany","UEFA",5,0,2,15,16,5,3,10,1
"Argentina","CONMEBOL",4,0,1,12,10,6,2,8,0
"Uruguay","CONMEBOL",3,2,2,11,11,8,3,13,2
"Brazil","CONMEBOL",3,1,1,10,9,4,2,9,2
"Ghana","CAF",2,2,1,8,5,4,1,12,0
"Japan","AFC",2,1,1,7,4,2,2,4,0
```

That's a lot of data for each team, and while we could try to come up with some kind of graphical object that encodes all 9 different data points simultaneously (plus labels), instead we'll use interactive and dynamic methods to provide access to the data.

3.1.2 Resources

Pre-generated content like hand-drawn SVG and HTML components are another kind of external file that you'll need to know how to load. You'll see some examples of these later on in the chapter. Each file contains enough code to draw the shape or traditional DOM elements you'll be adding to your page. We'll spend a lot more time with the contents of this folder later on in Section 3.3.2 and Section 3.3.3 when we deal with loading pre-generated content.

3.1.3 Images

Later on, you'll use a set of PNGs with the flags of each team represented in our dataset. We're going to name the PNGs with the same name as the team, so that it's easier to use the images with D3, as you'll see below. Every digital file is made of code, but we think of images as fundamentally different. This distinction breaks down when we deal with SVG when you're accustomed to treating SVG as if they're images. If you're dealing with SVG images as images and not as code that you want to manipulate in D3, then you should put them in your image directory and keep the others in your resources directory.

3.1.4 Stylesheets

While we won't focus on CSS in this chapter too much, you should be aware that there is much you can do not only with CSS but also with CSS compilers that affords support for variables in CSS and other improved functionality. Our stylesheet seen in Listing 3.1 has classes for the different states of the SVG elements we're dealing with, which includes SVG text elements that use a different syntax than traditional DOM elements when dealing with font.

Listing 3.1 d3ia.css

```
text {
font-size: 10px;
}

g > text.active {
font-size: 30px;
}

circle {
fill: pink;
stroke: black;
stroke-width: 1px;
}

circle.active {
fill: red;
}

circle.inactive {
fill: gray;
}
```

3.1.5 External libraries

For the example in this chapter, we'll only be using two more .js files besides d3.min.js, which is the minified D3 library. The first is soccerviz.js, which stores the functions we'll be building and using in this chapter. The second is colorbrewer.js, which also comes bundled with D3 and provides a set of predefined color palettes that you'll find useful and which we'll deal with below.

This is reflected through reference to these files in our much cleaner d3ia_2.html seen in Listing 3.2.

Listing 3.2 d3ia_2.html

```
<html>
<head>
  <title>D3 in Action Examples</title>
  <meta charset="utf-8" />
  <link type="text/css" rel="stylesheet" href="d3ia.css" />
</head>
<script src="d3.v3.min.js" type="text/javascript"></script>
<script src="colorbrewer.js" type="text/javascript"></script>
<script src="soccerviz.js" type="text/javascript"></script>
<body onload="createSoccerViz()">
<div id="viz">
<svg style="width:500px;height:500px;border:1px lightgray solid;" />
</div>
<div id="controls" />
</body>
</html>
```

The <body> has two elements, one with the id "viz" and the other with the id "controls". Notice that the <body> element has an onload property that runs one of our functions in soccerviz.js (seen in Listing 3.3), createSoccerViz(). This loads the data and binds it to create a labeled circle for each team. It's not much, but it's a start.

Listing 3.3 soccerviz.js

```
function createSoccerViz() {
  d3.csv("worldcup.csv", function(data) { #a
    overallTeamViz(data);
  })

function overallTeamViz(incomingData) {
d3.select("svg")
.append("g") #b
.attr("id", "teamsG")
.attr("transform", "translate(50,300)")
.selectAll("g")
.data(incomingData)
.enter()
.append("g")
.attr("class", "overallG")
.attr("transform", function (d,i) {return "translate(" + (i * 50) + ", 0)"})
#c

var teamG = d3.selectAll("g.overallG"); #d

teamG
.append("circle")
.attr("r", 20)
.style("fill", "pink")
.style("stroke", "black")
.style("stroke-width", "1px")

teamG
.append("text")
.style("text-anchor", "middle")
.attr("y", 30)
.style("font-size", "10px")
.text(function(d) {return d.team})
  }
}
```

#a Load the data and run createSoccerViz with the loaded data
#b By appending a <g> to the <svg> canvas, we can move it and center its contents more easily
#c We create a <g> for each team so that we can add labels or other elements as we get more ambitious
#d By assigning the selection to a variable, we can refer to it without typing out d3.selectAll() every time

While you might write an application entirely with D3 and your own custom code, for large-scale sustainable projects, you'll have to integrate more external libraries. We'll only be using one of those, colorbrewer.js, which isn't very intimidating. The colorbrewer library is a set of

arrays of colors, which are useful in information visualization and mapping. We'll see this library in action in the Section 3.3.2 of this chapter.

3.2 Interactive Style and DOM

Creating interactive information visualization is key to providing your users with the capacity to deal with large and complex datasets. And the key to building interactivity into your D3 projects is the use of events, which define behaviors based on user activity. Once you understand how to make your elements interactive, you'll need to understand D3 transitions, which allow you to animate the change from one color or size to another. With that in place, we'll spend some time understanding how to make changes to an element's position in the DOM so that you can make sure your graphics are drawn properly. Finally, we'll look more closely at color, which you'll use often in response to user interaction.

3.2.1 Events

To get started, let's update our visualization to add some buttons that change the appearance of our graphics to correspond with different data. We could hand code the buttons in HTML and tie them to functions like you would in traditional web development, but we can also use D3 to discover and examine the attributes in the data and create buttons dynamically. This has the added benefit of scaling to the data, so that if we add more attributes to our dataset, then this function will automatically create the necessary buttons.

```
 var dataKeys = d3.keys(incomingData[0]) #a
.filter(function (el) {return el != "team" && el != "region"})
   d3.select("#controls").selectAll("button.teams").data(dataKeys).enter().ap
pend("button")
   .on("click", buttonClick) #b
   .html(function(d) {return d}) #c

function buttonClick(datapoint) { #d
var maxValue = d3.max(incomingData,
function(d) {return parseFloat(d[datapoint])
});
var radiusScale = d3.scale.linear().domain([0,maxValue]).range([2,20]);
     d3.selectAll("g.overallG")
.select("circle")
.attr("r", function(d) {return radiusScale(d[datapoint])})
}
```

#a We're going to only build buttons based on the data that's numerical, so we know we want all the attributes except the team and region attributes, which store strings
#b This registers an onclick behavior for each button, with a wrapper that gives access to the data that was bound to it when it was created
#c Remember that dataKeys consists of an array of attribute names, so the d will correspond to one of those names, and make a good button title
#d The function each button is calling on click, with the bound data sent automatically as the first argument

In this case, we use `d3.keys` and we pass it one of the objects from our array. The `d3.keys` function returns the names of the attributes of an object as an array. We've made

sure to filter this array to remove the "team" and "region" attributes because these have non-numerical data and won't be suitable for the `buttonClick` functionality we're defining. Obviously, in a larger or more complex system, you'll want to have more robust methods for designating attributes than listing them by hand like this. We'll see that later when we deal with more complex datasets. In this case, we bind this filtered array to a selection to create buttons for all the remaining attributes, and give the buttons labels for each of the attributes, as seen in Figure 3.1.

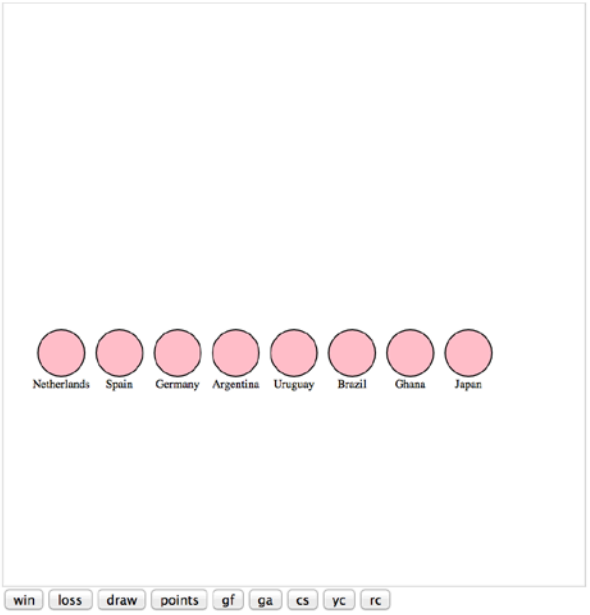


Figure 3.1 Buttons for each numerical attribute are appended to the "controls" div underneath the "viz" div.When clicked, it runs buttonClick.

The `.on` function is a wrapper for the traditional HTML mouse events, and accepts "click", "mouseover", "mouseout" and so on. If you want to, you can access those same events using `.attr`, such as `.attr("onclick", "console.log('click')")` but notice that in this case you are passing a string in the same way you would using traditional HTML. There's a D3-specific reason to use the `.on` function, though: it sends the bound data to the function automatically and in the same format as the anonymous inline functions we've been using to set style and attribute.

By creating buttons based on the attributes of the data and dynamically measuring the data based on the attribute bound to the button, we can resize the circles representing each team to reflect the teams with the highest and lowest values in each category like we do in Figure 3.2.

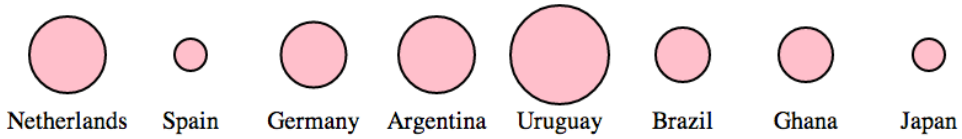


Figure 3.2 The effect of our initial buttonClick function resizes the circles based on the numerical value of the associated attribute. In this case, we see the size of each circle with the radius reflecting the number of goals scored against each team, kept in the "ga" attribute of each datapoint.

We can use .on() to tie events to any object, so let's add some interactivity to the circles by making them indicate whether teams are in the same FIFA region:

```
teamG.on("mouseover", highlightRegion)

function highlightRegion(d) {
 d3.selectAll("g.overallG").select("circle").style("fill", function(p)
{return p.region == d.region ? "red" : "gray"})
}
```

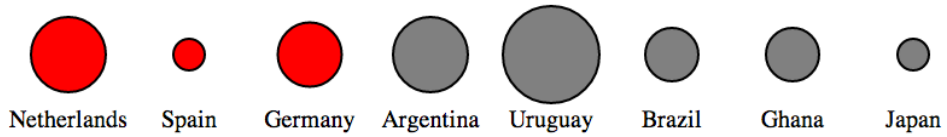


Figure 3.3 The effect of our initial highlightRegion selects elements with the same region attribute and colors them red, while coloring those that aren't in the same region gray.

This time we used d as our variable, which is typical in the examples you'll see on-line for D3 functionality. As a result, we changed the inline function variable to p, so that it wouldn't conflict. Here we see an "ifsie", which is an inline if statement that tests the region of each element in the selection and compares it to the region of the element that we moused over with results like those in Figure 3.3.

Restoring the circles to their initial color on mouse out is simple enough that the function can be declared inline with the `.on` function:

```
teamG.on("mouseout", function()
{d3.selectAll("g.overallG").select("circle").style("fill", "pink")})
```

If you want to define custom event handling, you would use `d3.dispatch`, which you can see in action in Chapter 9.

3.2.2 Graphical Transitions

One of the challenges of highly interactive, graphics-rich web pages is to ensure that the experience of graphical change isn't jarring. The instantaneous change in size or color that we've already implemented doesn't just look clumsy, it can actually interfere with a reader understanding the information you're trying to relay. In order to smooth things out a bit, we'll introduce transitions, which we saw briefly at the end of Chapter 1.

Transitions are defined for a selection, and can be set to occur after a certain delay using delay() or to occur over a set period of time using duration(). We can easily implement a transition in our buttonClick function:

```
d3.selectAll("g.overallG").select("circle").transition().duration(1000).attr(
"r", function(p) {return radiusScale(d[datapoint])})
```

Now when we click our buttons, we not only see the sizes of the circles change, but the change is animated. This isn't just for show, we're actually encoding new data, indicating the change between two datapoints using animation. Before, when there was no animation, we had to rely on the reader remembering if there was a difference between the ranking in draws and wins for Germany, now the reader has an animated indication in that Germany shrinks or grows visibly indicating the difference between these two datapoints.

The use of transitions also brings with it the ability to delay the application of the change through the .delay() function. Like the .duration() function, .delay() is set with the number of milliseconds to wait before implementing the change. Slight delays in the firing of an event from interaction can be useful to improve legibility of information visualization, allow the user to have a slight moment to re-orient themselves to shift from interaction to reading, but long delays will usually be misinterpreted as being poor web performance.

Why else would you want to delay the firing of an animation? Delays can also help to draw attention to visual elements when they're first drawn. By making the elements pulse when they first arrive on-screen, you let the user know that these are dynamic objects and tempt them to click or otherwise interact with them. Delays, like duration, can be dynamically set based on the bound data for each element. We can use delays with another feature: transition chaining, which is setting multiple transitions one after another which are each activated after the last transition has finished. If we amend the code in overallTeamViz() that first appends the <circle> elements to our <g> elements, we can see transitions of the kind that produce the screenshot in Figure 3.4.

```
teamG
.append("circle").attr("r", 0)
.transition()
.delay(function(d,i) {return i * 100})
.duration(500)
.attr("r", 40)
.transition()
.duration(500)
.attr("r", 20);
```

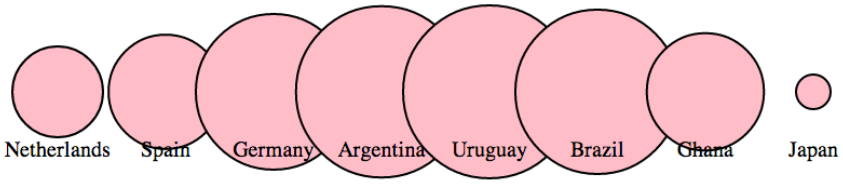


Figure 3.4 A screenshot of our data visualization in the middle of its initial drawing, which shows the individual circles growing to an exaggerated size and then shrinking to their final size in the order in which they appear in the bound dataset.

This causes a pulse because it uses transition chaining to set one transition followed by a second after the completion of the first. In this case, we start by drawing the circles with a radius of 0, so they're invisible. Each element has a delay set to its array position i times .1 seconds (100ms) after which it causes the circle to grow to a radius of 40px. After each circle grows to that size, it fires a second transition that shrinks the circles to 20px. The effect, which isn't very easy to present with a screenshot, is to cause the circles to pulse sequentially.

3.2.3 DOM Manipulation

Since these visual elements and buttons are all living in the DOM, it's important to know how to access and work with them both with D3 and using built-in JavaScript functionality.

While D3 selections are extremely powerful, there are times when we want to deal specifically with the DOM element that is bound to the data. These DOM elements comes with a rich set of built-in functionality in JavaScript. This can be accomplished in one of two ways:

1. Using `this` in the inline functions
2. Using the `.node()` function

Inline functions will always have access to the DOM element along with the datapoint and array position of that datapoint in the bound data. The DOM element, in this case, is represented by `this`. You can see it in action using the `.each()` function of a selection, which performs the same code for each element in a selection. We'll make a selection of one of our circles and then use .each() to send `d`, `i`, and `this` to the console to see what each corresponds to (which should look similar to the results in Figure 3.5).

```
d3.select("circle").each(function(d,i)
{console.log(d);console.log(i);console.log(this)})
```

```
> d3.select("circle").each(function(d,i) {console.log(d);console.log(i);console.log(this)})
  ▶ Object {team: "Netherlands", region: "UEFA", win: "6", loss: "0", draw: "1"…}
  0
  <circle r="20" style="fill: #ffc0cb; stroke: #000000; stroke-width: 1px;"></circle>
```

Figure 3.5 The console results of inspecting a selected element, which show first the datapoint in the selection, it's position in the array, and then the actual SVG element itself.

Unpacking this a bit, we can see the first thing echoed, d, is the data bound to the circle, which is a JSON object representing the Netherlands team. The second thing echoed, i, is the array position of that object in the array we used to create these elements, which in this case is 0 and means that incomingData[0] is the Netherlands JSON object. The last thing echoed to the console, `this`, is the actual `<circle>` DOM element itself.

We can also access this DOM element using the .node() function of a selection:

```
d3.select("circle").node();
```

```
d3.select("circle").node()
<circle r="20" style="fill: #ffc0cb; stroke: #000000; stroke-width: 1px;"></circle>
```

Figure 3.6 The results of running the node function of a selection in the console, which is the DOM element itself, in this case, an SVG <circle> element.

Getting to the actual DOM element like we see in Figure 3.6 lets us take advantage of built-in JavaScript functionality to do things like measure the length of a `<path>` element or clone an element. One of the most useful built-in functions of nodes when working with SVG is the ability to re-append a child element. Remember that SVG has no Z-levels, which means that the drawing order of elements is determined by their DOM order. Drawing order is important because it we don't want the graphical objects we're interacting with to look like they're beneath the objects that we're not interacting with. To see what I mean, let's first adjust our highlighting function so that it increases the size of the label when you mouse over each element:

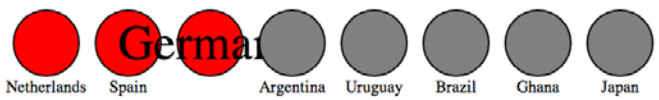
```
function highlightRegion2(d,i) {
 d3.select(this).select("text").classed("active", true).attr("y", 10);
 d3.selectAll("g.overallG").select("circle")
.each(function(d,i) { p.region == d.region ?
d3.select(this).classed("active",true) :
d3.select(this).classed("inactive",true)}) #a
}
```

#a By changing turning on "active" class for the <g> that we hover over, we take advantage of our "g > text.active" rule in CSS that makes any text elements in that <g> increase their font size

Because we're doing a bit more, we should change the mouseout event to point to a function, which we'll call unHighlight:

```
teamG.on("mouseout", unHighlight)
```

```
function unHighlight() {
 d3.selectAll("g.overallG").select("circle").attr("class", "");
 d3.selectAll("g.overallG").select("text").classed("highlight",
false).attr("y", 30);
```



}

Figure 3.7 The <text> element “Germany” is drawn at the same DOM level as the parent <g> which, in this case, is behind the element to its right.

As we can see in Figure 3.6, Germany was appended to the DOM before Argentina, and as a result when we increase the size of the graphics associated with Germany, those graphics remain behind any graphics for Argentina, creating a visual artifact that looks unfinished and distracting. We can rectify this by re-appending the node to the parent <g> during that same highlighting event which results in the label being displayed above the other elements as seen in Figure 3.8.

```
function highlightRegion2(d,i) {
 d3.select(this).select("text").classed("highlight", true).attr("y", 10);
 d3.selectAll("g.overallG").select("circle")
.classed( function(p) {return p.region == d.region ? "active" : "inactive"},
true);
 this.parentElement.appendChild(this);
```

}

Figure 3.8 By re-appending the <g> element for Germany to the <svg> element, it is moved to the end of that DOM region and therefore drawn above the other <g> elements.

You’ll see in this example that the mouseout event becomes less intuitive because the event is attached to the <g> element, which includes not only the circle but the text as well. As a result, mousing over the circle or the text will fire the event, and when you increase the size of

the text, and it overlaps a neighboring circle, it doesn't trigger a mouseout event. We'll get into event propagation later, but one thing you can do to easily disable mouse events on elements is to set the style property "pointer-events" of those elements to "none":

```
teamG.select("text").style("pointer-events","none");
```

3.2.4 Using Color Wisely

Color seems like a small and dull subject, but when you're representing data with graphics, color selection is of primary importance. There's a lot of good research done on the use of color in cognitive science and design, but that's an entire library. Here, we're going to deal with a few fundamental issues: Understanding how colors are mixed in color ramps, discrete colors for categorical data, and designing for accessibility factors raised by colorblindness.

Infoviz Terms: Color Theory

Attention to what makes a compelling visual is not decorative chart-junk. As Edward Tufte noted in *Envisioning Information*, chart-junk is empty decoration, while attention to visual clarity results in "signal enhancement of data through noise reduction,…[reducing] viewer fatigue and the accuracy of reading." As such, distracting and unharmonious color and value placement become unintentional chart junk. Tufte relies heavily on Josef Albers, a master of color theory, when making such assessments. And as Albers aptly noted, in the visual realm 1 +1 = 3. There are optical consequences of placing certain colors and shapes next to each other, resulting in simultaneous and successive contrast as well as accidental color.

It is thus worth studying the propertied of color--hue, value, intensity, and temperature--to ensure the most harmonious color relationships in a visualization. Leonardo da Vinci organized colors into psychological primaries, the colors the eyes sees unmixed, but the modern exploration of color theory, as with many other phenomena in physics, can be attributed to Sir Isaac Newton. Newton observed the separation of sunlight into bands of color via a prism in 1666, and called it a color spectrum. Newton also devised a color circle of seven hues, a precursor to the many future visualizations that would organize colors and their relationships. About a century later, J. C. Le Blon would identify the primary colors as red, yellow, and blue and their mixes as the secondaries. The work of other more modern color theoreticians like Josef Albers, who emphasized the effects of color juxtaposition, influence the standards for presentation in print and on the web.

Color is typically represented on the web in RGB, or red, green, and blue, using one of three formats: hex, RGB, or CSS color name. The first two both represent the same information, the level of Red, Green, and Blue in the color, but do so with either hexidecimal or comma-delimited decimal notation. CSS color names use vernacular names for colors, and of which there are 140 (You can read all about them on Wikipedia http://en.wikipedia.org/wiki/Web_colors#X11_color_names). Red, for instance, can be represented as:

```
"rgb(255,0,0)" #a
"#ff0000" #b
"red" #c
```

#a RGB, or Red-Green-Blue, encoded color
#b Hex, or hexidecimal, formatted RGB
#c CSS3 web color name

There are, as you might expect, a few helper functions in D3 to support working with colors. The first is d3.rgb(), which allows you to create a more feature-rich color object suitable for data visualization. It's pretty easy to use d3.rgb(), you just need to give it the red, green, and blue values of your color:

```
teamColor = d3.rgb("red");
teamColor = d3.rgb("#ff0000");
teamColor = d3.rgb("rgb(255,0,0)");
teamColor = d3.rgb(255,0,0);
```

These color objects have two useful methods, .darker() and .brighter(). They do exactly what you'd expect: returning a color that is a darker or brighter version of the color you started with. In our case, we can replace the gray and red that we've been using to highlight similar teams with darker and brighter versions of pink, the color we started with:

```
function highlightRegion2(d,i) {
 var teamColor = d3.rgb("pink")
 d3.select(this).select("text").classed("highlight", true).attr("y", 10)
 d3.selectAll("g.overallG").select("circle").style("fill", function(p)
{return p.region == d.region ? teamColor.darker(.75) :
teamColor.brighter(.5)})
 this.parentElement.appendChild(this);
}
```

Figure 3.9 Use of the darker and brighter functions of a d3.rgb object in the highlighting function show a darker version of the set color for teams from the same region and lighter colors for teams from different regions.

Notice that you can set the intensity for how much brighter or darker you want the color to be. Our new version as seen in Figure 3.9 now maintains the palette during highlighting, with darker colors foregrounding and lighter colors receding. Unfortunately, this means we lose the

ability to style with CSS because we're back to using inline styles. As a rule, you should use CSS whenever you can, but if you want access to things like dynamic colors and transparency using D3 functions, then you'll need to use inline styling.

There are other ways to represent color with various benefits, but we'll only deal with HSL, which stands for Hue, Saturation, and Lightness. The corresponding d3.hsl() will allow you to create HSL color objects in the same way that you would with d3.rgb(). The reason why we may want to use HSL is to avoid the muddying we see when we darken pink, which can also happen when we build color ramps and mix colors using D3 functions.

COLOR MIXING

In chapter 2, we saw that you can create a color ramp mapped to numerical data to create a spectrum of color to represent your datapoints. But the interpolated values for colors created by these ramps can be quite poor. As a result, a ramp that includes, say, yellow, can end up interpolating values that are very muddy and hard to distinguish. You may think this isn't important but when you're using a color ramp to indicate a value and your color ramp does not interpolate the color in a way that your reader expects, then you can end up showing wrong information to your users. Let's add a color ramp to our buttonClick function and use the color ramp to show the same information we did with radius.

```
var ybRamp = d3.scale.linear()
.domain([0,maxValue]).range(["yellow", "blue"]) #a
```

#a This is the same kind of color ramp we built back in Chapter 2, using the maxValue we calculated for our circle radius scale.

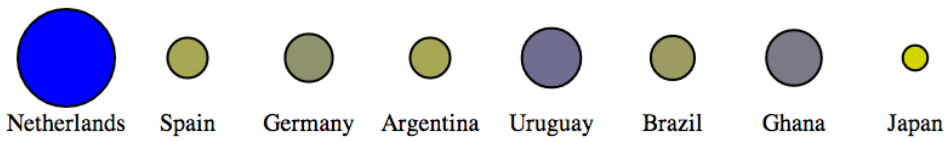


Figure 3.10 Color mixing between yellow and blue in the RGB scale results in muddy gray-like colors displayed for the values between yellow and blue.

You'd be forgiven if you expected the colors in Figure 3.10 to range from yellow to green to blue. The problem is that the default interpolator in the scale we used is mixing the red, green, and blue channels numerically. You can change the interpolator in the scale by designating one specifically, for instance using the HSL representation of color (Figure 3.11) that we looked at above:

```
var ybRamp = d3.scale.linear()
.interpolate(d3.interpolateHsl) #a
.domain([0,maxValue]).range(["yellow", "blue"])
```

#a Setting the interpolation method for a scale is necessary when you don't want it to use its default behavior, such as when we want to create a color scale with a method other than interpolating the RGB values.

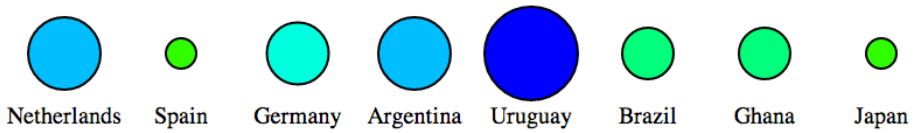


Figure 3.11 Interpolation of yellow to blue based on hue, saturation and lightness (HSL) results in a different set of intermediary colors from the same two starting values.

There are two other supported color interpolators in D3: HCL (Figure 3.12) and LAB (Figure 3.13), that each deal with the question of what colors are between blue and yellow in a different manner. First, the HCL ramp:

```
var ybRamp = d3.scale.linear()
.interpolate(d3.interpolateHcl)
.domain([0,maxValue]).range(["yellow", "blue"])
```

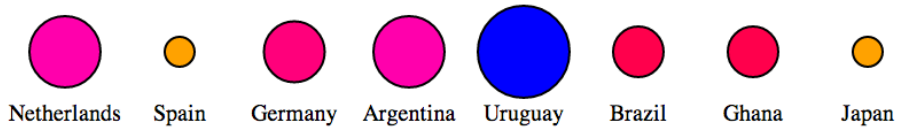


Figure 3.12 Interpolation of color based on Hue, Chroma, Luminosity (HCL) provides a different set of intermediary colors between yellow and blue.

Finally, the LAB ramp:

```
var ybRamp = d3.scale.linear()
.interpolate(d3.interpolateLab)
.domain([0,maxValue]).range(["yellow", "blue"])
```

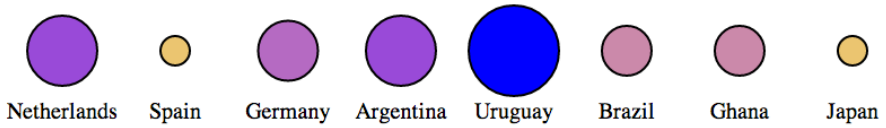


Figure 3.13 Interpolation of color based on Lightness and color opponent space (LAB) provides yet another different set of intermediary colors between yellow and blue.

As a general rule, you'll find the colors interpolated in RGB to tend toward muddy and gray, unless you break the color ramp into multiple stops. You can experiment with different color ramps, or choose to stick to ramps that emphasize hue or saturation (by using HSL). Or, you can rely on experts by using the built-in D3 functions for color ramps that are proven to be easier for a reader to distinguish, which we'll now look at.

DISCRETE COLORS

Oftentimes, we use color ramps to try to map colors to categorical elements. It's better to use the discrete color scales available in D3 for this purpose. The popularity of these scales is why so many D3 examples have the same palette. To get started, we need to use a new d3 scale, d3.scale.category10, which is built to map categorical values to particular colors. It works like a quantize scale where you cannot change the domain, because the domain is already defined as ten highly distinct colors. Instead you instantiate your scale with the values you want mapped to those colors. In our case, we want to distinguish the various regions in our dataset, which let's remember consists of the top 8 FIFA teams from the 2010 World Cup, representing four global regions. We want to represent these as different colors, and to do so we need to create a scale with those values in an array.

```
function buttonClick(datapoint) {
 var maxValue = d3.max(incomingData, function(el) {return
parseFloat(el[d])});
 var tenColorScale = d3.scale.category10(["UEFA", "CONMEBOL", "CAF", "AFC"]);
 var radiusScale = d3.scale.linear().domain([0,maxValue]).range([2,20]);
 d3.selectAll("g.overallG").select("circle").transition().duration(1000)
.style("fill", function(p) {return tenColorScale(p.region)})
.attr("r", function(p) {return radiusScale(p[d])})
}
```

The application of this scale is visible when you click on one of our buttons, seen in Figure 3.14, which now resizes the circles as it always has but also applies one of these distinct colors to each team.

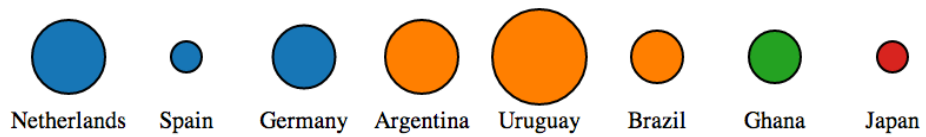


Figure 3.14 Application of the category10 scale in D3 assigns distinct colors to each class applied, in this case the four regions in our dataset.

COLOR RAMPS FOR NUMERICAL DATA

Another option is to use color schemes based on the work of Cynthia Brewer, who has lead the way in defining effective color use in cartography. Helpfully, D3 includes colorbrewer.js and

colorbrewer.css for just this purpose. Each array in colorbrewer.js corresponds to one of Brewer's color schemes, designed for a set number of colors. For instance, the reds scale is:

```
Reds: {
3: ["#fee0d2","#fc9272","#de2d26"],
4: ["#fee5d9","#fcae91","#fb6a4a","#cb181d"],
5: ["#fee5d9","#fcae91","#fb6a4a","#de2d26","#a50f15"],
6: ["#fee5d9","#fcbba1","#fc9272","#fb6a4a","#de2d26","#a50f15"],
7: ["#fee5d9","#fcbba1","#fc9272","#fb6a4a","#ef3b2c","#cb181d","#99000d"],
8:
["#fff5f0","#fee0d2","#fcbba1","#fc9272","#fb6a4a","#ef3b2c","#cb181d","#9900
0d"],
9:
["#fff5f0","#fee0d2","#fcbba1","#fc9272","#fb6a4a","#ef3b2c","#cb181d","#a50f
15","#67000d"]
}
```

This provides high legibility discrete colors in the red spectrum for our elements. Again, we'll color our circles by region but this time, we'll color them by their magnitude using our buttonClick function. This require using the quantize scale that we saw earlier in Chapter 2:, because the colorbrewer scales, despite being discrete scales, are designed for quantitative data that has been separated into categories. In other words, they're built for numerical data, but numerical data that has been sorted into ranges, such as when you take all the ages of adults in a census and break them down into categories of 18-35, 36-50, 51-65, and 65+.

```
function buttonClick(datapoint) {#a
 var maxValue = d3.max(incomingData, function(el) {return
parseFloat(el[d])});
 var colorQuantize =
d3.scale.quantize().domain([0,maxValue]).range(colorbrewer.Reds[3]); #b
 var radiusScale = d3.scale.linear().domain([0,maxValue]).range([2,20]);
 d3.selectAll("g.overallG").select("circle").transition().duration(1000).styl
e("fill", function(p) {return colorQuantize(p[d])}).attr("r", function(p)
{return radiusScale(p[d])})
}
```

#a Our new buttonClick function will sort the circles in our visualization into three categories with colors associated with them.

#b The quantize scale sorts the numerical data into as many categories as there are in the range. Since colorbrewer.Reds[3] is an array of three values, this means the dataset will be sorted into three discrete categories, and each category will have a different shade of red assigned.

One of the conveniences of using colorbrewer.js dynamically paired to a quantizing scale like this is that if we adjust the number of colors, for instance from colorbewer.Reds[3] (Seen in Figure 3.15) to colorbrewer.Reds[5], that's all it takes to see the range of numerical data represented with 5 colors instead of 3.

```
function buttonClick(datapoint) {
 var maxValue = d3.max(incomingData, function(el) {return
parseFloat(el[d])});
 var colorQuantize =
d3.scale.quantize().domain([0,maxValue]).range(colorbrewer.Reds[3]);
 var radiusScale = d3.scale.linear().domain([0,maxValue]).range([2,20]);
```

```
 d3.selectAll("g.overallG").select("circle").transition().duration(1000).styl
e("fill", function(p) {return colorQuantize(p[d])}).attr("r", function(p)
{return radiusScale(p[d])})
}
```

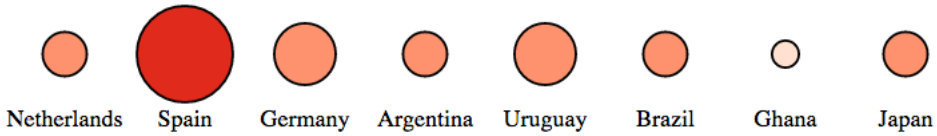


Figure 3.15 Automatic quantizing linked with the ColorBrewer 3-red scale produces distinct visual categories in the red family.

Color is important and it can behave strangely on the web. Colorblindness, for instance, is a key accessibility issue that is addressed by most of the colorbrewer scales. But even though color use and deployment is complex, smart people have been thinking about color for a while and D3 takes advantage of that.

3.3 Pre-generated Content

It's neither fun nor smart to create all your HTML elements using D3 syntax with nested selections and appending. More importantly, there's an entire ecosystem of tools out there for creating HTML, SVG, and static images that we'd be foolish to ignore just because we're using D3 for our general DOM manipulation and information visualization. Fortunately, loading externally generated resources--like images, HTML fragments and pre-generated SVG--and tying them into your graphical elements is straightforward and easy.

3.3.1 Images

In Chapter 1, I noted that gifs, despite their resurgent popularity, aren't tremendously useful for a rich interactive site. But that doesn't mean we'll get rid of images entirely. You'll find the addition of images to your data visualization can vastly improve them. In SVG, the image element is `<image>`, and its source is defined using the `xlink:href` attribute if it's located in your directory structure.

We already have files in our images directory that are PNGs of the respective flags of each national team. To add them to our data visualization, we simply select the `<g>` elements that have the team data already bound to them, and add an SVG image:

```
d3.selectAll("g.overallG").insert("image", "text")
.attr("xlink:href", function(d) {
return "images/" + d.team + ".png"
})
.attr("width", "45px").attr("height", "20px").attr("x", "-22")
.attr("y", "-10")
```

To make the images show up successfully, we used `insert()` instead of `append()` because that gives us the capacity to tell D3 to insert the images before the text elements. This keeps the labels from being drawn below the newly added images. Because each image name was the same as the team name of each data point, we can use an inline function to point to that value combined with strings for the directory and file extension. We also needed to define the height and width of the images because SVG images, by default, have no setting for height and width and will not display until these are set. We also need to manually center SVG images--here the x and y attributes are set to a negative value of one-half the respective height and width, which results in centering the images in their respective circles as seen in Figure 3.16.

Figure 3.16 Our graphical representations of each team now include a small PNG national flag, downloaded from Wikipedia and loaded using an SVG <image> element.

You can tie image resizing to the button events, but remember that raster images don't resize particularly well, and so you'll probably want to use them at fixed sizes.

Infoviz Terms: Chart Junk

Now that we're learning how to add images and icons to everything, let's remember that just because you can do something doesn't mean you should. When building information visualization, the key aesthetic principle should be to avoid cluttering your charts and interfaces with distracting and useless "chart junk" like unnecessary icons, decoration, or skeumorphic paneling. Remember, simplicity is force.

The term "chart junk" comes from Tufte, and in general refers to the kind of generic and useless clip art that typifies Powerpoint presentations. While icons and images are useful and powerful in many situations, and thus should not be avoided just to maintain an austere appearance, always make sure that your graphical representations of data are as uncluttered as you can make them.

3.3.2 HTML fragments

We already created some traditional DOM elements in this chapter using D3 data-binding for our buttons. If we want to, we can use the D3 pattern of selecting and appending to create very complex HTML objects such as forms and tables on-the-fly. But there are better authoring

tools for HTML and you'll likely be working with designers and other developers who want to use those tools and require that those HTML components be included in your application. For instance, let's build a modal dialogue box into which we can put the actual numbers associated with the teams. Say you want to see the stats on our teams—one of the best ways to do this is to build a dialogue box that pops up as you click on each team. A modal dialogue is just another way of referring to that "floating" area that typically only shows up when you click on an element. We can write just the HTML we need for the table itself in a separate file as we see in Listing 3.3.

Listing 3.3 modal.html

```
<table>
    <tr>
        <th>Statistics</th>
    </tr>
    <tr><td>Team Name</td><td class="data"></td></tr>
    <tr><td>Region</td><td class="data"></td></tr>
    <tr><td>Wins</td><td class="data"></td></tr>
    <tr><td>Losses</td><td class="data"></td></tr>
    <tr><td>Draws</td><td class="data"></td></tr>
    <tr><td>Points</td><td class="data"></td></tr>
    <tr><td>Goals For</td><td class="data"></td></tr>
    <tr><td>Goals Against</td><td class="data"></td></tr>
    <tr><td>Clean Sheets</td><td class="data"></td></tr>
    <tr><td>Yellow Cards</td><td class="data"></td></tr>
    <tr><td>Red Cards</td><td class="data"></td></tr>
</table>
```

And now we'll want to add some CSS rules for the table and the div that we want to put it in. As you see in Listing 3.4, we can use the position and z-index CSS styles because this is a traditional DOM element.

Listing 3.4 Update to d3ia.css

```
#modal {
  position:fixed;
  left:150px;
  top:20px;
  z-index:1;
  background: white;
  border: 1px black solid;
  box-shadow: 10px 10px 5px #888888;
}

tr {
  border: 1px gray solid;
}

td {
  font-size: 10px;
}
td.data {
  font-weight: 900;
```

```
}
```

Now that we have the table, all we need to do is to is add a click listener and associated function to populate this dialogue, as well as a function to create a div with ID “modal” into which we add the loaded HTML code using the .html() function.

```
d3.text("resources/modal.html", function(data)
{d3.select("body").append("div").attr("id", "modal").html(data)}); #a

.on("click", teamClick)

function teamClick(d) {
  d3.selectAll("td.data").data(d3.values(d)).html(function(p) {return p})
} #b
```

#a Create a new div with an id corresponding to one in our CSS and populate it with html content from modal.html
#b Select and update the td.data elements with the values of the team clicked

The results are immediately apparent when you reload the page. A div with the defined table in modal.html is created and, when you click on it, it populates the div with values from the data bound to the element you click on (Figure 3.17).

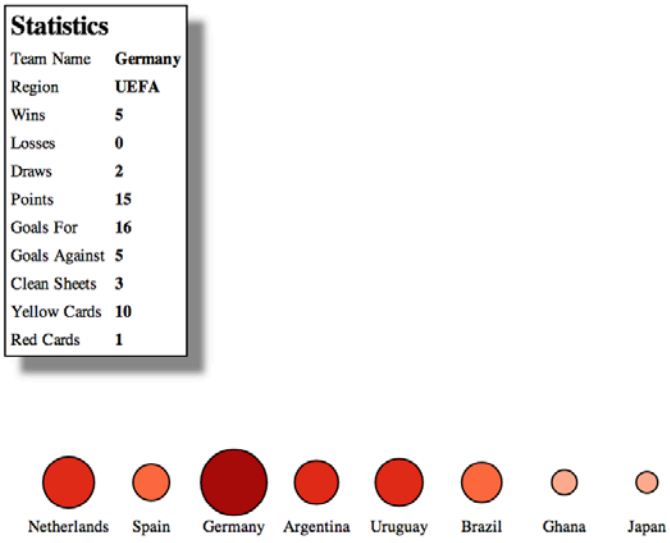


Figure 3.17 The modal dialogue is styled based on the defined style in CSS and is created by loading the HTML data from modal.html and adding it to the content of a newly created div.

We used `d3.text()` in this case because when working with HTML, it can be more convenient to load the raw HTML code like this and drop it into the `.html()` function of a

selected element that we've created as we've done here. If we use `d3.html()`, then we get HTML nodes that allow us to do more sophisticated manipulation, which we'll see now when as we work with pre-generated SVG.

3.3.3 Pre-generated SVG

SVG has been around since for a while, and there are, not surprisingly, robust tools for drawing SVG like Adobe Illustrator and the open source tool Inkscape. As such, you'll likely want pre-generated SVG for icons, interface elements, and other components of your work. If you're interested in icons, **The Noun Project** has an extensive repository of SVG icons, including the football in figure 3.18.

When you download an icon from the Noun Project, you get it in two forms: SVG and PNG. We've already learned how to reference images, and you can do the same with SVG by just pointing the xlink:href attribute of an <image> element at an SVG file. But loading SVG directly into the DOM gives you the capacity to manipulate it as you would any SVG elements that you created in the browser with D3.

Let's say you decided to replace your boring circles with balls, and you didn't want them to be static images because you wanted to be able to modify their color and shape like you would other SVG. In that case, you'd need to find a suitable ball icon and download it.

Figure 3.18 An icon for a football created by James Zamyslianskyj and available at http://thenounproject.com/term/football/1907/ from The Noun Project.

With the modal table we dealt with above, the assumption was that you're pulling in all the code found in modal.html, and so we can bring it in using d3.text() and drop the raw HTML as text into the .html() function of a selection. But in the case of SVG, especially SVG that you've downloaded, you often want to ignore the verbose settings in the document, which will include its own <svg> canvas as well as any <g> elements that have been not-so-helpfully added. Instead, you probably only want to deal with the graphical elements. In the case of our soccer ball, this means we want to only get the <path> elements. If we load the file using `d3.html()`, then the results are DOM nodes loaded into a document fragment that we can access and move around using D3 selection syntax. Using `d3.html()` is the same as using any of the other loading functions, where we designate the file to be loaded and the callback. You can see the results of this command in Figure 3.19.

```
d3.html("resources/icon_1907.svg", function(data) {console.log(data)})
```

Once you've loaded the SVG into the fragment, you can loop through the fragment to get all the paths very easily using the .empty() function of a selection, which checks to see if a selection still has any elements inside it and will eventually fire true once you've moved the paths out of the fragment into your main SVG. By including .empty() in a while statement, we can move all of the path elements out of the document fragment and load them directly onto the SVG canvas.

Figure 3.19 An SVG loaded using d3.html() that was created in Inkscape consists not only of the graphical <path> elements that make up the SVG but also much data that is often extraneous.

```
d3.html("resources/icon_1907.svg", loadSVG); #a
function loadSVG(svgData) {
 while(!d3.select(svgData).selectAll("path").empty()) {
   d3.select("svg").node().appendChild(d3.select(svgData).select("path").node
())
 }
```

```
 d3.selectAll("path").attr("transform", "translate(50,50)")
}
 #a The data variable will automatically be passed to loadSVG()
```

Notice how I've added a "transform" attribute so as to offset the paths so that they won't be clipped in the top right corner. Instead, we clearly see a football in the top corner of our <svg> canvas. Remember that document fragments are not a normal part of your DOM, so you don't have to worry about accidentally selecting the <svg> canvas in the document fragment, or any other elements.

But rather than a while loop like this, which can sometimes be necessary, typically the best and most efficient method is to use the .each() with your selection. Remember, .each() runs the same code on every element of a selection. In this case, the function we want to run is to select our <svg> canvas and append the path to that canvas.

```
function loadSVG(svgData) {
 d3.select(svgData).selectAll("path").each(function()
{d3.select("svg").node().appendChild(this)})
 d3.selectAll("path").attr("transform", "translate(50,50)")
}
```

We end up with a football floating in the top-left corner of our canvas as seen in Figure 3.20.

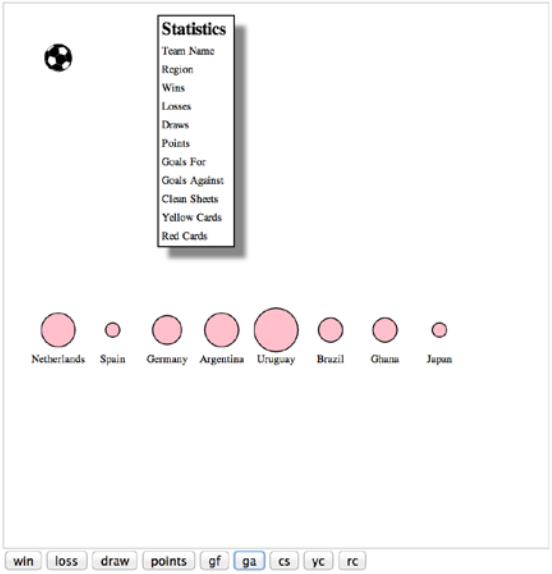


Figure 3.20 Along with the SVG elements and HTML elements we've created in our code, a hand-drawn football icon is loaded onto the <svg> canvas.

This is useful if you want to move individual nodes out of your loaded document fragment, but if you want to bind this to data, it's an added step that you can skip. You can't simply set the `.html()` of a <g> element to the text of your incoming elements like we did with the that we populated with the contents of modal.html. That's because SVG doesn't have a corresponding property to innerHTML and therefore the .html() function on a selection of SVG elements has no effect. Instead, we have to clone the paths and append them to each <g> element representing our teams.

```
d3.html("resources/icon_1907.svg", loadSVG);

function loadSVG(svgData) {
d3.selectAll("g").each(
function() {
var gParent = this;
d3.select(svgData).selectAll("path").each(
function() {
gParent.appendChild(this.cloneNode(true))
}
)
}
)
}
```

Selecting each <g> and then selecting each loaded <path> may seem like a backward way of doing things until you think about how `.cloneNode()` and `.appendChild()` work. What we have to do is take each <g> element and go through the <path> cloning process for every path in the loaded icon, which means we use nested `.each()` statements (one for each <g> element in our DOM and one for each <path> element in the icon). By setting gParent to the actual <g> node (the this variable) we can then append a cloned version of each path in order. The results are soccer balls for each team can be seen in Figure 3.21.

Figure 3.21 Each <g> element has its own set of paths cloned as child nodes, resulting in football icons overlaid on each element.

You could have easily done the same thing using the <image> syntax we used in the first example in this section, but with your SVG elements individually added to each And now you can style them in the same way you would any path element. You could use the national colors for each ball, but we'll settle for making them red, with the results seen in Figure 3.22.

```
d3.selectAll("path").style("fill", "darkred").style("stroke",
```

```
"black").style("stroke-width", "1px")
```

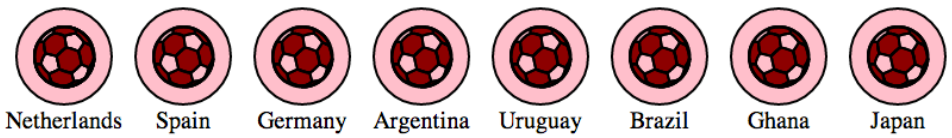


Figure 3.22 Football icons with a fill and stroke set by D3.

One drawback with this method is that the paths cannot take advantage of the D3 .insert() method's ability to place the elements below the labels or other visual elements. To get around this, you'll need to either append icons to <g> elements that have been placed in the proper order, or use the parentNode and appendChild function to move the paths around the DOM that we described earlier in this chapter.

The other drawback is that because these paths were added using cloneNode and not selection#append syntax, they have no data bound to them. You'll remember that we looked at rebinding data back in Chapter 1, and that if we select the <g> elements and then select the <path> element that this will re-bind data. However, we have numerous <path> elements under each <g> element, and selectAll does not rebind data. As a result, we have to take a more involved approach to bind the data from the parent <g> elements to the child <path> elements that have been loaded in this manner. The first thing we'll do is select all the <g> elements and then use .each() to select all the path elements under each <g>, at which point we will separately bind the data from the <g> to each <path> using .datum(). What's .datum()? Well, datum is the singular of data, so a piece of data is a datum, and the datum function is what you use when you're binding just one piece of data to an element. It's the equivalent of wrapping your variable in an array and binding it to .data(). After we've gone through this action, we can dust off our old scale from earlier and easily apply it to our new <path> elements. You can just run this code in the console to see the effects that should look like Figure 3.23.

```
d3.selectAll("g.overallG").each(function(d)
{d3.select(this).selectAll("path").datum(d)})

var tenColorScale = d3.scale.category10(["UEFA", "CONMEBOL", "CAF", "AFC"]);

d3.selectAll("path").style("fill", function(p) {return
tenColorScale(p.region)}).style("stroke", "black").style("stroke-width",
"2px");
```

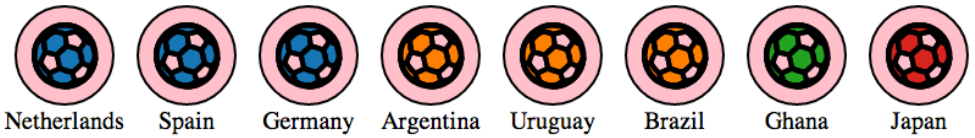


Figure 3.23 The paths now have the data from their parent element bound to them and respond accordingly when a discrete color scale based on region is applied.

Now you have data-driven icons. Use them wisely.

3.4 Summary

Throughout this chapter, we dealt with methods and functionality that typically is glossed over in D3 tutorials, especially the color functions and loading external content like external SVG and HTML. We also saw some of the more common D3 functionality, like animated transitions, especially tied to mouse events. Specifically, we covered:

- Project file structure and placing your D3 code in the context of traditional web development
- Some external libraries we might want to be aware of for D3 applications
- Using transitions and animation to highlight change and interaction
- Creating event listeners for mouse events on buttons and graphical elements
- Using color effectively for categories and numerical data and being aware of how color is treated in interpolations
- Accessing the DOM element itself from a selection
- Loading external resources, specifically images, HTML fragments and pre-generated SVG

D3 is a powerful library that can handle much of the needs for an interactive site, but you need to know when to rely on core HTML5 functionality or other libraries when it becomes more efficient. Moving forward, we're going to transition from the core functions of D3 and get into the higher level features of the library that allow you to build fully functional charts and chart components. We'll start in the next chapter by looking at generating SVG lines and areas from data as well as pre-formatted axis components for your charts. We'll also go into more detail about creating complex multi-part graphical objects from your data and use those techniques to produce complex examples of information visualization.

4

Chart Components

D3 provides an enormous library of examples of charts and Github is also packed with implementations. So, it's easy to simply format your data to match the existing data used in an implementation and, voila, you have a chart. Likewise, D3 includes layouts, which allow you to create complex data visualizations from a properly formatted dataset. But before you get started with default layouts, which will allow you to create basic charts like pie charts, as well as more exotic charts, you should first understand the basics of creating the elements that typically make up a chart. This chapter focuses on widely used pieces of charts created with D3, such as a labeled axis or a line. It also touches on the formatting, data modeling, or analytical methods most closely tied to creating charts.

Obviously, this isn't your first exposure to basic charts, since you've already created a scatterplot and bar chart in chapter 2. This chapter is going to introduce you to components and generators. A D3 component, like an axis, is a function for drawing all the graphical elements necessary for an axis. A generator, like d3.svg.line(), lets you draw the a straight or curved line across many points. The chapter starts by showing you how add axes to scatterplots as well as create line charts but by the end of it you'll create an exotic but still pretty simple chart: the streamgraph. By understanding how D3 generators and components work, you'll be able do more than just recreate the charts that other people have made and posted online (many of which they're just recreating from somewhere else).

A chart (and notice here that I don't use the term "graph" since that is a synonym for "network") is here used to refer to any flat layout of data in graphical manner. The datapoints, which as we know can be individual values or objects in arrays, and may contain categorical or quantitative or topological or unstructured data. In this chapter we're going to use several datasets to ultimately create the charts shown in Figure 4.1. While it might seem more useful to use a single dataset for the various charts, as the old saying goes, "Horses for courses", which is to say that different charts are more suitable to different kinds of datasets, as we'll see in this chapter.

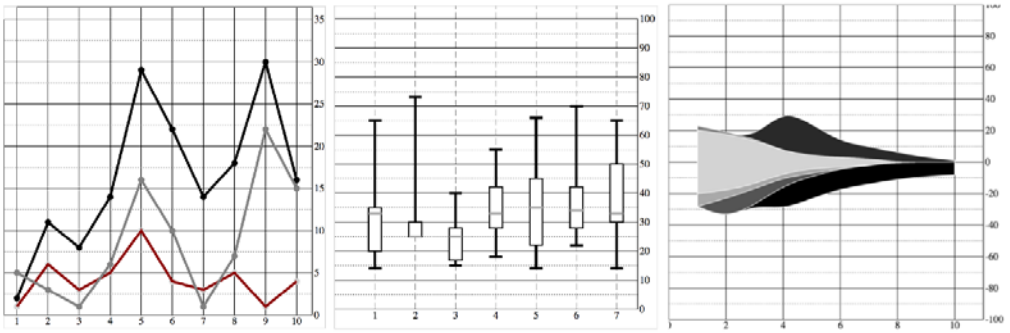


Figure 4.1 The charts we'll learn how to make in this chapter using D3 generators and components. From left to right, a line chart, a boxplot, and a streamgraph.

4.1 General Charting Principles

Any chart consists of several graphical elements that are drawn or derived from the dataset being represented. These graphical elements may be simple graphical primitives, like circles or rectangles, or more complex multi-part graphical objects like the boxplots we'll look at later in the chapter, or they may be supplemental pieces like axes and labels. While creating any of these elements in D3 uses the same general processes we've explored in the previous chapters, it's important to differentiate between the methods available in D3 to create graphics for charts.

We've already learned how to directly create simple and complex elements with data binding. We've also learned how to measure our data and transform it for display. Along with these two types of functions, there are three more broad categories into which D3 functionality can be placed: generators, components, and layouts, which you can see in Figure 4.2 along with a general overview of how they're used.

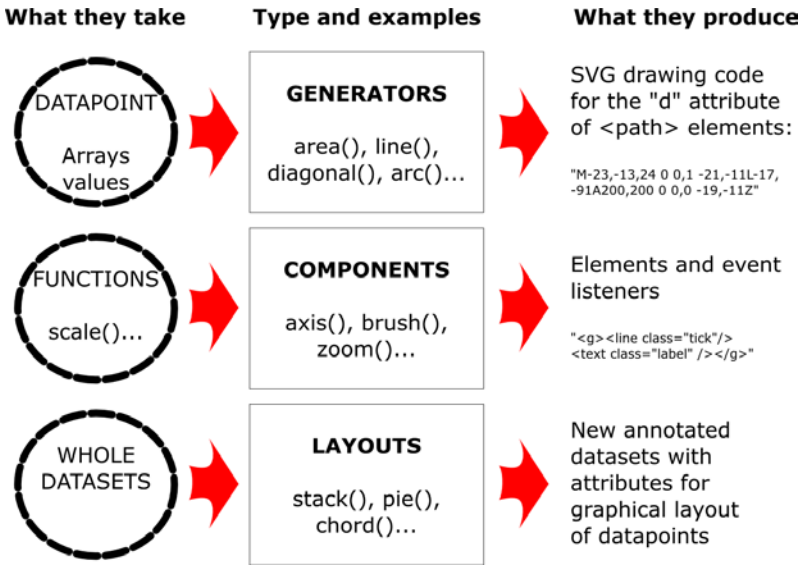


Figure 4.2 The three main types of functions found in D3 can be classified as generators, components and layouts. We'll see components and generators in detail in this chapter, and layouts in detail in the next chapter.

4.1.1 Generators

D3 generators consist of functions that take data and return the necessary SVG drawing code to create a graphical object based on that data. For instance, if you have an array of points and you want to draw a line from one point to another, or turn it into a polygon or an area, there are a few D3 functions that can help you with this process. These generators simplify the process of creating a complex SVG <path> by abstracting the process needed to write a <path> "d" attribute. In this chapter, we'll look at d3.svg.line and d3.svg.area, and in the next chapter we'll see d3.svg.arc, which is used to create the pie pieces of pie charts. Another generator that we'll see in Chapter 5 is d3.svg.diagonal, used for drawing curved connecting lines in dendrograms.

4.1.2 Components

In contrast to generators, which produce the actual "d" attribute string necessary for a <path> element, components create an entire set of graphical objects necessary for a particular chart component. The most commonly used D3 component, which we'll see in this chapter, is d3.svg.axis, which creates a bunch of <line>, <path>, <g>, and <text> elements that are needed for an axis based on the scale and settings you provide the function. Another component is d3.svg.brush, which we'll see later, and which creates all the graphical elements necessary for a brush selector.

4.1.3 Layouts

In contrast to generators and components, D3 layouts can be rather straight-forward, like the pie chart layout, or very complex, like a force-directed network layout. Layouts take in one or more arrays of data, and sometimes generators, and append attributes to the data necessary to draw that data in certain positions or sizes, either statically or dynamically. We'll see some of the more simple layouts in Chapter 5, and then focus on the force-directed network layout and other network layouts in Chapter 6.

4.2 Creating an Axis

Scatterplots are a simple and extremely effective charting method for displaying data that we've already worked with in Chapter 1 and 2. For most charts, the x position is a point in time and the y-position is magnitude, as when we placed our tweets in Chapter 2 along the x-axis according to when the tweets were made and along the y-axis according to their impact factor. In contrast, a scatterplot places a single symbol on a chart with its x and y position determined by quantitative data for that datapoint, for instance if we placed a tweet along its y-axis based on the number of favorites and the x-axis based on the number of retweets. Scatterplots are common in scientific discourse and have grown increasingly common in journalism and public discourse to present data such as the cost compared to the quality of health care.

4.2.1 Plotting Data

Scatterplots require multidimensional data. That just means that each datapoint needs to have more than 1 piece of data about it, and for a scatterplot it has to be numerical data. We need only have an array of data with 2 different numerical values for a scatterplot to work. For that, we'll use an array where every object represents a person for whom we know the number of friends they have and the amount of money they make. Now we can see if having more or less friends positively correlates to a high salary.

```
var scatterData = [{friends: 5, salary: 22000}, {friends: 3, salary: 18000},
{friends: 10, salary: 88000}, {friends: 0, salary: 180000}, {friends: 27,
salary: 56000}, {friends: 8, salary: 74000}]
```

If you think these salary numbers are too high or two low, just pretend they're in a foreign currency the exchange rate of which would make them more reasonable.

Representing this data graphically using circles is easy, we've already done it several times:

```
d3.select("svg").selectAll("circle").data(scatterData).enter().append("circle
").attr("r", 5).attr("cx", function(d,i) {return i * 10}).attr("cy",
function(d) {return d.friends})
```

By designating d.friends for the cy position, we get circles placed with their depth based on the value of the "friends" attribute. In other words, circles placed lower in the chart represent people in our dataset that have more friends. Circles are arranged from left to right using the old array position trick we learned earlier in Chapter 2. In Figure 4.3, we can see that it's not much of a scatterplot.

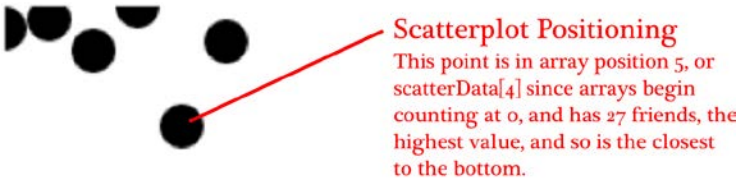


Figure 4.1 Circle positions indicate the number of friends and the array position of each datapoint.

As we've seen already, we need to build some scales to make this fit better on our SVG canvas:

```
xExtent = d3.extent(scatterData, function(d) {return d.salary});
yExtent = d3.extent(scatterData, function(d) {return d.friends});
xScale = d3.scale.linear().domain(xExtent).range([0,500]);
yScale = d3.scale.linear().domain(yExtent).range([0,500]);
d3.select("svg").selectAll("circle").data(scatterData).enter().append("circle
").attr("r", 5).attr("cx", function(d) {return xScale(d.salary)}).attr("cy",
function(d) {return yScale(d.friends)})
```

The result, in Figure 4.4, is a true scatterplot, with points representing people arranged by number of friends along the y-axis and amount of salary along the x-axis.

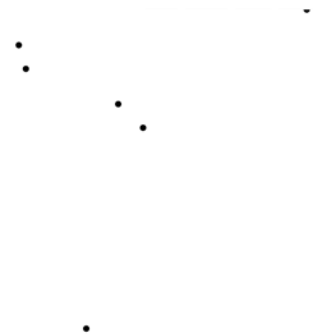


Figure 4.4 Any point closer to the bottom has more friends and any point closer to the right has a higher salary but that's not clear at all without labels, which we're going to make.

This chart, like most charts, is practically useless without some way of expressing to the reader what the position of the elements means. One way of accomplishing this is using well-formatted axis labels. While you could use the same method for binding data and appending

elements to create lines and ticks (which are just lines representing equidistant points along an axis) and labels for an axis, D3 provides `d3.svg.axis()` which can be used to create these elements based on the scales you've been using to display the data. Once you create an axis function, you define how you want your axis to appear and then you can draw it via a selection's `.call()` method from a selection on a `<g>` element where you want these graphical elements to be drawn.

```
yAxis = d3.svg.axis().scale(yScale).orient("right");
d3.select("svg").append("g").attr("id", "yAxisG").call(yAxis);
xAxis = d3.svg.axis().scale(xScale).orient("bottom");
d3.select("svg").append("g").attr("id", "xAxisG").call(xAxis);
```

Notice that the `.call() method of a selection` invokes a function with the selection which is active in the method chain, and is the equivalent of writing `xAxis(d3.select("svg").append("g").attr("id", "xAxisG"));`

Figure 4.5 shows a bit more legible result, with the x- and y-position of the circles denoted by labels in a pair of axes. The labels are derived from the scales that we've used to create each axis, and provide the kind of context necessary to interpret this chart.

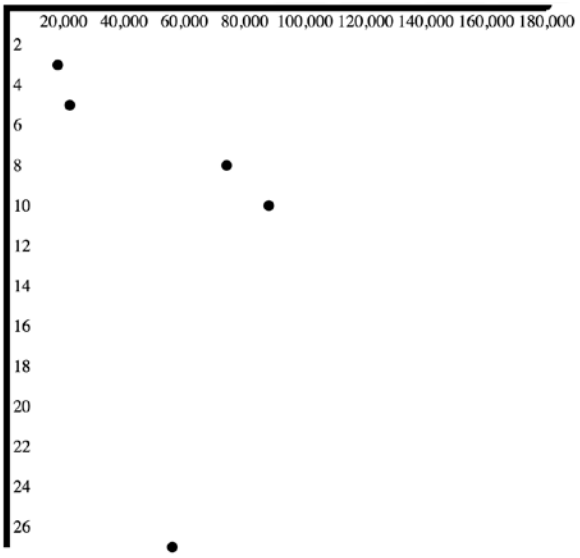


Figure 4.5 The same scatterplot from figure 4.2, but with a pair of labeled axes. The axis is drawn in such a way as to obscure one of the points.

The axis lines are thick enough to overlap with one of your scatterplot points because the domain of the axis being drawn is a path, and recall from Chapter 3 that paths are by default filled in black. We can adjust the display by setting the fill style of those two axis domain paths

to “none”. Doing so reveals that the ticks for the axes are not being drawn, because those elements do not have default “stroke” styles applied.

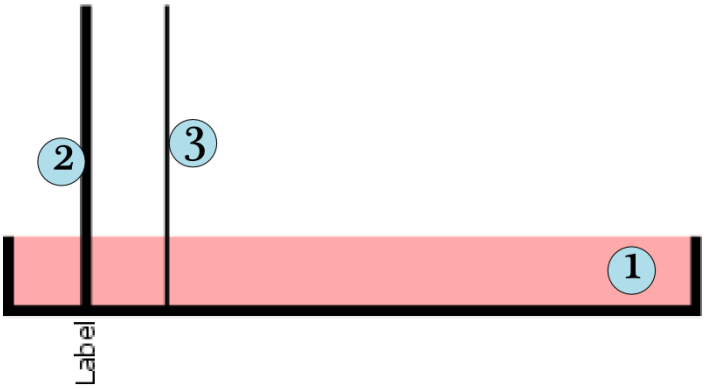


Figure 4.6 Elements of a created axis from d3.svg.axis are: 1) a <path.domain> with a size equal to the extant of the axis. 2) A <g.tick.major> that contains a <line> and a <text> for each major tick. 3) A <line.tick.minor> for each minor tick. Not shown, and invisible, is the <g> element that is called and into which these elements are created. In our example, region 1 is filled with black, and none of the lines have strokes, because that’s the default way that SVG draws <line> and <path> elements.

Figure 4.6 demonstrates why we don’t see any of our ticks and why we have thick black regions for our axis domains. To improve our axis, we need to style it properly.

4.2.2 Styling axes

Remember, these elements are standard SVG elements that are being created by the axis function, and do not have any more or less formatting than any other such elements would on first being created. This may seem counterintuitive, but SVG is meant to be paired with CSS so it’s better that elements don’t have any “helpful” styles assigned to them, or you’d have a hard time overwriting those styles with your CSS. For now, we can just set the domain path to fill:none and the lines to stroke: black using d3.select() and .style() to see what we’re missing:

```
d3.selectAll("path.domain").style("fill", "none").style("stroke", "black");
#a
d3.selectAll("line").style("stroke", "black"); #b
```

#a Notice we’re using selectAll because there are two of these paths, one for each axis we’ve called
#b You’ll want to be more specific in the future (“line.tick”), since it’s likely that whatever you’re working on will have more lines than just those used in your axes

The result is a bit more visually appealing:

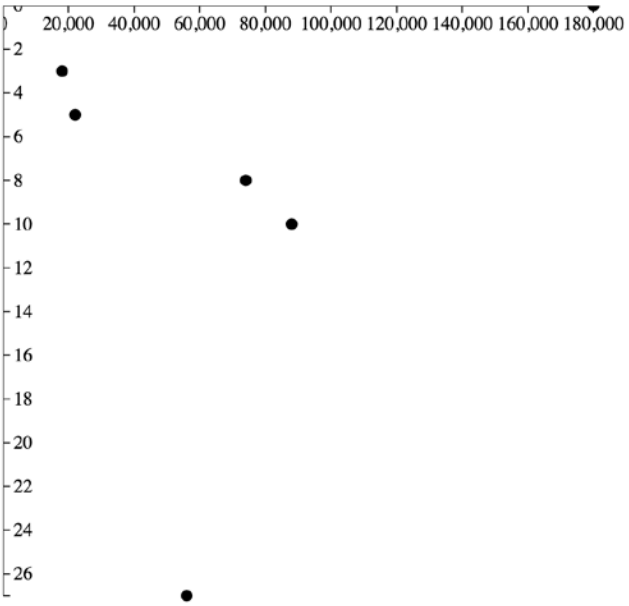


Figure 4.5 Changing the <path.domain> fill value to “none” as well as setting its and the <line> stroke values to “black” shows the ticks as well as the stroke of <path.domain>. It also reveals our hidden datapoint.

If you experiment with the axis code, you’ll notice that if you set the `.orient()` option of the y-axis to “left” or the `.orient()` option of the x-axis to “top” that they seem to not be drawn. This is because they’re drawn outside the canvas, like our earlier rectangles. To move your axes around, you need to adjust the `.attr(“translate”)` of their parent <g> elements, either when you draw them or later. Remember, this is why it’s important to assign an ID to your elements when you append them to the canvas. We can move the x-axis to the bottom of this drawing easily:

```
d3.selectAll("#xAxisD").attr("transform","translate(0,500)");
```

Here’s our updated code, which utilizes the `.tickSize()` function to change the ticks into lines, the `.tickSubdivide()` function to add minor unlabeled ticks to the yAxis, and manually sets the number of ticks using the ticks() function.

```
var scatterData = [{friends: 5, salary: 22000}, {friends: 3, salary: 18000},
{friends: 10, salary: 88000}, {friends: 0, salary: 180000}, {friends: 27,
salary: 56000}, {friends: 8, salary: 74000}];
xScale = d3.scale.linear().domain([0,180000]).range([0,500]); #a
yScale = d3.scale.linear().domain([0,27]).range([0,500]);

xAxis =
d3.svg.axis().scale(xScale).orient("bottom").tickSize(500).ticks(4);#b
d3.select("svg").append("g").attr("id", "xAxisG").call(xAxis);
```

```
yAxis =
d3.svg.axis().scale(yScale).orient("right").ticks(16).tickSize(500).tickSubdi
vide(true);
d3.select("svg").append("g").attr("id", "yAxisG").call(yAxis); #c

d3.select("svg").selectAll("circle").data(scatterData).enter().append("circle
").attr("r", 5).attr("cx", function(d) {return xScale(d.salary)}).attr("cy",
function(d) {return yScale(d.friends)})
```

#a Create a pair of scales to map the values in our dataset to the canvas
#b Here we use method chaining to create an axis and explicitly set its orientation, tick size and number of ticks
#c Append a <g> element to the canvas and call the axis from that <g> to create the necessary graphics for the axis

The results of the use of all these functions is, actually pretty uninspiring:

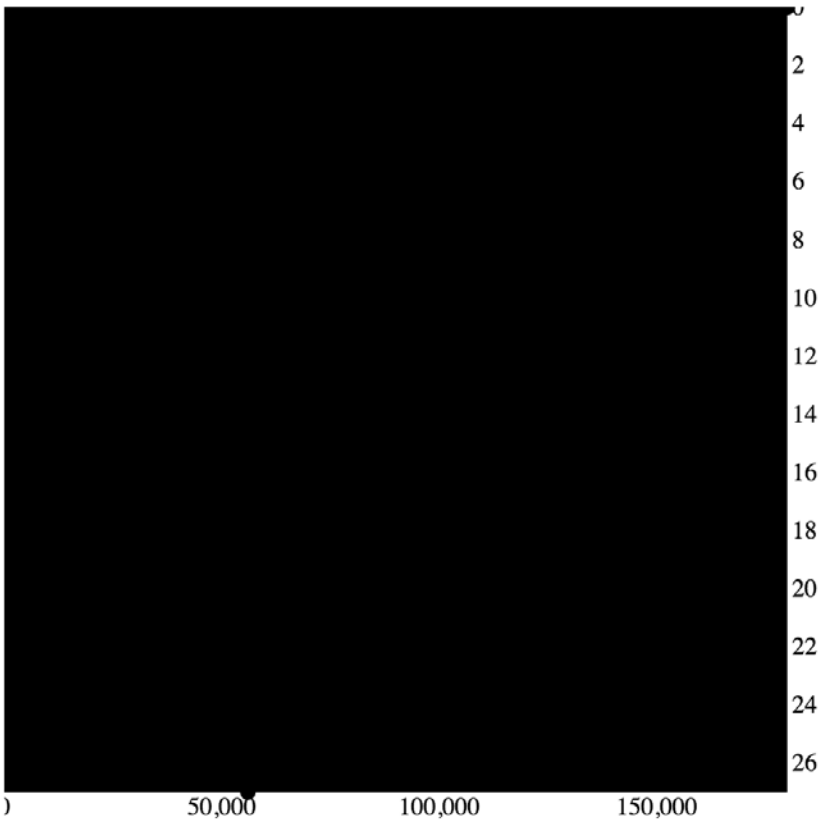


Figure 4.7 Setting axis ticks to the size of your canvas also sets <path.domain> to the size of your canvas. Since paths are, by default, filled with black, then the result is illegible.

Let's examine the elements created by the axis code and shown in Figure 4.7 as a giant black square. Inside of the <g> element that we created with the id of "xAxisG" are <g> elements that each have a line and some text:

```
<g class="tick major" transform="translate(0,0)" style="opacity: 1;">
<line x2="6" y2="0"></line>
<text x="9" y="0" dy=".32em" style="text-anchor: start;">0</text>
</g>
```

Notice that the <g> element has been created with classes, so you can style the child elements (your line and your label) using CSS or select them with D3. This is going to be necessary if we want our axes to be displayed properly, with lines corresponding to the labeled points. Why? Because along with lines and labels, the axis code has drawn the <path.domain> to cover the entire region contained by the axis elements. This domain element needs to be set to "fill: none" or you'll end up with a big black square. You'll also see examples where the tick lines are drawn with negative lengths to create a slightly different visual style. To make your axis make sense, you could continue to apply inline styles by using d3.select to modify the styles of the necessary elements, but instead you should use CSS because it's easier to maintain and doesn't require you to write styles on-the-fly in JavaScript. Here's a short CSS stylesheet that corresponds to the elements created by the axis function:

Listing 4.1 ch4stylesheet.css

```
<style>
line { #a
  shape-rendering: crispEdges;
  stroke: #000;
}

line.minor  {
  stroke: #777;
  stroke-dasharray: 2,2;
}

path.domain {
  fill: none;
  stroke: black;
}
</style>
```

#a This will apply to all your lines, which includes the major lines that you would otherwise need to reference with 'g.major > line'

With this in place, we get something a bit more legible:

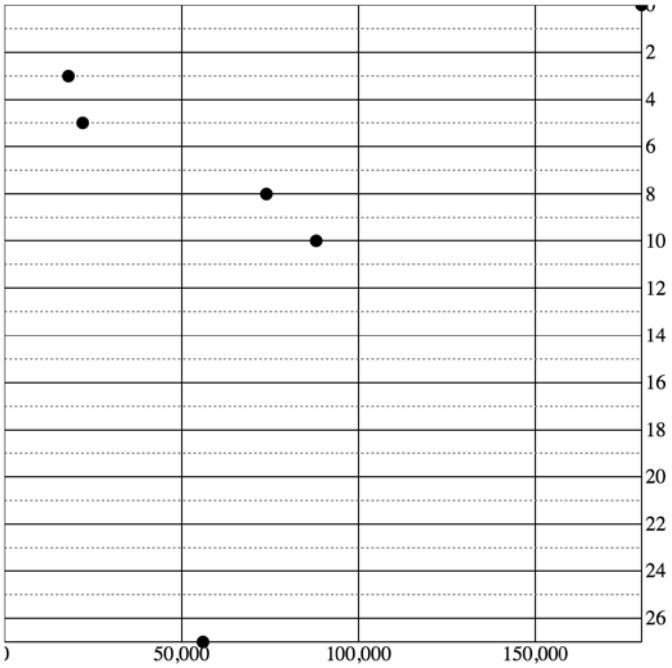


Figure 4.8 With <path.domain> fill set to “none” and CSS settings also corresponding to the minor and major tick <line> elements, we can draw a rather attractive grid based on our two axes.

Take a look at the elements created by the axis() function in Figure 4.8 and see in Figure 4.9 how the CSS classes vary depending on whether they are minor or major ticks:

```
  <line class="tick minor" x2="500" y2="0"
  transform="translate(0,425.9259259259259)"
  style="opacity: 1;"></line>
  <line class="tick minor" x2="500" y2="0"
  transform="translate(0,462.96296296296293)"
  style="opacity: 1;"></line>
▶<g class="tick major" transform=
"translate(0,0)" style="opacity: 1;">…</g>
▼<g class="tick major" transform=
"translate(0,37.03703703703704)" style="opacity:
1;">
    <line x2="500" y2="0"></line>
    <text x="503" y="0" dy=".32em" style="text-
    anchor: start;">2</text>
  </g>
▶<g class="tick major" transform=
"translate(0,74.07407407407408)" style="opacity:
1;">…</g>
```

Figure 4.9 The DOM shows how minor tick <line.minor.tick> elements are appended directly to the <g> that calls the axis, while major tick <line> elements are appended along with a <text> element for the label to one of a set of <g.tick.major> elements corresponding to the number of major ticks

As you create more complex information visualization, you'll get used to the need to create your own elements with classes referenced by your stylesheet as well as understand where D3 components create elements in the DOM and how they are classed so that you can style them properly.

4.3 Complex Graphical Objects

Using simple circles or rectangles for your data won't work with some datasets, especially if an important aspect of the data you're examining has to do with distribution like user demographics or statistical data. Often, the distribution of data gets lost in information visualization, or is only noted with a reference to standard deviation or other first-year statistics terms that indicate the average doesn't tell the whole story. One particularly useful way of representing data that has a distribution (such as a fluxuating stock price) is the use of a boxplot in place of a traditional scatterplot, which uses a complex graphic that encodes distribution in its shape. The box in a boxplot typically looks like the one seen in Figure 4.10. This uses quartiles that have been pre-processed, but you could easily use d3.scale.quartile() to create your own values from your own dataset.

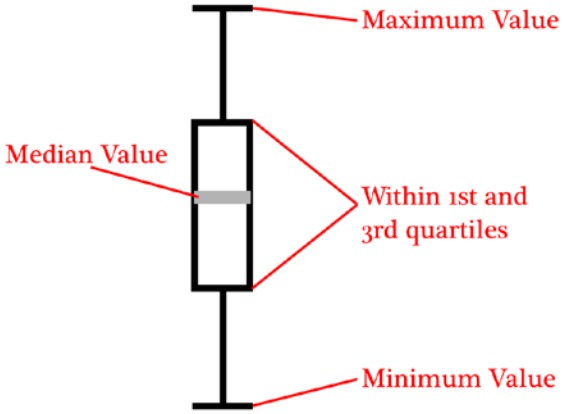


Figure 4.10 A box from a boxplot consists of five pieces of information encoded in a single shape. 1) The maximum value. 2) The high value of some distribution, such as the third quartile. 3) The median or mean value. 4) The corresponding low value of the distribution, such as the first quartile. 5) The minimum value.

Take a moment to examine the amount of data that's encoded in this simple graphic in Figure 4.10. The median value is represented as a gray line. The rectangle shows the amount of whatever you're measuring that falls within a set range that represents the majority of the data. The two lines above and below the rectangle indicate the minimum and maximum values. Everything except the information in the gray line is lost when you only map the average or median value at a datapoint.

To build a reasonable boxplot, you'll need a set of data with some interesting variation in those areas. Let's assume you want to plot the number of registered visitors coming to your website by day of the week so that you can compare your stats week to week (or so that you can present this info to your boss or for whatever interesting reason you have). You have the data for the age of the visitors (based on their registration details) and derived the quartiles from that. Maybe you used Excel, or maybe you used Python, or maybe you used d3.scale.quartile(), or maybe it was just part of a dataset you downloaded. As you work with data, you'll be exposed to common statistical summaries like this and expected to represent it as part of your charts, so don't be too intimidated by it. We'll use a csv format for the information.

Listing 4.1 shows our dataset with the number of registered users that visit the site each day, and the quartiles of their ages.

Listing 4.1 boxplots.csv

```
day,min,max,median,q1,q3,number
1,14,65,33,20,35,22
2,25,73,25,25,30,170
3,15,40,25,17,28,185
4,18,55,33,28,42,135
5,14,66,35,22,45,150
```

```
6,22,70,34,28,42,170
7,14,65,33,30,50,28
```

When we map the median age as a scatterplot as we do in Figure 4.11 and it looks like there's not too much variation in your user base throughout the week. We do that by drawing scatterplot points for each day at the median age of the visitor for that day. We'll also invert the y-axis so that it makes a bit more sense.

Listing 4.x Basic scatterplot of average age

```
    d3.csv("boxplot.csv", scatterplot)

function scatterplot(data) {
    xScale = d3.scale.linear().domain([1,8]).range([20,470]);
    yScale = d3.scale.linear().domain([0,100]).range([480,20]); #a

yAxis = d3.svg.axis()
.scale(yScale)
.orient("right")
.ticks(8)
.tickSize(-470)
.tickSubdivide(true);

d3.select("svg").append("g")
.attr("transform", "translate(470,0)") #b
.attr("id", "yAxisG")
.call(yAxis);

xAxis = d3.svg.axis()
.scale(xScale)
.orient("bottom")
.tickSize(-470)
.tickValues([1,2,3,4,5,6,7]); #c

d3.select("svg").append("g")
.attr("transform", "translate(0,480)")
.attr("id", "xAxisG")
.call(xAxis);

d3.select("svg").selectAll("circle.median")
.data(data)
.enter()
.append("circle")
.attr("class", "tweets")
.attr("r", 5)
.attr("cx", function(d) {return xScale(d.day)})
.attr("cy", function(d) {return yScale(d.median)})
.style("fill", "darkgray");
  }
```

#a Notice our scale is inverted, so higher values will be drawn higher on up and lower values toward the bottom
#b Offset the <g> containing the axis
#c Specify the exact tick values to correspond with the numbered days of the week

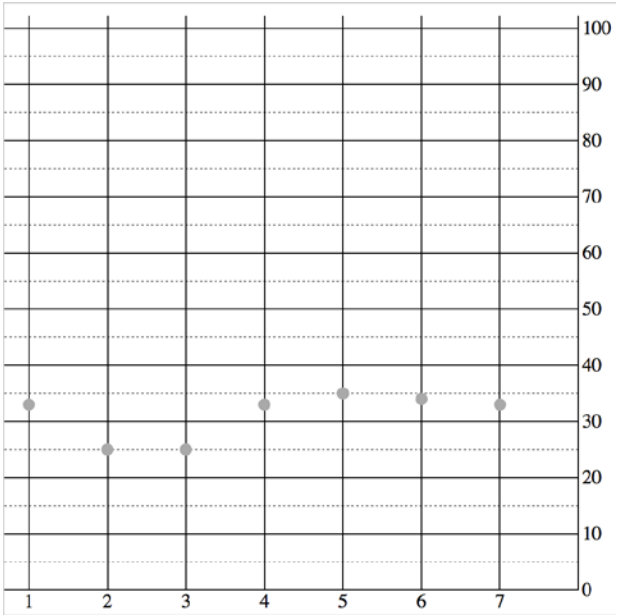


Figure 4.11 The median age of visitors (y-axis) by day of the week (x-axis) as represented by a scatterplot shows a slight dip in age on the second and third day.

But to get a better view of this data, you'll need to create a boxplot. Building a boxplot is very similar to building a scatterplot, but instead of appending circles for each point of data, you're going to append a <g> element. It's a good rule to always use <g> elements for your charts, because they allow you to apply labels or other important information to your graphical representations. But that means you're going to need to get used to the "transform" attribute, which is how <g> elements are positioned on the canvas. Remember that elements appended to a <g> base their coordinates off of the coordinates of their parent, so when applying "x" and "y" attributes to child elements you need to set them relative to the parent <g>.

Rather than selecting all the <g> elements and appending child elements one at a time, as we've done in earlier chapters, we're going to use the .each() function of a selection, which allows us to perform the same code on each element in a selection, to create the new elements. Like any D3 selection function, .each() allows you to access the bound data, array position, and actual DOM element. We achieved the same functionality earlier on by using selectAll to select the <g> elements and directly append <circle> and <text> elements like we did in Chapter 1. That's a very clean method and the only reasons to use .each() to add child elements is because you prefer the syntax, or you plan on doing complex operations involving each data element, or you want to add conditional tests to change whether or what child elements you're appending. You can see it in action in Listing 4.x, which takes advantage of the

scales we created in the first example and draws rectangles on top of the circles we've already drawn.

Listing 4.x Initial boxplot drawing code

```
d3.select("svg").selectAll("g.box")
.data(data)
.enter()
.append("g")
.attr("class", "box")
.attr("transform", function(d) {return "translate(" + xScale(d.day) +"," +
yScale(d.median) + ")"})
.each(function(d,i) {
d3.select(this) #a
.append("rect")
.attr("width", 20)
.attr("height", yScale(d.q3) – yScale(d.q1)); #b
})
```

#a Because we're inside the .each(), we can select(this) to append new child elements.
#b The d and i variables are declared in the .each() anonymous function, so each time we access it, we get the data bound to the original element.

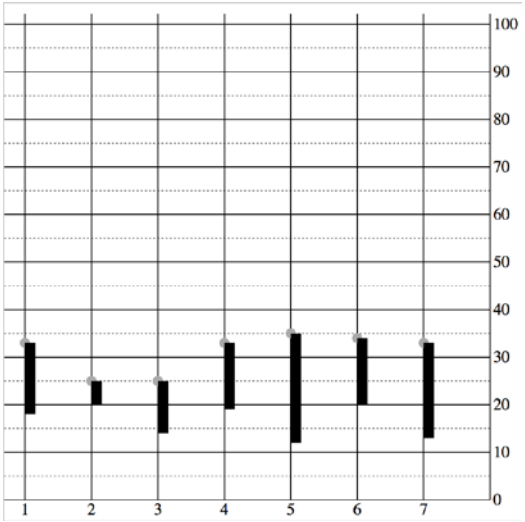


Figure 4.12 <rect> elements representing the scaled value the range of the first and third quartiles of visitor age are placed on top of a gray <circle> in each <g> element which is placed on the chart at the median age. The rectangles are drawn, as per SVG convention, from the <g> down and to the right.

The new rectangles which we've drawn in Figure 4.12, which indicate the distribution of visitor ages, is not only offset to the right, but also showing the wrong values. Day 7, for instance, should range in value from 30 to 50, and instead is shown as ranging from 13 to 32.

We know it's doing that because that's the way SVG draws rectangles. We have to update our code a bit to make it accurately reflect the distribution of visitor ages:

```
…
.each(function(d,i) {
d3.select(this)
.append("rect")
.attr("width", 20)
.attr("x", -10) #a
.attr("y", yScale(d.q3) – yScale(d.median)) #b
.attr("height", yScale(d.q1) - yScale(d.q3))
.style("fill", "white")
.style("stroke", "black")
})
```

#a Setting a negative offset of half the width centers a rectangle horizontally.
#b The height of the rectangle is equal to the difference between its q1 and q3 values, which means we need to offset the rectangle by the difference between the middle of the rectangle—the median—and the high end of the distribution—q3

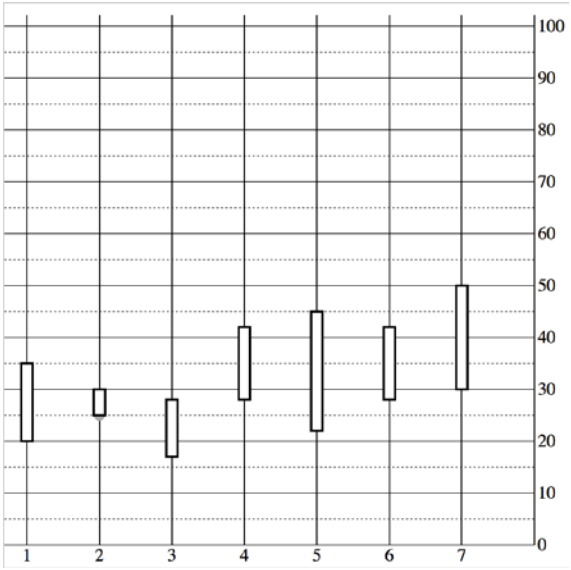


Figure 4.13 <rect> elements are now properly placed so that their top and bottom correspond with the visitor age between the first and third quartiles of visitors for each day. The circles are completely covered, except for the second rectangle where the first quartile value is the same as the median age, and so we can see half the gray circle peeking out from underneath it.

We'll use the same technique we used to create the chart in Figure 4.13 to add the remaining elements of the boxplot (described in detail below in Figure 4.14) by including several append functions within the .each() function. All of them select the parent <g> element created during the data-binding process and append the shapes necessary to build a boxplot.

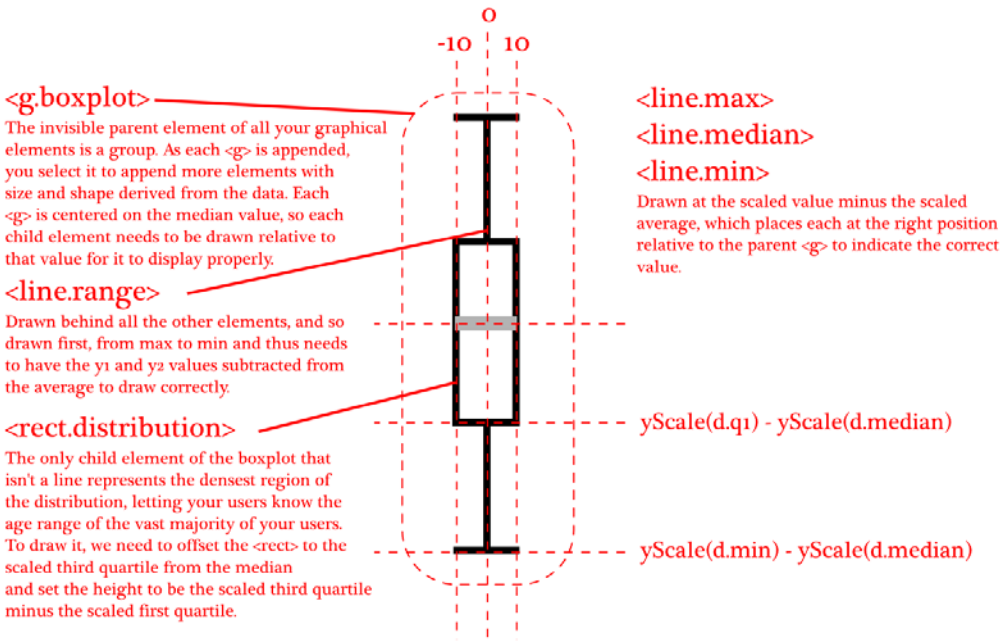


Figure 4.14 How a boxplot can be drawn in D3. Pay particular attention to the relative positioning necessary to draw child elements of a <g>. The 0 positions for all elements are where the parent <g> has been placed, so that <line.max>, <rect.distribution> and <line.range> all need drawn with an offset placing their top-left corner above this center, while <line.min> is drawn below the center and <line.median> has a 0 y-value since our center is the median value.

Listing 4.2 The .each() function of the appended boxplot <g> drawing five child elements.

```
…
.each(function(d,i) {
d3.select(this)
.append("line")
.attr("class", "range")
.attr("x1", 0)
.attr("x2", 0)
.attr("y1", yScale(d.max) - yScale(d.median)) #a
.attr("y2", yScale(d.min) - yScale(d.median))
.style("stroke", "black")
.style("stroke-width", "4px");

d3.select(this)
.append("line")
.attr("class", "max")
.attr("x1", -10)
.attr("x2", 10)
.attr("y1", yScale(d.max) - yScale(d.median)) #b
.attr("y2", yScale(d.max) - yScale(d.median))
.style("stroke", "black")
```

```
    .style("stroke-width", "4px");

    d3.select(this)
    .append("line")
    .attr("class", "min")
    .attr("x1", -10)
    .attr("x2", 10)
    .attr("y1", yScale(d.min) - yScale(d.median)) #c
    .attr("y2", yScale(d.min) - yScale(d.median))
    .style("stroke", "black")
    .style("stroke-width", "4px");
    d3.select(this)
    .append("rect")
    .attr("class", "range")
    .attr("width", 20)
    .attr("x", -10)
    .attr("y", yScale(d.q3) - yScale(d.median)) #d
    .attr("height", yScale(d.q1) - yScale(d.q3))
    .style("fill", "white")
    .style("stroke", "black")
    .style("stroke-width", "2px");

    d3.select(this)
    .append("line") #e
    .attr("x1", -10)
    .attr("x2", 10)
    .attr("y1", 0)
    .attr("y2", 0)
    .style("stroke", "darkgray")
    .style("stroke-width", "4px");
    })
```

#a This is how we draw the line from the min to the max value
#b The top bar of the min-max line
#c The bottom bar of the min-max line
#d The offset so that the rectangle is centered on the median value
#e The median line doesn't need to be moved, since the parent <g> is centered on the median value

Along with this new code to create the shapes necessary to build a boxplot, we should add an x-axis to remind us which day each box is associated with. This will take advantage of the explicit .tickValues() function we saw earlier. It will also use negative tickSize() and the corresponding offset of the <g> that we use to call the axis function.

```
        xAxis = d3.svg.axis().scale(xScale).orient("bottom")
    .tickSize(-470) #a
    .tickValues([1,2,3,4,5,6,7]); #b
        d3.select("svg").append("g").attr("transform",
    "translate(0,470)").attr("id", "xAxisG").call(xAxis); #c
        d3.select("#xAxisG > path.domain").style("display", "none") #d
```

#a A negative tickSize will draw the lines above the axis, but we need to make sure to offset the axis by the same value
#b Setting specific tickValues forces the axis to only show the corresponding values, which is useful when you want to override the automatic ticks created by the axis. Sometimes, the specified tick values are
#c Here we offset the axis to correspond with our negative tickSize
#d We can hide this, since it has extra ticks on the ends that will distract our readers.

The end result of all of this is a chart where each of our datapoints is represented not by a single circle but by a multipart graphical element designed to emphasize distribution.

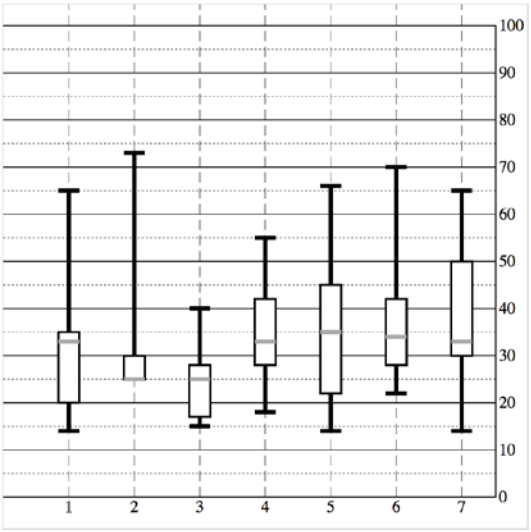


Figure 4.15 Our final boxplot chart. Each day now shows not only the median age of visitors but the range of visiting ages, allowing for a more extensive examination of the demographics of site visitorship.

Each boxplot in Figure 4.15 encodes not just the median age of visitors for that day, but the minimum, maximum and distribution of the age of the majority of visitors. This expresses in much more detail the demographics of visitorship clearly and cleanly. It doesn't manage to include the number of visitors, but you could encode that with color, or make it available on a click of each boxplot or make the width of the boxplot correspond to the number of visitors.

We looked at boxplots because a boxplot allows us to explore the creation of multipart objects while using relatively simple lines and rectangles. But what's the value of a visualization like this that shows distribution? It encodes a graphical summary of the data, answering quickly the questions you'd ask about visitor age for the site on Wednesday, such as, "Most of them were between the age of 18 and 28. The oldest was 40. The youngest was 15. The median age was 25." It also allows you to quickly perform visual queries, checking to see if the median age of one day was within the majority of visitor ages of another day.

We'll stop exploring boxplots, and begin to take a look at a different kind of complex graphical object: an interpolated line.

4.4 Line Charts and Interpolations

Drawing connections between points is how we create line charts. A line connecting points, as well as shaded regions inside or outside the area constrained by the line, invest narrative into

the representation of data. While technically a static data visualization, any line chart is a representation of change, typically over time.

We'll start with a new dataset that better represents change over time. Let's imagine you have a Twitter account and you've been tracking the number of tweets, favorites and retweets to try to see what times you have the most response to your social media. While you'll ultimately deal with this kind of data as JSON, you'll probably want to start with a comma-delimited file, since it's the most efficient for this kind of data:

Listing 4.3 tweetdata.csv

```
day,tweets,retweets,favorites
1,1,2,5
2,6,11,3
3,3,0,1
4,5,2,6
5,10,29,16
6,4,22,10
7,3,14,1
8,5,7,7
9,1,35,22
10,4,16,15
```

First we pull this CSV in using d3.csv() as we did in Chapter 2, and then we create circles for each datapoint and do this for each variation on the data, with the .day attribute determining x position and the other datapoint determining y position. We create the usual x and y scales to draw the shapes within the confines of our canvas. We also have a couple axes to frame our results. Notice that we've tried to differentiate between the three data types by coloring them differently:

Listing 4.4 Callback function to draw a simple scatterplot from tweetdata

```
d3.csv("tweetdata.csv", lineChart)

function lineChart(data) {

xScale = d3.scale.linear().domain([1,10.5]).range([20,480]); #a
yScale = d3.scale.linear().domain([0,35]).range([480,20]);

xAxis = d3.svg.axis()
.scale(xScale)
.orient("bottom")
.tickSize(480)
.tickValues([1,2,3,4,5,6,7,8,9,10]);#b

d3.select("svg").append("g").attr("id", "xAxisG").call(xAxis);

yAxis = d3.svg.axis()
.scale(yScale)
.orient("right")
.ticks(10)
.tickSize(480)
.tickSubdivide(true);
```

```
d3.select("svg").append("g").attr("id", "yAxisG").call(yAxis);

d3.select("svg").selectAll("circle.tweets")
.data(data)
.enter()
.append("circle")
.attr("class", "tweets")
.attr("r", 5)
.attr("cx", function(d) {return xScale(d.day)})
.attr("cy", function(d) {return yScale(d.tweets)})
.style("fill", "black") #c

d3.select("svg").selectAll("circle.retweets")
.data(data)
.enter()
.append("circle")
.attr("class", "retweets")
.attr("r", 5)
.attr("cx", function(d) {return xScale(d.day)})
.attr("cy", function(d) {return yScale(d.retweets)})
.style("fill", "lightgray")

d3.select("svg").selectAll("circle.favorites")
.data(data)
.enter()
.append("circle")
.attr("class", "favorites")
.attr("r", 5)
.attr("cx", function(d) {return xScale(d.day)})
.attr("cy", function(d) {return yScale(d.favorites)})
.style("fill", "gray")
}
```

#a Our scales, as usual, have margins built in
#b Fix the ticks of the x axis to correspond to the days
#c Each of these uses the same dataset, but bases the y-position on tweets, retweets and favorites values, respectively

The graphical results of the code in Figure 4.16, which take advantage of the CSS rules we defined earlier, are not very easily interpreted:

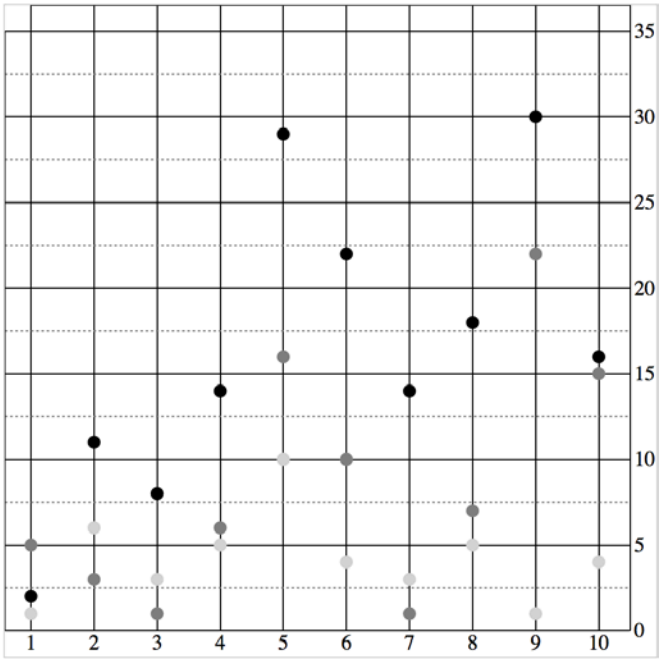


Figure 4.16 A scatterplot showing the datapoints for ten days of activity on Twitter, with the number of tweets in light gray, the number of retweets in dark gray, and the number of favorites in black.

By drawing a line that intersected each point of the same category, we could compare the number of tweets and retweets and favorites. We can start by drawing a line for tweets by using `d3.svg.line()`. This line generator expects an array of points as data, for which you'll need to tell the generator what values constitute the x and y coordinates for each point. By default, this generator expects a two-part array where the first part is the x value and the second part is the y value. We can't use that, because our x value is based on the day of the activity and our y value is based on the amount of activity.

This means that the .x() accessor function of the line generator needs to point at the scaled day value, while the .y() accessor function needs to point to the scaled value of the appropriate activity. The line function itself will take the entire dataset that we've loaded from tweetdata, and will return the SVG drawing code necessary for a line between the points in that dataset. To generate three lines, we'll use the dataset three times, with a slightly different generator for each. That means we need to not only write the generator function and define how it accesses the data it will be using to draw the line, we also need to append a <path> to our canvas and set its "d" attribute to equal the generator function we defined:

Listing 4.6 New line generator code inside the callback function

```
tweetLine = d3.svg.line()
```

```
        .x(function(d) { #a
        return xScale(d.day)
    })
        .y(function(d) { #b
        return yScale(d.tweets)
    })

    d3.select("svg")
        .append("path")
        .attr("d", intLine(data)) #c
        .attr("fill", "none")
        .attr("stroke", "darkred")
        .attr("stroke-width", 2)
    })
```

#a You need to define an accessor for data like ours, in this case we take the day attribute and pass it to xScale first.
#b This accessor does the same for the number of tweets
#c The appended path is drawn according to the generator with the loaded tweetdata passed to it.

We draw the line above the circles we've already drawn, and can see the line generator in action in Figure 4.17.

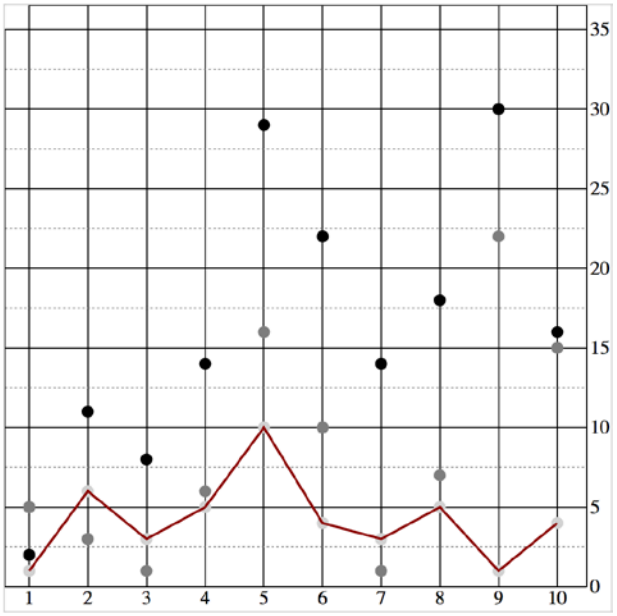


Figure 4.17 The line generator takes the entire dataset and draws a line where every point of the line has the x,y position on the canvas based on its accessor. In this case, that means each point on the line corresponds to the day,tweets scaled to fit the x and y scales we created to display the data on the canvas.

If we build a line constructor for each data type in our set and call each with its own path, then we can see the variation over time for each of our data points:

Listing 4.7 Line generators for each tweetdata

```
tweetLine = d3.svg.line() #a
    .x(function(d) {
    return xScale(d.day)
})
    .y(function(d) {
    return yScale(d.tweets)
})

retweetLine = d3.svg.line()
    .x(function(d) {
    return xScale(d.day)
})
    .y(function(d) {
    return yScale(d.retweets) #b
})

favLine = d3.svg.line()
    .x(function(d) {
    return xScale(d.day)
})
    .y(function(d) {
    return yScale(d.favorites)
})

d3.select("svg")
    .append("path") #c
    .attr("d", tweetLine(data))
    .attr("fill", "none")
    .attr("stroke", "darkred")
    .attr("stroke-width", 2)

    d3.select("svg")
    .append("path")
    .attr("d", retweetLine(data))
    .attr("fill", "none")
    .attr("stroke", "gray")
    .attr("stroke-width", 3)

    d3.select("svg")
    .append("path")
    .attr("d", favLine(data))
    .attr("fill", "none")
    .attr("stroke", "black")
    .attr("stroke-width", 2)
})
```

#a A more efficient way to do this would be to define one line generator, and then modify the .y() accessor on the fly as you call it for each line. But it's easier to see the functionality this way.
#b Notice how only the y accessor is different between each line generator
#c Each line generator needs to be called by a corresponding new <path> element

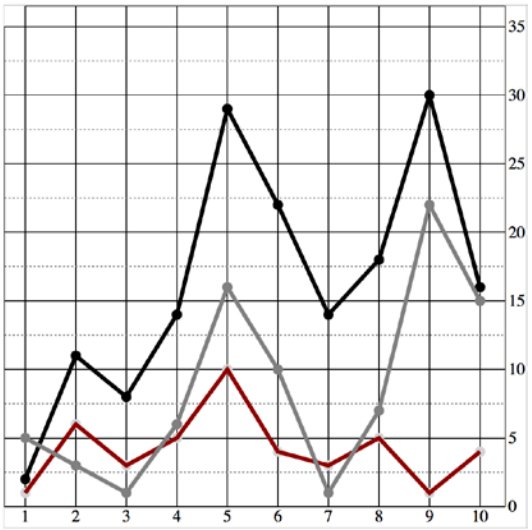


Figure 4.18 The dataset is first used to draw a set of circles, creating the simple scatterplot from the beginning of this section, and then used three more times to draw each line.

D3 provides a number of interpolation methods with which to draw these lines, so that they can more accurately represent the data. In cases like tweetdata, where we have discrete points that represent data accurately and not samples, then the default "linear" method seen in Figure 4.18 is appropriate. But in other cases, a different interpolation method for the lines like the ones seen in Figure 4.19 may be appropriate. Here's the same data but with the d3.svg.line() generator using different interpolation methods.

```
tweetLine.interpolate("basis"); #a
retweetLine.interpolate("step");
favLine.interpolate("cardinal");
```

#a You can add this code right after you create your line generators and before you call them to change the interpolate method or you could set .interpolate() as you're defining the generator.

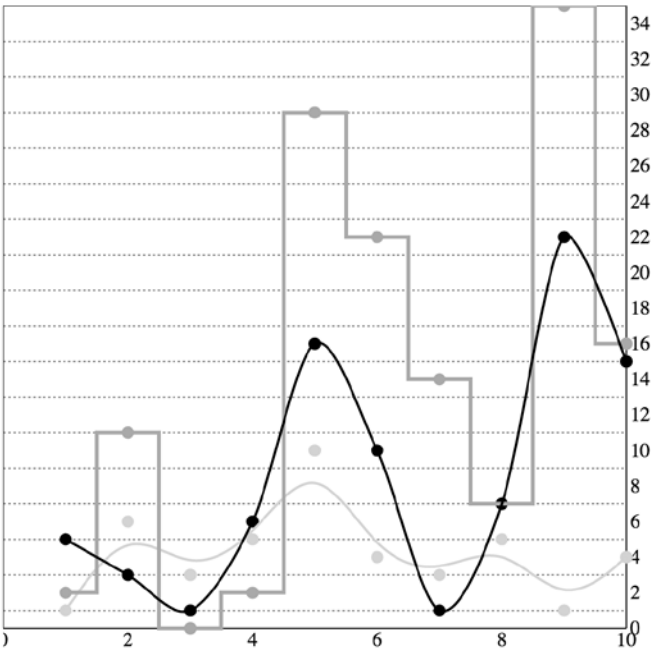


Figure 4.19 Light gray: “basis” interpolation, Dark gray: “step” interpolation, Black: “cardinal” interpolation

What’s the best interpolation?

Interpolation modifies the representation of data. Experiment with this drawing code to see how the different interpolation settings show different information than other interpolators. Data can be visualized in different ways, all correct from a programming perspective, and it’s up to you to make sure the information you’re visualizing reflects the actual phenomena.

Data visualization deals with the visual representation of statistical principles, which means its subject to all the dangers of misuse of statistics. Interpolating lines, because they change a clunky looking line into a smooth, “natural”, line, is particularly vulnerable to misuse.

4.5 Complex Accessor Functions

Each of the previous types of chart we built were based on points. The scatterplot is just points on a grid, while the boxplot consists of complex graphical objects in place of points, and line charts just use the points as the basis for drawing a line. In this chapter and earlier chapters, we've dealt with rather staid examples of information visualization, of a kind you might easily create in any traditional spreadsheet. But you didn't get into this business to make Excel charts, you want to wow your audience with beautiful data, and win awards for your aesthetic

je ne sais quoi and evoke deep emotional responses with your representation of change over time. You want to make streamgraphs like the one in Figure 4.20.

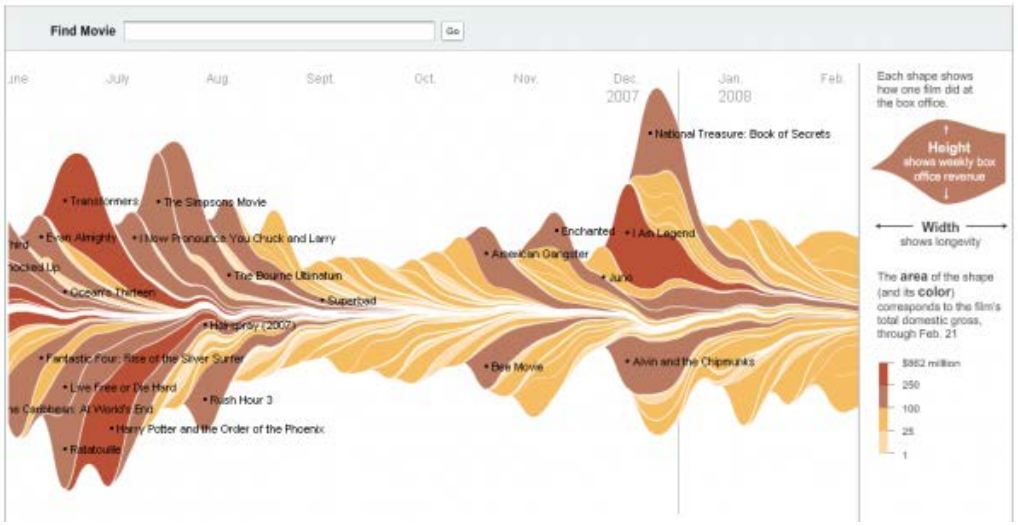


Figure 4.20 Behold the glory of the streamgraph. Look on my works, ye mighty, and despair!

It is a sublime piece of information visualization that like the boxplot represents variation and change. It may seem like a difficult thing to create, until you start to put the pieces together. Ultimately, a streamgraph is what's known as a stacked chart, where the layers accrete upon each other and adjust the area of the elements above and below based on the space taken up by the components closer to the center. It appears organic because that accretive nature mimics the way many organisms grow, and seems to imply the kinds of emergent properties that govern the growth and decay of organisms. We'll get back to interpreting its appearance, but first lets figure out how to build it.

The reason we're looking at a streamgraph is because it's not actually that exotic. A streamgraph is simply a stacked graph, which means it is fundamentally similar to our earlier line charts. By learning how to make it, we can better understand another kind of generator, d3.svg.area(). The first thing we need is some data that is amenable to this kind of visualization. Let's mimic the New York Times, from which we get the streamgraph in figure 4.x, and work with the grosses for six movies over the course of nine days. Each data point is therefore the amount of money a movie made on a particular day.

Listing 4.8 movies.csv

```
day,movie1,movie2,movie3,movie4,movie5,movie6
1,20,8,3,0,0,0
2,18,5,1,13,0,0
3,14,3,1,10,0,0
```

```
4,7,3,0,5,27,15
5,4,3,0,2,20,14
6,3,1,0,0,10,13
7,2,0,0,0,8,12
8,0,0,0,0,6,11
9,0,0,0,0,3,9
10,0,0,0,0,1,8
```

We're going to build a streamgraph because in order to do so we need to get more sophisticated with the way we're accessing data and feeding it to generators in order to draw lines. In our earlier example, we created three different line generators for our dataset, but that's terribly inefficient. We also used pretty simple functions to draw the lines, but we'll need more than that to draw something like a streamgraph. So, even if you don't ever think you'll want to draw streamgraphs (and there are reasons why you may not, which we'll get into at the end of this section) the important thing to focus on here is the way that accessors are used with D3's line, and later area, generators.

Listing 4.9 The callback function to draw movies.csv as a linechart

```
for (x in data[0]) {
  if (x != "day") {#a

movieArea = d3.svg.line() #b
    .x(function(d) {
    return xScale(d.day) #c
})
    .y(function(d) {
    return yScale(d[x]) #d
})
    .interpolate("cardinal")

d3.select("svg")
    .append("path")
    .style("id", x + "Area")
    .attr("d", movieArea(data))
    .attr("fill", "none")
    .attr("stroke", "black")
    .attr("stroke-width", 3)
    .style("opacity", .75)

  }
}
```

#a We'll iterate through our data attributes with a for-loop where x is going to be the name of each column from our data ("day", "movie1", "movie2", etc.) which allows us to dynamically create and call generators.
#b Instantiate a line generator for each movie.
#c Every line uses the day column for its x-value.
#d Dynamically set the the y-accessor function of our line generator to grab the data from the appropriate movie for our y variable.

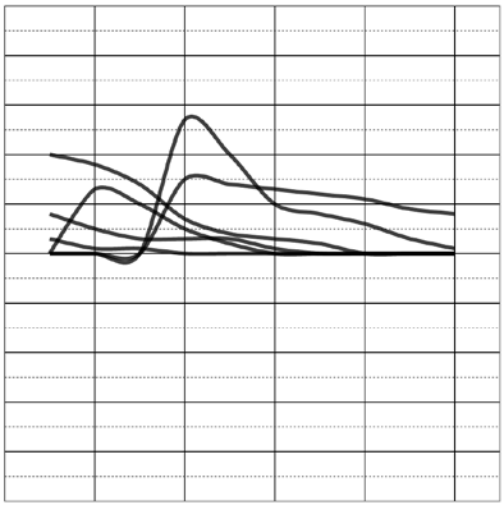


Figure 4.21 Each movie column is drawn as a separate line. Notice how the “cardinal” interpolation creates a graphical artifact where it seems like some movies made negative money.

The line drawing code produces a cluttered line chart in Figure 4.21. As we learned in Chapter 1, lines and filled areas are almost exactly the same thing in SVG, differentiated only by the existence of a “Z” at the end of the drawing code that indicates the shape is closed, or the presence or absence of a “fill” style. With that in mind, it’s important to note that D3 provides d3.svg.line and d3.svg.area generators to draw lines or areas. Both of these constructors produce <path> elements but d3.svg.area provides helper functions to bound the lower end of your path in a way amenable to producing areas in charts. This means you need to define a .y0() accessor that corresponds to your y accessor and determines what the shape of the bottom of your area. Let’s see how d3.svg.area() works:

```
for (x in data[0]) {
  if (x != "day") {

movieArea = d3.svg.area()
    .x(function(d) {
    return xScale(d.day)
})
    .y(function(d) {
    return yScale(d[x])
})
    .y0(function(d) { #a
    return yScale(-d[x])
})
    .interpolate("cardinal")

d3.select("svg")
    .append("path")
    .style("id", x + "Area")
```

```
        .attr("d", movieArea(data))
        .attr("fill", "darkgray")
        .attr("stroke", "lightgray")
        .attr("stroke-width", 2)
        .style("opacity", .5)

    }
  }
```

#a This new accessor provides you with the ability to define where the bottom of the path is. In this case, we'll start by making the bottom equal to the inverse of the top, which will just mirror the shape.

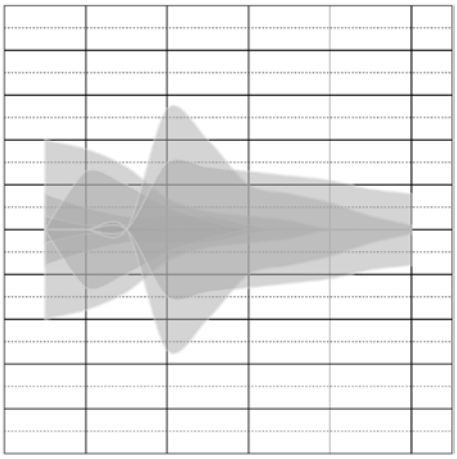


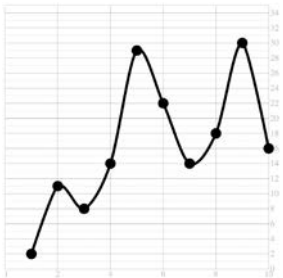
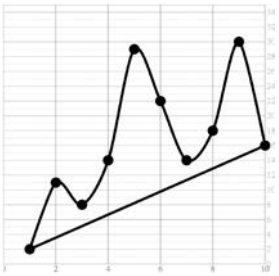
Figure 4.22 By using an area generator and defining the bottom of the area as the inverse of the top, we can mirror our lines to create an area chart. Here they're drawn with semi-transparent fills, so that you can see how they overlap.

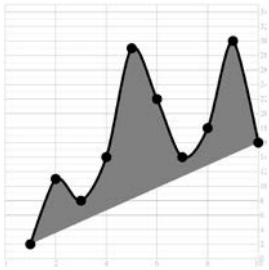
Should we always draw filled paths with d3.svg.area?

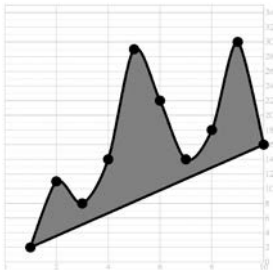
No. Counter intuitively, you should use d3.svg.line to draw filled areas. To do so, though, you need to append "Z" to the created "d" attribute. This indicates that the path is closed.

Open Path	Closed Path Changes	Explanation

movieArea = d3.svg.line()
.x(function(d) {
return xScale(d.day)
})
.y(function(d) {
		Writing the constructor for the line drawing code is the same regardless of whether you want a line or shape or filled or unfilled.

return yScale(d[x]) }) .interpolate("cardinal");		
d3.select("svg") .append("path") **.attr("d", movieArea(data))** .attr("fill", "none") .attr("stroke", "black") .attr("stroke-width", 3);	d3.select("svg") .append("path") **.attr("d", movieArea(data) + "Z")** .attr("fill", "none") .attr("stroke", "black") .attr("stroke-width", 3);	It is only in calling the constructor when you append a <path> element that you determine whether the line is "closed" or not. This is done by concatenating a "Z" to the string created by your line constructor for the **d** attribute of the <path>.
		The result of adding a "Z" to the end of an SVG <path> element's d attribute is to draw a line connection the two end points.
d3.select("svg") .append("path") .attr("d", movieArea(data)) **.attr("fill", "none")** .attr("stroke", "black") .attr("stroke-width", 3);	d3.select("svg") .append("path") .attr("d", movieArea(data) + "Z") **.attr("fill", "gray")** .attr("stroke", "black") .attr("stroke-width", 3);	You might think that only a closed path could be filled, but the fill of a path is the same whether or not you close the line by appending "Z".





The area of a path filled is always the same, whether it is closed or not.

You use d3.svg.line when you want to draw most shapes and lines, whether filled or unfilled or closed or open. You should use d3.svg.area() when you want to draw a shape where the bottom of the shape can be calculated based on the top of the shape as you're drawing it. It's suitable for drawing bands of data, such as that found in a stacked area chart or streamgraph.

By defining the y0 function of d3.svg.area we've mirrored the path created and filled it as seen in Figure 4.22, which is a step in the right direction. Notice that we're presenting inaccurate data, now, since the area of the path is twice the area of the data. We want our areas to draw one on top of the other, which means we need .y0() to point to a complex stacking function that makes the bottom of an area equal to the top of the previously drawn area. D3 comes with a stacking function, .stack(), which we'll look at later, but for the purpose of our example, we're going to write our own.

Listing 4.10 Callback function for drawing stacked areas

```
fillScale = d3.scale.linear() #a
   .domain([0,5])
   .range(["lightgray","black"]);
var n = 0; #b
for (x in data[0]) {
if (x != "day") { #c
movieArea = d3.svg.area() #d
        .x(function(d) {
        return xScale(d.day)
    })
        .y(function(d) {
        return yScale(simpleStacking(d,x))
    })
        .y0(function(d) {
        return yScale(simpleStacking(d,x) - d[x]);
    })
        .interpolate("basis")
d3.select("svg") #e
        .append("path")
        .style("id", x + "Area")
        .attr("d", movieArea(data))
        .attr("fill", fillScale(n))
```

```
        .attr("stroke", "none")
        .attr("stroke-width", 2)
        .style("opacity", .5)
        n++; #f
        }
    }
function simpleStacking( #g
incomingData, incomingAttribute) {
      var newHeight = 0;
      for (x in incomingData) {
        if (x != "day") {
          newHeight += parseInt(incomingData[x]);
          if (x == incomingAttribute) {
            break;
          }
        }
      }
      return newHeight;
    }
```

#a Create a color ramp that corresponds to the six different movies
#b Each movie will correspond to one iteration through the for loop, so you'll increment *n* to use in the color ramp. You could also create an ordinal scale assigning a color for each movie.
#c We're not going to draw a line for the *day* value of each object, because this is what provides us with our x-coordinate
#d A d3.svg.area() generator for each iteration through the object that corresponds to one of our movies using the *day* value for the x-coordinate but will iterate through the values for each movie for the y-coordinates.
#e Draw a path using the current constructor--remember we'll have one for each attribute not named "day".
Give it a unique id based on which attribute we're drawing an area for.
Fill the area with a color based on the color ramp we built.
#f Finish the for loop and increment to the next attribute in the object and increment *n* to color the next area.
#g This function takes the incoming bound data and the name of the attribute and loops through the incoming data, adding each value until it reaches the current named attribute.
As a result, it returns to total value for every movie during this day up to the movie you've sent.

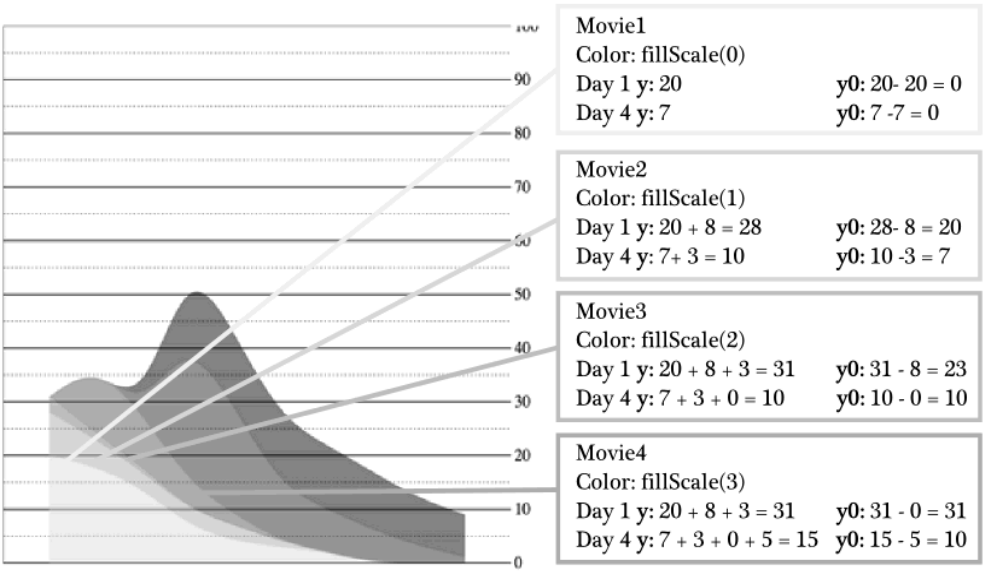


Figure 4.23 The results of our stacked area code is that each area representing a movie has the bottom of that area equal to the total amount of money made by any movies drawn earlier for that day.

The stacked area chart in Figure 4.23 is already pretty complex. To make it a proper streamgraph, the stacks need to alternate. This requires a more complicated stacking function:

Listing 4.11 A stacking function that alternates between drawing an area above or below the last area.

```
…
movieArea
.y(function(d) {
return yScale(alternatingStacking(d,x,"top")) #a
})
.y0(function(d) {
return yScale(alternatingStacking(d,x,"bottom"));
})
…
function alternatingStacking(incomingData,incomingAttribute,topBottom) #b
{
var newHeight = 0;
var skip = true;
for (x in incomingData) {
if (x != "day") { #c
if (x == "movie1" || skip == false) { #d
newHeight += parseInt(incomingData[x]);
if (x == incomingAttribute) {
break; #e
}
```

```
if (skip == false) {
skip = true;
}
else {
n%2 == 0 ? skip = false : skip = true;
}
}
else {
skip = false;
}
}
}
if(topBottom == "bottom") { #f
newHeight = -newHeight;
}
if (n > 1 && n%2 == 1 && topBottom == "bottom") {
newHeight = 0;
}
if (n > 1 && n%2 == 0 && topBottom == "top") {
newHeight = 0;
}
return newHeight;
}
```

#a You can create whatever complex accessor function you want for your generators
#b We need not only the data, but wehter we are drawing the top or bottom of the area, which alternates as we move through the dataset
#c Always skip day, since that's just our x position
#d Skip the first movie (our center) and then skip every other movie to get the alternating pattern
#e Stop when you reach this movie, that will give you the baseline
#f The height is negative for areas on the bottom side of the streamgraph, and positive for those on the top side

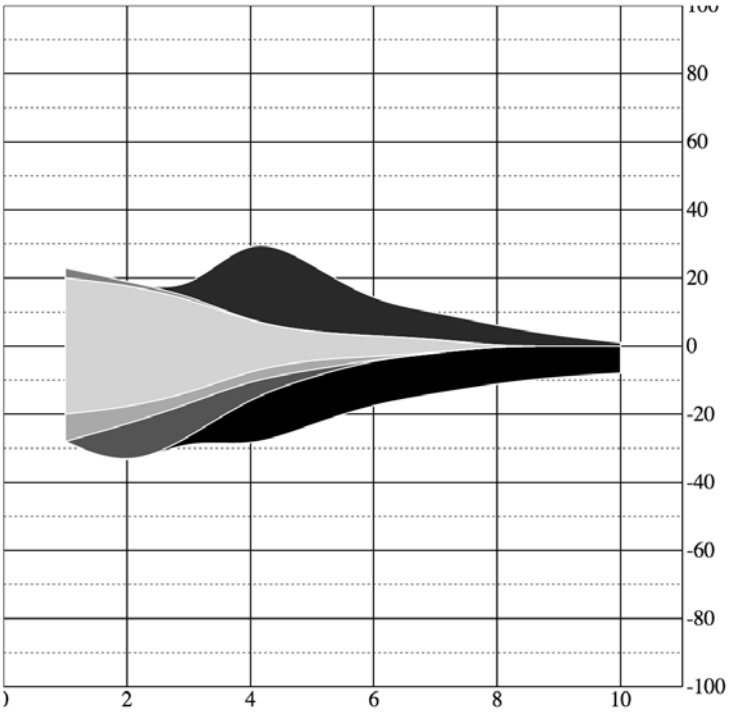


Figure 4.24 A simple streamgraph that shows the accreted values for movies by day. The problems of using different interpolation methods are clear, since the basis method here shows some innacuracies, and the difficulty of labeling the scale is also apparent.

This streamgraph from Figure 4.24 has some obvious issues, but we're not going to correct them. For one thing, we're overrepresenting the gross of the first movie by drawing it at twice the height. If we wanted to, we could easily make the stacking function account for this by halving the values of that first area. Another issue is that the actual areas being drawn are different from the areas being displayed, which isn't a problem when your data visualization is only going to be read from one perspective and not from multiple perspectives.

But the purpose of this section was to focus on building complex accessor functions to create, from scratch, the kinds of data visualization you've seen and likely thought of as exotic. Let's assume this data is correct and take a moment to analyze effectiveness of this admittedly attractive method of visualizing data. Is it really a better way to show movie grosses than a more simple stacked graph or line chart? That depends on the scale of the questions being addressed by the chart. If you're trying to discover overall patterns of variation in movie grosses, as well as spot interactions between them (for instance, seeing if a particularly high-grossing-over-time movie interferes with the opening of another movie) then it might be useful. If you're trying to simply impress an audience with a complex-looking chart, it would

also be useful. But otherwise, you're probably going to be better off with something more simple than this. But even if you only ever build less visually impressive charts, you'll still use the same techniques we've gone over in this section.

4.6 Summary

In this chapter you've learned the basics of creating charts, which includes:

- Integrating generators and components with the selection and binding process.
- Learning about D3 components and the axis component to create chart elements like an x-axis and y-axis
- Interpolating graphical elements, such as lines or areas from point data, using D3 generators.
- Creating complex SVG objects that leverage the <g> element's ability to create child shapes using .each()that can be drawn based on the bound dataset.
- Exploring representation of multidimensional data using boxplots.
- Combining and extending these methods to implement a sophisticated charting method, the streamgraph, while learning how such charts may outstrip their audience's ability to successfully interpret such data.

These skills and methods will help you to understand better the layouts that are included with D3 and which we'll be exploring in more detail in the following chapters. The incredible breadth of data visualization techniques possible with D3 is based on the fundamental similarity between different methods of displaying data, at the visual level, at the functional level, and at the data level. By understanding how the basic processes work and how they can be combined to create more interactive and rich representation, you'll be better equipped to choose and deploy the right one for your data.

5 Layouts

D3 contains a variety of functions referred to as layouts that help you format your data so that it can be presented using a popular charting method. In this chapter, we'll look at several different layouts so that you can understand general layout functionality and know how to deal with D3's layout structure, as well as deploy one of these layouts with your data.

In each case, as we'll see with the following examples, when a dataset is associated with a layout, each of the objects in the dataset has attributes populated on it that allow for drawing the data. Layouts do not draw the data, nor are they called like components or referred to in the drawing code like generators. Rather, they are a pre-processing step that formats your data such that it's ready to be displayed in the form you've chosen. This means that a layout can be updated and then if you rebind that altered data to your graphical objects, you can use the D3 enter/update/exit syntax we first encountered in Chapter 2 to update your layout. Paired with animated transitions, this can provide you with the framework for an interactive, dynamic chart.

This chapter gives an overview of layout structure by implementing popular layouts such as the histogram, pie chart, tree and circle packing. There are more layouts than those found in this chapter such as the chord layout and more exotic ones, but they follow the same principles and should be easy to understand after looking at these. We'll get started with a kind of chart you've already worked with, the bar chart or histogram, which has its own layout that helps abstract the process of building this kind of chart.

5.1 Histograms

Before we get into charts that you'll need layouts for, let's take a look at a chart that you could, and have, easily make without a layout. In chapter 2 we made a bar chart based on our Twitter data by using d3.nest(). But D3 has a layout, d3.layout.histogram(), that will bin values automatically and provide you with the necessary settings to draw a bar chart based on a scale that you've defined. Many people who first get started with D3 think it's a charting library, and that they'll find a function like d3.layout.histogram and it will create a bar chart in a

when it's run, but D3 layouts don't result in charts, they result in the settings necessary for charts, which means you have to put in a bit of extra work for simple charts, but you have enormous flexibility (as we'll see later in this chapter and in later chapters) that allow you to make diagrams and charts like you would not be able to find in other libraries.

You can see in Listing 5.x the code necessary to create a histogram layout and associate it with a particular scale. I've also included a simple example of how you can use interactivity to adjust the original layout and rebind the data to your shapes to change the histogram from showing the number of tweets that were favorited to the number of tweets that were retweeted.

Listing 5.x Histogram Code

```
var xScale = d3.scale.linear().domain([0,5]).range([0,500]);
var yScale = d3.scale.linear().domain([0, 10]).range([400, 0]);

var xAxis = d3.svg.axis()
.scale(xScale)
.ticks(5)
.orient("bottom");

  var histoChart = d3.layout.histogram(); #a

  histoChart
.bins([0,1,2,3,4,5]) #b
.value(function(d) {return d.favorites.length}); #c

  var histoData = histoChart(tweetsData); #d

d3.select("svg").selectAll("rect")
.data(histoData)
.enter()
.append("rect")
.attr("x", function(d) {return xScale(d.x)})
.attr("y", function(d) {return yScale(d.y)})
.attr("width", 48)
.attr("height", function(d) { return 400 - yScale(d.y); }) #e
.on("click", retweets)

d3.select("svg").append("g")
.attr("class", "x axis")
.attr("transform", "translate(0,400)")
.call(xAxis);

d3.select("g.axis").selectAll("text").attr("dx", 50); #f

function retweets() {
        histoChart.value(function(d) {return d.retweets.length}); #g

        histoData = histoChart(tweetsData);

        d3.selectAll("rect").data(histoData) #h
        .transition().duration(500)
        .attr("x", function(d) {return xScale(d.x)})
        .attr("y", function(d) {return yScale(d.y)})
```

```
            .attr("height", function(d) { return 400 - yScale(d.y); });
    }

    }
```

#a Create a new layout function
#b Determine the values the histogram bins for
#c The value the layout is binning for from the datapoint
#d The layout formats the data
#e The formatted data is used to draw the bars
#f Center the axis labels under the bars
#g Change the value being measured
#h Bind and redraw the new data

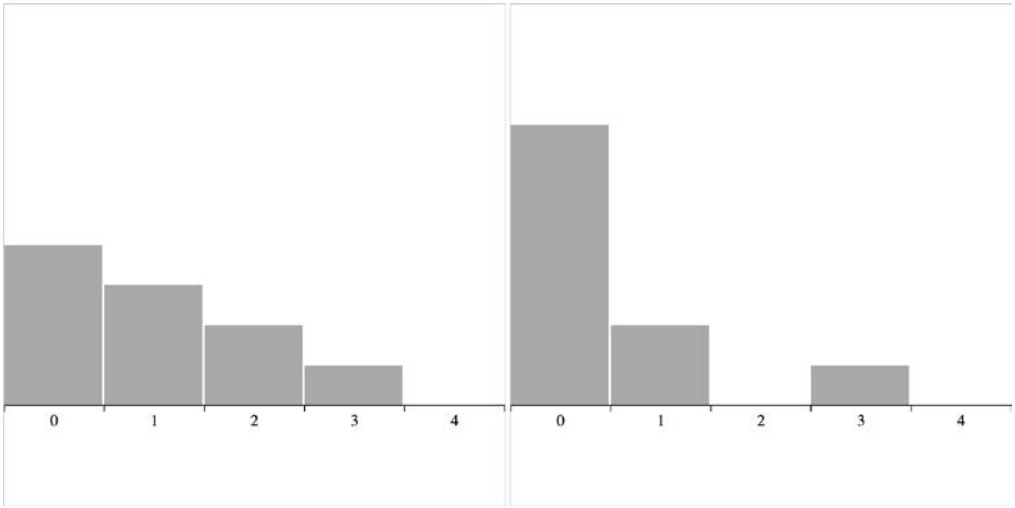


Figure 5.x The histogram in its initial state (left) and after we change the measure from favorites to retweets (right) by clicking on one of the bars.

You're not expected to follow the process of using the histogram to create the results in Figure 5.x with this example. Instead, we're going to get into that as we look at more layouts throughout this chapter. Just notice a few general principles: First, a layout formats the data for display, as I pointed out in the beginning of Chapter 4. Second, you still need the same scales and components that you needed when you were creating a bar chart from raw data without the help of a layout. Third, the histogram is useful because it automatically bins data, whether it is whole numbers like this or it falls within a range of values in a scale. Finally, if you want to dynamically change a chart using a different dimension of your data, you don't need to remove the original, you just need to reformat your data using the layout and rebind it to the original elements, preferably with a transition. We'll see this in more detail in our next example, which uses another type of basic charts: pie charts.

5.2 Pie Charts

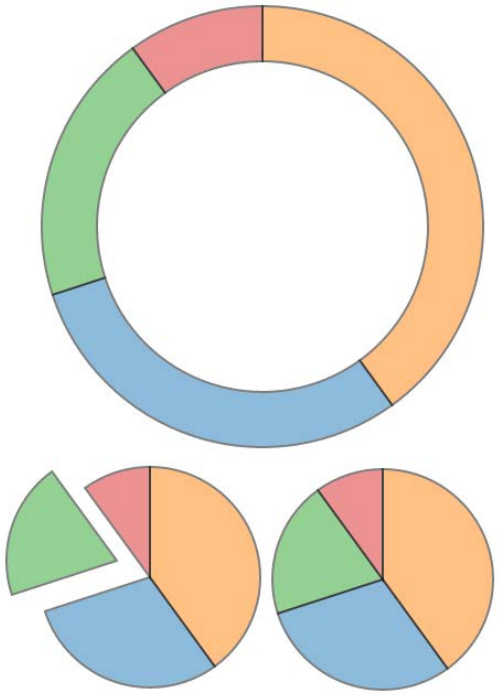


Figure 5.x The traditional pie chart (bottom right) represents proportion as an angled slice of a circle. With some slight modification, it can be turned into a donut or ring chart (top) or an exploded pie chart (bottom left).

One of the most straightforward layouts available in D3 is the pie layout, which is used to make pie charts like those seen above in Figure 5.x. Like all layouts, a pie layout can be created and assigned to a variable where it is used as both an object and a function. In this section, we'll learn how to create a simple pie chart, transform it into a ring chart and learn how to use tweening to properly transition it when you change its data source. Once you create it, you can pass it an array of values (which I'll refer to as a dataset) and it will compute the necessary starting and ending angles for each of those values to draw a pie chart. This can be seen clearly when we pass a simple array of numbers as our dataset to a pie layout in the console with the following code, which will not produce any kinds of graphics but rather result in the response we see below in Figure 5.x.

```
var pieChart = d3.layout.pie();
var yourPie = pieChart([1,1,2]);
```

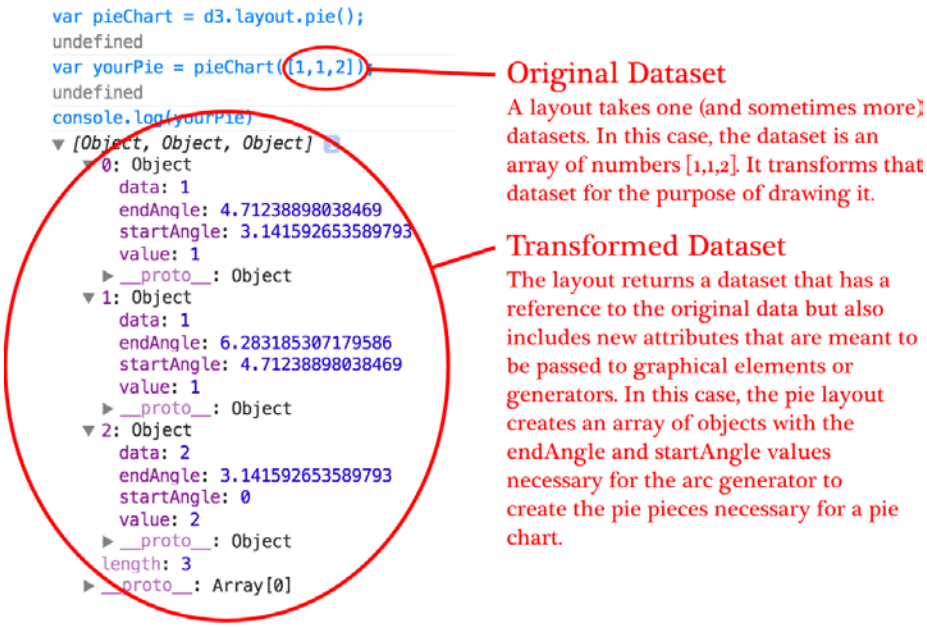


Figure 5.x The results of a newly created pie layout applied to an array of [1,1,2] shows objects created with a start angle, end angle, and value attribute corresponding to the dataset, as well as the original data, which in this case is a simple number.

As you can see, your pieChart function created a new array of three objects, with the corresponding startAngle and endAngle for each of the data values such that it would draw a simple pie chart with one piece from 0 degrees to pi, the next from pi to 1.5pi and the last from 1.5pi to 2pi. But this isn't a drawing, or any kind of SVG code like the line and area generators produced.

5.2.1 Drawing the Pie Layout

Instead, these are settings that need to be passed to a generator to make each of the pieces of your pie chart. This particular generator is d3.svg.arc and it's instantiated just like the generators we worked with in Chapter 4. While it has a few settings, the only one you need to deal with for this first example is the outerRadius() function, which allows you to set a dynamic or fixed radius for your arcs.

```
var newArc = d3.svg.arc();
newArc.outerRadius(100); #a
console.log(newArc(yourPie[0])); #b
```

#a To give our arcs and resulting pie chart a radius of 100px
#b The function returns the "d" attribute necessary to draw this arc as a <path> element:
"M6.123031769111886e-15,100A100,100 0 0,1 -100,1.2246063538223773e-14L0,0Z"

Now that we know how the arc constructor works and that it will work with our data, all we need to do is to bind the data created by our pie layout and pass it to <path> elements to draw our pie chart. The pie layout is centered on the 0,0 point in the same way that a circle is drawn, so if we want to draw it at the center of our canvas, we need to create a new <g> element to hold the <path> elements we'll draw and then move the <g> to the center of the canvas.

```
d3.select("svg")
.append("g") #a
.attr("transform","translate(250,250)")
.selectAll("path")
.data(yourPie) #b
.enter()
.append("path")
.attr("d", newArc) #c
.style("fill", "blue")
.style("opacity", .5)
.style("stroke", "black")
.style("stroke-width", "2px")
```

#a We'll append a new <g> and move it to the middle of the canvas so that it'll be easier to see the results
#b We're going to bind the array that was created using the pie layout, not our original array or the pie layout itself
#c Each path drawn based on that array will need to pass through the newArc function, which sees the startAngle and endAngle attributes of the objects and produces the commensurate SVG drawing code

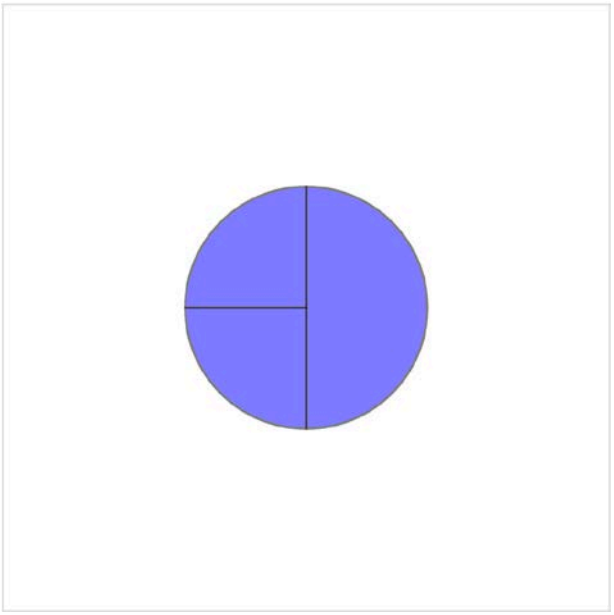


Figure 5.x A simple pie chart showing three pie pieces that subdivide the circle between the values in the array [1,1,2].

The pie chart layout, like most layouts, grows a bit more complicated when you want to work with JSON object arrays rather than simple number arrays. Let's bring back our tweetdata.csv from Chapter 2, which we can nest and measure again to transform it from an array of tweets into an array of Twitter users with their number of tweets computed.

```
      nestedTweets = d3.nest()
      .key(function (el) {return el.user})
      .entries(incData);

      nestedTweets.forEach(function (el) {
        el.numTweets = el.values.length
        el.numFavorites = d3.sum(el.values, function (d) {return
d.favorites.length}); #a
        el.numRetweets = d3.sum(el.values, function (d) {return
d.retweets.length}); #b
      })
```

#a Gives us the total number of favorites by summing the favorites array length of all the tweets
#b Gives us the total number of retweet by doing the same for the Retweets array length

5.2.2 Creating a ring chart

If we try pieChart(nestedTweets) like we did with the earlier array it will fail, because it doesn't know that the numbers we should be using to size our pie pieces come from the .numTweets attribute. Most layouts, pie included, come with the capacity to define where the values are in your array by defining an accessor function necessary to get to those values. In the case of nestedTweets, this means defining pieChart.value() to point at the numTweets attribute of the dataset it's being used on. While we're at it, let's set a value for our arc generator's innerRadius() so that you end up creating a donut chart instead of a pie chart. With those changes in place, we can use the same code we did before to draw the pie chart in Figure 5.x.

```
pieChart.value(function(d) {return d.numTweets});
newArc.innerRadius(20)
yourPie = pieChart(nestedTweets);
```

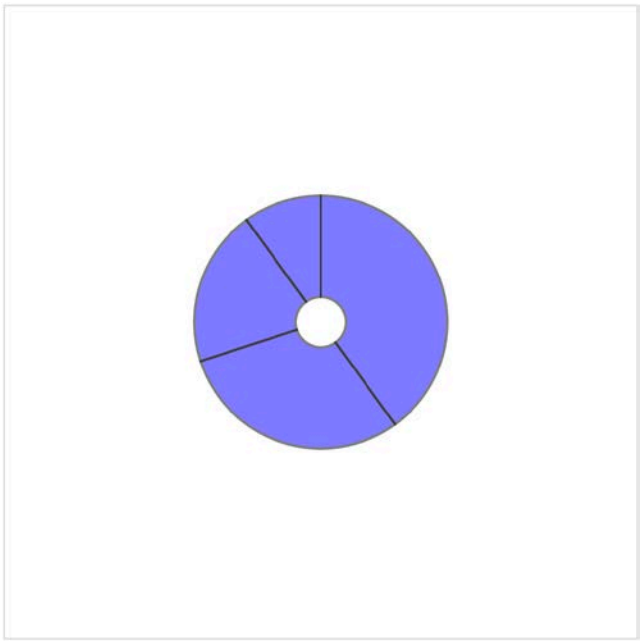


Figure 5.x A simple donut chart showing the number of tweets from our four users represented in the nestedTweets dataset.

5.2.3 Transitioning

You'll notice that for each value in nestedTweets we didn't just total the number of tweets, but also used d3.sum() to total the number of retweets and favorites (if any). Because we have this data, we can adjust our pie chart to show pie pieces based not on the number of tweets but on those other values. One of the core uses of a layout in D3 is that all we need to do to update the graphical chart is to make changes to the data or layout then rebind the data to the existing graphical elements. By using a transition, we can see the pie chart change from one form to the other. Running the code below will first transform the pie chart to represent number of favorites instead of number of tweets, and the next block will cause the pie chart to represent number of retweets. The final forms of the pie chart after running that code are seen in Figure 5.x.

```
pieChart.value(function(d) {return d.numFavorites});
d3.selectAll("path").data(pieChart(nestedTweets)).transition().duration(1000)
.attr("d", newArc);

pieChart.value(function(d) {return d.numRetweets});
d3.selectAll("path").data(pieChart(nestedTweets)).transition().duration(1000)
.attr("d", newArc);
```

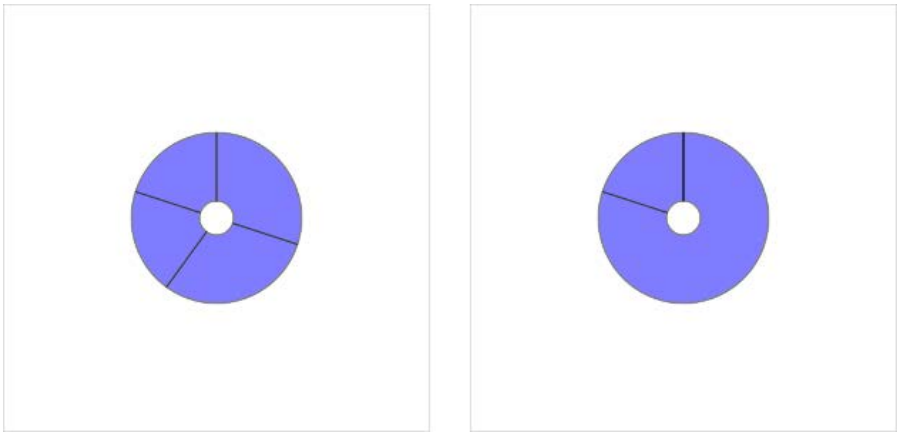


Figure 5.x The pie charts representing, on the left, total number of favorites, and on the right, the total number of retweets.

While the results in Figure 5.x are what we want, the transition can leave a lot to be desired. In Figure 5.x, we can see snapshots of the transition from the pie chart representing the number of tweets to the pie chart representing the number of favorites. As you'll see by running the code, and which we can see in these snapshots, the pie chart does not smoothly transition from one state to another but instead distorts quite significantly.

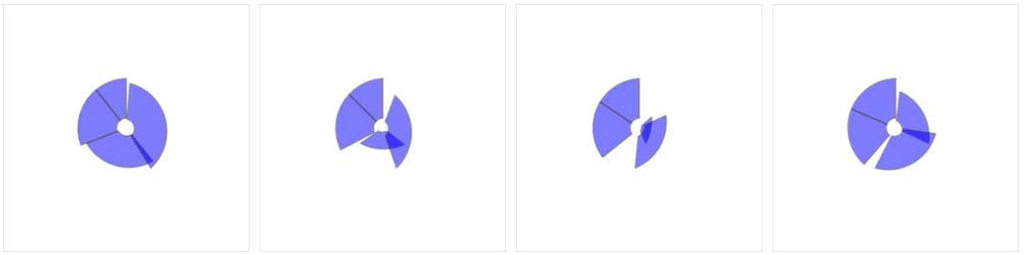


Figure 5.x Snapshots of the transition from the pie chart representing the number of tweets to the pie chart representing the number of favorites. This transition highlights the need to assign key values for data binding, as well as utilize tweens for some types of graphical transition, such as that used for arcs.

The reason we see this wonky transition is because, as we learned earlier, the default data-binding key is array position. The pie layout, when it measures data, also sorts it in order from largest to smallest, so as to create a more readable chart. But that means when we re-call the layout, it re-sorts the dataset, which means that the data objects are bound to different pieces in the pie chart, and so when we transition between them graphically, we see the effect above. To prevent this from happening, we need to disable this sort:

```
pieChart.sort(null);
```

The results of disabling the sort is a smooth graphical transition between numTweets and numRetweets because the object position in the array remains unchanged, and so the transition in the drawn shapes is straightforward. But if you look closely, you'll notice that the circle deforms a bit because the default transition() behavior doesn't deal with arcs well. To be very specific, it's not transitioning the degrees in your arcs, instead it's treating each arc as a basic geometric shape and transitioning from one to another.

This becomes obvious when we look at the transition from either of those versions of our pie chart to one that shows numFavorites, because some of the objects in our dataset have 0 values for that attribute, and one of them changes size quite dramatically. To clean this all up and make our pie chart transition properly, we need to institute a few changes to the code. Some of this we've already dealt with, like using key values for our created elements and using that in conjunction with exit and update behavior. But to make our pie pieces transition in a smooth graphical manner, we need to extend our transitions to include a custom tween to define how an arc can grow or shrink graphically into a different arc.

Listing 5.x Updated binding and transitioning for pie layout

```
pieChart.value(function(d) {return d.numRetweets}); #a

d3.selectAll("path").data(
pieChart(nestedTweets.filter(function(d) {return d.numRetweets > 0})), #b
function (d) {return d.data.key} #c
)
.exit() #d
.remove();

d3.selectAll("path").data(
pieChart(nestedTweets.filter(function(d) {return d.numRetweets > 0})),
function (d) {return d.data.key}
)
.transition()
.duration(1000)
.attrTween("d", arcTween); #e

    function arcTween(a) {
       var i = d3.interpolate(this._current, a);
       this._current = i(0);
     return function(t) {
     return newArc(i(t)); #f
     };
   }
```

#a Update the function that defines the value for which we're drawing arcs
#b Instead of binding the entire array, only bind the objects that have values
#c The user id becomes our key value, this same key value needs to be used in the initial enter() behavior
#d Remove the elements that have no corresponding data
#e Call a tween on the "d" attribute
#f Use the arc generator to tween the arc by calculating the shape of the arc explicitly

The result of the code in Listing 5.x is a pie chart that cleanly transitions the individual arcs or removes them when there is no data that corresponds to the pie pieces. We'll see more of attrTween and styleTween, as well as a deeper investigation of easing and other transition properties in later chapters.

At this point, you could label each pie piece <path> element, or color it according to a measurement or category, or add interactivity. But rather than spend a chapter creating the greatest pie chart application you've ever seen, we're going to move on to another kind of layout that's often used, the circle pack.

5.3 Pack

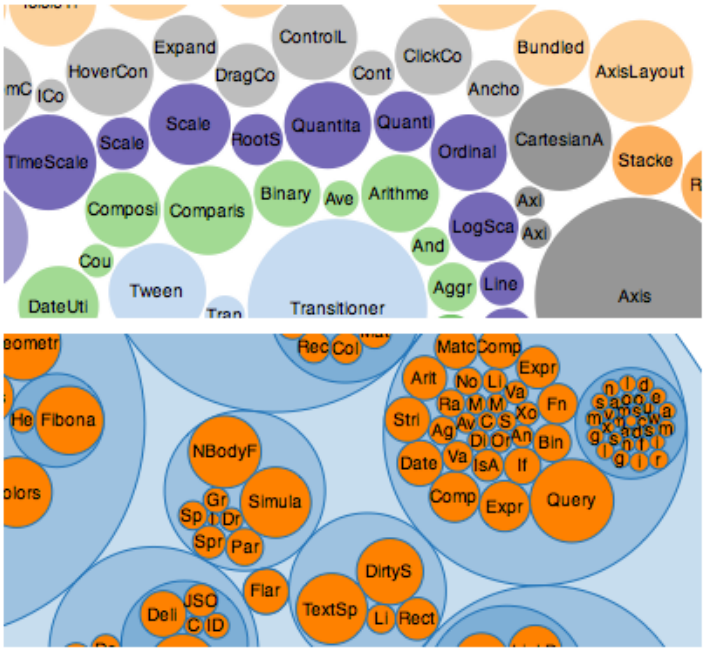


Figure 5.x Pack layouts, whether they are flattened (top) or visually represent hierarchy (bottom) are useful for representing nested data. (examples from Bostock)

Hierarchical data is amenable to an entire family of layouts. One of the most popular is circle packing, seen above in Figure 5.x. This places each object graphically inside the hierarchical parent of that object. It visually displays the hierarchical relationship in a readable manner. As with all layouts, the pack layout expects a default representation of data that may not align with the data you're working with. Specifically, pack expects a JSON object array where the child elements in a hierarchy are stored in a "children" attribute that points to an array. When reading through examples of layout implementations on the web, it's typical to

see the data formatted to match the expected data format, which in our case would formatting our tweets to look like this:

```
{id: “All Tweets”, children: [
{id: “Al’s Tweets”, children: [{id: “tweet1”}, {id: “tweet2”}]},
{id: “Roy’s Tweets”, children: [{id: “tweet1”}, {id: “tweet2”}]}
...
```

But it’s better to get accustomed to adjusting the accessor functions of the layout to match your data. This doesn’t mean you don’t have to do any data formatting. You still need to create a root node for circle packing to work (what is referred to as “All Tweets” above), but we’ll adjust the accessor function .children() to match the structure of the data as it’s represented in nestedTweets, which stores the child elements in the “values” attribute. We’ll also override the .value() setting that it uses to determine the size of circles to set it to a fixed value, as you’ll see in Figure 5.x.

Listing 5.x Simple circle-packing of nested tweets data

```
nestedTweets = d3.nest()
     .key(function (el) {return el.user})
     .entries(incData);

     packableTweets = {id: "All Tweets", values: nestedTweets} #a

     var depthScale = d3.scale.category10([0,1,2]); #b

     packChart = d3.layout.pack();
     packChart.size([500,500]) #c
     .children(function(d) {return d.values}) #d
     .value(function(d) {return 1}) #e

     d3.select("svg")
     .selectAll("circle")
     .data(packChart(packableTweets)) #f
     .enter()
     .append("circle")
     .attr("r", function(d) {return d.r}) #g
     .attr("cx", function(d) {return d.x})
     .attr("cy", function(d) {return d.y})
     .style("fill", function(d) {return depthScale(d.depth)}) #h
     .style("stroke", "black")
     .style("stroke", "2px")
```

#a First we’ll put the array that d3.nest creates inside a “root” object that acts as the top-level parent
#b We’ll want a color scale to color each depth of the circle pack differently
#c Set the size of the circle-packing chart to the size of our canvas
#d Set the pack accessor function for child elements to look for “values”, which matches the data created by d3.nest
#e Create a simple function that just returns 1 when determining the size of leaf nodes
#f Bind the results of packChart transforming packableTweets
#g Radius and XY coordinates are all computed by the pack layout
#h Each node is given a depth attribute that we can use to color them distinctly by depth

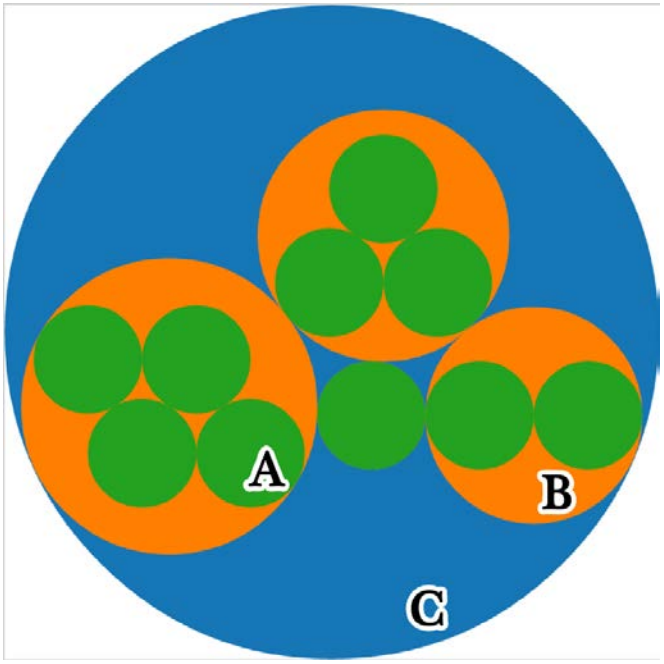


Fig 5.x Each tweet is represented by a green circle (A) nested inside an orange circle (B) that represents the user that made the tweet which are nested inside a blue circle (C) that represents our "root" node.

Notice that when the pack layout has a single child, as in the case of Sam, who only made one tweet, that the size of the child node is the same as the size of the parent. This can read visually to seem like Sam's tweet is at the same hierarchical level as the other Twitter users who made more tweets. To deal with this, we can modify the radius of the circle to account for its depth in the hierarchy, which can act as a margin of sorts:

```
.attr("r", function(d) {return d.r - (d.depth * 10)})
```

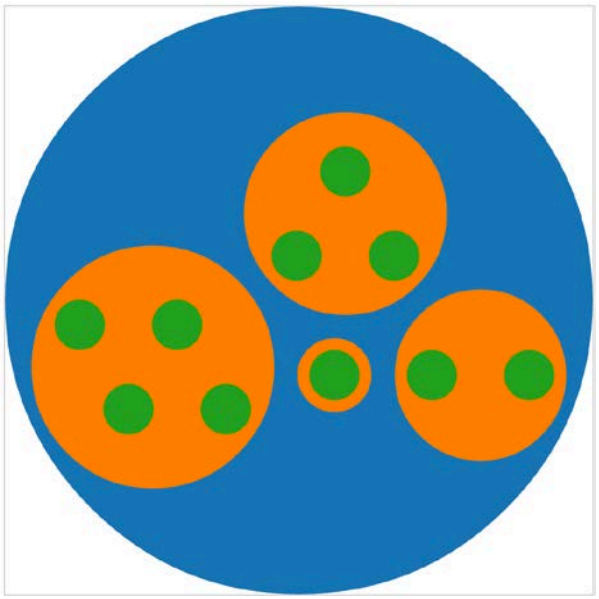


Figure 5.x A fixed margin based on hierarchical depth implemented by adjusted the circle size of each node to be reduced based on its computed "depth" value.

If you want to implement this in the real world, remember to use something more sophisticated than just the depth times ten, since this will scale poorly with a hierarchical dataset with many levels or with a crowded circle packing layout. If there was one or two more levels in this hierarchy, our fixed margin would result in negative radius values for the circles, so using a d3.scale.linear() or other method to set the margin should be your approach.

I glossed over the .value() setting on the pack layout earlier, but if you have some numerical measurement for your leaf nodes, then you can use that to set their size and therefore influence the size of their parent nodes. In our case, we can base the size of our leaf nodes (tweets) on the number of favorites and retweets each has received (the same value we used in Chapter 4 to as our "impact factor"):

```
      .value(function(d) {return d.retweets.length + d.favorites.length +
1})#a
```

#a Add one so that tweets with no retweets or favorites still have a value greater than zero and are displayed

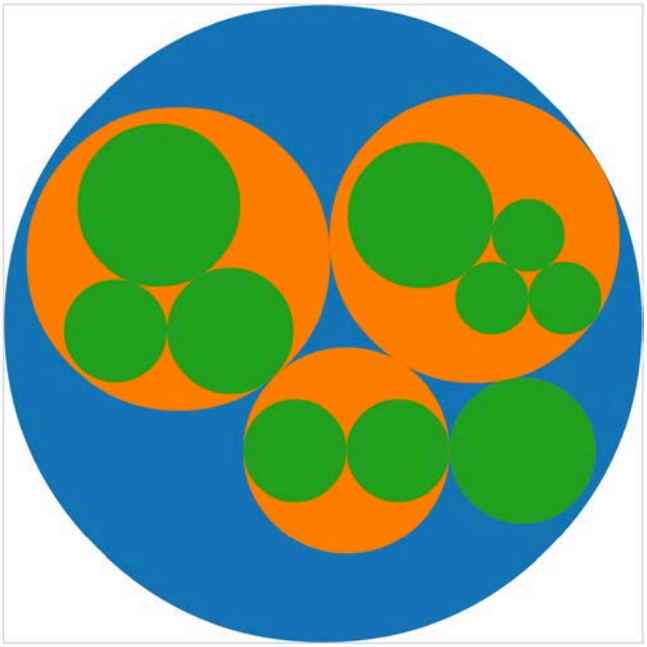


Figure 5.x A circle packing layout with the size of the leaf nodes set to the impact factor of those nodes.

Remember that layouts, like generators and components, are amenable to method chaining, you'll see examples where the settings and data are all strung together in long chains. As with the pie chart, we could assign interactivity to the nodes or adjust the colors but this chapter is more focused on the general structure of layouts, which is better served by moving on. Before we do, notice that circle packing is quite similar to another hierarchical layout known as treemaps, which pack space more effectively because they're built out of rectangles, but can be harder to read. The next layout is another hierarchical layout, known as a dendrogram, which more explicitly draws the hierarchical connections in our data.

5.4 Trees

Another way to show hierarchical data is to lay it out as one would a family tree, with the parent nodes connected to the child nodes in what is known as a dendrogram (Figure 5.x).

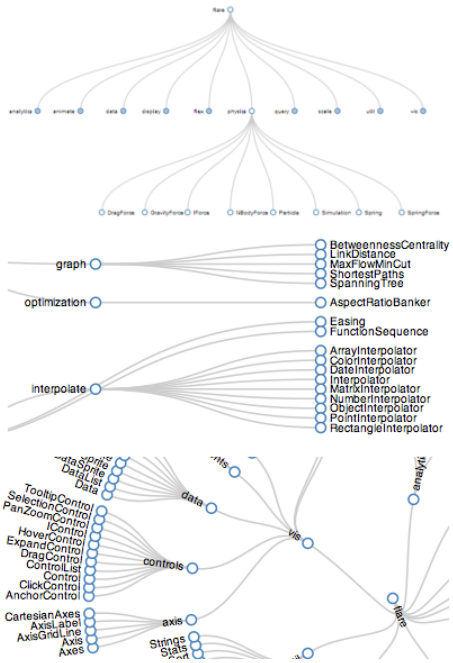


Figure 5.x Tree layouts are another useful method for expressing hierarchical relationships and are often seen laid out vertically (top), horizontally (middle), or radially (bottom). (examples from Bostock)

The word “dendro” just means tree, and in D3 the layout is d3.layout.tree. It follows much the same setup that the pack layout requires, except that to draw the lines connecting the nodes, we need a new generator, d3.svg.diagonal, which is used to draw a curved line from one point to another.

Listing 5.x Callback function to draw a dendrogram

```
        treeChart = d3.layout.tree();
        treeChart.size([500,500])
        .children(function(d) {return d.values});

        var linkGenerator = d3.svg.diagonal(); #a

        d3.select("svg")
        .append("g") #b
        .attr("id", "treeG")
        .selectAll("g") #c
        .data(treeChart(packableTweets)) #d
        .enter()
        .append("g")
        .attr("class", "node")
        .attr("transform", function(d) {return "translate(" +d.y+","+d.x+")"});
    #e
```

```
    d3.selectAll("g.node")
    .append("circle") #f
    .attr("r", 10)
    .style("fill", function(d) {return depthScale(d.depth)}) #d
    .style("stroke", "white")
    .style("stroke-width", "2px");

    d3.selectAll("g.node")
    .append("text")
    .text(function(d) {return d.id || d.key || d.content}) #g

    d3.select("#treeG").selectAll("path")
    .data(treeChart.links(treeChart(packableTweets))) #h
    .enter().insert("path","g")
    .attr("d", linkGenerator) #i
    .style("fill", "none")
    .style("stroke", "black")
    .style("stroke-width", "2px");
```

#a We can create a diagonal generator with the default settings
#b Create a parent <g#treeG> to put all these elements in
#c This time we'll create <g> elements so we can label them
#d Uses packableTweets and depthScale from the previous example
#e Like the pack layout, the tree layout computes the XY coordinates of each node
#f A little circle to represent each node that we'll color with the same scale we used for the circle pack
#g A text label for each node, with the text being either the id, key or content attribute, whichever the node has
#h The .links function of the layout will create an array of links between each node that we can use to draw these links
#hi Just like all the other generators

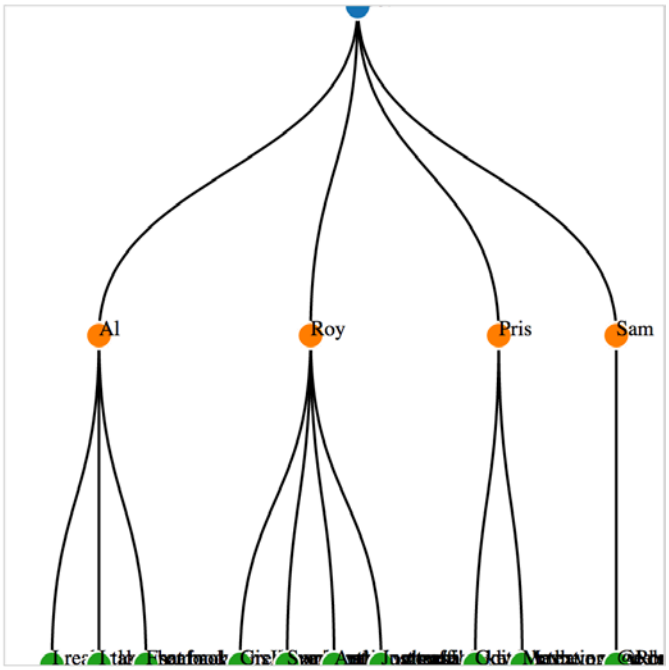


Figure 5.x A dendrogram laid out vertically using data from tweetdata.csv showing the level 0 “root” node (which we created to contain the users) in blue, the level 1 nodes (which represent users) in orange, and the level 2 “leaf” nodes (which represent tweets) in green.

The dendrogram in Figure 5.x is a bit hard to read like that. To turn this on its side, we need to adjust the positioning of the <g> elements by flipping the x and y coordinates, which allows us to orient the nodes horizontally. We also need to adjust the .projection() of the diagonal generator, which allows you to orient the lines horizontally.

```
      linkGenerator.projection(function (d) {return [d.y, d.x]})
...

      .append(“g”)
...
      .attr("transform", function(d) {return "translate(" +d.y+","+d.x+")"});
```

The results, seen in Figure 5.x, is more legible since the text isn’t overlapping on the bottom of the canvas, but you’ve still ended up with critical aspects of the chart drawn off the canvas. We only see half of the root node and the leaf nodes (the blue and green circles) and can’t read any of the labels of the leaf nodes, which represent our tweets.

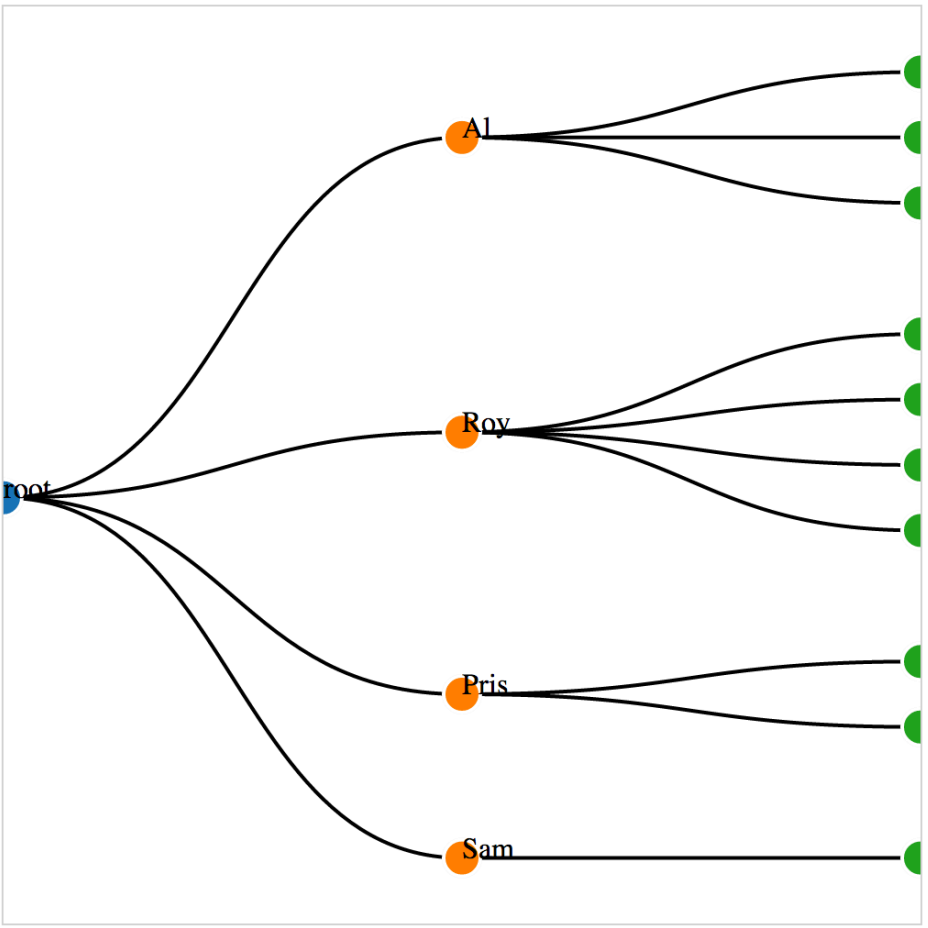


Figure 5.x The same dendrogram as figure 5.x but here laid out horizontally.

You could try to create some margins in the height and width of the layout as we've done earlier. Or you could provide information about each node as a modal information box that opens when you click on it, like we did with the soccer data. But a better option is to give the user the ability to drag the canvas up and down and left and right to see more of the visualization.

To do this, we use the D3 zoom behavior, d3.behavior.zoom, which creates a set of event listeners. A behavior is like a component but instead of creating graphical objects, it creates events, in this case for drag, mousewheel and double-click, and ties those events to the element that calls the behavior. With each of these events, a zoom object changes its .translate() and/or .scale() values to correspond to traditional dragging and zooming interaction. We'll use these changed values to adjust the position of graphical elements in

response to user interaction. Like a component, the zoom behavior needs to be called by the element to which you want these events attached. Typically, this means calling the zoom from the base <svg> element because that means it fires whenever we click on anything in our graphical area. When creating the zoom component, you need to define what functions are called on “zoomstart”, “zoomend” and “zoom”, which correspond (as you might think) to the beginning of a zoom event, the actual event itself, and the end of the event. Since “zoom” fires continuously as a user drags their mouse, you might want to have functions that are resource-intensive take place only at the beginning or end of the zoom event. We’ll see some more complicated zoom strategies, as well as the use of scale, in chapter 7 when we look at geospatial mapping, which uses zooming extensively.

When starting with a zoom component, as with other components, you create a new instance and set any attributes of it as you might need. In our case, we only want the default zoom component, with the “zoom” event triggering a new function, zoomed(), that changes the position of the <g> element that holds our chart and allows the user to drag it around.

```
      treeZoom = d3.behavior.zoom(); #a
      treeZoom.on("zoom", zoomed); #b
      d3.select("svg").call(treeZoom); #c

      function zoomed() {
        var zoomTranslate = treeZoom.translate();#d
        d3.select("g.treeG").attr("transform",
"translate("+zoomTranslate[0]+","+zoomTranslate[1]+")")#e
      }
```

#a First create a new zoom component
#b Key the “zoom” event to the zoomed() function
#c Call our zoom component with the SVG canvas
#d The transform attribute changes to reflect the zoom behavior
#e Updating the <g> to set it to the same translate setting of the zoom component will update the position of the <g> and all its child elements

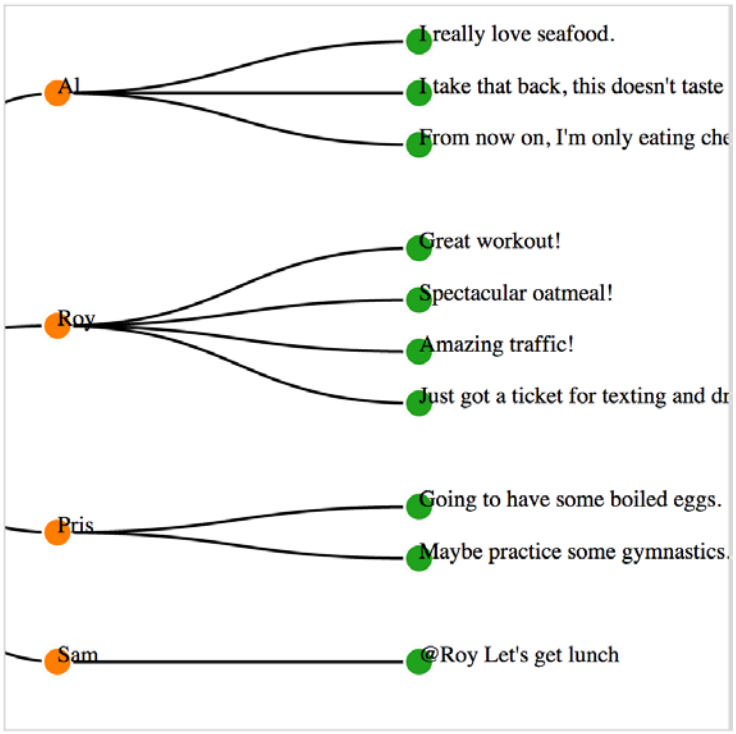


Figure 5.x The dendrogram when dragged to the left shows the labels for the tweets.

Now we can drag and pan our entire chart left and right and up and down. In figure 5.x, we can finally read the text of the tweets by dragging the chart to the left. The ability to zoom and pan gives you powerful interactivity to enhance your charts. It may seem odd that we learned how to use something called zoom and haven't even dealt with zooming in and out, but panning tends to be more universally useful with charts like these, while changing scale becomes a necessity when dealing with maps.

Finally, drawing your tree from top to bottom and left to right are not your only choices. If you tie the position of each node to an angle, and utilize a diagonal generator subclass created for radial layouts, you can draw your tree diagrams in a radial pattern:

```
var linkGenerator = d3.svg.diagonal.radial()
.projection(function(d) { return [d.y, d.x / 180 * Math.PI]; });
```

To make this work well, you need to reduce the size of your chart, since the radial drawing of a tree layout in D3 will use the size to determine the maximum radius, and will be drawn out from the 0,0 point of its container like a <circle> element would be:

```
treeChart.size([200,200])
```

With these changes in place, we need only change the positioning of the nodes to take rotation into account:

```
.attr("transform", function(d) { return "rotate(" + (d.x - 90) +
")translate(" + d.y + ")"; })
```

The results of these changes can be seen in Figure 5.x.

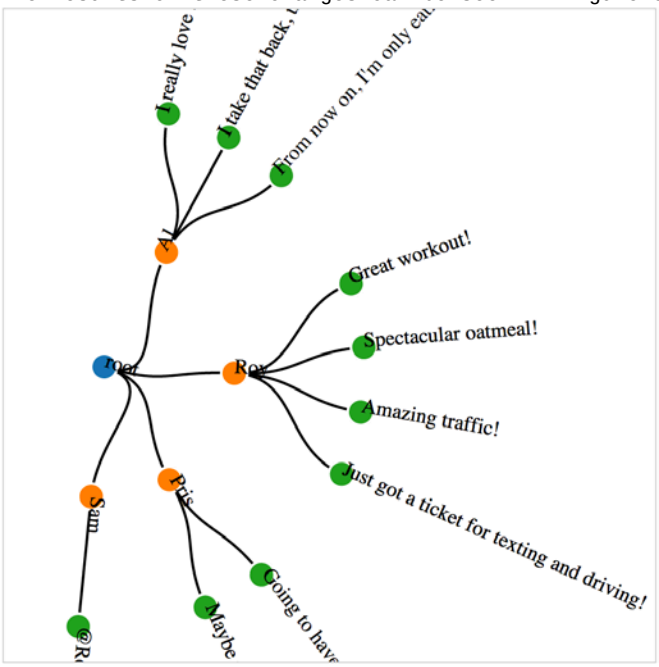


Figure 5.xThe same dendrogram laid out in a radial manner. Notice that the <g> elements are rotated, so their child <text> elements are rotated in the same manner.

The dendrogram is a generic way of displaying information and can be repurposed for menus or information you might not think of as traditionally hierarchical. One example is from the work of Jason Davies, who used the dendrogram functionality in D3 to create word trees.

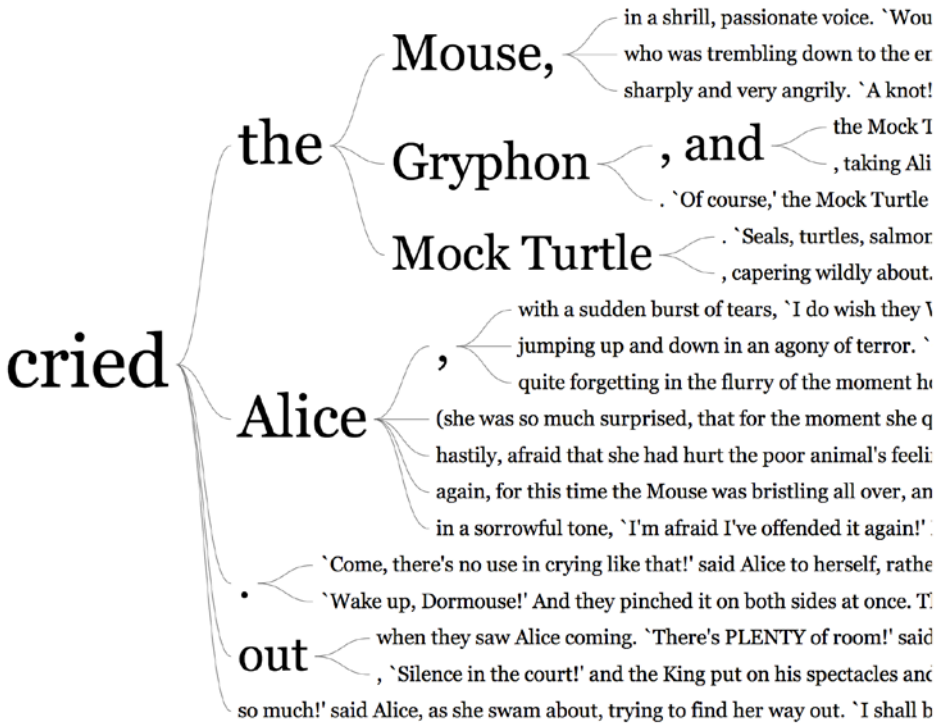


Figure 5.x Example of dendrogram use in word trees by Jason Davies http://www.jasondavies.com/wordtree/

Hierarchical layouts are common and well-understood by readers. This gives you the option to emphasize the nested container nature of a hierarchy, as we did with the circle pack layout, or the links between parent and child element, as with the dendrogram.

5.5 Stack Layout

We already saw the effects of the stack layout in the last chapter when we created our own streamgraph by writing a simple, and then a more complex, stacking function. As I pointed out then, D3 actually implements a stack layout, which formats your data in such a way that it can be easily passed to d3.svg.area to draw a stacked graph or streamgraph.

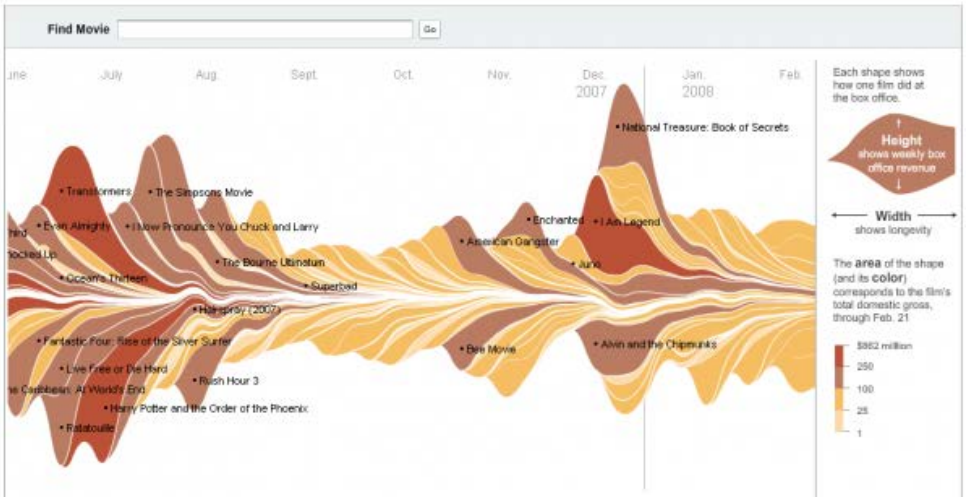


Figure 5.x The streamgraph used in a New York Times piece on movie grosses.

As you can guess, to implement this, we'll need to use the area generator in tandem with the stack layout. This general pattern should be familiar to you by now:

1. Process the data to match the requirements of the layout.
2. Set the accessor functions of the layout to align it with the dataset.
3. Use the layout to format the data for display.
4. Send the modified data either directly to SVG elements or paired with a generator like d3.svg.diagonal, d3.svg.arc, or d3.svg.area.

The first step is taking our original streamdata.csv data and transforming it into an array of movies objects that each have an array of values at points that correspond to the thickness of the section of the streamgraph that they represent:

Listing 5.x Stack layout example

```
d3.csv("streamdata.csv", function(error,data) {dataViz(data)});

  function dataViz(incData) {
    expData = incData;
    stackData = [];

    var xScale = d3.scale.linear()
    .domain([0, 10])
    .range([0, 500]);

    var yScale = d3.scale.linear()
    .domain([0, 100])
    .range([500, 0]);
```

```
    var movieColors =
d3.scale.category10(["movie1","movie2","movie3","movie4","movie5","movie6"]);

    var stackArea = d3.svg.area()
    .interpolate("basis")
    .x(function(d) { return xScale(d.x); })
    .y0(function(d) { return yScale(d.y0); })
    .y1(function(d) { return yScale(d.y0 + d.y); });

    for (x in incData[0]) {
      if (x != "day") { #a
        var newMovieObject = {name: x, values: []}; #b
        for (y in incData) {
          newMovieObject.values.push({x: parseInt(incData[y]["day"]) ,y:
parseInt(incData[y][x])}) #c
        }
        stackData.push(newMovieObject);
      }
    }

    stackLayout = d3.layout.stack()
    .offset("silhouette")
    .order("inside-out")
    .values(function(d) { return d.values; });

    d3.select("svg").selectAll("path")
    .data(stackLayout(stackData))
    .enter().append("path")
    .style("fill", function(d) {return movieColors(d.name)})
    .attr("d", function(d) { return stackArea(d.values); })
  }
```

#a We want to skip the day column, because in this case, the day will become our x-value
#b For each movie, we create an object with an empty array named "values"
#c Fill the "values" array with objects that list the x-coordinate as the day and the y-coordinate as the amount of money made by a movie on that day

The results of this reformatting of the initial dataset is an object array with its data structured in a way that the stack layout is designed to deal with:

```
[
{"name":"movie1","values":[{"x":1,"y":20},{"x":2,"y":18},{"x":3,"y":14},{"x":
4,"y":7},{"x":5,"y":4},{"x":6,"y":3},{"x":7,"y":2},{"x":8,"y":0},{"x":9,"y":0
},{"x":10,"y":0}]},
{"name":"movie2","values":[{"x":1,"y":8},{"x":2,"y":5},{"x":3,"y":3},{"x":4,"
y":3},{"x":5,"y":3},{"x":6,"y":1},{"x":7,"y":0},{"x":8,"y":0},{"x":9,"y":0},{
"x":10,"y":0}]}
...
```

In this case, we're looking at the x-value as the day and the y-value as the amount of money made by the movie that day, which will correspond to thickness. As with other layouts, if you didn't format your data this way, you'd need to adjust the .x() and .y() accessor to match your data names for those values. One of the benefits of formatting your data to match the expected data model of the layout is that it makes for a very simple layout function:

```
stackLayout = d3.layout.stack()
.values(function(d) { return d.values; }); #a
```

#a notice we're function chaining on the newly created stack() layout function

The results of our stackLayout function processing our dataset are available when we run stackLayout(stackData). The layout creates x, y and y0 functions corresponding to the top and bottom of the object at the x-position. If you're using the stack layout to create a streamgraph as we are, then it requires a corresponding area generator.

```
var stackArea = d3.svg.area()
.x(function(d) { return xScale(d.x); }) #a
.y0(function(d) { return yScale(d.y0); })
.y1(function(d) { return yScale(d.y0 + d.y); });
```

#a We almost always need to pass the data at some point to a scale function in order to fit it to the screen

Once we have our data, our layout, and our area generator in order, we can call them all as part of the selection and binding process to give a set of SVG <path> elements the necessary shapes to make our chart.

```
d3.select("svg").selectAll("path")
.data(stackLayout(stackData)) #a
.enter()
.append("path")
.style("fill", function(d) {return movieColors(d.name)}) #b
.attr("d", function(d) { return stackArea(d.values); }) #c
```

#a Remember that the data being bound is stackData processed by stackLayout()
#b A color scale that associates a unique color with each object in the array
#c The actual SVG drawing code comes from the area generator taking the values from our data processed by the layout

The result, as seen in figure 5.x, isn't a streamgraph but rather a stacked area chart, which isn't that different from a streamgraph, as we'll soon find out.

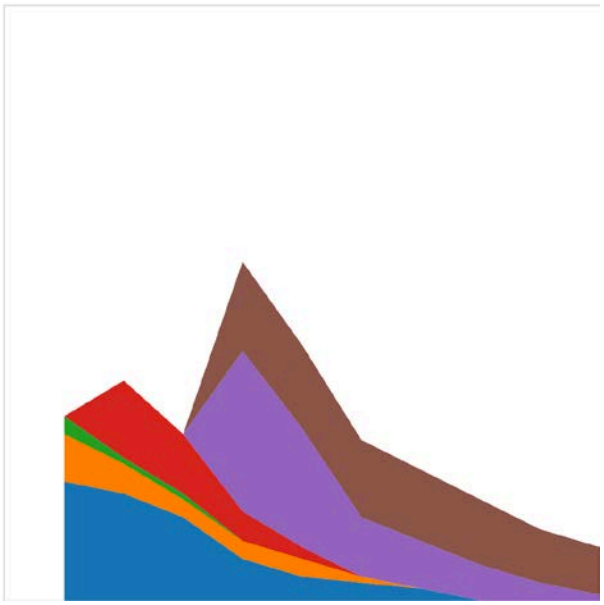


Figure 5.x The stack layout default settings, when tied to an area generator, will produce a stacked area chart like this one.

The stack layout has an .offset() function that determines where the areas that make up the chart are drawn in relation to. While you can write your own offset functions to create really exotic charts, it recognizes few keywords that allow the layout to achieve the typical effects you're looking for. We'll use the "silhouette" keyword, which centers the drawing of the stacked areas around the middle. Another function useful for creating streamgraphs is the .order() function of a stack layout, which determines the order in which areas are drawn, so that you can alternate them in the way that streamgraphs tend to do. We'll use "inside-out" because that gives the best streamgraph effect. The last change is to the area constructor, which we'll update to use the "basis" interpolator since that gave the best look in our earlier streamgraph example.

```
stackLayout.offset("silhouette").order("inside-out");
stackArea.interpolator("basis");
```

The results are a cleaner streamgraph than our example from Chapter 4:

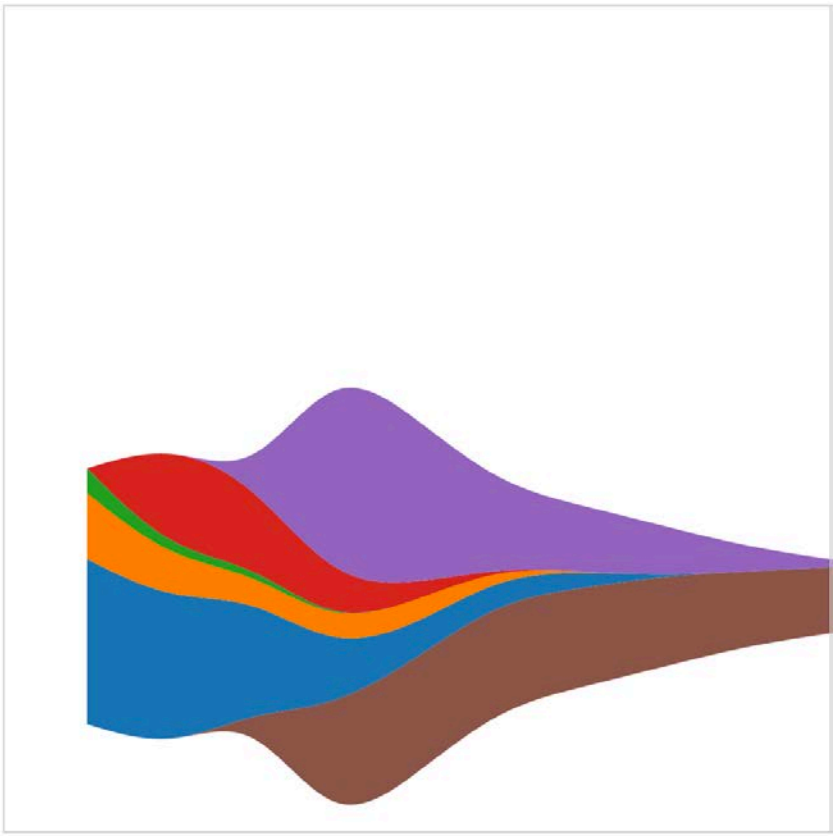


Figure 5.x The streamgraph effect from a stack layout with basis interpolation for the areas and using the "silhouette" and "inside-out" settings for the stack layout. This is similar to our hand-built example from Chapter 4 and shows the same graphical artifacts from the basis interpolation.

The last time we made a streamgraph, we explored the question of whether or not it was a useful chart. It is, for various reasons, not least of which is because the area in the chart corresponds graphically to the aggregate phenomena of each movie.

But sometimes a simple stacked bar graph is better. Layouts can be used for various types of charts, and the stack layout is no different. If we restore the .offset() and .order() back to the default settings, we can use the stack layout to create a set of rectangles that makes a traditional stacked bar chart.

```
stackLayout = d3.layout.stack()
.values(function(d) { return d.values; });

 var heightScale = d3.scale.linear()
 .domain([0, 70])
 .range([0, 480]);
```

```
d3.select("svg").selectAll("g.bar")
.data(stackLayout(stackData))
.enter()
.append("g")
.attr("class", "bar")
.each(function(d) {

d3.select(this).selectAll("rect")
.data(d.values)
.enter()
.append("rect")
.attr("x", function(p) { return xScale(p.x) - 15; })
.attr("y", function(p) { return yScale(p.y + p.y0); })
.attr("height", function(p) { return heightScale(p.y); })
.attr("width", 30)
.style("fill", movieColors(d.name))
})
```

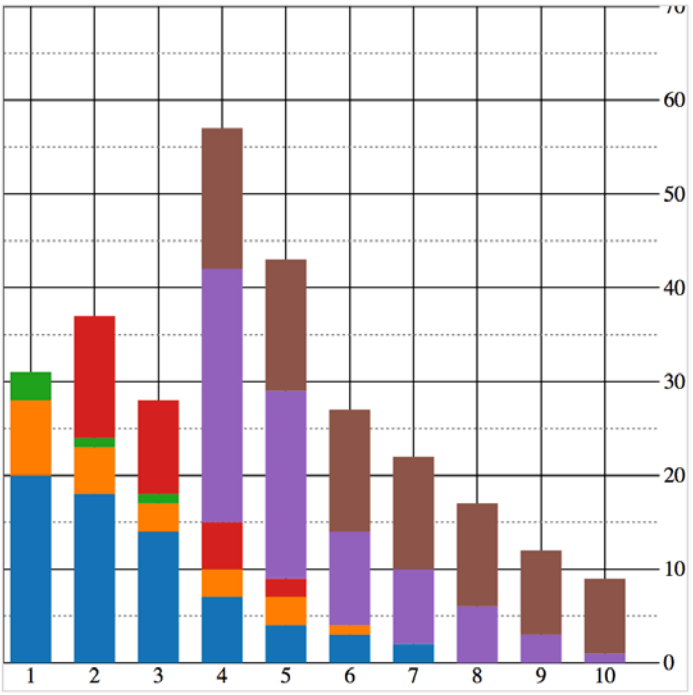


Figure 5.x A stacked bar chart using the stack layout to determine the position of the rectangles that make up each day's stacked bar.

In many ways, the stacked bar chart is much more readable than the streamgraph, presenting the same information but easily annotated with a y-axis that tells you exactly how much money a movie made. There's a reason why bar charts, line charts and pie charts are the

standard chart types found in your spreadsheet. The key takeaway here is that a streamgraph and a stacked bar chart and a stacked area chart are fundamentally the same thing, and rely on the stack layout to format your dataset in such a way as to draw them. Since you can deploy them equally easily, your decision whether or not to use one or the other can be based on user testing rather than your ability to create awesome dataviz.

The layouts we've looked at so far, as well as the associated methods and generators, have broad applicability. Now we'll look at a pair of layouts that don't come with D3 that are designed for more specific kinds of data: the Sankey Diagram and the Word Cloud. Even though these layouts aren't as generic as the layouts included in the core D3 library that we've looked at so far, they have some prominent examples and can come in handy.

5.6 Plugins

The examples we've touched on in this chapter are just a few of the layouts that come with the core D3 library. We'll see a few more in later chapters, and focus specifically on the force layout in Chapter 6. But layouts exist outside of core D3 that may be useful to you. These layouts tend to use very specifically formatted datasets or different terminology for layout functions.

5.6.1 Sankey Diagram

The Sankey diagram provides you with the ability to map flow from one category to another. It's the kind of diagram used in Google Analytics (Figure 5.x) to show you event flow or user flow from one part of your website to another. Sankey diagrams consist of two types of objects: nodes and edges. In this case, the nodes are the actual web pages or events, while the edges are the traffic between them. This is different from the hierarchical data we worked with before because nodes can have many, overlapping connections.

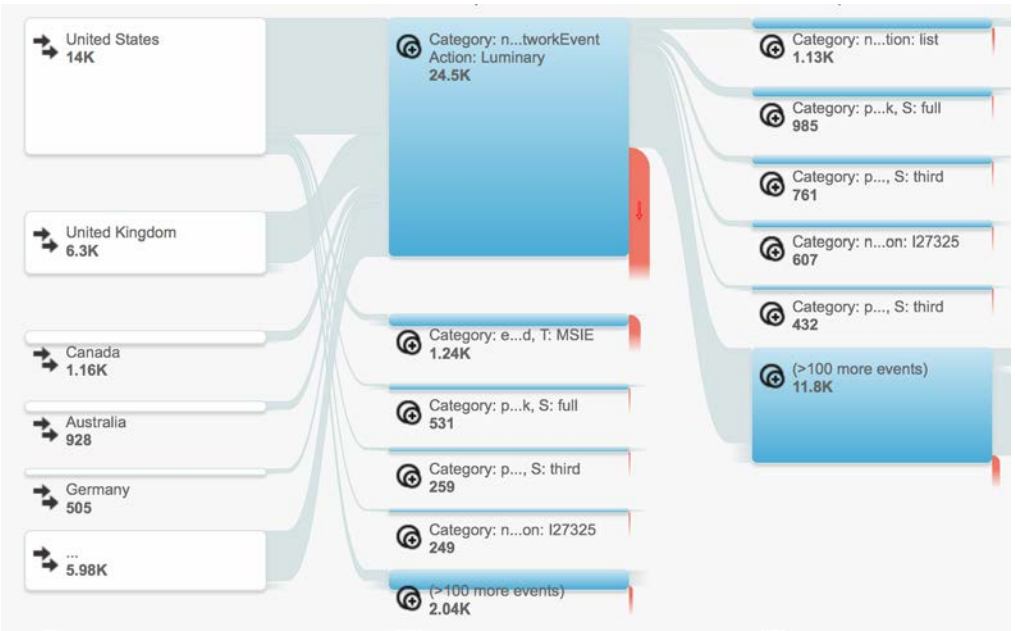
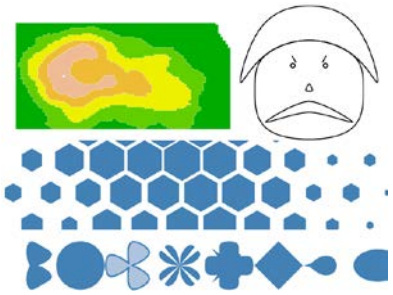


Figure 5.x Google Analytics uses Sankey Diagrams to chart event and user flow for website visitors.

The D3 version of the Sankey layout is a plugin written by Mike Bostock a couple years ago, which you can find at https://github.com/d3/d3-plugins along with some other interesting D3 plugins. The Sankey layout has a couple examples and sparse documentation—one of the drawbacks of non-core layouts. Another minor drawback is that they don't always follow the patterns of the core layouts in D3. To understand the Sankey layout, you need to examine the format of the data, the examples, and the code itself.

D3 PLUGINS



The core d3.js library that you download comes with quite a few layouts and useful functions, but you can find even more at https://github.com/d3/d3-plugins. Besides the two non-core layouts we look in this chapter, we'll look at the geo plugins in Chapter 7 when we deal with maps. Also available is a fisheye distortion lens, a canned boxplot layout, a layout for horizon charts, and even more exotic plugins for Chernov faces and superformula.

The data takes the form of a JSON array of nodes and a second JSON array of links. Get used to this format, since it's how most of the network data we're going to use in Chapter 6 is formatted. For our example, we're going to look at the traffic flow in a website that sells milk and milk-based products. We want to see how visitors move through the site from the homepage to the store page to the various product pages. In the parlance of the data format we need to work with, the web pages are nodes and the number of visitors that go from one page to another (if any) represent the links and the value of the links (total number of visitors that move from that page to the next).

Listing 5.x sitestats.json

```
{"nodes":[ #a
{"name":"index"},
{"name":"about"},
{"name":"contact"},
{"name":"store"},
{"name":"cheese"},
{"name":"yoghurt"},
{"name":"milk"}
],
"links":[
{"source":0,"target":1,"value":25}, #b
{"source":0,"target":2,"value":10},
{"source":0,"target":3,"value":40},
{"source":1,"target":2,"value":10},
{"source":3,"target":4,"value":25},
{"source":3,"target":5,"value":10},
{"source":3,"target":6,"value":5},
{"source":4,"target":6,"value":5},
{"source":4,"target":5,"value":15}
]
}
```

#a Each entry in this array represents a web page
#b Each entry in this array represents the number of times someone navigated from the "source" page to the "target" page

In reading Listing 5.x, the nodes array is pretty clear--each object represents a web page. The links array is a bit more opaque, until you realize that the format of the data is that the numbers represent the array position of nodes in the node array. So, when links[0] reads "source": 0, it means that the source is nodes[0], which is the "index" page of the site, and it connects to nodes[1], the "about" page, and indicates that 25 people navigated from the home page to the about page. That defines our flow—the flow of traffic through a site.

The Sankey layout is initialized like any layout:

```
var sankey = d3.sankey()
   .nodeWidth(20) #a
   .nodePadding(200) #b
   .size([460, 460])
   .nodes(data.nodes)
   .links(data.links)
   .layout(200); #c
```

#a Where to start and stop drawing the flows between nodes
#b The distance between nodes vertically, a lower value will create longer bars representing our web pages
#c The number of times to run the layout to try to optimize placement of of flows

We've only seen .size() before, which controls the graphical extent that the layout will use. The rest you'd need to figure out by looking at the example, experimenting with different values, or reading the actual sankey.js code itself. Most of it will quickly make sense, especially if you're familiar with the .nodes() and .links() convention which is used in network visualizations in D3. The .layout() setting is pretty hard to understand without diving into the code, but I'll explain that below.

Once we have our Sankey layout defined, we need to draw the chart by selecting and binding the necessary SVG elements. In this case, that typically consists of <rect> elements for the nodes and <path> elements for the flows, to which we'll also add some <text> elements to label the nodes.

Listing 5.x Sankey drawing code

```
   var intensityRamp = d3.scale.linear().domain([0,d3.max(data.links,
function(d) {return d.value})]).range(["black", "red"])

 d3.select("svg").append("g").attr("transform",
"translate(20,20)").attr("id", "sankeyG"); #a

 d3.select("#sankeyG").selectAll(".link")
     .data(data.links)
   .enter().append("path")
     .attr("class", "link")
     .attr("d", sankey.link()) #b
     .style("stroke-width", function(d) {return d.dy}) #c
     .style("stroke-opacity", .5)
     .style("fill", "none")
     .style("stroke", function(d){return intensityRamp(d.value)}) #d
     .sort(function(a, b) { return b.dy - a.dy; })
     .on("mouseover", function() {d3.select(this).style("stroke-opacity",
.8)}) #e
     .on("mouseout", function() {d3.selectAll("path.link").style("stroke-
opacity", .5)})

 d3.select("#sankeyG").selectAll(".node")
     .data(data.nodes)
   .enter().append("g")
     .attr("class", "node")
     .attr("transform", function(d) { return "translate(" + d.x + "," + d.y +
")"; }); #f
```

```
d3.selectAll(".node").append("rect")
    .attr("height", function(d) { return d.dy; })
    .attr("width", 20)
    .style("fill", "pink")
    .style("stroke", "gray")

d3.selectAll(".node").append("text")
    .attr("x", 0)
    .attr("y", function(d) { return d.dy / 2; })
    .attr("text-anchor", "middle")
    .text(function(d) { return d.name; })
```

#a Offset the parent <g> of the entire chart
#b The sankey layout's .link() function is a path generator
#c Note that the layout expects you to use a thick stroke and not a filled area
#d We set the stroke color using our intensity ramp, black to red indicating weak to strong
#e Emphasize the link when you mouse over it by making it less transparent
#f Node position is calculated as x and y coordinates on your data

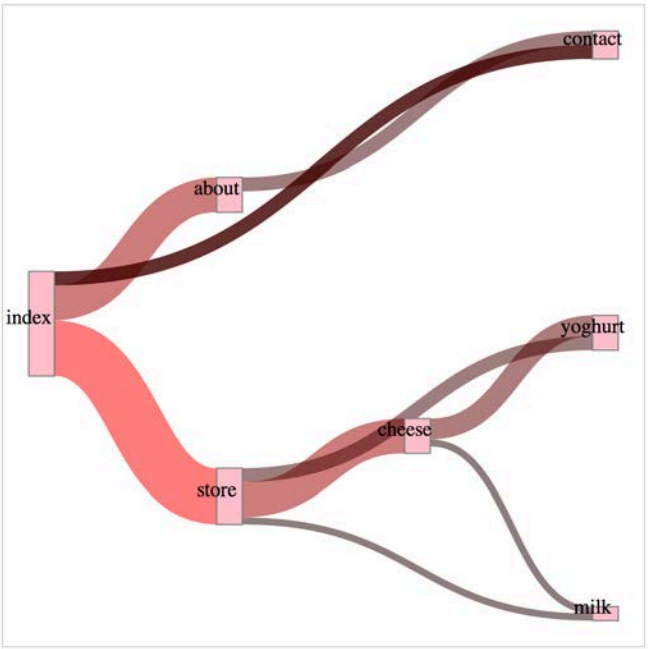


Figure 5.x A simple Sankey diagram where the number of visitors is represented in the color of the path. The flow between index and contact has an increased opacity as the result of a mouseover event.

There's some interactivity in the implementation of this layout as seen in Figure 5.x, and not just to remind us that charts can be interactive. Diagrams like these, with wavy paths

overlapping other wavy paths, need interaction to make them legible to your site visitor. In this case, it's to differentiate one flow from another.

With a Sankey diagram like this at your disposal, you can track the flow of goods, visitors, or anything else through your organization, website, or other system. While we could expand on this example in any number of ways, I think one of the most useful is one of the most simple. Remember, layouts are not tied to particular shape elements. In some cases, like with the flows in the Sankey diagram, you'll have a hard time adapting the layout data to any element other than a <path> but the nodes don't need to be <rect> elements. If you adjust your code a bit, you can easily make nodes that are circles.

```
 sankey.nodeWidth(1);

d3.selectAll(".node").append("circle")
    .attr("height", function(d) { return d.dy; })
    .attr("r", function(d) { return d.dy / 2; })
    .attr("cy", function(d) { return d.dy / 2; })
    .style("fill", "pink")
    .style("stroke", "gray")
```

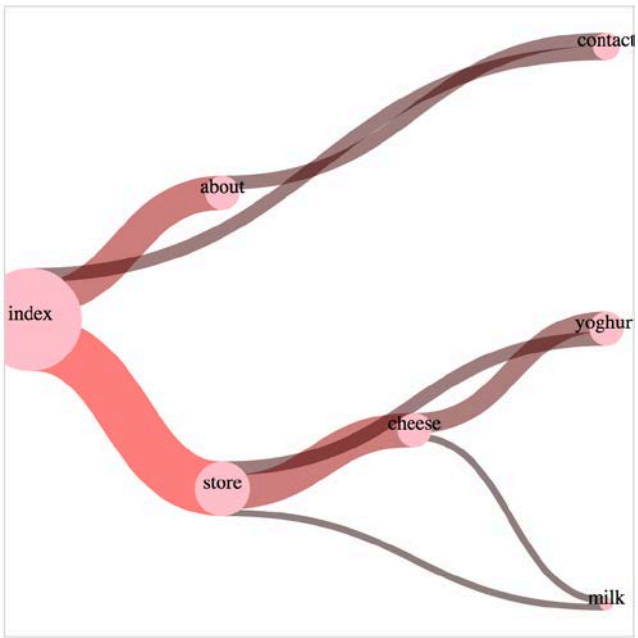


Figure 5.x A squid-like Sankey diagram

Don't shy away from experimenting with tweaks to "traditional" charting methods. Circles instead of rectangles may seem frivolous, but it might better fit visually or distinguish your Sankey from all the boring sharp-edged Sankeys out there. By the same vein, don't be afraid of

leveraging D3's capacity for information visualization to teach yourself how a layout works. You'll remember that d3.layout.sankey has a layout() function and that you might discover the operation of that function by reading the code. But there's another way for you to see how this function works: By using transitions and creating a function that updates the .layout() property dynamically, we can see what this function does to the chart graphically.

> **VISUALIZING ALGORITHMS** While you might think of all the graphics in this book as data visualization, it's also simultaneously a graphical representation of the methods you used to process the data. In some cases like the Sankey diagram here or the force-directed network visualization we'll see in the next chapter, the algorithm used to sort and arrange the graphical elements is front and center. Once you have your layout displaying properly, you can play with the settings and update the elements like we've done with the Sankey diagram to better understand how the algorithm works visually.

First we need to add an onclick function to make the chart interactive. I'm going to attach this function to the <svg> element itself, but you could just as easily add a button like we did in Chapter 3.

The moreLayouts() function will do two things:

- Update the sankey.layout() property by incrementing a variable and setting it to the new value of that variable.
- Select the graphical elements that make up our chart (the <g> and <path> elements) and re-draw them with the updated settings. By using transition() and delay(), we'll see the chart dynamically adjust.

Listing 5.x Visual layout function for the Sankey diagram

```
var numLayouts = 1;
d3.select("svg").on("click", moreLayouts);
sankey.layout(numLayouts); #a

function moreLayouts() {
   numLayouts += 20; #b
   sankey.layout(numLayouts);

   d3.selectAll(".link")
   .transition()
   .duration(500)
   .attr("d", sankey.link()) #c

   d3.selectAll(".node")
   .transition()
   .duration(500)
   .attr("transform", function(d) { return "translate(" + d.x + "," + d.y +
")"; });
}
```

#a We initialize the sankey with only a single layout pass
#b I chose 20 passes because it shows some change without requiring you to click too much

#c Because the layout updates the dataset, we just have to call the drawing functions again and they automatically update

The end result is you visually experience the effect of the .layout() function, which determines the number of passes that d3.layout.sankey makes to determine the best position of the lines representing flow. You can see some snapshots of this in Figure 5.x showing the lines sort out and get out of each other's way. This kind of optimizing of position is a common technique in information visualization, and drives the force-directed network layout that we'll see in Chapter 6. In the case of our Sankey example, even one pass of the layout provides pretty good positioning. That's because this is a simple dataset, and it stabilizes pretty quickly. As you can see as you click on your chart and in Figure 5.x, the layout doesn't change much with progressively higher numbers of passes in the layout() setting.

Figure 5.x The Sankey layout algorithm attempts to optimize the positioning of nodes to reduce overlap. Here we see the chart reflecting the position of nodes after (from left to right) 1 pass, 20 passes, 40 passes, and 200 passes.

It should be clear by this example that updating the settings of the layout afford you the opportunity to update the visual display of the layout, and using animations and transitions is, by design, as simple as calling the elements as we've done here and setting their drawing code or position to reflect the changed data. We'll see much more of this in later chapters.

5.6.2 Word Clouds

One of the most popular information visualization charts is also one of the most maligned: the word cloud. Also known as a tag cloud, the word cloud uses text and text size to represent the importance or frequency of words. Figure 5.x shows a thumbnail gallery of 50 word clouds derived from text in a species biodiversity database. Oftentimes, word clouds rotate the words to set them at right angles or jumble them at random angles to improve the appearance of the graphics. Word clouds, like streamgraphs, receive a lot of criticism for being hard to read or presenting too little information. However, they are both surprisingly popular with audiences.

Figure 5.x A word or tag cloud uses the size of a word to indicate its importance or frequency in a text, allowing for a visual summary of text. These word clouds were created by the popular online word cloud generator Wordle.

The word clouds in figure 5.x were created with the popular Java applet Wordle, which provides an easy user interface and a few aesthetic customization choices. Because it let anyone create visually arresting but problematic graphics simply by dropping text onto a page, it flooded the Internet with word clouds. This caused much consternation among data visualization experts, who think word clouds are evil because they embed in the visualization no analysis and only highlight superficial data such as the quantity of words in a blog post.

But word clouds are not evil. First of all, they're popular with audiences. But more than that, words are remarkably effective graphical objects and so if you can identify a numerical attribute that indicates the significance of a word, then scaling the size of a word in a word cloud will relay that significance to your reader.

So let's start by assuming we have the right kind of data for a word cloud. Fortunately, I do: the top twenty words used in this chapter, with the number of each word.

Listing 5.x worddata.csv

```
text,frequency
layout,63
function,61
data,47
return,36
attr,29
chart,28
array,24
style,24
layouts,22
values,22
need,21
nodes,21
pie,21
use,21
figure,20
circle,19
we'll,19
zoom,19
append,17
elements,17
```

To create a word cloud with D3, we have to use another layout that isn't in the core library, created by Jason Davies (who created the sentence trees using the tree layout seen in Figure 5.x) and implementing an algorithm written by Jonathan Feinberg (http://static.mrfeinberg.com/bv_ch03.pdf). This layout, d3.layout.cloud(), is available on GitHub at https://github.com/jasondavies/d3-cloud. The layout requires that you define what attribute will determine word size and what size you want the word cloud to lay out for.

Unlike most other layouts, cloud() fires a custom event "end" that indicates it's done calculating the most efficient use of space to generate the word cloud, to which it passes the processed dataset with the position, rotation and size of the words. Because of this, we can run the cloud layout without ever referring to it again, and don't even need to assign it to a variable. Of course, if you plan to reuse it and adjust the settings, you would assign it to a variable just like we did any other layout.

```
wordScale=d3.scale.linear().domain([0,75]).range([10,160]); #a

 d3.layout.cloud()
  .size([500, 500])
  .words(data) #b
  .fontSize(function(d) { return wordScale(d.frequency); }) #c
  .on("end", draw)
  .start(); #d

  function draw(words) { #e

  var wordG = d3.select("svg").append("g").attr("id",
"wordCloudG").attr("transform","translate(250,250)");

  wordG.selectAll("text")
  .data(words)
```

```
        .enter()
        .append("text")
        .style("font-size", function(d) { return d.size + "px"; })
          .style("opacity", .75)
          .attr("text-anchor", "middle")
          .attr("transform", function(d) {
            return "translate(" + [d.x, d.y] + ")rotate(" + d.rotate + ")";
          }) #f
          .text(function(d) { return d.text; });
        }
```

#a A scale for the font rather than using raw values
#b You assign data to the cloud layout using .words()
#c Setting the size of each word using our scale
#d The cloud layout needs to be initialized, when it's done it will fire "end" and run whatever function "end" is associated with
#e We've assigned draw() to "end", which automatically passes the processed dataset as the words variable
#f Translation and rotation are calculated by the cloud layout

The results of this code is to create an SVG <text> element that is rotated and placed according to the code. None of our words are rotated, so we get a pretty staid word cloud seen in Figure 5.x.

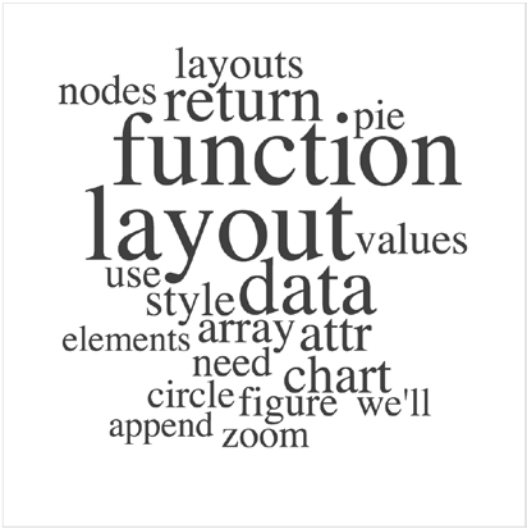


Figure 5.x A simple word cloud with words that are arranged horizontally.

Defining rotation, though, is simple enough, and only requires that you set some rotation value in the cloud layout's .rotate() function:

```
randomRotate=d3.scale.linear().domain([0,1]).range([-20,20]); #a

 d3.layout.cloud()
```

```
    .size([500, 500])
    .words(data)
    .rotate(function() {return randomRotate(Math.random())} ) #b
    .fontSize(function(d) { return wordScale(d.frequency); })
    .on("end", draw)
    .start();
```

#a This scale will take a random number between 0 and 1 and return an angle between -20 degrees and 20 degrees
#b Set the rotation for each word

Figure 5.x A word cloud using the same worddata.csv but with slightly perturbed words created by randomizing the rotation property of each word.

At this point, you have your traditional word cloud, and you can tweak the settings and colors to create anything you've seen on wordle. But now that we have our word cloud, and I've noted that word clouds aren't evil, let's take a look at why word clouds get such a bad reputation. We've taken an interesting dataset, the most common words in this chapter, and other than size them by their frequency, we've done little more than place them on screen and jostle them a bit. Remember that there are different channels for expressing data visually, and in this case the best channels that we have, besides size, are color and rotation.

With that in mind, let's imagine that we have a keyword list for this book, and that each of these words is in a glossary in the back of the book. We'll place those keywords in an array and use them to highlight the words in our word cloud that appear in the glossary. We'll also rotate shorter words 90 degrees and leave the longer words unrotated so that they'll be easier to read.

Listing 5.x Word cloud layout with key word highlighting

```
var keywords = ["layout", "zoom", "circle", "style", "append", "attr"] #a
```

```
    d3.layout.cloud()
      .size([500, 500])
      .words(data)
      .rotate(function(d) { return d.text.length > 5 ? 0 : 90; }) #b
      .fontSize(function(d) { return wordScale(d.frequency); })
      .on("end", draw)
      .start();

      function draw(words) {

      var wordG = d3.select("svg").append("g").attr("id",
"wordCloudG").attr("transform","translate(250,250)");

      wordG.selectAll("text")
      .data(words)
      .enter()
      .append("text")
      .style("font-size", function(d) { return d.size + "px"; })
        .style("fill", function(d) { return (keywords.indexOf(d.text) > -1 ?
"red" : "black"); }) #c
        .style("opacity", .75)
        .attr("text-anchor", "middle")
        .attr("transform", function(d) {
          return "translate(" + [d.x, d.y] + ") rotate(" + d.rotate + ")";
        })
        .text(function(d) { return d.text; });
      }
```

#a Our array of keywords
#b The rotate function makes every word with 6 or more characters have a rotation of 90 degrees
#c If the word appears in the keyword list, color it red, otherwise color it black

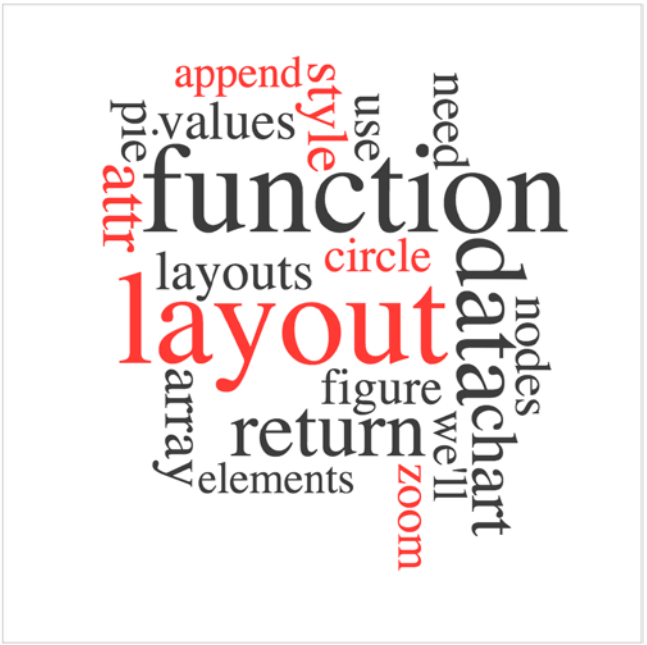


Figure 5.x Word cloud highlighting key words and placing longer words horizontally and shorter words vertically.

The result seen in figure 5.x is fundamentally the same word cloud, but instead of using color and rotation for aesthetics, we used them to encode information in the dataset. There are more controls over the format of your word cloud that you can see in the layout's documentation at https://www.jasondavies.com/wordcloud/about/ including selecting fonts and padding.

Layouts like word cloud aren't suitable for as wide a variety of data as some other layouts, but because they're so easy to deploy and customize, you can combine them with other charts to represent the multiple facets of your data. We'll see this kind of synchronized charts in chapter 9.

5.7 Summary

In this chapter, we took an in-depth look at D3 layout structure and experimented with several datasets. In doing so, we learned how to use layouts not just to draw one particular chart, but variations on that chart. We also experimented with interactivity and animation

In particular, we learned:

- Layout structure and functions common to D3 core layouts
- Arc and diagonal generators for drawing arcs and connecting links

- How to make pie charts and donut charts using the pie layout
- Using tweens to better animate the graphical transition for arc segments (pie pieces)
- How to create circle packing diagrams and format them effectively using the pack layout
- How to create vertical, horizontal and radial dendrograms using the tree layout
- How to create stacked area charts, streamgraphs and stacked bar charts using the stack layout
- How to use non-core D3 layouts to build Sankey diagrams and word clouds

Now that you understand layouts in general, in the next chapter, we're going to focus on representing networks. Among the methods we'll look at, we're going to spend the most time dealing with the force-directed layout, which has much in common with general layouts but is distinguished from them because it's designed to be interactive and animated. Because the chapter deals with network data, like the kind you used for the Sankey layout in this chapter, you'll also learn a few tips and tricks for processing and measuring networks.

6
Network Visualization

With the growth of online social networks like Twitter and Facebook, as well as social media and linked data more generally in what was known as Web 2.0, network analysis and network visualization has become increasingly common. Networks are particularly interesting because they focus on how things are related, and so represent systems more than data.

This chapter is focused on representing networks, so it's important that we understand some basic network terminology. In general, when dealing with networks we refer to the things (like people) being connected as nodes and the connections (such as being a friend on Facebook) between them as edges or links. Sometimes you might hear nodes referred to as vertices, since that's where to edges join. While it might seem useful here to have a figure with nodes labeled and edges labeled, one of the lessons from this chapter is that there is no one way to represent a network. A network may also be referred to as a graph, since this is what they are called in mathematics. Finally, the measure of the importance of a node in a network is typically referred to as centrality. There's more, but that should be enough to get you started.

Networks are not just a data format, they are a perspective on data. When you are working with network data, you typically are trying to discover and display patterns of the network or of parts of the network, and not of individual nodes in the network. While you might use a network visualization because it makes for a cool graphical index, like a mind map or a network map of a website, in general you'll find that the typical information visualization techniques are designed to showcase network structure, and not individual nodes.

6.1 Static Network Diagrams

Network data is different from hierarchical data in that networks present the possibility of many-to-many connections like the Sankey layout from Chapter 5, whereas in hierarchical data a node can have many children but only one parent like the tree and pack layouts from Chapter 5. A network does not have to be a social network, and many different structures are amenable to being representing using this format, such as transportation networks and linked open data.

In this chapter, we'll look at four common forms for representing networks: as data, as adjacency matrices, as arc diagrams and especially using force-directed network diagrams.

In each case, the actual graphical representation will be quite different. For instance, in the case of a force-directed layout, we'll represent the nodes as circles and the edges as lines. But in the case of the adjacency matrix, nodes with be positioned on an x- and y- axis and the edges will be filled squares. There is no default way to represent a network, but the examples we'll see in this chapter are the most common.

6.1.1 Network Data

While there are several data formats that networks can be stored in, the most straightforward to work with is known as the edge list. An edge list is typically represented as a CSV like that seen in Listing 6.1 with a source column and a target column using a string or number to indicate which nodes are connected. Each edge may also have other attributes, indicating the type of connection or its strength, or the time period when the connection is valid, or its color or any other information you want to store about a connection. The important thing to recognize is that only the source and target columns are necessary.

In the case of directed networks, the source and target columns indicate the direction of connection between nodes. A directed network just means that nodes may be connected in one direction but not in the other, so for instance you could follow a user on Twitter but that doesn't necessarily mean that they follow you. Undirected networks still typically have the columns listed as "source" and "target" but the connection is the same in both directions, so for instance if your network was made up of connections that indicated people had shared classes then if I was in a class with you then you were likewise in a class with me. We'll see directed and weighted networks represented throughout this chapter.

Listing 6.1 edgelist.csv

```
source,target,weight
sam,pris,1
roy,pris,5
roy,sam,1
tully,pris,5
tully,kim,3
tully,pat,1
tully,mo,3
kim,pat,2
kim,mo,1
mo,tully,7
mo,pat,1
mo,pris,1
pat,tully,1
pat,kim,2
pat,mo,5
lee,al,3
```

Our network also has a "weight" value for the connections, which indicates the strength of connections. In our case, our edge list represents how many times the source favorited the

tweets of the target. So, Sam favorited one tweet made by Pris and Roy favorited 5 tweets made by Pris, and so on. This is a "weighted network" because our edges have a value, and it's a directed network because our edges have direction, so we have a weighted directed network, and need to account for both weight and direction in our network visualizations.

Technically, you only need an edge list to create a network, since you can derive a list of nodes from the unique values in the edge list. This is done by traditional network analysis software packages like Gephi, and while you can derive a node list with JavaScript, it's more common that you will have a corresponding node list that provides more information about the nodes in your network, like we have in Listing 6.2.

Listing 6.2 nodelist.csv

```
id,followers,following
sam,17,500
roy,83,80
pris,904,15
tully,7,5
kim,11,50
mo,80,85
pat,150,300
lee,38,7
al,12,12
```

Since these are Twitter users, then we have some more information about them based on their Twitter stats, in this case the number of followers and the number of people they follow. As with the edge list, it's not necessary to have more than an id, but having access to more data gives us the chance to modify our network visualization to reflect the node attributes.

How we represent a network depends on its size and the nature of the network. If a network does not represent discrete connections between similar things, but rather the flow of goods or information or traffic, then you could use a Sankey diagram like we did in Chapter 5. Recall that the data format for the Sankey is exactly the same as what we have here: a table of nodes and a table of edges. But while the Sankey Diagram is only suitable for specific kinds of network data, there are a few different chart types that are more generically useful for network data, the first is an adjacency matrix.

Before we gets started with code to create our network visualizations, let's put together a CSS page so that we can set color based on class and use inline styles as little as possible. Listing 6.3 gives the CSS necessary for all the examples in this chapter. Of course, there will still be some inline style setting going on when we want the numerical value of an attribute to relate to the data bound to that graphical element, such as making the stroke-width of a line be based on the strength of that line.

Listing 6.3 networks.css

```
.grid {
      stroke: black;
      stroke-width: 1px;
      fill: red;
```

```
}
.arc {
       stroke: black;
       fill: none;
}
.node {
      fill: lightgray;
      stroke: black;
      stroke-width: 1px;
}
circle.active {
            fill: red;
}
path.active {
            stroke: red;
}
```

6.1.2 Adjacency Matrix

As it becomes more and more common to see networks represented graphically, it seems like the only way to represent a network is with a circle or square that represents the node and some kind of line (whether straight or curvy) that represents the edge. As such, it might surprise you that one of the most effective network visualizations has no connecting lines at all. Instead, the adjacency matrix represents connections between nodes by using a grid.

The principle of an adjacency matrix is simple: you place the nodes along the x axis and then place the same nodes along the y axis. If two nodes are connected, then the corresponding grid square is filled, otherwise it's left blank. In our case, since it's a directed network, then the nodes along the y axis are considered the source and the nodes along the x axis are considered the target as we'll see in a few pages. Since our network is also weighted, then we'll use saturation to indicate weight, with lighter colors indicating a lower strength of connection and darker colors indicating a stronger connection.

The only problem with building an adjacency matrix in D3 is that there isn't an existing layout, which means we have to build it like hand like we did with the bar chart, scatterplot and boxplot. There's a really impressive example by Mike Bostock at http://bost.ocks.org/mike/miserables/ but we can make something that's pretty functional without too much code. In doing so, though, we need to process the two JSON arrays that are created from our CSVs and format the data so that it's easy to work with. This is very close to writing your own layout, something we'll do in Chapter 10, and a good idea generally.

Listing 6.4 The simple adjacency matrix function

```
    function adjacency() {
      queue() #a
      .defer(d3.csv, "nodelist.csv")
      .defer(d3.csv, "edgelist.csv")
      .await(function(error, file1, file2) { createAdjacencyMatrix(file1,
file2); });

      function createAdjacencyMatrix(nodes,edges) {
        var edgeHash = {}; #b
        for (x in edges) {
```

```
        var id = edges[x].source + "-" + edges[x].target;
        edgeHash[id] = edges[x];
      }
      matrix = [];
      for (a in nodes) {
        for (b in nodes) { #c
          var grid = {id: nodes[a].id + "-" + nodes[b].id, x: b, y: a,
weight: 0}; #d
          if (edgeHash[grid.id]) {
            grid.weight = edgeHash[grid.id].weight; #e
          }
          matrix.push(grid);
        }
      }

      d3.select("svg")
      .append("g")
      .attr("transform", "translate(50,50)")
      .attr("id", "adjacencyG")
      .selectAll("rect")
      .data(matrix)
      .enter()
      .append("rect")
      .attr("class", "grid")
      .attr("width", 25)
      .attr("height", 25)
      .attr("x", function (d) {return d.x * 25})
      .attr("y", function (d) {return d.y * 25})
 .style("fill-opacity", function (d) {return d.weight * .2})

      var scaleSize = nodes.length * 25;
      var nameScale = d3.scale.ordinal()
.domain(nodes.map(function (el) {return el.id})) #f
.rangePoints([0,scaleSize],1); #g

      xAxis = d3.svg.axis().scale(nameScale).orient("top").tickSize(4); #h
      yAxis = d3.svg.axis().scale(nameScale).orient("left").tickSize(4);
      d3.select("#adjacencyG").append("g").call(yAxis);
      d3.select("#adjacencyG").append("g").call(xAxis)
.selectAll("text")
.style("text-anchor", "end")
.attr("transform", "translate(-10,-10) rotate(90)"); #i

    }
  }
```

#a We need to load two datasets before we can get started, and queue let's us move the asynchronous loaders into a synchronous format
#b A hash will allow us to test if a source-target pair has a link
#c Create all possible source-target connections
#d Set the xy coordinates based on the source-target array positions
#e If there's a corresponding edge in our edge list, give it that weight
#f Create an ordinal scale from the node IDs
#g Used for ordinal values
#h Both axes use the same scale
#i Rotate the text on the y-axis

There are a few new things going on here in Listing 6.4. For one, we're using a new scale: d3.scale.ordinal, which takes in an array of distinct values and allows you to place them on an axis like we do with the names of our nodes in this example. This necessitates that we use a scale function that we haven't seen before, rangePoints, which associates each of those unique values with a value evenly divided out of the array given with each point having an offset declared in the second, optional variable. The other new piece of code is the use of queue.js, which we need because we're loading two CSV files and we don't want to run our function until those two CSVs are loaded. We're also building this matrix array of objects that may seem obscure but if you examine it in your console you'll see, as in Figure 6.1, it's just a list of every possible connection and the strength of that connection, if it exists.

Figure 6.1 The array of connections we're building. Notice that every possible connection is stored in the array and only those connections that exist in our dataset have a weight value other than 0. Notice, also that our CSV import has resulted in the weight value being a string.

The result seen in figure 6.2 is a simple adjacency matrix based on the node list and edge list.

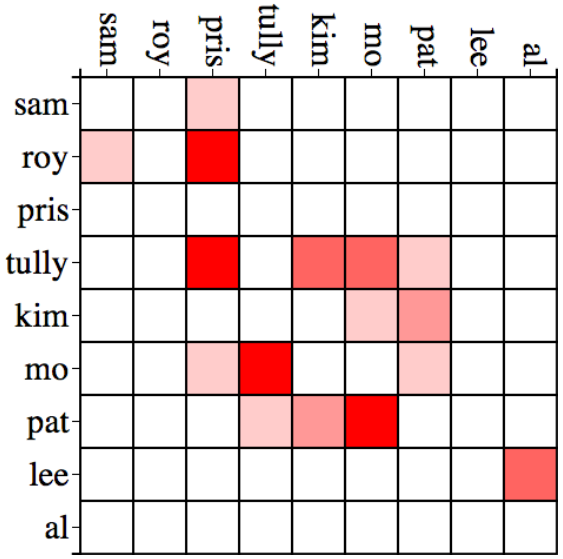


Figure 6.2 A weighted, directed adjacency matrix where lighter red indicates weaker connections and darker red indicates stronger connections. The source is on the y-axis and the target is on the x-axis, indicating that Roy favorited tweets by Sam but Sam did not favorite any tweets by Roy.

You'll notice in many adjacency matrices that the square indicating the connection from a node to itself is always filled. This is what's known in network parlance as a "self-loop" which occurs when a node is connected to itself. In our case, the definition of what we call a connection would mean that someone would need to favorite their own tweet, and fortunately no one in our dataset is a big enough loser to do that.

INTERACTIVITY

If we want, we can add some simple interactivity to help make it more readable. Grids can be hard to read without something to highlight the row and column of a square. It's quite simple to add highlighting to our matrix. All we have to do is add a mouseover event listener that fires a gridOver function which highlights all rectangles that have the same x or y value.

```
d3.selectAll("rect.grid").on("mouseover", gridOver);

      function gridOver(d,i) {
        d3.selectAll("rect").style("stroke-width", function (p) {return p.x
== d.x || p.y == d.y ? "3px" : "1px"})
      }
```

Now we can see in Figure 6.3 how moving your cursor over a grid square highlights the row and column of that grid square.

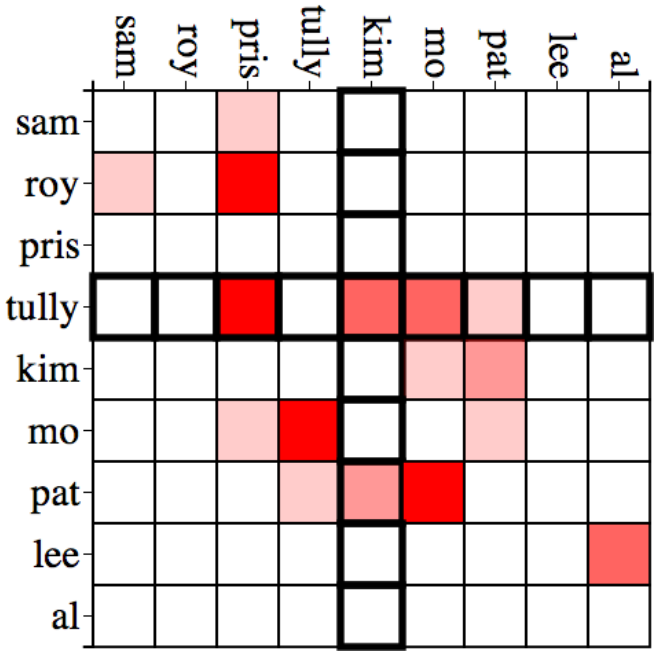


Figure 6.3 Adjacency highlighting column and row of the grid square. Here it is with the mouse over the Tully to Kim edge. We can see that Tully favorited tweets by four people, one of whom was Kim, and that Kim only had tweets favorited by one other person: Pat.

6.1.3 Arc Diagram

Another way to graphically represent networks is through the use of an Arc Diagram, which arranges the nodes along a line and draws the links as arcs above and/or below that line. Again, there isn't a handy layout available for arc diagrams, and there are even fewer examples, but the principle is actually rather simple once you see the code. This requires us to build another pseudo-layout like we did with the adjacency matrix, but this time we need to do some processing of the nodes as well as the links like we do in Listing 6.5.

Listing 6.5 Arc Diagram Code

```
    function arcDiagram() {
      queue()
      .defer(d3.csv, "nodelist.csv")
      .defer(d3.csv, "edgelist.csv")
      .await(function(error, file1, file2) { createArcDiagram(file1, file2);
});
      function createArcDiagram(nodes,edges) {

        var nodeHash = {};
        for (x in nodes) {
          nodeHash[nodes[x].id] = nodes[x]; #a
```

```
        nodes[x].x = parseInt(x) * 40; #b
      }
      for (x in edges) {
        edges[x].weight = parseInt(edges[x].weight);
        edges[x].source = nodeHash[edges[x].source]; #c
        edges[x].target = nodeHash[edges[x].target];
      }

      linkScale = d3.scale.linear()
.domain(d3.extent(edges, function (d) {return d.weight}))
.range([5,10])

      var arcG = d3.select("svg").append("g")
.attr("id", "arcG").attr("transform", "translate(50,250)");

      arcG.selectAll("path")
      .data(edges)
      .enter()
      .append("path")
 .attr("class", "arc")
 .style("stroke-width", function(d) {return d.weight * 2})
      .style("opacity", .25)
 .attr("d", arc) #d

      arcG.selectAll("circle")
      .data(nodes)
      .enter()
      .append("circle")
      .attr("class", "node")
      .attr("r", 10)
      .attr("cx", function (d) {return d.x}) #e

      function arc(d,i) {
        var draw = d3.svg.line().interpolate("basis");
        var midX = (d.source.x + d.target.x) / 2; #f
        var midY = (d.source.x - d.target.x) * 2;
        return draw([[d.source.x,0],[midX,midY],[d.target.x,0]])
      }
    }
  }
```

#a Create a hash that associates each node JSON object with its ID value
#b Set each node with an x-position based on its array position
#c Replace the string ID of the node with a pointer to the JSON object
#d Draw the links using the arc function
#e Draw the nodes as circles at each node's x-position
#f Draw a basis-interpolated line from the source node to a computed middle point above them to the target node

Notice that the edges array that we're building is just using a hash with the id value of our edges to create object references. By building objects that have references to the source and target nodes, it allows us to easily calculate the graphical attributes of the <line> or <path> element we're using to represent the connection. This is the same method used in the force layout that we'll look at later in the chapter. The results of our code is our first arc diagram seen in Figure 6.6.

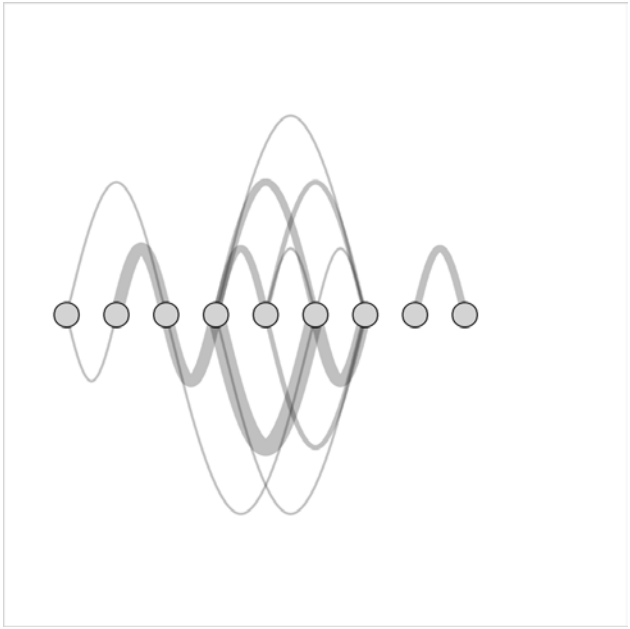


Figure 6.6 A simple arc diagram, with connections between nodes represented as arcs above and below the nodes. Arcs above the nodes indicate the connection is from left to right, while arcs below the nodes indicate the source is on the right and the target is on the left.

With abstract charts like these, we're getting to the point where interactivity is no longer optional. Even though the links follow simple rules, and we're not dealing with too many nodes or edges, it can be hard to make out what is connected to what and how. We can add some useful interactivity by making the edges highlight the connecting nodes on mouseover, as well as make the nodes highlight connected edges on mouseover by adding two new functions as seen in Listing 6.6 and with results seen in Figure 6.7.

Listing 6.6 Arc diagram interactivity

```
d3.selectAll("circle").on("mouseover", nodeOver);
d3.selectAll("path").on("mouseover", edgeOver);

      function nodeOver(d,i) {
        d3.selectAll("circle").classed("active", function (p) {return p == d
? true : false}) #a
        d3.selectAll("path").classed("active", function (p) {return p.source
== d || p.target == d ? true : false}) #b
      }

      function edgeOver(d) {
        d3.selectAll("path")
.classed("active", function(p) {return p == d ? true : false})
        d3.selectAll("circle")
```

```
.style("fill",
function(p) {return p == d.source ? "blue"
: p == d.target ? "green" : "lightgray"}) #c
}
    }
```

#a Make a selection of all nodes to set the class of the node being hovered over to “active”
#b Any edge where the selected node shows up as source or target render as red
#c This nested if checks to see if a node is the source, which is set to blue, or if it’s the target and set to green, or if it’s neither and set to gray

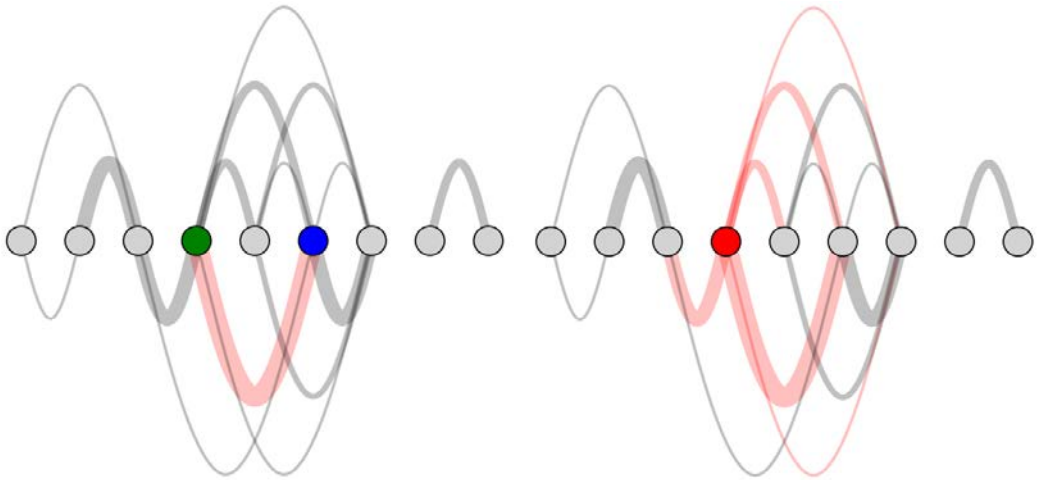


Figure 6.7 Mouseover behavior on edges (left) indicates the edge being moused over in pink, the source node in blue, and the target node in green. Mouseover behavior on nodes (right) indicates the node being moused over in red and the connected edges in pink.

If you’re interested in exploring arc diagrams further and want to use it for larger datasets, you’ll also want to look into hive plots, which are arc diagrams arranged on spokes. We won’t deal with hive plots in this book but there’s a plugin layout for hive plots that you can see at https://github.com/d3/d3-plugins/tree/master/hive. Both the adjacency matrix and arc diagram benefit from the control you have over how to sort the nodes and place them, as well as the linear manner in which they’re laid out. The next method for network visualization, which will remain our focus for the rest of the chapter, uses entirely different principles of determining how and where to place nodes and edges.

6.2 Force-Directed Layout

The force layout gets its name from the method by which it determines the most optimal graphical representation of a network. Like the word cloud and the Sankey diagram from Chapter 5, the force() layout dynamically updates the positions of its elements to try to find the best fit. Unlike those layouts, it does it continuously in real-time rather than as a pre-processing step before rendering. The basic principle behind a force layout is the interplay

between three forces, shown in Figure 6.8. These forces push nodes away from each other, attract connected nodes to each other, and keep nodes from flying out of sight.'

In this section, we'll learn how to make force-directed layouts, how they work, and some general principles from network analysis that help you better understand them. We'll also learn how to add and remove nodes and edges as well as adjust the settings of the layout on-the-fly.

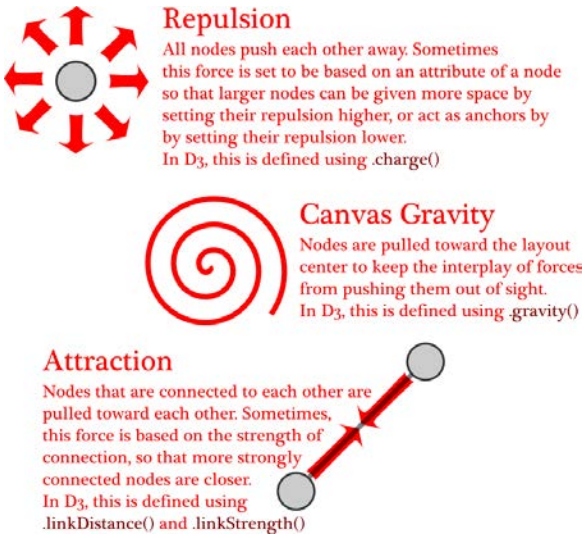


Figure 6.8 The basic forces in a force-directed algorithm: repulsion, gravity, and attraction. Other factors, such as hierarchical packing and community detection, can also be factored into force-directed algorithms, but these basic features are the most common. Forces are approximated for larger networks to improve performance.

6.2.1 Creating a Force-Directed Network Diagram

The force() layout you see initialized in Listing 6.x has some settings you've already seen before, the most obvious being size(), which takes an array indicating the width and height of your layout region and uses that to calculate the necessary force settings. The nodes() and links() settings are the same as what we saw when we used the Sankey layout in Chapter 5. They take, as you would expect, arrays of data that correspond to the nodes and links. We're creating our own source and target references in our links array, just like we did with the arc diagram, and that's the formatting that force() expects. It will also accept integer values where the integer values correspond to the array position of a node in the nodes array, like the formatting of data for the Sankey diagram links array from Chapter 5. As you can see in Listing 6.8, the one setting that is new is charge(), which determines how much each node pushes away other nodes. There's also a new event listener, "tick", that needs to get associated with a tick function that updates the position of our nodes and edges.

Listing 6.8 Force layout function

```
    function forceDirected() {

      queue()
      .defer(d3.csv, "nodelist.csv")
      .defer(d3.csv, "edgelist.csv")
      .await(function(error, file1, file2) { createForceLayout(file1, file2);
});
      function createForceLayout(nodes,edges) {
        var nodeHash = {};
        for (x in nodes) {
          nodeHash[nodes[x].id] = nodes[x];
        }
        for (x in edges) {
          edges[x].weight = parseInt(edges[x].weight);
          edges[x].source = nodeHash[edges[x].source];
          edges[x].target = nodeHash[edges[x].target];
        }

        var weightScale = d3.scale.linear()
.domain(d3.extent(edges, function(d) {return d.weight}))
.range([.1,1]);

        force = d3.layout.force()
        .charge(-1000) #a
        .size([500,500])
.nodes(nodes)
.links(edges)
.on("tick", forceTick); #b

        d3.select("svg").selectAll("line.link")
.data(edges, function (d) {return d.source.id + "-" + d.target.id}) #c
.enter()
        .append("line")
        .attr("class", "link")
        .style("stroke", "black")
        .style("opacity", .5)
        .style("stroke-width", function(d) {return d.weight});

        var nodeEnter = d3.select("svg").selectAll("g.node")
.data(nodes, function (d) {return d.id})
.enter()
.append("g")
.attr("class", "node");

        nodeEnter.append("circle")
        .attr("r", 5)
        .style("fill", "lightgray")
        .style("stroke", "black")
        .style("stroke-width", "1px");

        nodeEnter.append("text")
        .style("text-anchor", "middle")
        .attr("y", 15)
        .text(function(d) {return d.id})
```

```
        force.start(); #d

        function forceTick() {
        d3.selectAll("line.link")
        .attr("x1", function (d) {return d.source.x}) #e
        .attr("x2", function (d) {return d.target.x})
        .attr("y1", function (d) {return d.source.y})
        .attr("y2", function (d) {return d.target.y});

        d3.selectAll("g.node")
        .attr("transform", function (d) {return "translate("+d.x+","+d.y+")"})
        }
}
```

#a How much each node pushes away each other, if set to a positive value, nodes will attract each other
#b "tick" events are fired continuously, running the associated function
#c Key values for our nodes and edges will help when we update the network later
#d Initializing the network will start firing "tick" events as well as calculate the degree centrality of nodes
#e The tick function updates the edge drawing code and node drawing code based on the newly calculated node positions

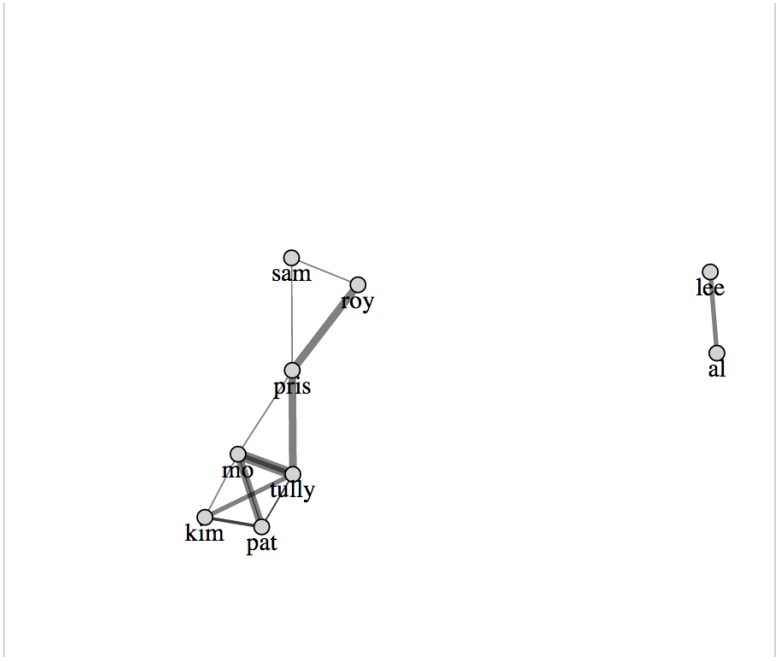


Figure 6.9 A force-directed layout based on our dataset and organized graphically using default settings in the force layout.

The animated nature of the force layout is lost on the page, but you can see in Figure 6.9 general network structure that is less prominent in an adjacency matrix or arc diagram. It's

readily apparent that there are four nodes (Mo, Tully, Kim and Pat) are all connected to each other (forming what is in network terms is called a “clique”) and three nodes (Roy, Pris, and Sam) that are more peripheral. Over on the right are two nodes (Lee and Al) that are only connected to each other. The only reason those nodes are still on-screen is because the layout’s gravity pulls unconnected pieces toward the center.

The thickness of the lines corresponds to the strength of connection. But while we have edge strength, we’ve lost the direction of the edges in this layout and can only tell that the network is directed because the links are drawn as semi-transparent, so we can see when there are two links of different weights overlapping each other. We need to use some method to show if these links are to or from a node, and one way to do this is to turn our simple lines into arrows using SVG markers.

6.2.2 SVG Markers

There are times when you want to place a symbol on a line or path that you’ve drawn, such as an arrowhead. In that case, you have to define a marker in your svg:defs and then associate that marker with the element on which you want it to draw. You can define your marker statically in HTML or you can create it dynamically like you would any SVG element as we’ll do below. The marker you define can be any sort of SVG shape, but we’ll use a path since it lets us draw an arrowhead. A marker can be drawn at the start, end or middle of a line, and has settings to determine its direction relative to its parent element.

Listing 6.9 Marker definition and application

```
var marker = d3.select("svg").append('defs')
     .append('marker')
     .attr("id", "Triangle")
     .attr("refX", 12)
     .attr("refY", 6)
     .attr("markerUnits", 'userSpaceOnUse') #a
     .attr("markerWidth", 12)
     .attr("markerHeight", 18)
     .attr("orient", 'auto')
     .append('path')
     .attr("d", 'M 0 0 12 6 0 12 3 6');

d3.selectAll("line").attr("marker-end", "url(#Triangle)"); #b
```

#a The default setting for markers bases their size off the stroke-width of the parent, which in our case would result in difficult to read markers

#b A marker is assigned to a line by setting the marker-end, marker-start or marker-mid attribute to point to the marker

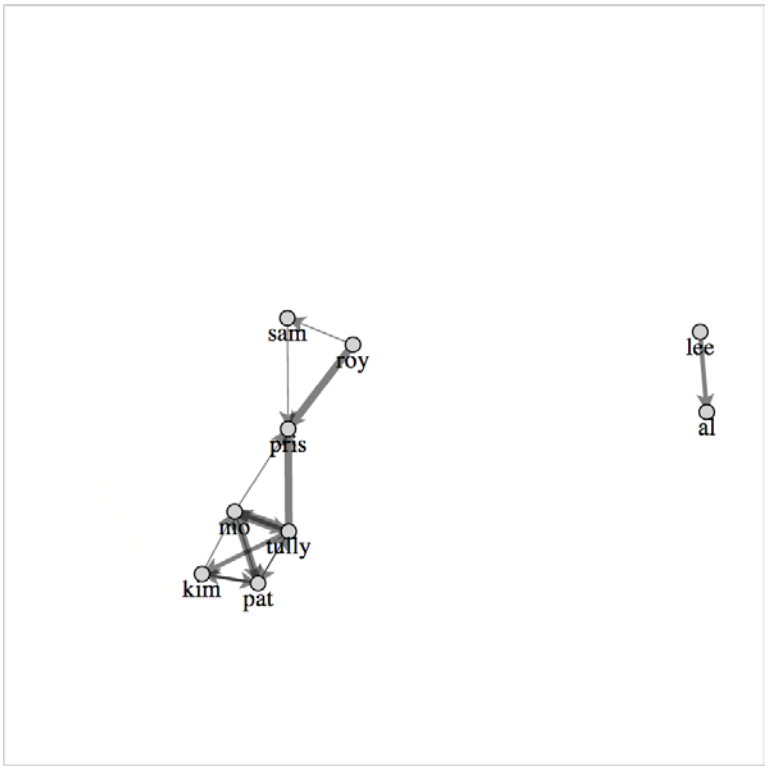


Figure 6.10 Edges now display markers (arrowheads) indicating the direction of connection. Notice that all the arrowheads are the same size

With the markers defined in Listing 6.9 we can now read the network as shown in Figure 6.10 more effectively and see which way nodes are connected to each other as well as spot which nodes have what are known as reciprocal ties with each other (where nodes are connected in both directions). Reciprocation is important to identify, since there's a big difference between people who favorite Katy Perry's tweets and people whose tweets are favorited by Katy Perry (the current Twitter user with the most followers). So, direction of edges is important, but there are other ways to represent direction, such as using curved edges or edges that grow fatter on one end than the other. To do something like that, you would need to use a <path> rather than a <line> for the edges like we did with the Sankey layout or the arc diagram.

If you've run this code on your own, your network probably looks a little different than the what's seen in Figure 6.x. That's because network visualizations made using force-directed layouts are the result of the interplay of forces and, even with a small network like this, that interplay can result in different positions for nodes. This can cause some confusion for users, who think that these variations indicate different networks. One way around this is to generate

a network using a force-directed layout and then fix it in place to create a network basemap, and then apply any later graphical changes to that fixed network. The concept of a basemap comes from geography and has been applied in network visualization to refer to the use of the same layout with differently sized and/or colored nodes and edges. It allows readers to identify regions of the network that are significantly different according to different measures. You can see this concept of a basemap in use in Figure 6.11, which shows how one network can be measured in multiple ways.

Infoviz Term: Hairball

Network visualizations are very impressive, but can also be so complex as to prove unreadable. For this reason, you'll encounter critiques of networks that are too dense to be readable. These network visualizations are often referred to as "hairballs" due to extensive overlap of edges that make them resemble a mass of unruly hair.

In cases where you feel the force-directed layout is hard to read, you can pair it with another network visualization, such as an adjacency matrix, and highlight both as the user navigates either visualization. We'll see techniques for pairing visualizations like this in Chapter 11.

The force-directed layout provides the added benefit of seeing larger structures. Depending on the size and complexity of your network, they may be enough. But there are other network measurements that you might need to represent when working with network data.

6.2.3 Network Measures

Networks have been studied for a long time--at least decades and, if you want to look at graph theory in mathematics, centuries. As such, there are a few terms and measures that you may encounter when working with networks. This is only meant to be a brief overview. If you want to learn more about networks, I would suggest reading Scott Weingart's excellent introduction to networks and network analysis at:

http://www.themacroscope.org/?page_id=337

EDGE WEIGHT

You'll notice that our dataset contains a "weight" value for each link. This represents the strength of the connection between two nodes. In our case, we are assuming that the more favorites, the stronger a connection that one Twitter user demonstrates to. We've already drawn the lines thicker in the case where there is a higher weight, but we can also adjust the way the force layout works based on that weight, as you'll see below.

CENTRALITY

Networks are representations of systems, and one of the things we want to know about the nodes in a system is which ones are more important than the others, referred to as centrality. Central nodes are considered to have more power or influence in a network. There are many different measurements of centrality, a few of which are shown in Figure 6.x, and different

measures more accurately assess centrality in different network types. One simple measure of centrality is computed by D3's force() layout: degree centrality.

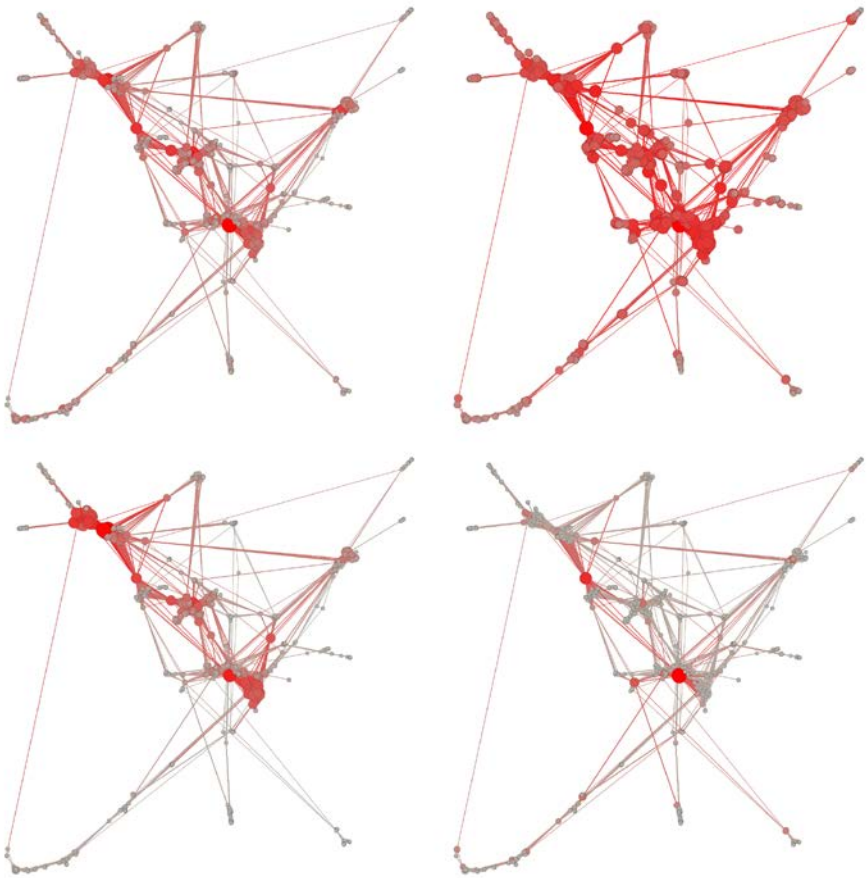


Figure 6.11 The same network measured using Degree Centrality (top left), Closeness Centrality (top right), Eigenvector Centrality (bottom left), and Betweenness Centrality (bottom right). More central nodes are larger and bright red, whereas less central nodes are smaller and gray. Notice that while some nodes are central according to all measures, their relative centrality varies as does the overall centrality of other nodes.

DEGREE

Degree, also known as degree centrality, is simply the total number of links that are connected to a node. In our example data, Mo has a degree of 6, because there are 6 links that he is the source or target of. Degree is a rough measure of the importance of a node in a network, since we assume that people or things with more connections have more power or influence in a network. Weighted Degree is used to refer to the total value of the connections to a node,

which would give Mo a value of 18. Further, you can differentiate degree into In Degree and Out Degree, which are used to distinguish between incoming and outgoing links, in which case Mo would have a score of 4 and 2, respectively.

Every time you start the force() layout, D3 computes the total number of links per node, and updates that node's weight attribute to reflect it. We'll use that to affect the way that the force layout runs. For now, let's just add a button that resizes the nodes based on their weight attribute.

```
d3.select("controls").append("button").on("click", sizeByDegree).html("Degree
Size")

    function sizeByDegree() {
      force.stop();
      d3.selectAll("circle")
      .attr("r", function(d) {return d.weight * 2})
      }
```

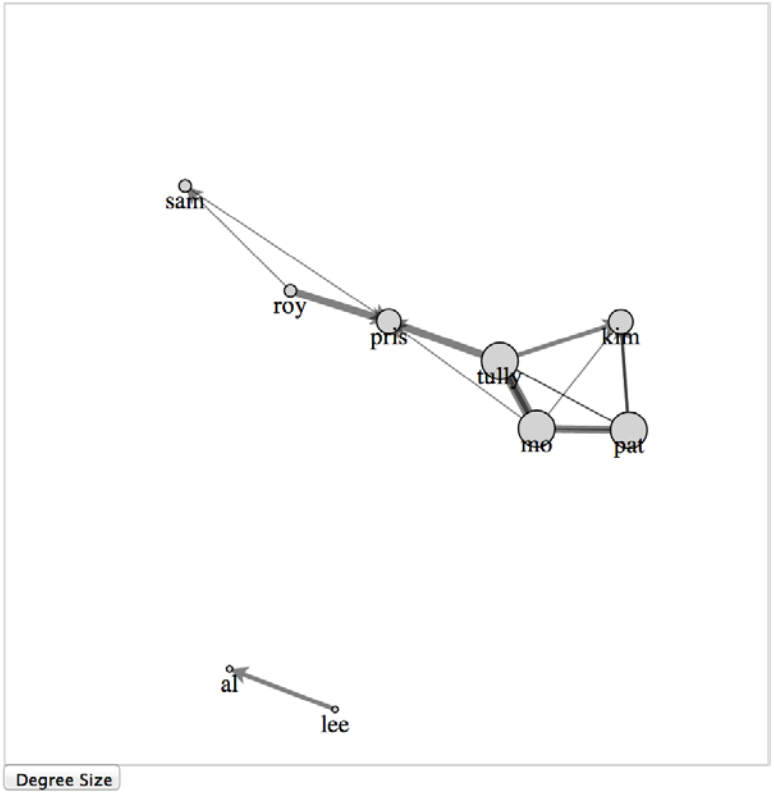


Figure 6.12 Sizing nodes by weight indicates the number of total connections for each node by setting the radius of the circle equal to the weight times two.

The results in Figure 6.12 show the value of the degree centrality measure. While we can see and easily count the connections and nodes in this small network, being able to spot at a glance the most and least connected nodes is extremely valuable. Notice that we're counting links in both directions, so that even though Tully is connected to more people, he's the same size as Mo and Pat, who are connected as many times but to fewer people.

CLUSTERING AND MODULARITY

One of the most important things to find out about a network is whether any communities exist in that network and what they look like. This is done by looking at whether some nodes are more particularly connected to each other than they are to the rest of the network, known as modularity, and also by looking at whether nodes are particularly interconnected, known as clustering. Cliques, mentioned earlier, are part of the same measurement, and a clique is just a term for a group of nodes that are fully connected to each other.

Notice that this interconnectedness and community structure is supposed to arise visually out of a force-directed layout. We see the four highly connected users in a cluster, and visually see the other users farther away. If you'd prefer to measure your networks to try to reveal these structures, you can see an implementation of a community detection algorithm implemented by David Mimno with D3 at http://mimno.infosci.cornell.edu/community/. This algorithm runs in the browser and can be integrated with your network quite easily to color your network based on community membership.

6.2.4 Force Layout Settings

When we initialized our force layout, we started out with a charge setting of -1000. Along with charge, there are a few different settings that give you more control over the way that the force layout runs.

CHARGE

Charge sets the rate at which nodes push each other away. If you don't set charge, then it has a default setting of -1000. The reason we set charge to -1000 was because the default settings for charge with our network would have resulted in a tiny network on-screen (see Figure 6.13).

Figure 6.13 The layout of our network with the default charge, which displays the nodes too closely together to be easily read.

Along with setting fixed values for charge, you can use an accessor function to base the charge values on an attribute of the node. For instance, we could base the charge on the weight (the degree centrality) of the node so that nodes with many connections push nodes away more, giving them more space on the chart.

Charge values are negative to represent repulsion in a force-directed layout, but you could set them to positive if you wanted your nodes to exert an attractive force. This would likely cause problems with a traditional network visualization but may come in handy for a more complicated visualization.

GRAVITY

With nodes pushing each other, the only thing to stop them from flying off the edge of your chart is what's known as canvas gravity, which pulls all nodes toward the center of the layout. When gravity isn't specifically set, it defaults to .1. If we had set the gravity of our force differently, we can see the results of increasing or decreasing the gravity (with our original charge(-1000) setting) in Figure 6.14.

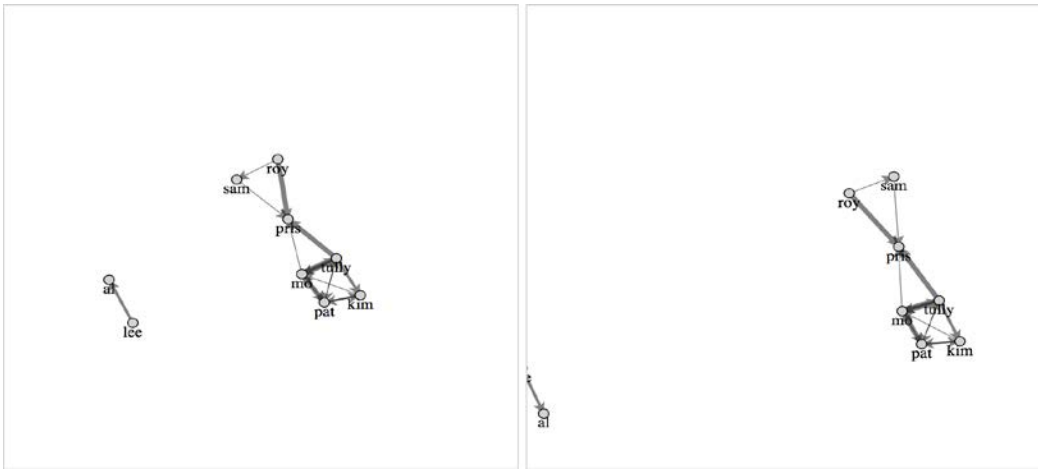


Figure 6.14 Increasing the gravity to .2 (left) pulls the two components closer to the center of the layout area. Decreasing the gravity to .05 (right) allows for the small component to drift off-screen.

Gravity, unlike charge, does not accept an accessor function and only accepts a fixed setting.

LINKDISTANCE

Attraction between nodes is determined by setting the linkDistance property, which is the optimal distance between connected nodes. One of the reasons we needed to set our charge so high was because the linkDistance defaults to 20. If we set it to 50, then we can reduce the charge to -100 and produce the results in Figure 6.15.

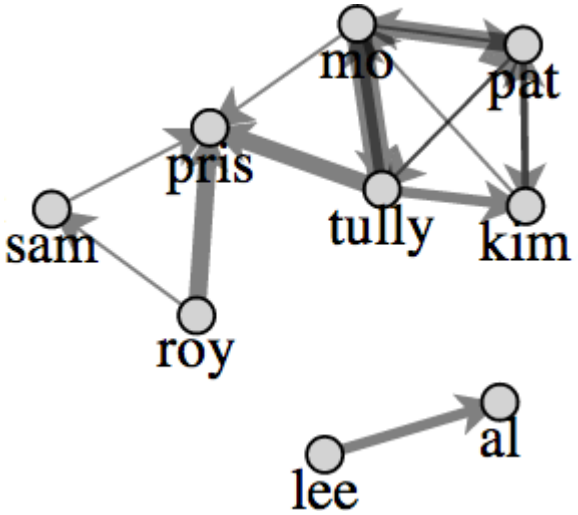


Figure 6.15 With linkDistance adjusted, our network becomes much more readable.

Setting your linkDistance parameter too high will cause your network to fold back in on itself, which you can identify by the presence of prominent triangles in the network visualization. In Figure 6.16, we see this folding occur with linkDistance set to 200.

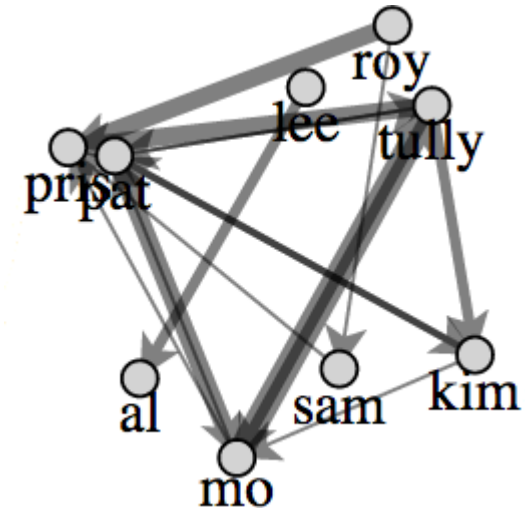


Figure 6.16 Distortion based on high linkDistance makes it look like node X is connected to nodes Y and Z.

You can set linkDistance to be a function and associate it with edge weight to make edges with higher or lower weight values to have lower or higher distance settings, but a better way to achieve that effect is to use linkStrength.

LINKSTRENGTH

A force layout is a physical simulation, meaning it uses physical metaphors to arrange the network to its optimal graphical shape. If your network has stronger and weaker links, like our example does, then it makes sense to have those edges exert stronger and weaker effects on the controlling nodes. You can achieve this by using linkStrength, which can accept a fixed setting but can also take an accessor function to set the strength of an edge to be based on an attribute of that edge.

```
force.linkStrength(function (d) {return weightScale(d.weight)})
```

The results of this code are dramatically demonstrated in Figure 6.17, which reflects the weak nature of some of the connections.

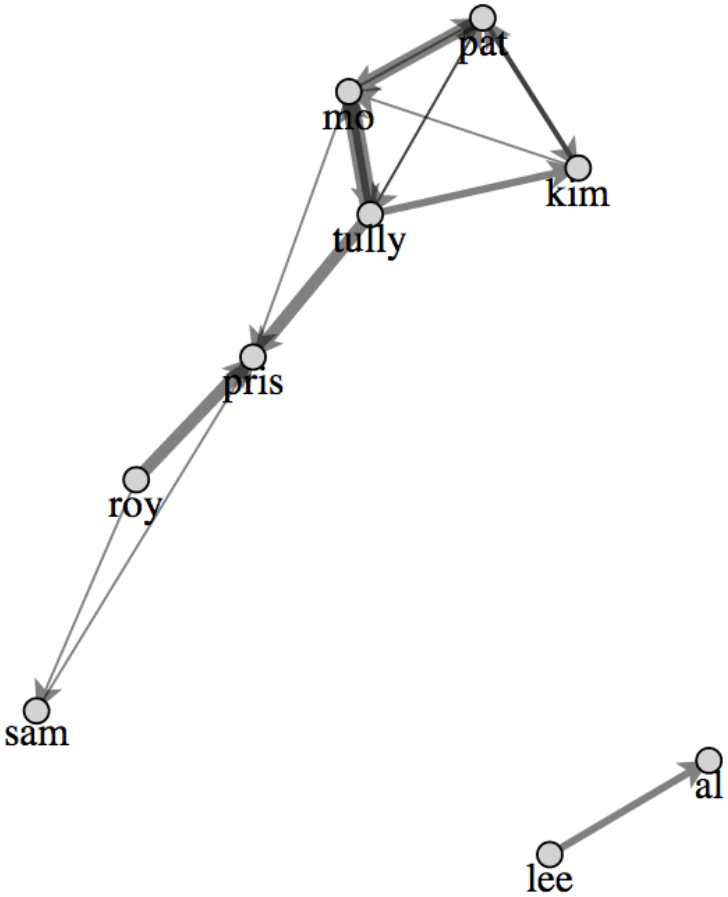


Figure 6.17 By basing the strength of the attraction between nodes on the strength of the connects between nodes, we see a dramatic change in the structure of the network, especially as the weaker connections between x and y allow that part of the network to drift away.

6.2.5 Updating the Network

On creating your network, you'll find that you want to provide your users with the ability to add or remove nodes to the network, or drag them around. You might also want to adjust the various settings dynamically rather than changing them when you first create the force layout.

STOPPING AND RESTARTING THE LAYOUT

The force layout is designed to "cool off" and eventually stop once the network is laid out well enough that the nodes no longer move to new positions. Once the layout has stopped like this, you'll need to restart it if you want to see it begin to animate again. Also, if you've made any changes to the force settings or want to add or remove parts of the network, then you'll need to stop it and restart it.

FORCE.STOP()

You can turn off the force interaction by using force.stop(), which stops running the simulation. It's good to stop the network when there's an interaction with a component elsewhere on your web page or some change in the styling of the network.

FORCE.START()

To begin or restart the animation of the layout, you use force.start(). We've already seen .start(), since we used it in our initial example to get the force layout going.

FORCE.RESUME()

If you haven't made any changes to the nodes or links in your network and you want the network to start moving again, you can use force.resume(). It resets a cooling parameter which will cause the force layout to start moving again.

FORCE.TICK()

Finally, if you want to move the layout forward one step, you can use force.tick(). Force layouts can be very resource intensive and you might want to use one for just a few seconds rather than let it run continuously.

FORCE.DRAG()

Traditional network analysis programs provide the user with the ability to drag nodes to new positions. This is implemented using the behavior force.drag(). A behavior is like a component in that it's called by an element using .call() but instead of creating SVG elements, it creates a set of event listeners.

In the case of force.drag(), those event listeners correspond to dragging events that give you the ability to click on and drag your nodes around while the force layout runs. We can enable dragging on all of our nodes by simply selecting them and calling force.drag() on that selection:

```
d3.selectAll("g.node").call(force.drag());
```

FIXED

When a force layout is associated with nodes, each node has a boolean attribute called fixed that determines whether or not the node is affected by the force during ticks. One interaction technique that proves effective is to set a node as fixed when the user interacts with it. This allows users to drag nodes to a position on the canvas so they can visually sort the important nodes. To differentiate fixed nodes from unfixed nodes, we'll also have the function give fixed nodes a thicker stroke-width. The effect of dragging some of our nodes can be seen in Figure 6.18.

```
d3.selectAll("g.site").on("click", fixNode);

function fixNode(d) {
  d3.select(this).select("circle").style("stroke-width", 4);
  d.fixed = true;
}
```

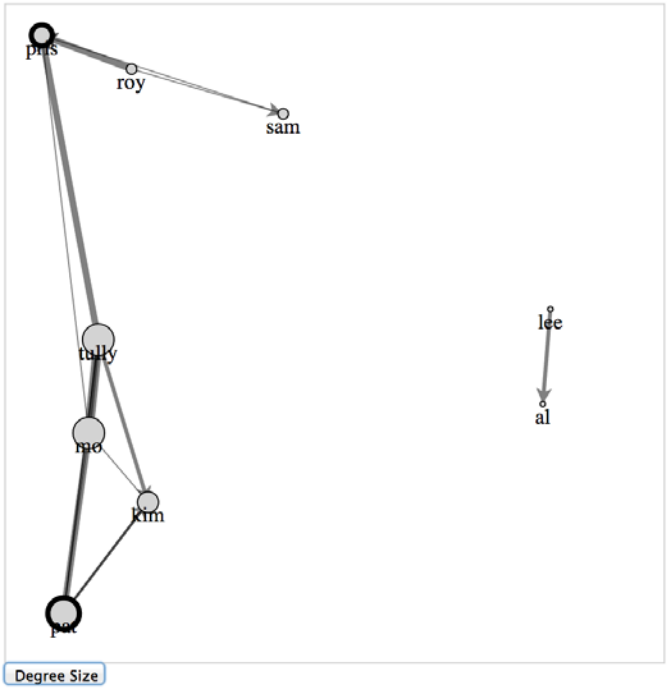


Figure 6.18 The node representing Pat has been dragged to the bottom-left corner and fixed in position while the node representing Pris has been dragged to the top-left corner and fixed in position. the remaining unfixed nodes have taken their position based on the force-directed layout.

6.2.6 Removing and Adding Nodes and Links

When dealing with networks, there are times when you're going to want to filter the networks or give the user the ability to add or remove nodes. To filter a network, you need to stop() it, then remove any nodes and links that are no longer part of the network, then rebind those arrays to the force layout, and then start() the layout.

This can be done as a simple filter on the array that makes up your nodes. For instance, we might want to only see the network of people with more than 20 followers, because we want to see how the most influential people are connected.

But that's not enough, because we would still have links in our layout that reference nodes that no longer exist. That means we'll need a more involved filter for our links array. By using the .indexOf function of an array, though, we can actually create our filtered links pretty easily by checking to see if the source and target are both in our filtered nodes array. Because we used key values when we first bound our arrays to our selection in Listing 6.8, we can use the selection.exit() behavior to easily update our network. We can see how to do this in Listing 6.10 and the effects in Figure 6.19.

Listing 6.10 Filtering a network

```
function filterNetwork() {
      force.stop()
      originalNodes = force.nodes(); #a
      originalLinks = force.links();
      influentialNodes = originalNodes.filter(function (d) {return
d.followers > 20});
      influentialLinks = originalLinks.filter(function (d) {
return influentialNodes.indexOf(d.source) > -1 &&
influentialNodes.indexOf(d.target) > -1
}); #b

      d3.selectAll("g.node")
      .data(influentialNodes, function (d) {return d.id})
      .exit()
      .transition() #c
      .duration(4000)
      .style("opacity", 0)
      .remove();

      d3.selectAll("line.link")
      .data(influentialLinks, function (d) {return d.source.id + "-" +
d.target.id})
      .exit()
      .transition()
      .duration(3000)
      .style("opacity", 0)
      .remove();

      force
      .nodes(influentialNodes)
      .links(influentialLinks)

      force.start()
}
```

#a Access the current array of nodes and array of links associated with the force layout
#b Only make an array of links out of those that reference existing nodes
#c By setting a transition on the .exit() it only applies the transition to those nodes being removed and waits until the transition is finished to remove them.

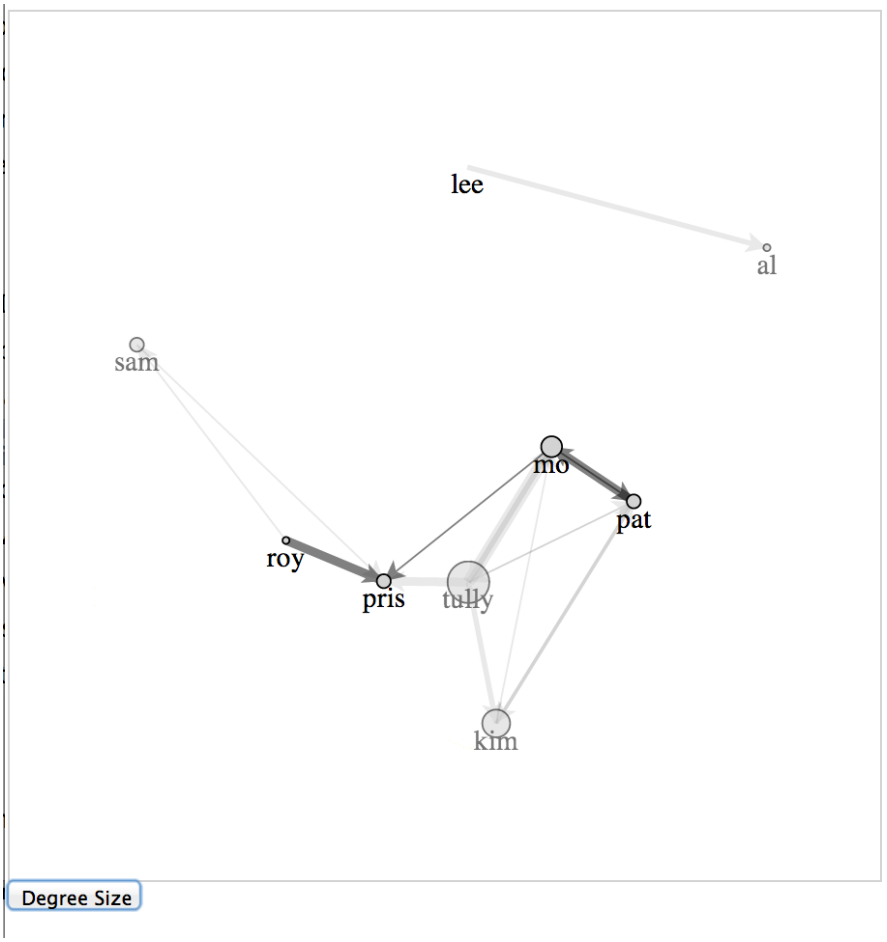


Figure 6.19 The results of filtering the network to only show nodes with more than 20 followers, after clicking the Degree Size button. Notice that Lee, with no connections, has a degree of 0 and so the associated circle has a radius of 0, rendering it invisible. The transition from full to 0 opacity of nodes and edges being removed is caught in mid-stream.

Because the force algorithm is restarted after the filtering, we can see how the shape of the network changes with the removal of so many nodes. That animation is important because it reveals structural changes in the network.

ADDING TO THE NETWORK

Putting more nodes and edges into the network is easy, as long as you properly format your data. You just need to stop the force layout, add the properly formatted nodes or edges to the respective arrays, and re-bind the data as we've done in the past. If, for instance, we want to add an edge between Sam and Al like we see in Figure 6.20, we need to stop the force layout

like we did earlier, create a new datapoint for that edge and add it to the array we're using for the links. Then we rebind the data and append a new line element for that edge before we restart the force layout. You can see the code for this in Listing 6.11.

Listing 6.11 A function for adding edges

```
function addEdge() {
    force.stop();
    var oldEdges = force.links();
    var nodes = force.nodes();
    newEdge = {source: nodes[0], target: nodes[8], weight: 5};
    oldEdges.push(newEdge);
    force.links(oldEdges);
    d3.select("svg").selectAll("line.link")
    .data(oldEdges, function(d) {return d.source.id + "-" + d.target.id})
    .enter()
    .insert("line", "g.node")
    .attr("class", "link")
    .style("stroke", "red")
    .style("stroke-width", 5)
    .attr("marker-end", "url(#Triangle)");

    force.start();
}
```

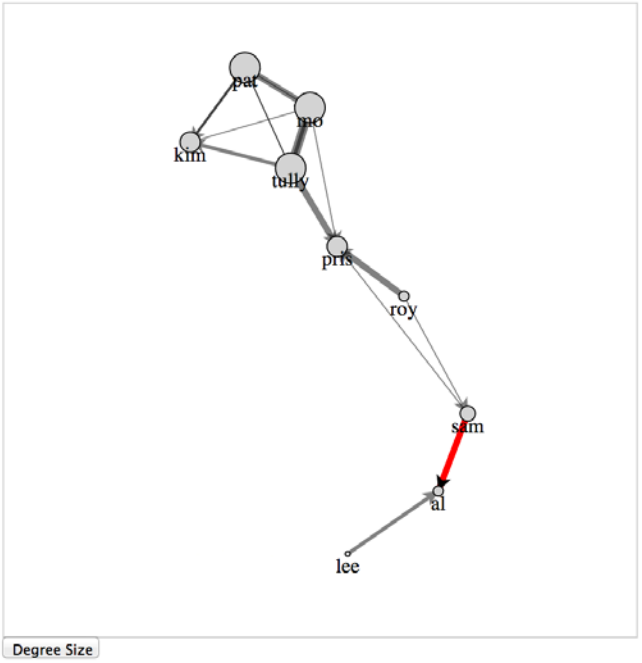


Figure 6.20 Network with a new edge added. Notice that because we re-initialized the force layout that it has correctly recalculated the weight for Al.

If we want to add new nodes like we see in Figure 6.21, we'll also want to add edges at the same time. Not because we have to, but because otherwise they'll float around in space and not be connected to our current network. The code and process, which you can see in Listing 6.12, should look familiar to you by now.

Listing 6.12 Function for adding nodes and edges

```
function addNodesAndEdges() {
    force.stop();
    var oldEdges = force.links();
    var oldNodes = force.nodes();
    newNode1 = {id: "raj", followers: 100, following: 67};
    newNode2 = {id: "wu", followers: 50, following: 33};
    newEdge1 = {source: oldNodes[0], target: newNode1, weight: 5};
    newEdge2 = {source: oldNodes[0], target: newNode2, weight: 5};
    oldEdges.push(newEdge1,newEdge2);
    oldNodes.push(newNode1,newNode2);
    force.links(oldEdges).nodes(oldNodes);

    d3.select("svg").selectAll("line.link")
    .data(oldEdges, function(d) {return d.source.id + "-" + d.target.id})
    .enter()
    .insert("line", "g.node")
    .attr("class", "link")
    .style("stroke", "red")
    .style("stroke-width", 5)
    .attr("marker-end", "url(#Triangle)");

      var nodeEnter = d3.select("svg").selectAll("g.node").data(oldNodes,
function (d) {return d.id}).enter()
      .append("g")
      .attr("class", "node")
      .call(force.drag())

      nodeEnter.append("circle")
      .attr("r", 5)
      .style("fill", "red")
      .style("stroke", "darkred")
      .style("stroke-width", "2px");

      nodeEnter.append("text")
      .style("text-anchor", "middle")
      .attr("y", 15)
      .text(function(d) {return d.id})

      force.start();
}
```

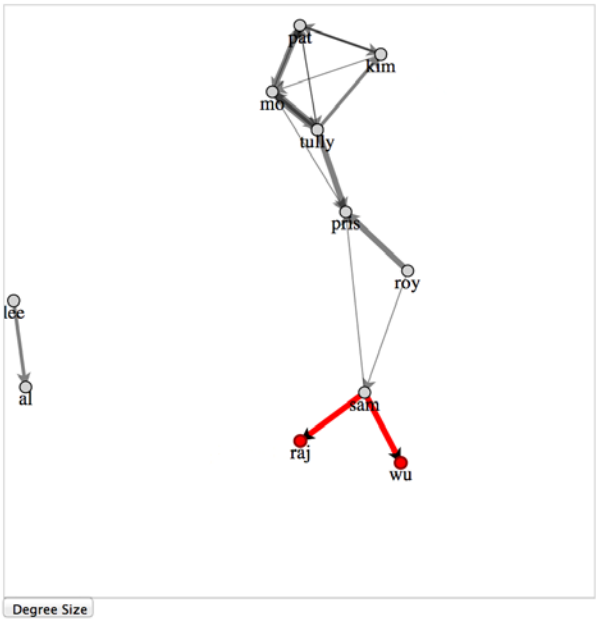


Figure 6.21 Network with two new nodes (Raj and Wu) added, both added with links to Sam.

6.2.7 Manually Positioning Nodes

Remember, the force-directed layout doesn't actually move your elements, instead it calculates the position of elements based on the x and y attributes of those elements in relation to each other. During each tick, it updates those x and y attributes. What updates the position of your elements is the selection on the <line> and <g> elements that takes place in the tick function to move them to these updated x and y values.

So, when you want to move your elements manually, you can do so just like you normally would, though you first need to stop the force so that you prevent that tick function from overwriting your elements' positions. Let's lay out our nodes like a scatterplot, looking at the number of followers by the number that each node is following. We'll also add axes to make it readable. You can see the code in Listing 6.13 and the results in Figure 6.22.

Listing 6.13 Moving your nodes manually

```
function manuallyPositionNodes() {
    var xExtent = d3.extent(force.nodes(), function(d) {return
parseInt(d.followers)})
    var yExtent = d3.extent(force.nodes(), function(d) {return
parseInt(d.following)})
    var xScale = d3.scale.linear().domain(xExtent).range([50,450])
    var yScale = d3.scale.linear().domain(yExtent).range([450,50])
```

```
    force.stop();
    d3.selectAll("g.node")
    .transition()
    .duration(1000)
    .attr("transform", function(d) {return "translate("+ xScale(d.followers)
+","+yScale(d.following) +")"})

    d3.selectAll("line.link")
    .transition()
    .duration(1000)
    .attr("x1", function(d) {return xScale(d.source.followers)})
    .attr("y1", function(d) {return yScale(d.source.following)})
    .attr("x2", function(d) {return xScale(d.target.followers)})
    .attr("y2", function(d) {return yScale(d.target.following)})

    xAxis = d3.svg.axis().scale(xScale).orient("bottom").tickSize(4);
    yAxis = d3.svg.axis().scale(yScale).orient("right").tickSize(4);
    d3.select("svg").append("g").attr("transform",
"translate(0,460)").call(xAxis);
    d3.select("svg").append("g").attr("transform",
"translate(460,0)").call(yAxis);

    d3.selectAll("g.node").each(function(d){
      d.x = xScale(d.followers);
      d.px = xScale(d.followers);
      d.y = yScale(d.following);
      d.py = yScale(d.following);
    })

   }
```

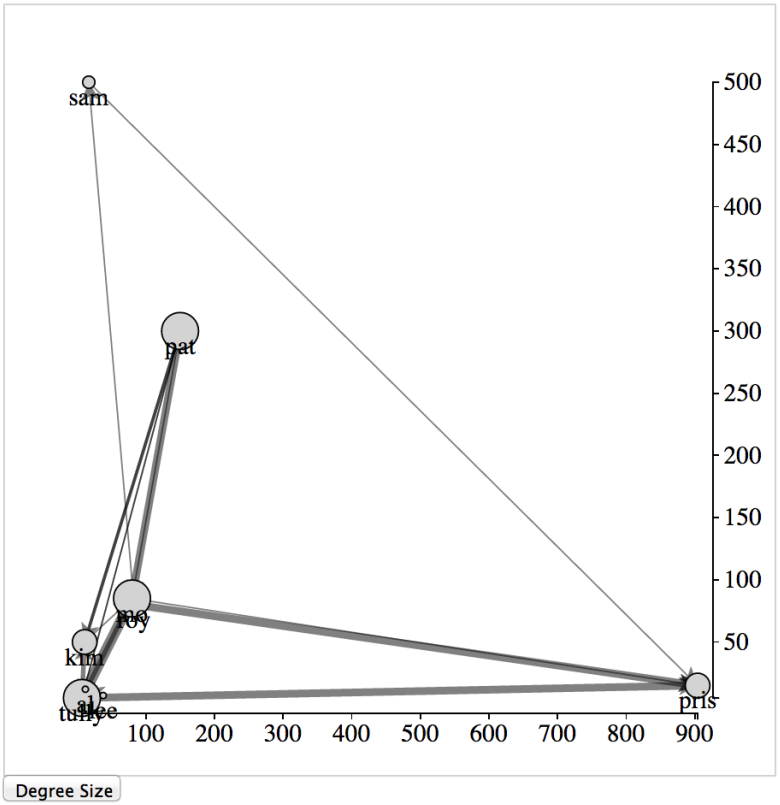


Figure 6.22 When presenting a network as a scatterplot, the links increase the visual clutter. It provides a useful contrast to the force-directed layout, but can be hard to read on its own.

Notice that we need to update the x and y attributes of each node, but we also need to update the px and py attributes of each node. The px and py attributes are the previous x and y coordinates of the node before the last tick, and if you don't update them, then the force layout thinks that the nodes have very high velocity, and will move violently from their new position.

If we didn't update the x,y,px,py attributes, then the next time we start the force layout, the nodes will immediately return to their position before we moved them. This way, when we restart the force layout with force.start(), the nodes and edges animate from their current position.

6.2.8 Optimization

The force layout is extremely resource intensive, that's why it cools off and stops running by design. And, if you have a large network running with the force layout, you can really tax a user's computer and it becomes practically unusable. The first tip to optimization, then, is to

try to limit the number of nodes in your network, as well as the number of edges. A good rule of thumb is no more than 100 nodes, unless you know your audience is going to be using the browsers that perform best with SVG, like Safari and Chrome.

But if you have to present more nodes and want to reduce the performance hit, you can use force.chargeDistance() to set a maximum distance for the layout to compute the repulsive charge for each node. The lower this setting, more unstructured the force layout will be, but the faster it will run. Because networks vary so much, you'll have to experiment with different values for chargeDistance to find the best one for your network.

6.3 Summary

In this chapter you learned several methods for displaying network data, and looked in-depth at the force layout available for network data in D3. There is no one way to visually represent a network, and so now you have multiple methods along with static, dynamic and interactive variations, with which to work. Specifically, you learned to:

- Format a node and edge list in the manner D3 typically uses
- Build a weighted, directed adjacency matrix and add interaction to explore it
- Build an interactive weighted, directed arc diagram
- Learn simple techniques to find links to a node
- Build and customize force-directed layout
- Understand the basics of network terminology and statistics, such as edge, node, degree and centrality
- Use accessors to create dynamic forces
- Add interactivity to update node size based on degree centrality

We focused on network information visualization because your world is awash in network data. In the next chapter, we'll look at another broadly applicable but specific domain: geographic information visualization. Just as we've seen several different ways to represent networks in this chapter, in Chapter 7 you'll learn different ways of making maps, including tiled maps, globes, and traditional data-driven polygon maps.

7

Geospatial Information Visualization

One of the most common categories of data you'll encounter is geospatial data. This can come in the form of administrative regions like states or counties, points that represent cities or places where a person was when they made a tweet, or satellite imagery of the surface of the Earth.

It used to be that if you wanted to make a web map, that you'd need a specialized library like Google Maps or Leaflet or OpenLayers. However, D3 provides enough core functionality to make any kind of map you've seen on the web, and since you're already working with D3, you can make that map far more sophisticated and distinctive than the kind of out-of-the-box maps you typically see. The major reason to continue to use a dedicated library like Google Maps API is because of the added functionality that comes from being in that ecosystem, such as Street View of Google tiles or integrated support for Fusion Tables. But if you're not going to use the ecosystem, then it might be a smarter move to build the map with D3, since it means you won't have to invest in learning a different syntax and abstraction layer, along with the greater flexibility D3 mapping affords you.

Because map-making and geographic information systems/science (known as GIS and GIScience respectively) have been in practice for so long, there are well-developed methods for representing this kind of data. D3 has built-in robust functionality to load and display geospatial data, and a related library that we'll get to know in this chapter, TopoJSON, provides even more geospatial information visualization functionality.

In this chapter, we'll start with making simple maps that combine points, lines and polygons using data from CSV and GeoJSON formatted sources. We'll learn how to style those maps and provide interactive zooming by revisiting d3.zoom() and exploring it in more detail. After that, we'll look closely at the TopoJSON data format and why it provides significantly smaller data files but also its built-in functionality that leverages topology. Finally, we'll learn how to make maps using tiles, allowing you to show terrain and satellite imagery.

7.1 Basic Mapmaking

Before we get into pushing the boundaries for what a map might be, we need to make a simple map. In D3, the most simple map you can make is a vector map using SVG <path> and <circle> elements to represent countries and cities. We can bring back cities.csv which we used in Chapter 2, and finally take advantage of its coordinates, but we need to look a bit further to find the data necessary to represent those countries. Once we have that data, we can render it as areas, lines or points on a map. After that, we can add interactivity like highlighting a region when you move your mouse over it or computing and showing its center.

Before we get started, though, let's take a look at the CSS for this chapter in Listing 7.1.

Listing 7.1 ch7.css

```
path.countries {
     stroke-width: 1;
     stroke: black;
     opacity: .5;
     fill: red;
}

circle.cities {
     stroke-width: 1;
     stroke: black;
     fill: white;
}

circle.centroid {
     fill: red;
     pointer-events: none;
}
rect.bbox {
     fill: none;
     stroke-dasharray: 5 5;
     stroke: black;
     stroke-width: 2;
     pointer-events: none;
}

path.graticule {
     fill: none;
stroke-width: 1;
stroke: black;
}

path.graticule.outline {
     stroke: black;
}
```

7.1.1 Finding Data

Making a map requires data, and there's an enormous amount of data available to you. Geographic data can come in several forms, and if you're familiar with GIS then you'll be familiar with one of the most common forms for complex geodata: the shapefile, which is a

format developed by ESRI and is most commonly found in desktop GIS applications But the most human readable form of geodata comes in the form of simple latitude and longitude (or X/Y like we list in our file) coordinates when dealing with points like cities, often times in a CSV. We'll use cities.csv, the same CSV we measured in Chapter 2 that had the locations of eight cities from around the world.

Listing 7.2 cities.csv

```
"label","population","country","x","y"
"San Francisco", 750000,"USA",-122,37
"Fresno", 500000,"USA",-119,36
"Lahore",12500000,"Pakistan",74,31
"Karachi",13000000,"Pakistan",67,24
"Rome",2500000,"Italy",12,41
"Naples",1000000,"Italy",14,40
"Rio",12300000,"Brazil",-43,-22
"Sao Paolo",12300000,"Brazil",-46,-23
```

One thing you'll notice if you are at all familiar with geodata is that our latitude and longitude are imprecise. San Francisco, for instance, is not at 37,-122 but rather 37.783, -122.417. That means when we plot these cities, they're going to look pretty off as we zoom in. Obviously, you'll want to use more accurate coordinates for your maps but for this example, which mostly uses maps that are zoomed way out, this should be fine.

If you only have city names or addresses and need to get latitude and longitude, you can take advantage of geocoding services that will provide you with xy from addresses. These exist as APIs and are available on the web for small batches. You can see an example of these services maintained by Texas A&M at http://geoservices.tamu.edu/Services/Geocode/.

When dealing with more complex geodata like shapes or lines we are necessarily dealing with more complex data formats. To do that, you'll want to use GeoJSON, which has become the standard for web mapping data.

GeoJSON

GeoJSON (geojson.org) is, like it sounds, a way of encoding geodata in JSON format. Each "feature" in a "featureCollection" is a JSON object that stores the border of the feature in a "coordinates" array as well as metadata about the feature in a "properties" hash object. For instance, if you wanted to draw a square that went around the island of Manhattan, then it would have corners at [-74.0479, 40.6829], [-74.0479, 40.8820], [-73.9067, 40.8820], [-73.9067, 40.6829] as seen in figure 7.1. You can easily export shapefiles into GeoJSON using QGIS (a desktop GIS application - qgis.org), PostGIS (a spatial database run on Postgres - postgis.net), GDAL (a library for manipulation of geospatial data – gdal.org) and other tools and libraries.

Figure 7.1 A polygon drawn at the coordinates [-74.0479, 40.8820], [-73.9067, 40.8820], [-73.9067, 40.6829], [-74.0479, 40.6829].

When you draw a rectangle over a geographic feature like this, it's known as a "bounding box" and it's often represented with only two coordinate pairs: the upper left and bottom right corners. But any polygon data, such as the irregular border of a state or coastline, can be represented by an array of coordinates like this. In Listing 7.3, we have a fully compliant GeoJSON "FeatureCollection" with only one feature, the very simplified borders of the very small nation of Luxembourg.

Listing 7.3 GeoJSON example of Luxembourg

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

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

You're not going to be creating your own GeoJSON in this chapter, and unless you get into serious GIS, you may never create your own GeoJSON. Instead, you can easily get by with downloading existing geodata and either use it without editing it or edit it in a GIS application and export it. In our examples in this chapter, we'll be using world.geojson (available at emeeks.github.io/d3ia/world.geojson), a file that consists of the countries of the world in the same very simplified, low-resolution representation that we see in Listing 7.x.

PROJECTION

Entire books have been written on creating web maps, and an entire book can be written on using D3.js for crafting maps. Since this is only a chapter, I'll be forced to gloss over many deep issues. One of these is projection. Projection is a term in geographic information systems that refers to the process of rendering data that refers to points on a globe, like the actual Earth, onto a flat plane, like your computer monitor. There are many different ways to project geographic data for representation on your screen, and in this chapter we'll look at a few different methods.

But to start, we'll use one of the most common geographic projections, the Mercator projection, which is used in most web maps because it's the projection used by Google Maps and so became the de facto standard. To use Mercator projection requires us to include an extension of D3, `d3.geo.projection.js`, which we'll want for some of the more interesting

work we'll do later in the chapter. By defining a projection, we'll also be able to take advantage of d3.geo.path, which draws geoData on-screen based on your selected projection. Once you've defined a projection and have geo.path() ready, that's all you really need, which means the entire code in Listing 7.4 is all that's required to draw the map seen in Figure 7.2.

Listing 7.4 Initial Mapping Function

```
    d3.json("world.geojson", createMap);

    function createMap(countries) {
      var aProjection = d3.geo.mercator(); #a
      var geoPath = d3.geo.path().projection(aProjection); #b
      d3.select("svg").selectAll("path").data(countries.features)
       .enter()
.append("path")
.attr("d", geoPath) #c
.attr("d", "countries")
}
```

#a Projection functions have many options that we'll see later
#b d3.geo.path() defaults to albersUSA, which is a projection that is only suitable for maps of the United States.
#c d3.geo.path() takes properly formatted geoJSON features and returns SVG drawing code for SVG paths

Figure 7.2 A map of the world using the default settings for D3's Mercator projection results in most of the Western Hemisphere, some of Europe and Africa, and the rest of the world rendered out of sight.

Why do we only see part of the world in Figure 7.x? Because the default settings of the Mercator projection are such that we need to adjust the settings so as to show the entire map of the world in our SVG canvas. Each projection has a .translate() and .scale() that follow the syntax of the transform convention in SVG, but have different effects with different projections.

SCALE

There are some tricks for setting the right scale for certain projects. For instance, with our Mercator projection if we divide the width of the available space by 2 and divide the quotient by Math.pi, then the result will be the proper scale to display the entire world in the available space. Figuring out the right scale for your map and your projection is typically done through experimenting with different values, but it's made easier when you include zooming, as we'll see below in section 7.2.2.

Different families of projections have different scale defaults. The d3.geo.albersUsa projection defaults to 1070, while d3.geo.mercator defaults to 150. As with most D3 functions like this, we can see the default by calling the function without passing it a value:

```
d3.geo.mercator().scale() #a
d3.geo.albersUsa().scale() #b
```

#a 150
#b 1070

By adjusting the translate and scale as we do in Listing 7.5, we can adjust the projection to show different parts of the geodata we're working with--in our case the world.

Listing 7.5

```
  function createMap(countries) {
    var width = 500; #a
    var height = 500;
    var aProjection = d3.geo.mercator()
    .scale(80) #b
    .translate([width / 2, height / 2]) #c
    var geoPath = d3.geo.path().projection(aProjection);
    d3.select("svg").selectAll("path").data(countries.features)
     .enter()
     .append("path")
     .attr("d", geoPath)
.attr("d", "countries")

}
```

#a by defining the size of our SVG as variables, we can refer to them throughout our visualization code
#b Scale values are different for different families of projections, 80 works well in this case
#c Moves the center of the projection to the center of our canvas

Figure 7.3 The mercator-projected world from our data now fitting our SVG area. Notice the enormous distortion in size of regions near the poles, such as Greenland and Antarctica.

7.1.2 Drawing Points on a Map

Projection is not just used to display areas, it is also used to place individual points. Typically, we think of cities or people as represented not by their spatial footprint (though we do this with particularly large cities) but with a single point on a map, which is sized based on some variable such as population. A D3 projection can be used not only in a geo.path() but also as a function on its own. When you pass it an array with a pair of latitude and longitude coordinates, it will return the screen coordinates necessary to place that point. For instance, if we want to know where to place a point representing San Francisco (roughly speaking -122 latitude, 37 longitude) then we could simply pass those values to our projection:

```
aProjection([-122,37]) #a
 #a [79.65586500535346, 194.32096033997914]
```

We can use this, paired with loading the data from cities.csv as we do in Listing 7.6, to add cities to our map.

Listing 7.6

```
queue()
    .defer(d3.json, "world.geojson")
    .defer(d3.csv, "cities.csv")
```

```
    .await(function(error, file1, file2) { createMap(file1, file2); });

  function createMap(countries, cities) {
    var width = 500;
    var height = 500;
    var projection = d3.geo.mercator()
    .scale(80)
    .translate([width / 2, height / 2])
    var geoPath = d3.geo.path().projection(projection);

    d3.select("svg").selectAll("path").data(countries.features)
    .enter()
    .append("path")
    .attr("d", geoPath)
    .style("fill", "gray") #a

    d3.select("svg").selectAll("circle").data(cities) #b
    .enter()
    .append("circle")
     .style("fill", "red")
.attr("class", "cities")
.attr("r", 3)
     .attr("cx", function(d) {return projection([d.x,d.y])[0]}) #c
     .attr("cy", function(d) {return projection([d.x,d.y])[1]})
```

#a Overriding the fill style so it will be easier to see our cities

#b We want to draw the cities over the countries, so we append them second

#c Projection returns an array, which means we need to take the [0] value for cx and the [1] value for cy

Figure 7.4 Our map with our eight world cities added to it. At this distance, we can't tell how inaccurate these points are, but if we zoom in, we'll see that both of our Italian cities are actually in the Mediterranean.

One thing to note from Listing 7.6 is that coordinates are often given in the real world in the order of "latitude, longitude" and since latitude corresponds to the y-axis and longitude corresponds to the x-axis, then you have to flip them to provide the "x, y" coordinates necessary for GeoJSON and D3.

AREA

Depending on what projection you use, the graphical size of your geographic objects will appear different. This is because it's impossible to perfectly display spherical coordinates on a flat surface, and so different projections are designed to either maintain geographic area on land; or distance; or particular shapes. Since we included d3.geo.projection.js, we have access to quite a few more projections to play with, one of which is the Molleweide projection. In the code in Listing 7.7, we can see the settings necessary to properly display a Molleweide projection of our geodata, and we'll use the calculated area to of the countries (the graphical area, not their actual physical area) to color each country. The results are quite distinct from the same code running on our Mercator projection, as seen in Figure 7.5. The world as displayed with Molleweide curves the edges, rather than stretching them into a rectangle like Mercator does.

Listing 7.7 Molleweide Projected World

```
      queue()
      .defer(d3.json, "world.geojson")
      .defer(d3.csv, "cities.csv")
      .await(function(error, file1, file2) { createMap(file1, file2); });

    function createMap(countries, cities) {

      var width = 500;
      var height = 500;

      var projection = d3.geo.mollweide()
      .scale(120) #a
      .translate([width / 2, height / 2])

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

      var featureSize = d3.extent(countries.features,
  function(d) {return geoPath.area(d)});

      var countryColor =
  d3.scale.quantize().domain(featureSize).range(colorbrewer.Reds[7]); #b


      d3.select("svg").selectAll("path").data(countries.features)
      .enter()
      .append("path")
       .attr("d", geoPath)
       .attr("class", "countries")
       .style("fill", function(d) {return countryColor(geoPath.area(d))}) #c

      d3.select("svg").selectAll("circle").data(cities)
      .enter()
      .append("circle")
  .attr("class", "cities")
       .attr("r", 3)
       .attr("cx", function(d) {return projection([d.x,d.y])[0]})
       .attr("cy", function(d) {return projection([d.x,d.y])[1]})
```

#a For a Molleweide projection, this will show the entire world
#b Measure the features and assign the size classes to a color ramp
#c Color each country based on its size

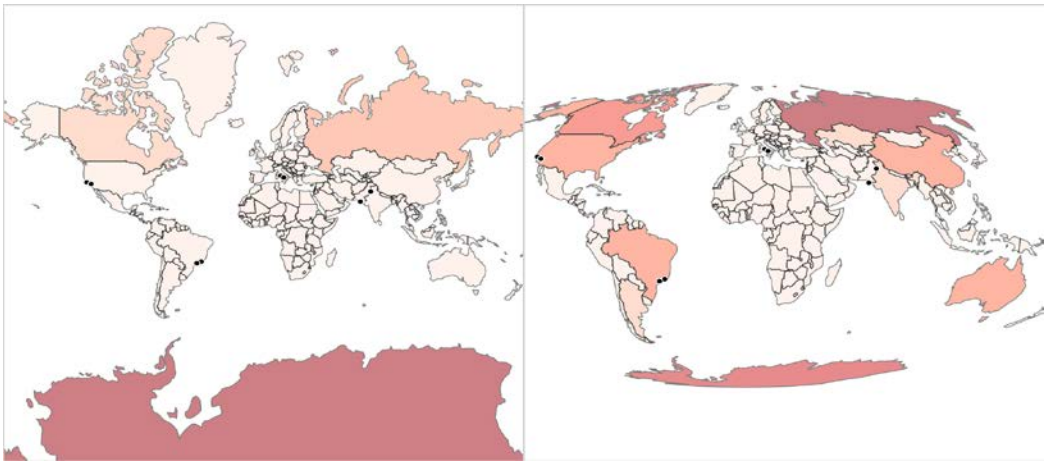


Figure 7.5 Mercator (left) dramatically distorts the size of Antarctica so much that no other shape looks as large. In comparison, the Molleweide projection maintains the actual physical area of the countries and continents in our geodata, at the cost of distorting their shape and angle. Notice that geo.path.area measures the graphical area and not the actual physical area of the features.

Picking the right projection is never easy, and depends on the goals of the map you're making. If you're working with traditional tile mapping, then you're probably going to stick to Mercator. If you're working on the world scale, it's usually best to use an equal-area projection like Molleweide which does not distort the visual area of geographic features. But because there are so many different projections available in D3, you should experiment to see which best suits the particular map you're creating.

Infoviz Term: Chloropleth Map

As you encounter more map-making, you'll hear this kind of map referred to as a chloropleth map. You can use the existing geographic features, in this case countries, to display statistical data, such as the GDP of a country, or its population, or its most-spoken language. You can do this in D3 either by getting geodata where the properties field has that information or by linking a table of data to your geodata where they both have the same unique ID values in common.

Keep in mind that chloropleth maps, while useful, are subject to what's known as the "areal unit problem" which is what happens when you draw boundaries in such a way, or select existing features, so that they disproportionately represent your statistics. This is what happens with gerrymandering, when political districts are drawn in such a way so as to create majorities for one political party or another.

7.1.3 Interactivity

Much of the geo aspect of D3 comes with built-in functionality that you'll typically need when working with geodata. Along with determining the area like we did to color our features, there are other useful functions. Two that are commonly used in mapping are the ability to quickly calculate the center of a geographic area (known as a centroid) and its bounding box. In Listing 7.8, we can see how to add mouseover events to the paths we created and draw a circle at the center of each geographic area, as well as a bounding box around it.

Listing 7.8

```
d3.selectAll("path.countries")
.on("mouseover", centerBounds)
    .on("mouseout", clearCenterBounds)

    function centerBounds(d,i) {
      var thisBounds = geoPath.bounds(d);
      var thisCenter = geoPath.centroid(d); #a
      d3.select("svg")
      .append("rect")
      .attr("class", "bbox")
      .attr("x", thisBounds[0][0]) #b
      .attr("y", thisBounds[0][1])
      .attr("width", thisBounds[1][0] - thisBounds[0][0])
      .attr("height", thisBounds[1][1] - thisBounds[0][1])

      d3.select("svg")
      .append("circle")
      .attr("class", "centroid")
      .attr("r", 5)
      .attr("cx", thisCenter[0]).attr("cy", thisCenter[1]) #c
      .style("pointer-events", "none")
    }
    function clearCenterBounds() { #d
      d3.selectAll("circle.centroid").remove();
      d3.selectAll("rect.bbox").remove();
    }
```

#a Functions of geo.path that give results based on the associated projection
#b The bounding box is the top-left and bottom right coordinates as an array
#c The centroid is a simple array with the x and y coordinates of the center of a feature
#d Remove the shapes when you mouse off a feature

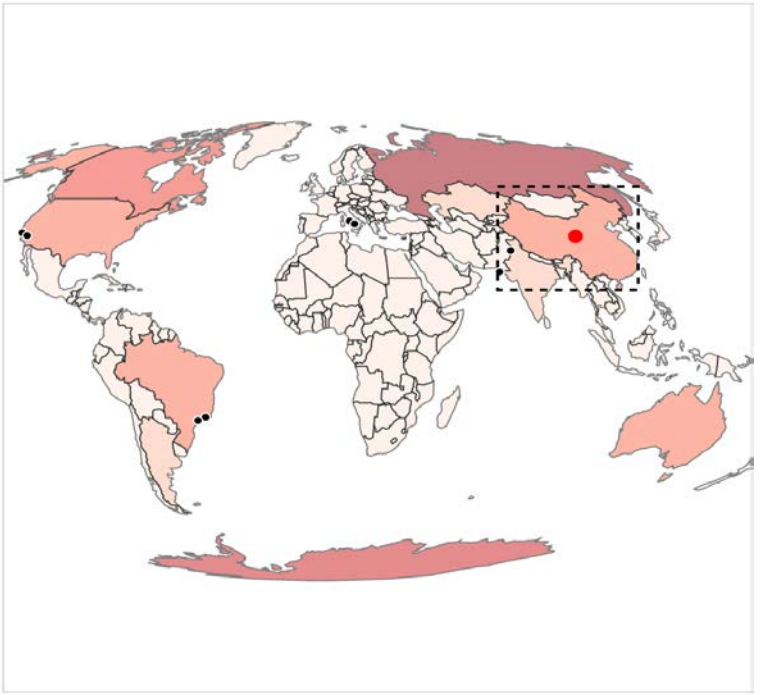


Figure 7.6 The effect of our interactivity is to provide a bounding box around each country and a red circle representing its graphical center. Here we see the bounding box and centroid of China. The D3 implementation of a centroid is weighted, so that it is the center of most area, and not just the center of the bounding box.

At this point you've learned the core geo functions that allow you to make maps with D3: geo.projection and geo.path. By using these functions, you can create maps with a distinct look-and-feel, as well as provide your users with the ability to interact with them not only as shapes but as geographic features. D3 provides even more functionality than this, which we'll start to dive into.

7.2 Better Mapping

To make your maps more readable, you can use a few more built-in features from d3.geo: the graticule generator and the zoom behavior. One provides grid lines that make it easier to read a map, while the other allows you to pan and zoom around your map. Both of these follow the same format and functionality of other behaviors and generators in D3, but are particularly useful for maps.

7.2.1 Graticule

A Graticule is just a grid line on a map. Just as D3 has generators for lines and areas and arcs, it has a generator for graticules to make your maps more beautiful. The graticule generator will create gridlines (you can specify where and how many, or use the default as we will) and will also create an outline that can provide a useful border. In listing 7.9, we'll see how to draw a graticule beneath the countries we've already drawn. Here we're using .datum instead of .data, which is just a convenience function that allows you to bind a single datapoint to a selection and just means it doesn't need to be in an array. In other words, .datum(yourDatapoint) is the same as .data([yourDatapoint]).

Listing 7.9 Adding a graticule

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

d3.select("svg").insert("path", "path.countries")
.datum(graticule)
.attr("class", "graticule line")
.attr("d", geoPath);

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

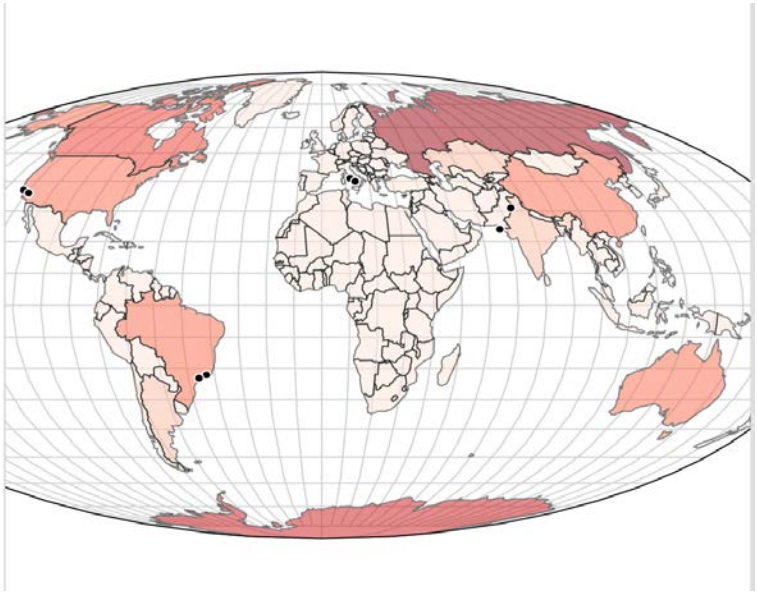


Figure 7.7 Our map with a graticule (in light gray) and a graticule outline (the black border around the edge of the map).

But how are we drawing so many graticule lines in Figure 7.7 from a single datapoint? The geo.graticule function creates a feature known as a "Multilinestring". A multilinestring, as you might already have figured out, is an array of arrays of coordinates, each corresponding to separate individual components of a feature. Multilinestrings and their counterpart, multipolygons, have always been a part of GIS because countries like the United States or Indonesia are made up of disconnected features such as states and regions, and there needed to be a way for that to be stored in the data. As a result, d3.geo.path knows when it gets a multipolygon or multilinestring to draw a <path> element made up of multiple, disconnected pieces.

7.2.2 Zoom

We dealt with zoom a little bit in Chapter 5, when we saw how the zoom behavior can easily allow you to pan a chart around the screen. Now it's time we start zooming with zoom. When we first looked at the zoom behavior, we used it to adjust the transform attribute of a <g> element that held our chart. This time, we'll use the scale and translate values of the zoom behavior to update the settings of our projection, the end result of which will be the ability to zoom and pan our map.

We'll create a zoom behavior and call it from our <svg> element. This means that whenever we have a drag event on anything in the <svg>, or a mousewheel event, or a double-click, then it will trigger zoom. When we worked with zoom before, we only dealt with the dragging, which updates the zoom.translate() value and which we can use to update the translate value of whatever element we want to update. This time, we'll also use the zoom.scale() value, which gives us an increasing (when we double-click or roll our mousewheel forward) or decreasing (when we roll our mousewheel backward) value. To use zoom with a projection, you'll want to overwrite the initial zoom.scale() value with scale value of the projection, and do the same with the zoom translate value. After that, any time you have an event that triggers zoom, you'll use the new values to update your projection, as seen in Listing 7.10

Listing 7.10

```
var mapZoom = d3.behavior.zoom()
.translate(projection.translate()) #a
.scale(projection.scale())
.on("zoom", zoomed);

    d3.select("svg").call(mapZoom);

    function zoomed() {
      projection.translate(mapZoom.translate()).scale(mapZoom.scale()); #b

      d3.selectAll("path.graticule").attr("d", geoPath); #c
      d3.selectAll("path.countries").attr("d", geoPath);

      d3.selectAll("circle.cities")
      .attr("cx", function(d) {return projection([d.x,d.y])[0]}) #d
      .attr("cy", function(d) {return projection([d.x,d.y])[1]})
```

```
    }
#a Overwrite the translate and scale of the zoom to match the projection
#b Whenever the zoom behavior is called, overwrite the projection to match the updated zoom values
#c Any path will be properly redrawn by calling your d3.geo.path associated with the updated
projection
#d Also calling the now updated projection
```

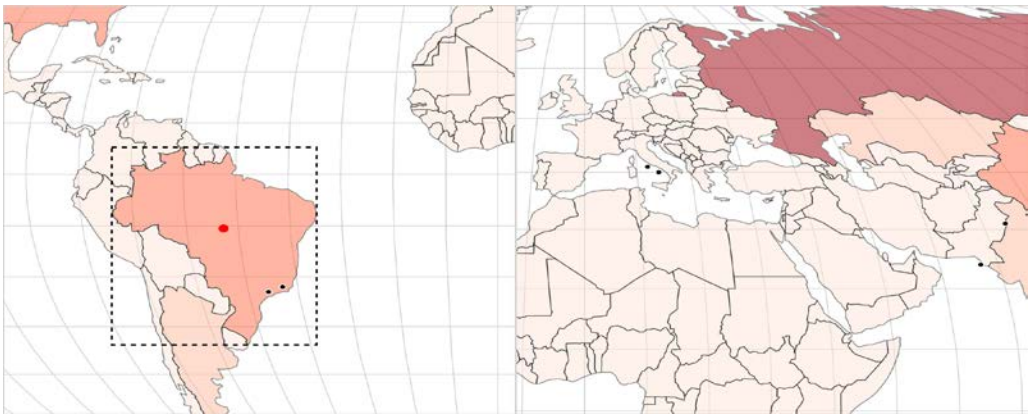


Figure 7.8 Our basic map with zooming enabled. Panning occurs with the drag behavior and zooming with mousewheel and/or double-clicking. Notice that the bounding box and centroid function still works, because it's based on our constantly updating projection.

The zoom behavior is really very simple, updating its .translate() array in reference to your dragging behavior, and increasing or decreasing the .scale() value in reference to your mousewheel and double-click behavior. Because it's designed to work with SVG transform and D3 geographic projections, the above function is all you need for basic pan-and-zoom functionality.

Infoviz term: Semantic Zoom

When you think about zooming in on things, you naturally think about increasing their size. But from mapping we know that we don't just increase the size or resolution as we zoom in, we also change the kind of data that we present to the reader. This is known as "semantic zoom" in contrast to "graphical zoom". It's most clear when you look at a zoomed out map and see only country boundaries and a few major cities, but as you zoom in you see roads, smaller cities, parks, and so on.

You should try to use semantic zoom whenever you're letting your user zoom in and out of any data visualization, not just a chart. It allows you to present strategic or global information when zoomed out, and high-resolution data when zoomed in.

The default zoom behavior assumes a user will know that their mousewheel and double-clicking are associated with zooming. But sometimes you want a zoom button, whether because you cannot assume the user will know that interaction or because you want to constrain or control the zooming process in a more complicated manner. The code in Listing 7.11 creates a zoom button function and adds the necessary buttons.

Listing 7.11

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


  mapZoom.scale(newZoom).translate([newX,newY]) #c
  zoomed(); #d
}

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

#a Calculating the new scale is easy
#b Calculating the new translate settings is not so easy and requires that we recalculate the center
#c Set the zoom behavior's scale and translate settings to our new settings
#d This will redraw the map based on the updated settings

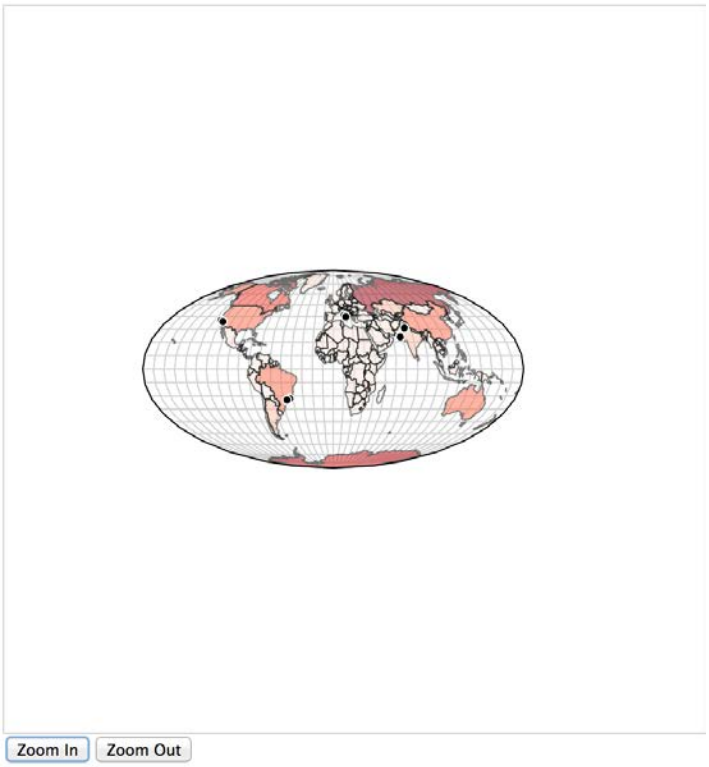


Figure 7.9 Zoom buttons and the effect of pressing “Zoom Out” five times. Because the zoom buttons modify the zoom behavior’s translate and scale, then any mouse interaction afterward will reflect the updated settings.

With this kind of styling and interactivity you have in place, you can make a map for most any application. Zooming and panning is especially important for maps because users expect to be able to zoom in and out, and also expect for the details of the map to change when they do so. In that way, geospatial information visualization is one of the most powerful forms of information visualization because there is so much literacy among users when it comes to reading and interacting with maps. But that also means that users expect a map to have certain features and functionality, and when it’s missing then they think it’s broken. So, make sure that when you create your map, it either includes this functionality or you have a good reason to leave it out.

7.3 Advanced Mapping

Up until this point, we’ve covered the aspects of creating maps that you’ll likely end up using with all your maps. There are many variations on the above that you could explore. You might want to scale your <circle> elements based on population, or use <g> elements so that you

can also provide labels like we've done earlier. But if you're making a map, it will probably have polygons and points and take advantage of bounding boxes or centroids, and will likely be tied to a zoom behavior. The exciting thing about D3 is that it lets you explore more complex ways of representing geography, with just a little more effort.

7.3.1 Creating and Rotating Globes

There's only one thing you'll do in 3D in this entire book, and that's create a globe. To do so, you don't need to load three.js or learn WebGL. Instead, we're going to take advantage of a simple trick of one of the geographic projections available in D3. That projection is the orthographic projection, which renders geographic data as it would appear from a distant point looking upon the entire globe. All we need to do is update our projection to refer to the orthographic projection and have a slightly different scale, as in Listing 7.12.

Listing 7.12

```
projection = d3.geo.orthographic()
.scale(200)
.translate([width / 2, height / 2])
.center([0,0])
```

And with that new projection, we can already see what looks like a globe in Figure 7.x.

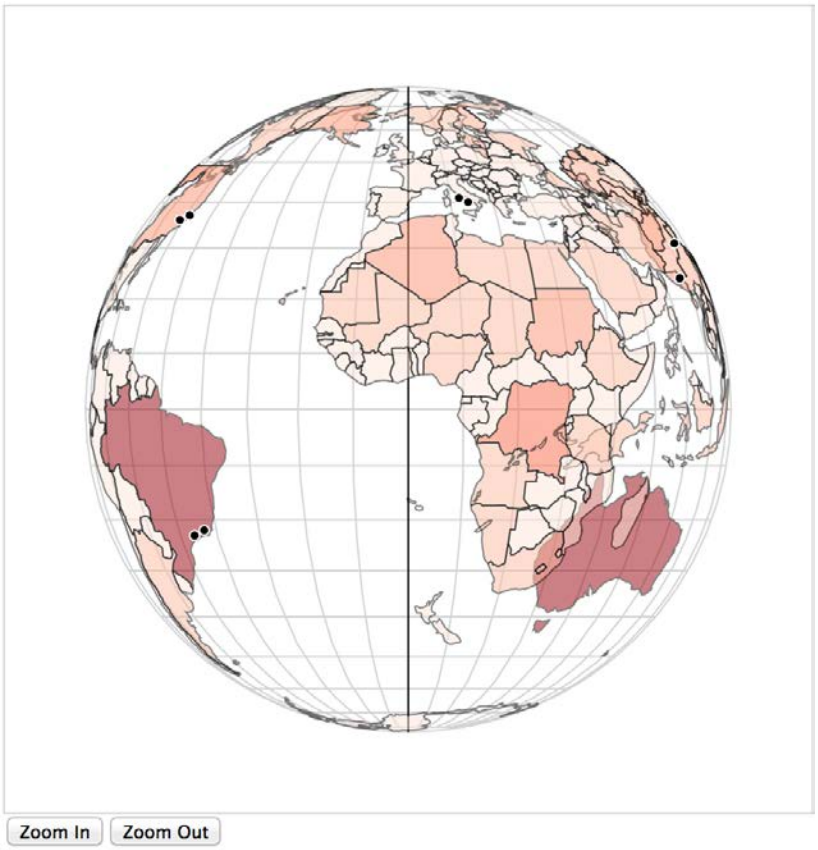


Figure 7.10 An orthographic projection shows our map in such a way as to look like a globe. Notice that even though the paths for countries are drawn over each other, they are still drawn above the graticules. Also notice that while zooming in and out works, that panning does not spin the globe but simply moves it around the canvas. Notice that the coloration of our countries is once again based on the graphical size of the country.

To make it rotate we need to use d3.mouse, which returns the current position of the mouse on the SVG canvas. By pairing this with event listeners to turn on and off a mousemove listener on the canvas, we can simulate dragging the globe, which we'll use to only rotate it along the x-axis. Because we're introducing some new behavior and its been a while since we've looked at the full code, Listing 7.13 has the entire code for creating the globe.

Listing 7.13 A draggable globe in D3

```
queue()
.defer(d3.json, "world_g.json")
.defer(d3.csv, "cities.csv")
.await(function(error, file1, file2) { createMap(file1, file2); });
```

```
function createMap(countries, cities) {

var width = 500;
     var height = 500;
     var projection = d3.geo.orthographic()
      .scale(200)
      .translate([width / 2, height / 2]);

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

    var rotateScale = d3.scale.linear() #a
.domain([0, width])
.range([-180, 180]);

    d3.select("svg")
.on("mousedown", startRotating) #b
.on("mouseup", stopRotating);

    function startRotating() {
    d3.select("svg").on("mousemove", function() { #c
            var p = d3.mouse(this);
            projection.rotate([rotateScale(p[0]), 0]);
            rotate();
});
    }

    function stopRotating() {
      d3.select("svg").on("mousemove", null);
}

    function rotate() { #d
      var currentRotate = projection.rotate()[0];

      d3.selectAll("path.graticule").attr("d", geoPath);
      d3.selectAll("path.countries").attr("d", geoPath);

      d3.selectAll("circle.cities")
      .attr("cx", function(d) {return projection([d.x,d.y])[0]})
      .attr("cy", function(d) {return projection([d.x,d.y])[1]})
    }
}
```

#a Map the width of the canvas to degrees on a sphere
#b These turn on and off the mousemove listener
#c Set the rotation angle to equal the mouse position
#d This is almost identical to our earlier zoomed() function

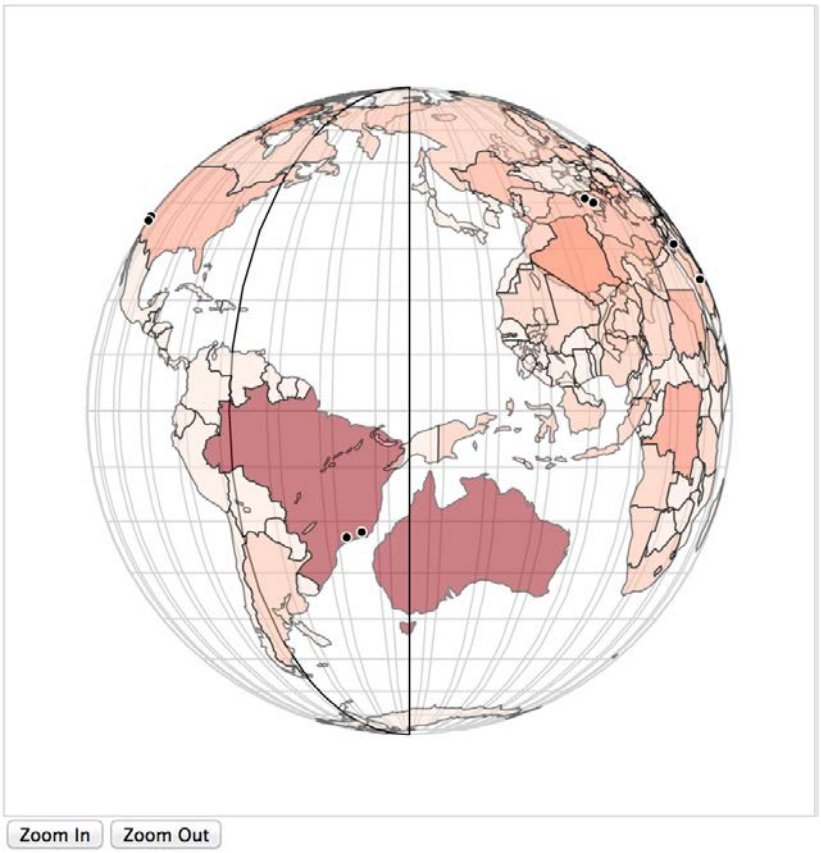


Figure 7.11 A globe with a transparent surface. We can see Australia through the globe because the projection does not by default clip this. Cities are drawn at the correct coordinates but are uniformly drawn above the features because the <circle> elements are drawn on top of the <path> elements in the DOM.

There is also a plugin by Jason Davies known as d3.geo.zoom available that abstracts this functionality and is available at: https://www.jasondavies.com/maps/rotate/

But this map still has the problem of a graphical artifact from the graticule outline, which must be removed when drawing globes. Another problem is seeing the other side of the globe through your globe. This might be a fine idea, if it weren't for the fact that it also muddles the SVG drawing code so that the shapes are drawn poorly when they get near the border (notice how poorly Antarctica looks in Figure 7.11). Also, our cities are being drawn above the paths, even when they are ostensibly on the other side of the world (like Karachi).

The path drawing can be handled with the clipAngle property of the projection which clips any paths drawn with that projection if they fall outside of a particular angle from its center.

This can be useful to show only small parts of your dataset for performance or display purposes. Here's how it would look in our new projection code:

```
projection = d3.geo.orthographic()
.scale(200)
.translate([width / 2, height / 2])
.clipAngle(90);
```

This won't work for the circles we're using for our cities because clipAngle only applies to data that's created by d3.geo.path(). For the circles, we have to ensure that they are only displayed if they fall within that clip angle. Taking this into account, we can pass a simple test in the zoomed function to determine whether a city should be displayed or not based on its coordinates.

Listing 7.13 Hiding cities on the other side of a rotated globe

```
function zoomed() {
var currentRotate = projection.rotate()[0];
projection.scale(mapZoom.scale());
d3.selectAll("path.graticule").attr("d", geoPath);
d3.selectAll("path.countries").attr("d", geoPath);

d3.selectAll("circle.cities")
.attr("cx", function(d) {return projection([d.x,d.y])[0]})
.attr("cy", function(d) {return projection([d.x,d.y])[1]})
.style("display", function(d) {
return parseInt(d.y) + currentRotate < 90 && #a
parseInt(d.y) + currentRotate > -90 ?
"block" : "none"})
    }
```

#a If this city's y position is within 90 degrees of the current rotation of the globe, then display it, otherwise hide it

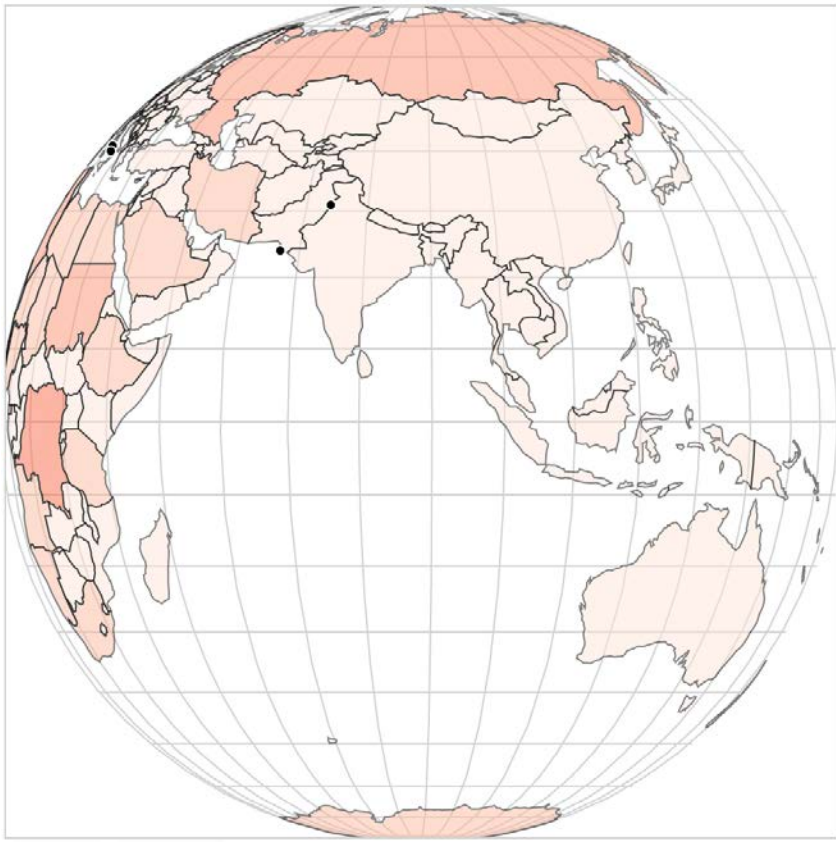


Figure 7.12 Our rotating and properly clipped globe.

You may think we're done, but there's one related issue to address now. We draw all the countries when the globe is first initialized, but many of them are clipped, and so our geo.path.area() function, which determines the area as the shape was drawn, shows even worse issues than we had with the Mercator projection. For instance, in Figure 7.12, Australia is colored as if it had an area similar to Madagascar. Fortunately, D3 also includes d3.geo.area() which determines the spherical area of a shape, which corresponds to its actual geographic area.

We could rewrite the draw code to use d3.geo.area, but instead lets just recolor our existing globe. But how do we get the data? Until now, we've assumed that the data array was exposed somewhere that our functions could get to, but what if it's outside our current scope? In this case, we can use #selectAll.data() and get an array of data associated with whatever we select (which will include undefined elements if we select HTML elements that are not bound with data). We'll see this in action more in the next chapter:

```
var featureData = d3.selectAll("path.countries").data();

var realFeatureSize =
d3.extent(featureData, function(d) {return d3.geo.area(d)});

var newFeatureColor =
d3.scale.quantize().domain(realFeatureSize).range(Reds[7]);

d3.selectAll("path.countries")
.style("fill", function(d) {return newFeatureColor(d3.geo.area(d))});
```

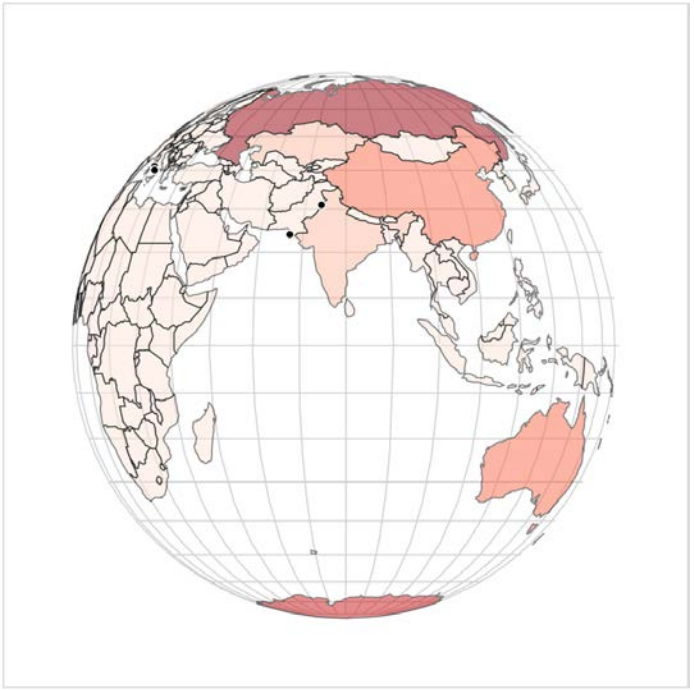


Figure 7.13 Our globe with countries colored by their geographic area, rather than their graphical area.

The spherical area of a shape as measured by d3.geo.area() is given in steradians, which means it's only a roughly proportionate area. If you want the actual square kilometers of a country or other shape, you'll still need to calculate that in a GIS package like QGIS or get that information from another source.

There are still some issues with this globe. Because we don't update the projection.center(), and we base the rotation off the current position of the mouse, then any time we drag the globe, it resets. We also don't clip the cities when we first draw them. Further, you can make a D3 globe drag in any of the three directions you could rotate a normal globe. But if you're looking for that level of functionality, then you're better off exploring the many and robust

examples available online (such as Jason Davies' http://www.jasondavies.com/maps/voronoi/capitals/). Instead, we're going to look at another exotic way of representing geodata, the satellite projection.

7.3.2 Satellite Projection

Isometric views of the world are powerful tools for storytelling. By presenting your map looking out from above some piece of the world, you're visually imparting that view on your data visualization. Imagine we had to create a map related to how the Middle East has a changing view of Europe. By crafting a satellite view looking out over the Mediterranean from the Middle East like we see in Figure 7.14, you invite your map reader to see a distant Europe from a geographical perspective in the Middle East.

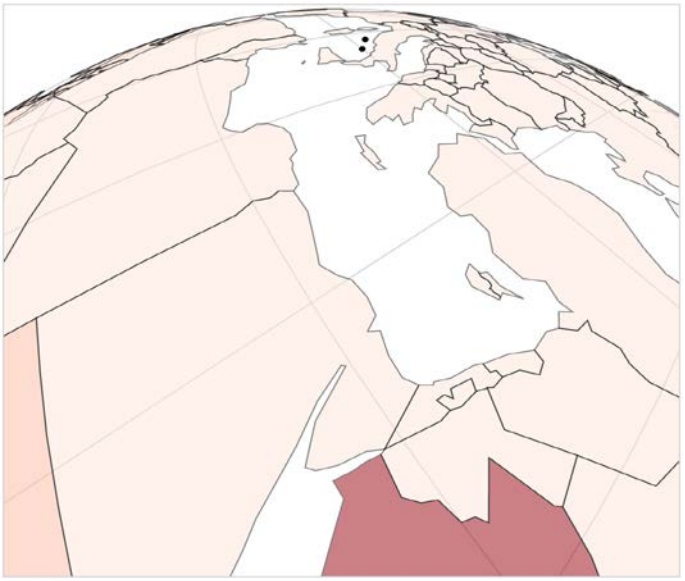


Figure 7.14 A satellite projection of our data from the Middle East facing Europe.

This is a projection just like the orthographic, mercator, and molleweide projections we previously used but, as you see in Listing 7.14, it has very specific settings for scale and rotate. It also uses new settings, tilt and distance, to determine the angle of the satellite projection.

Listing 7.14 Satellite projection settings

```
projection = d3.geo.satellite()
.scale(1330)
.translate([250,250])
.rotate([-30.24, -31, -56])
.tilt(30) #a
.distance(1.199) #b
```

```
    .clipAngle(45)
  #a The angle of the perspective on the geographic features
  #b The distance of the surface from your perspective
```

Tilt is the angle of the perspective on the data, while distance is the percentage of the radius of the Earth (so 1.119 is 11.9% of the radius of the Earth above the Earth). How did we come up with such exact settings? You have two options. The first is to understand how a tilted projection like this is mathematically described. If you have a degree in math or geography, you can look into literature for calculating this. If, like me, you don't have that kind of background, then I would suggest building a simple tool, using just the code we already explored in this chapter, to adjust the rotation, tilt, distance and scale settings interactively. That's how I did it, and you can play with this tool here:

http://bl.ocks.org/emeeks/10173187

This goes back to my advice in regard to understanding how the Sankey layout works. Use information visualization to visualize how the functions work so that you can better understand them and find the right settings. Otherwise, you're going to need to take a course in GIS or wait for someone to write "D3.js Mapping in Action". We're going to shift gears now from visualization and back into geodata structure as we explore a library that was developed by Mike Bostock and intimately tied to D3 mapping: Topojson.

7.4 TopoJSON Data and Functionality

TopoJSON (https://github.com/mbostock/topojson) is, fundamentally, three different things. First of all, it is a data standard for geographic data, and an extension of GeoJSON. Secondly, it is a library that runs in node.js to create TopoJSON-formatted files from GeoJSON files. Thirdly, it is a library that runs in JavaScript that processes TopoJSON-formatted files to create the data objects necessary to render them with libraries like D3. We will not deal with the second form at all, and only examine the first in a cursory manner as we learned about rendering TopoJSON data, merging it, and using it to find a feature's neighbors.

7.4.1 TopoJSON the file format

The difference between GeoJSON files and TopoJSON files is that while GeoJSON records for each feature an array of longitude and latitude points that describe a point, line or polygon, TopoJSON stores for each feature an array of arcs. An arc is any distinct segment of a line that is shared by one or more features in your dataset. The shared border between the United States and Mexico is a single arc that is referred to in the arcs array of the feature for the United States and the arcs array of the feature for Mexico.

Because most datasets have shared segments, TopoJSON often produces significantly smaller datasets. This is one of its appeals. Another appeal is that if you know what segments are shared, then you can do interesting things with the data, like easily calculating the neighboring features, the shared border, or even merging features.

TopoJSON stores the arcs as a reference to a particular arc in a master list of arcs that defines the actual coordinates of that arc. Which is why you need the Topojson.js library included in any website you're using to create maps with TopoJSON, it's what changes TopoJSON into a format that D3 can read and create graphics from.

7.4.2 Rendering TopoJSON

Because TopoJSON stores its data in a very different format from the GeoJSON structure that is expected by d3.geo.path(), you need to include Topojson.js and use it to process TopoJSON data to produce GeoJSON features. This is rather straightforward and can be done within a call to our new datafile as seen in Listing 7.x and showing the properly formatted features in your console as we see in Figure 7.x.

Listing 7.x

```
queue()
    .defer(d3.json, "world.TopoJSON")
    .defer(d3.csv, "cities.csv")
    .await(function(error, file1, file2) { createMap(file1, file2); });

  function createMap(file1, file2) {

    var worldFeatures = Topojson.feature(file1, file1.objects.countries) #a
    console.log(worldFeatures);
  }
```

#a Notice that your actual TopoJSON file has a property "objects" which all TopoJSON files have, but "countries" is specific to this file and might be "rivers" or "land" or other property names in other files

Figure 7.x TopoJSON data formatted using Topojson.feature() results in an array of objects with geometry represented as an array of coordinates just like the features that come out of a geojson file.

Now that it's in the format we want, we can send it to our existing code and draw this array of features just like we did with the features we loaded from world.geojson, replacing our earlier "countries" with the worldFeatures variable declared in Listing 7.x. That's all that most people do with TopoJSON, and they're happy for it because TopoJSON data is significantly smaller than geojson data. But because we know the topology of the features in a TopoJSON data file, we do some interesting geographic tricks with it.

7.4.3 Merging

The TopoJSON library provides you with the capacity to create new features by merging existing features. This can be used to create a new feature for "North America" by merging the countries in North America, or "The United States in 1912" by merging the states that were part of the United States in 1912. In Listing 7.15, we can see the code to not only draw a map using our new TopoJSON data file, but also to merge all the countries that have a center west of 0° longitude. The results, in Figure 7.15, show that merging doesn't just combine contiguous features but separate features into a multipolygon.

Listing 7.15 Rendering and merging TopoJSON

```
     queue()
      .defer(d3.json, "world.TopoJSON")
      .defer(d3.csv, "cities.csv")
      .await(function(error, file1, file2) { createMap(file1, file2); });

    function createMap(topoCountries, cities) {
      var countries =
topojson.feature(topoCountries, topoCountries.objects.countries)
      var width = 500;
      var height = 500;
      var projection = d3.geo.mollweide()
      .scale(120)
      .translate([width / 2, height / 2])
      .center([20,0])

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

      var featureSize =
d3.extent(countries.features, function(d) {return geoPath.area(d)});
      var countryColor =
d3.scale.quantize().domain(featureSize).range(Reds[7]);

      var graticule = d3.geo.graticule();

      d3.select("svg").append("path")
      .datum(graticule)
      .attr("class", "graticule line")
      .attr("d", geoPath)
      .style("fill", "none")
      .style("stroke", "lightgray")
      .style("stroke-width", "1px");

      d3.select("svg").append("path")
      .datum(graticule.outline)
```

```
    .attr("class", "graticule outline")
    .attr("d", geoPath)
    .style("fill", "none")
    .style("stroke", "black")
    .style("stroke-width", "1px");

    d3.select("svg").selectAll("path.countries")
    .data(countries.features) #a
    .enter()
    .append("path")
    .attr("d", geoPath)
    .attr("class", "countries")
    .style("fill", function(d) {return countryColor(geoPath.area(d))})
    .style("stroke-width", 1)
    .style("stroke", "black")
    .style("opacity", .5)

    d3.select("svg").selectAll("circle").data(cities)
    .enter()
    .append("circle")
    .style("fill", "black")
    .style("stroke", "white")
    .style("stroke-width", 1)
    .attr("r", 3)
    .attr("cx", function(d) {return projection([d.x,d.y])[0]})
    .attr("cy", function(d) {return projection([d.x,d.y])[1]})

    mergeAt(0); #b

    function mergeAt(mergePoint) {

    var filteredCountries = topoCountries.objects.countries.geometries
.filter(function(d) { #c
      var thisCenter = d3.geo.centroid(
      topojson.feature(expCountries, d) #d
      );
      return thisCenter[1] > mergePoint? true : null; #e
    }
      )

  d3.select("svg").insert("g", "circle")
       .datum(topojson.merge(countries1, filteredCountries)) #f
       .insert("path")
       .style("fill", "gray")
       .style("stroke", "black")
       .style("stroke-width", "2px")
       .attr("d", geoPath)
    }
  }
```

#a Once processed by Topojson.features, we use exactly the same methods to render the features
#b Our simple merge function
#c We're working with the TopoJSON dataset
#d To use geo.centroid, we convert each feature into geojson
#e This will result in an array of only the corresponding geometries
#f Use datum because merge returns a single multipolygon

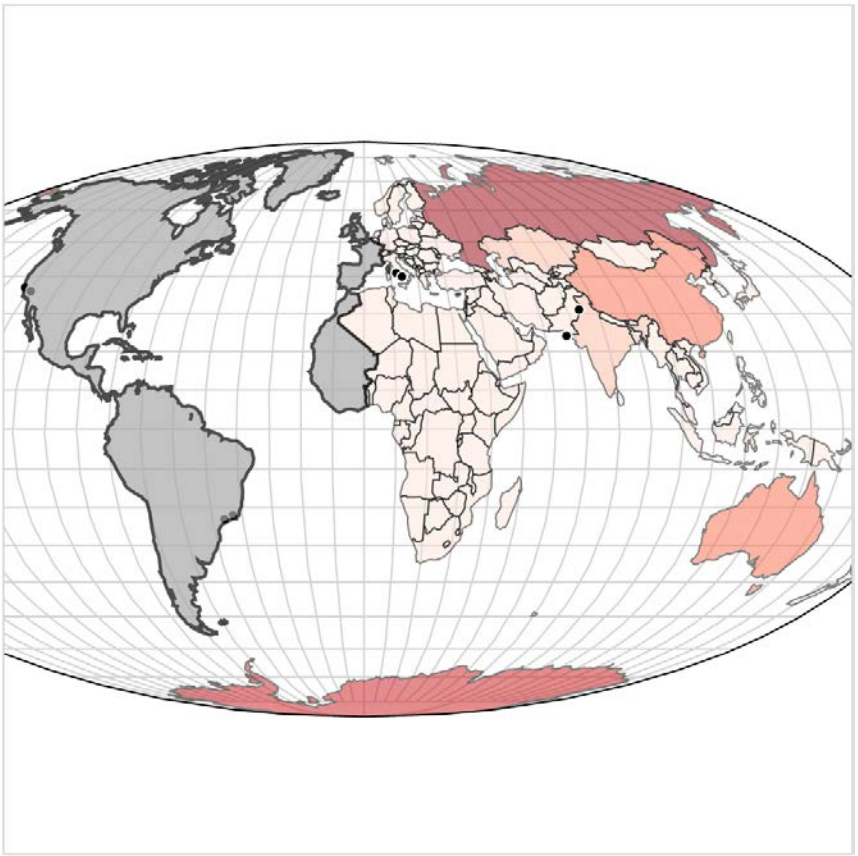


Figure 7.15 Merging based on the centroid of a feature results in the feature in gray as a single merged feature made up of many separate polygons.

If you adjust the mergeAt test just slightly to look at the x coordinate or to see features where the value is greater rather than less, you can see a single feature for each permutation of less than or greater than 0° latitude or longitude in Figure 7.16. Notice in each case that it's a single feature but not a single polygon.

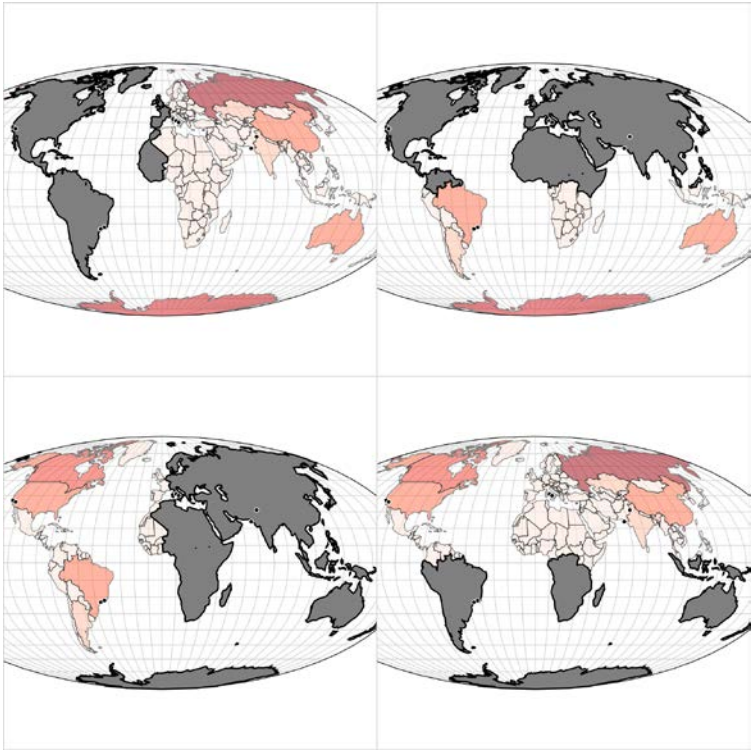


Figure 7.16 With slight adjustment of the merge settings, we can achieve something of a northern, southern, eastern and western hemisphere as a merged feature. Notice that since this is based on centroid, then at the bottom left we can see a piece of Eastern Russia as part of our merged feature, along with Antarctica.

A quick note for those who might want to continue working in topologies: Topojson.merge has a sister function mergeArcs that will allow you to merge shapes but keep them in TopoJSON format. Why would you want to maintain arcs? Because then you could continue to use TopoJSON functionality like merging, creating meshes or finding neighbors of your newly merged features.

7.4.4 Neighbors

Since we already know when features share arcs, we also know what features neighbor each other. Because of this, there's a simple function, Topojson.neighbors, that will build an array of all the features that share a border each feature. You can use this array to easily identify neighboring countries in our dataset using the code in Listing 7.17. The results of the interaction provided by this code can be seen in Figure 7.17.

Listing 7.17 Calculating Neighbors and Interactive Highlighting

```
var neighbors =
```

```
topojson.neighbors(topoCountries.objects.countries.geometries); #a

    d3.selectAll("path.countries")
    .on("mouseover", findNeighbors)
    .on("mouseout", clearNeighbors)

    function findNeighbors (d,i) {
      d3.select(this).style("fill", "red"); #b
      d3.selectAll("path.countries")
.filter(function (p,q) {return neighbors[i].indexOf(q) > -1})
.style("fill", "green") #c
    }

    function clearNeighbors () {
      d3.selectAll("path.countries").style("fill", "gray") #d
      }
```

#a creates an array indicating neighbors by their array position
#b color the country you hover over red
#c color all neighbors green
#d color all countries gray to "clear" results

Figure 7.17 Hover behavior displaying the neighbors of France using TopoJSON's neighbor function. Because of the overseas French department of Guyana, France is considered to be neighbors with Brazil and Suriname. This is because France is represented as a multipolygon in the data, and any neighbors with any of its shapes are returned as neighbors.

TopoJSON is a powerful new technology that provides tremendous power for web map development. Learning how it models data and the functionality that it provides are key to creating maps that impress users. As we move into exploring traditional web tile mapping, you'll see that you can combine more traditional web mapping techniques with the advanced functionality provided by TopoJSON and D3's geo functions to make incredibly sophisticated web maps.

7.5 Tile Mapping with d3.geo.tile

So far we've made chloropleth maps, some of which are pretty simple and others, like the satellite projection or the globe, are rather exotic. But none of our maps have terrain, or satellite imagery. That's because that kind of data--raster or image data--is not nearly as lightweight as vector data. Just think about the size of a picture you take with the camera on your phone and imagine how large an image must be if you want to give your user the ability to zoom in to any street in the world.

To get around the problem of these massive images, web mapping uses tiles to display satellite and terrain data. A high resolution satellite image of a city, for instance, would be cut into 256px by 256px tiles at as many zoom levels as are appropriate and stored on a server in directories indicating the zoom and position of those tiles. It sounds like it might be a lot of work to make tiles, but fortunately, you don't have to since companies like Mapbox (mapbox.com) provide you with tiles and the tools like TileMill to customize them (both free versions and commercial versions, depending on how many visitors your site will receive).

If you open it up and take a look at it, you'll see that tile.js is a pretty small file. That's because geotiles are pretty simple. Each tile is simply a raster image (typically a PNG) that represents one square of the Earth somewhere, with its filename indicating where and at what zoom level. The d3.geo.tile() function parses that filename and directory structure for you so that you can use these tiles in your map. To use it, though, you have to calibrate the scale and translate of your projection as well as your zoom behavior.

Listing 7.18 a simple Tile Map

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

  d3.select("svg").append("g").attr("id", "tiles"); #a

var tile = d3.geo.tile() #b
    .size([width, height]);

var projection = d3.geo.mercator()
    .scale(120)
    .translate([width / 2, height / 2]);

var center = projection([12, 42]);

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

```
var zoom = d3.behavior.zoom()
    .scale(projection.scale() * 2 * Math.PI)
    .translate([width - center[0], height - center[1]])
    .on("zoom", redraw);

    d3.select("svg").call(zoom);
    redraw();

    function redraw() {
      var tiles = tile #c
      .scale(zoom.scale())
      .translate(zoom.translate())
      ();

      var image = d3.select("#tiles")
.attr("transform",
"scale(" + tiles.scale + ") translate(" + tiles.translate + ")") #d
    .selectAll("image")
      .data(tiles, function(d) { return d; }); #e

      image.exit() #f
      .remove();

      image.enter().append("image") #g
      .attr("xlink:href",
function(d) { return "http://" + ["a", "b", "c", "d"][Math.random() * 4 | 0]
+ ".tiles.mapbox.com/v3/examples.map-zgrqqx0w/" + d[2] + "/" + d[0] + "/" +
d[1] + ".png"; }) #h
      .attr("width", 1)
      .attr("height", 1)
      .attr("x", function(d) { return d[0]; })
      .attr("y", function(d) { return d[1]; });
    }
```

#a A group to hold our tiles underneath any other drawn features
#b This will be the function we use to create our tiles
#c This will be the dataset we use to create the images
#d Proper transform settings are generated based on the current zoom
#e Bind the tiles data to svg:image elements
#f Remove any that are off-screen
#g Append new ones
#h The path to the tiles is generated by tile.js for services like MapBox

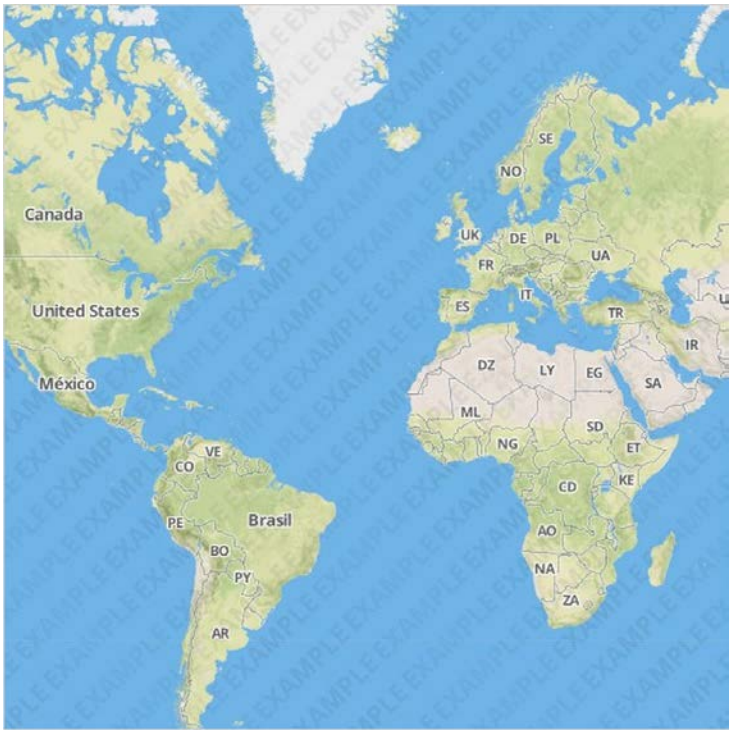


Figure 7.18 Your first tiled map, using pregenerated tiles from Mapbox

Of course, you'll want to add your points and polygons to this map, and to do so isn't very different from the code we just saw in Listing 7.19 and the code we've been working with throughout the chapter. We'll use the same data, but add a simple function on the display styling of the countries to make half of them disappear, so we can see our tiles in Figure 7.19.

Listing 7.19 a simple Tile Map

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

d3.select("svg").append("g").attr("id", "tiles");

var tile = d3.geo.tile()
    .size([width, height]);

var projection = d3.geo.mercator()
    .scale(120)
    .translate([width / 2, height / 2]);

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

```
var zoom = d3.behavior.zoom()
    .scale(projection.scale() * 2 * Math.PI)
    .translate([width - center[0], height - center[1]])
    .on("zoom", redraw);

    d3.select("svg").call(zoom);
    redraw();

    d3.select("svg").selectAll("path.countries").data(countries.features)
    .enter()
    .append("path")
    .attr("d", geoPath)
    .attr("class", "countries")
    .style("fill", function(d) {return countryColor(geoPath.area(d))})
    .style("stroke-width", 1)
    .style("stroke", "black")
    .style("opacity", .5)

    function redraw() {
      var tiles = tile
      .scale(zoom.scale())
      .translate(zoom.translate())
      ();

      var image = d3.select("#tiles").attr("transform", "scale(" +
tiles.scale + ")translate(" + tiles.translate + ")")
    .selectAll("image")
      .data(tiles, function(d) { return d; });

      image.exit()
      .remove();

      image.enter().append("image")
      .attr("xlink:href", function(d) { return "http://" + ["a", "b", "c",
"d"][Math.random() * 4 | 0] + ".tiles.mapbox.com/v3/examples.map-zgrqqx0w/" +
d[2] + "/" + d[0] + "/" + d[1] + ".png"; })
      .attr("width", 1)
      .attr("height", 1)
      .attr("x", function(d) { return d[0]; })
      .attr("y", function(d) { return d[1]; });

      projection
      .scale(zoom.scale() / 2 / Math.PI) #a
      .translate(zoom.translate());

      d3.selectAll("path.countries")
      .attr("d", geoPath);

      d3.selectAll("circle")
    .attr("cx", function(d) {return projection([d.x,d.y])[0]})
    .attr("cy", function(d) {return projection([d.x,d.y])[1]})
    }
```

#a Note that we're not taking zoom.scale() directly like we did before but processing it to get the properly formatted scale for our mercator projection.

Figure 7.19 A tile map overlaid with the point and polygon data we've worked with throughout this chapter.

7.6 Further Reading for Web Mapping

As I said in the beginning of this chapter, there's so much you can do with D3's mapping capabilities that it could fill an entire book. Here are a few other capabilities we didn't cover in this chapter:

7.6.1 Transform Zoom

The method we used for our zoom behavior in this chapter is known as "projection zoom" and recalculates mathematically the shape of features based on a change in scale and translation. If, however, you're using a projection like Mercator that is flat, then you can achieve faster performance by tying the zoom behavior change in scale and translate to your features' SVG transform. One issue you'll run into very quickly is that font-size and stroke-width are affected by SVG transform, and so you'll need to adjust those settings on-the-fly.

7.6.2 Canvas drawing

There is a .context function of d3.geo.path that allows you to easily draw your vector data to a <canvas> element, which can dramatically improve speed in certain cases and also allows you to use dataUri to create a PNG dynamically for users to save or share on social media.

7.6.3 Raster Reprojection

Jason Davies and Mike Bostock have both provided examples on bl.ocks.org/mbostock/ and www.jasondavies.com/maps/raster/satellite/ of reprojecting not just vector data but the tile data used in tile maps. You can use this to show a satellite-projected terrain map, or a terrain map with the Molleweide projection we used earlier.

7.6.4 Hexbins

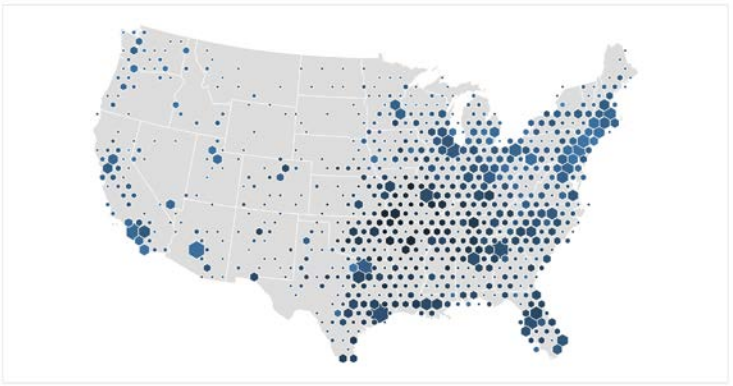


Figure 7.20 An example of hexbinning by Mike Bostock used to show the locations of Wal-Mart stores in the United States. Available at http://bl.ocks.org/mbostock/4330486

The d3.hexbin plugin allows you to easily create hexbin overlays for your maps that can be tremendously effective in relaying quantitative data that you have in point form and want to aggregate by area.

7.6.5 Voronoi Diagrams

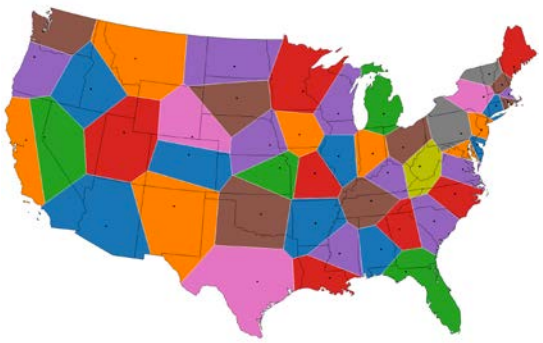


Figure 7.21 An example of a Voronoi diagram used to split the United States into polygons based on the closest state capital. Available at http://www.jasondavies.com/maps/voronoi/us-capitals/

As with hexbins, if you only have point data and want to create some kind of area data from it, you can use the d3.geom.voronoi function to derive polygons from points.

7.6.6 Cartograms

If you distort the area or length of a geographic object to show other information, that's known as a cartogram. So, you could distort the streets of your city based on the actual time it would take to get to drive along them, or make the size of countries on a world map bulge or shrink based on population. While there are no simple functions that create these, there are several examples of how to create them in D3 such as one created by Jason Davies at http://www.jasondavies.com/maps/dorling-world/ and one created by Mike Bostock at http://bl.ocks.org/mbostock/4055908 or the cost cartogram I built at orbis.stanford.edu.

7.7 Summary

In this chapter, we've covered the incredible breadth of geospatial information visualization capabilities present in D3. Maps are a core aspect of information visualization and the creation of rich interactive web sites, and D3's geo functions allow you to make maps that are much richer than the pushpin web maps that you typically see on the web. To make those maps, we needed to walk through a massive amount of functions and concepts, including:

- Understanding the GeoJSON spatial data format
- Creating simple maps
- Creating map components like graticules
- Computing geospatial attributes like centroids and bounding boxes
- Giving the user rich interactive panning and zooming
- Using different projections
- Creating globes
- Rendering TopoJSON and using it to merge features and find neighbors
- Creating tile maps with TopoJSON overlays.

In the next chapter, we'll start using D3 selections and data-binding to create galleries and tables using traditional DOM elements.

8

Traditional DOM Manipulation with D3

Many introductions to D3 start with sizing elements to create a bar chart. They do this because they figure you are probably a web developer and that you won't be as intimidated by a div as you would be by an SVG rectangle. This book even began by creating a set of <p> elements the first time we saw data-binding in action in Chapter 1. But then these tutorials (and this book) quickly transition into the creation of SVG elements, with an emphasis on the graphical display of information. This is at odds with traditional web development, which is focused on the presentation of blocks of text, images, buttons, lists and so on. As a result, it seems like D3 is for data visualization, but somehow not for manipulating traditional DOM elements like paragraphs and divs and lists. The benefit of using D3 to create these kinds of elements is that you can use D3 transitions, data-binding and other functionality to make a more interactive and dynamic website.

The principles at work in D3 not only can be used for traditional DOM elements, but in many cases should be used that way. In this chapter, we'll use D3 to create a simple spreadsheet as well as an image gallery. We'll also explore how to use HMTL5 canvas to draw and save images. This won't include an exhaustive example of canvas, because that's beyond the scope of the book, but it will give you the fundamentals to deploy it in tandem with D3 in your applications. In each case, we'll use D3 data-binding, transitions and selections the same way we have to make charts but instead to make more traditional HTML elements.

By using the same datasets and functions to deal with your DOM elements as you do with your SVG elements, you make it easier to tie them together and reduce the amount of syntax you need to learn to deploy rich sites. In later chapters, we'll see these different methods of presenting information working in together in tandem.

8.1 Setup

As you might expect, we'll need to make a few changes to the files we're working with now that we're going to do coding that resembles more traditional web development. In one case, this means simplifying, because our HTML page will lose the <svg> element necessary for representing SVG graphics, but in another sense it means making things more complex, because while we used CSS primarily for graphical changes with SVG, we need to use it for more than that when working with traditional DOM elements.

8.1.1 CSS

There's going to be more CSS when you're working with traditional DOM elements, because if you want to manipulate them in the way you manipulate SVG elements, you typically need to set them up a bit differently, for instance if you want to place HTML elements precisely like you do with SVG elements. Also, most of the graphical aspects of these elements are not set with attributes like they are in SVG, but with styles. If it's not clear to you the difference between styles and attributes, you can look back at Chapter 1 where I covered the difference between styles, attributes and properties. This shouldn't be a surprise for anyone who's had some experience working with CSS, because it's usually the case in the complex examples and under the hood when you use JavaScript libraries (for instance, if you look at the CSS of various libraries that provide autocomplete or more sophisticated UI elements, you'll see that they typically combine JavaScript with a variety of styles assigned to complex CSS selectors). In Listing 8.x you'll see the stylesheet we'll use for this chapter. Some of these elements, like <img.infinite>, won't be seen until the end of the chapter.

Listing 8.x Stylesheet for Chapter 8

```
tr {
  border: 1px gray solid;
}

td {
  border: 2px black solid;
}

div.table {
  position:relative;
}
div.data {
  position: absolute;
  width: 90px;
  padding: 0 5px;
}

div.head {
  position: absolute;
}

div.datarow {
  position: absolute;
  width: 100%;
```

```
  border-top: 2px black solid;
  background: white;
  height: 35px;
  overflow: hidden;
}

div.gallery {
  position: relative;
}

img.infinite {
  position: absolute;
  background: rgba(255,255,255,0);
  border-width: 1px;
  border-style: solid;
  border-color: rgba(0,0,0,0);
}
```

8.1.2 HTML

The HTML is going to be pretty simple: a single with the ID value of “traditional” in our <body> element, shown in Listing 8.x. Of course, we’ll still need a reference to d3.js, but otherwise it’s a pretty spartan HTML page. We’ll either modify or add new elements to that div for every example.

Listing 8.x chapter8.html

```
<!doctype html>
<html>
<script>d3v3.min.js</script>
<body>
<div id="traditional">
</div>
</body>
<html>
```

8.2 Spreadsheet

Let’s assume we want to take our tweets data that we’ve been working with throughout the book and present it as a spreadsheet. It might help to first think of spreadsheets as a kind of information visualization. They have an x-axis (columns) and a y-axis (rows) and visual channels to express information (not only color applied to text and cells but position and font styling). This is especially true of large spreadsheets, because they also use aggregated functions to tally results.

8.2.1 Making a spreadsheet with table

The easiest way to make a spreadsheet is to leverage the HTML <table> element and simply use data-binding to create rows and cells. As we’ve done in the past, we’ll create key values by using d3.keys on one of the entries in our dataset, which will be the venerable tweets.json. After we bind the dataset to the table, we need to create individual cells, which we can accomplish by taking each JSON object and applying d3.entries() to it, which turns an object into an array of key-value pairs perfectly suited for D3 data-binding.

Listing 8.x Simple Spreadsheet Example

```
d3.json("tweets.json",function(error,data) {
createSpreadsheet(data.tweets)});

      function createSpreadsheet(incData) {

        var keyValues = d3.keys(incData[0]) #a

        d3.select("#traditional")
        .append("table")

        d3.select("table")
        .append("tr")
        .attr("class", "head")
        .selectAll("th")
        .data(keyValues) #b
        .enter()
        .append("th")
        .html(function (d) {return d})

        d3.select("table")
        .selectAll("tr.data")
        .data(incData).enter() #c
        .append("tr")
        .attr("class", "data")

        d3.selectAll("tr")
        .selectAll("td")
        .data(function(d) {return d3.entries(d)}) #d
        .enter()
        .append("td")
        .html(function (d) {return d.value})
      }
```

#a This won't work if your objects have differing numbers of attributes but usually that's not the case
#b Your header row is created from your keys
#c Each row is created for a tweet
#d Each cell is created for an entry in each datapoint

The result of Listing 8.x is a decent tabular presentation of our tweets data as seen in Figure 8.x. Notice that the arrays have been transformed into comma-delimited strings.

user	content	timestamp	retweets	favorites
Al	I really love seafood.	Mon Dec 23 2013 21:30 GMT-0800 (PST)	Raj,Pris,Roy	Sam
Al	I take that back, this doesn't taste so good.	Mon Dec 23 2013 21:55 GMT-0800 (PST)	Roy	
Al	From now on, I'm only eating cheese sandwiches.	Mon Dec 23 2013 22:22 GMT-0800 (PST)		Roy,Sam
Roy	Great workout!	Mon Dec 23 2013 7:20 GMT-0800 (PST)		
Roy	Spectacular oatmeal!	Mon Dec 23 2013 7:23 GMT-0800 (PST)		
Roy	Amazing traffic!	Mon Dec 23 2013 7:47 GMT-0800 (PST)		
Roy	Just got a ticket for texting and driving!	Mon Dec 23 2013 8:05 GMT-0800 (PST)		Sam,Sally,Pris
Pris	Going to have some boiled eggs.	Mon Dec 23 2013 18:23 GMT-0800 (PST)		Sally
Pris	Maybe practice some gymnastics.	Mon Dec 23 2013 19:47 GMT-0800 (PST)		Sally
Sam	@Roy Let's get lunch	Mon Dec 23 2013 11:05 GMT-0800 (PST)	Pris	Sally,Pris

Figure 8.x A simple tabular display of the data found in tweets.json using <table>, <tr> and <td> elements.

As you can see, it's a fundamentally simple task to take data and bind it to create traditional DOM elements in the same way we've bound data to create SVG elements. You could have created an <ol> element and appended <li> elements to it from your dataset just as easily. You can also use D3's .on function to assign event listeners to highlight cells or rows by changing their background or font color. But rather than do that with a spreadsheet built using <table> we're going to build another spreadsheet entirely out of elements.

8.2.2 Making a spreadsheet with divs

Why use elements? Because we're going to start moving around our cells and rows however we want, and by the time we override all the styles that make a table and its constituent elements work, we're better off just starting fresh with a div. By setting the position to absolute, you can use D3 transitions to move them around in the same way we move around SVG in our earlier examples. This means we need to apply a bit more CSS to make the elements take up the right amount of space, whereas <table> does that for you, but the added flexibility is worth it. A quick note for those of you, like me, that always forget the one crazy rule of positioning DOM elements: Elements set to position:relative need to have a parent set to position:relative or position:absolute, which is why we create a parent (div.table) with position:relative to hold the elements that make up our table.

Listing 8.x A spreadsheet made of divs

```
d3.json("tweets.json",function(error,data) {
createSpreadsheet(data.tweets)});

      function createSpreadsheet(incData) {

        var keyValues = d3.keys(incData[0])

        d3.select("#traditional")
        .append("div")
```

```
        .attr("class", "table") #a

        d3.select("div.table")
        .append("div")
        .attr("class", "head")
        .selectAll("div.data")
        .data(keyValues) #b
        .enter()
        .append("div")
        .attr("class", "data")
        .html(function (d) {return d})
        .style("left", function(d,i) {return (i * 100) + "px"}); #c

        d3.select("div.table")
        .selectAll("div.datarow")
        .data(incData, function(d) {return d.content})
        .enter()
        .append("div")
        .attr("class", "datarow")
        .style("top", function(d,i) {return (40 + (i * 40)) + "px"});

        d3.selectAll("div.datarow")
        .selectAll("div.data")
        .data(function(d) {return d3.entries(d)})
        .enter()
        .append("div")
        .attr("class", "data")
        .html(function (d) {return d.value})
        .style("left", function(d,i,j) {return (i * 100) + "px"});
```

#a It's a <div.table> not a <table>
#b Same as before
#c Instead of x/y or transform, HTML elements have top/bottom/left/right

There are some obvious oversimplifications in this code. As we can see in Figure 8.x, each column being the same width doesn't make much sense, and while we could create a method for measuring the maximum size of the text in that field, that's not where I want to go with this chapter, which is more about showing a general overview of manipulating elements like these and not making the ultimate D3 spreadsheet.

user	content	timestamp	retweets	favorites
Al	I really love seafood.	Mon Dec 23 2013 21:30	Raj,Pris,Roy	Sam
Al	I take that back, this	Mon Dec 23 2013 21:55	Roy	
Al	From now on, I'm only	Mon Dec 23 2013 22:22		Roy,Sam
Roy	Great workout!	Mon Dec 23 2013 7:20		
Roy	Spectacular oatmeal!	Mon Dec 23 2013 7:23		
Roy	Amazing traffic!	Mon Dec 23 2013 7:47		
Roy	Just got a ticket for	Mon Dec 23 2013 8:05		Sam,Sally,Pris
Pris	Going to have some boiled	Mon Dec 23 2013 18:23		Sally
Pris	Maybe practice some	Mon Dec 23 2013 19:47		Sally
Sam	@Roy Let's get lunch	Mon Dec 23 2013 11:05	Pris	Sally,Pris

Figure 8.x Our improved spreadsheet built with elements. You can see how each div is the same width, and that because of our overflow settings it displays as much of the text as it can.

8.2.3 Animating your spreadsheet

It's time now to add some interactivity to the static chart we see in Figure 8.x. One traditional interaction technique applied to spreadsheets is the ability to sort them. We can do that with our spreadsheet by simply sorting the data and rebinding it to the cells, just like we've done in the past with SVG elements. By tying this to the same transition() behavior we've used before, we can also animate that sorting.

Listing 8.x Sorting Functions

```
        d3.select("#traditional").insert("button", ".table").on("click",
sortSheet).html("sort")
        d3.select("#traditional").insert("button", ".table").on("click",
restoreSheet).html("restore") #a

function sortSheet() {
          var dataset = d3.selectAll("div.datarow").data();
            dataset.sort(function(a,b) {
              var a = new Date(a.timestamp); #b
              var b = new Date(b.timestamp);
            return a>b ? 1 : (a<b ? -1 : 0);
            d3.selectAll("div.datarow")
            .data(dataset, function(d) {return d.content})
            .transition()
            .duration(2000)
```

```
            .style("top", function(d,i) {return (40 + (i * 40)) + "px"});
        }
        function restoreSheet() {
          d3.selectAll("div.datarow")
            .transition()
            .duration(2000)
            .style("top", function(d,i) {return (40 + (i * 40)) + "px"});
        }
```

#a Simple controls for our spreadsheet
#b Cast as date and sort the array so that earlier tweets are lower in the array

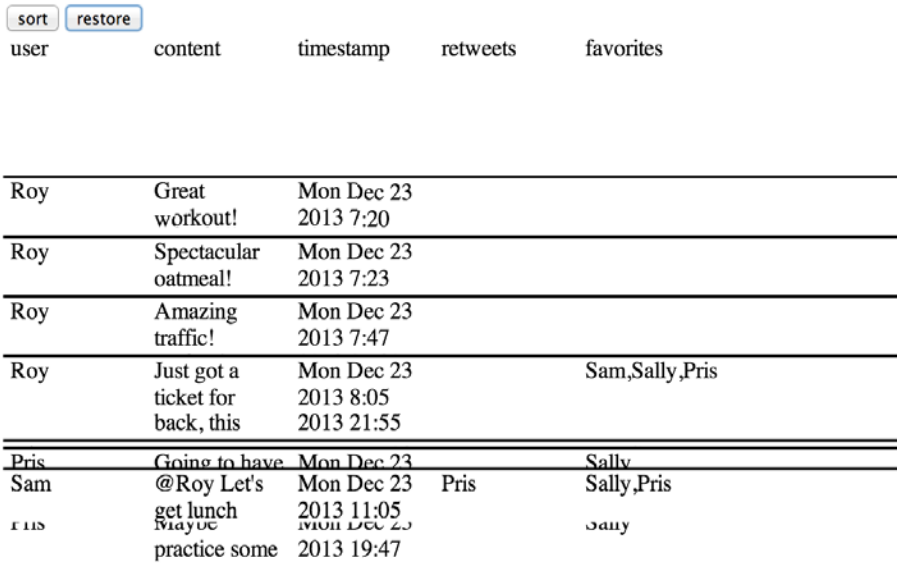


Figure 8.x The rows of our spreadsheet in the middle of the sort function.

Now we have a spreadsheet with sortable rows which float over and under each other after you click the sorting button. In Figure 8.x we see that animation caught in an intermediate state. If we want to sort the columns, though, we need to do something slightly different as we can see in Listing 8.x.

Listing 8.x Column sorting

```
        d3.select("#traditional").insert("button", ".table").on("click",
sortColumns).html("sort columns ")
        d3.select("#traditional").insert("button", ".table").on("click",
```

```
restoreColumns).html("restore columns")

        function sortColumns() {
          d3.selectAll("div.datarow")
          .selectAll("div.data")
            .transition()
            .duration(2000)
        .style("left", function(d,i,j) {return (Math.abs(i - 4) * 100) +
"px"});
        }

        function restoreColumns() {
          d3.selectAll("div.datarow")
          .selectAll("div.data")
            .transition()
            .duration(2000)
        .style("left", function(d,i,j) {return (i * 100) + "px"});
        }
```

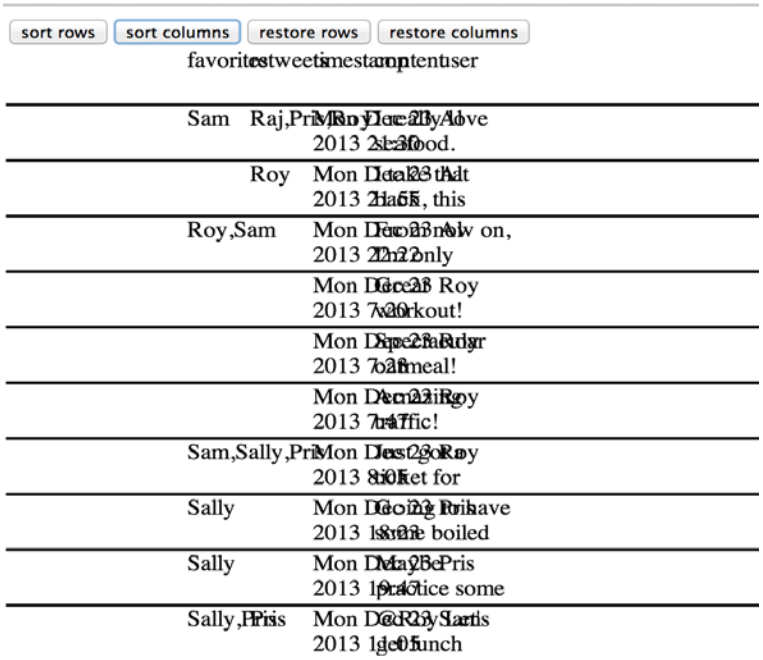


Figure 8.x Sorting columns in our sheet. Because we didn't define a background value for the divs, the text floats over itself. In this screenshot, you can see that I've added all the buttons for sorting and restoring columns and divs.

There you have it, a sortable animated spreadsheet that, if we catch it in mid-transition as we have in Figure 8.x, is rather messy. It's animated and interactive and data-driven with no SVG at all. Rather than adding more interactivity to our spreadsheet, we're going to switch gears and focus on a second kind of traditional component of a web page: image galleries. But before we get to that, we'll need some images. Rather than load them from external files, though, we're going to draw our own PNGs using HTML5 canvas, an API made for drawing static images. We're not going to dive deep into canvas, we're just going to use it to create some circles with numbers on them to stand in for whatever images we might put into a gallery.

8.3 Canvas

Even though we're not going to use canvas too much here, you should begin to recognize that while the canvas drawing syntax like that in Listing 8.x is different from SVG, it's something you could easily tie to D3. You might do that because you want to create images like we're doing here. Or you might use canvas because you can achieve much-improved performance using canvas to draw your graphics if you're dealing with very large datasets. There are a number of examples online (especially with maps like this: http://bl.ocks.org/mbostock/3783604 but also this implementation of a voronoi diagram in canvas: http://bl.ocks.org/mbostock/6675193) that use canvas instead of SVG for D3. But for our purposes, we don't need much code to create our image.

We're going to use canvas to draw some circles with some numbers in them. We're going to do this so that we can have a set of images that we can use for our gallery. Your gallery will probably have a set of images in a directory or called from an API, but since we don't have that here, we're just going to create them on the fly. At the same time, you'll get a sense of the functionality of the canvas API, especially in regard to how it can be used alongside D3.

8.3.1 Drawing with canvas

The first thing we're going to draw with canvas isn't going to use much of any D3 code at all. What little it does use, such as d3.select() and .node(), could easily be replaced with native JavaScript. It's not until later, when we start drawing many different images, and pass those images on to other elements, that we will see the kind of D3 functionality we've grown used to.

Listing 8.x Canvas Drawing Code

```
  d3.select("#traditional")
.append("canvas")
.attr("height", 500)
.attr("width", 500);

  var context = d3.select("canvas").node().getContext("2d");

  context.strokeStyle = "black";
  context.lineWidth = 2;
  context.fillStyle = "red";
  context.beginPath();
  context.arc(250,250,200,0,2*Math.PI);
```

```
context.fill();
context.stroke();

context.textAlign = "center";
context.font="200px Georgia";
context.fillStyle = "black";
context.textAligh = "center";
context.fillText("1",250,250);
```

The result is the circle in Figure 8.x. You'll notice a few important differences from the code we've been using earlier. First, we hardly use D3 in this example, and could easily have skipped it entirely by using the built-in selectors in core JavaScript. Second, we draw with canvas not on a <svg> element but on a <canvas> element that needs to be created in your DOM. Third, canvas has a very distinct syntax from SVG.

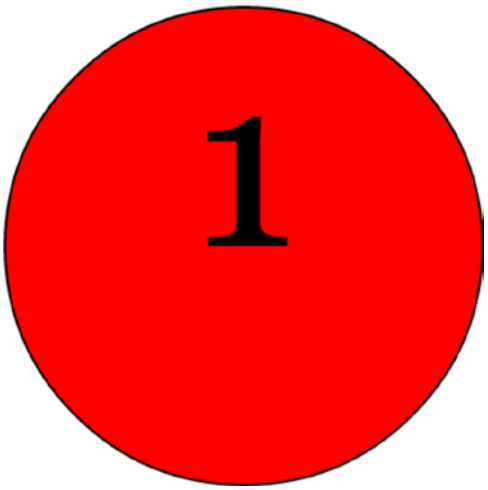


Figure 8.x A circle and text drawn using HTML5 canvas.

But there's one more major difference between the graphics created using canvas and the graphics created using SVG. You can see it if you inspect that circle, as shown in Figure 8.x. Anything drawn in canvas is drawn to a bitmap, so there is no individual text or circle element that you could assign an event listener to or later modify the appearance or text content of. That also means that it's not vector-based, so if you try to zoom the image you'll see the pixilation you're familiar with from zooming photos and other raster imagery. Because HTML5 canvas doesn't create separate DOM elements, it benefits from higher performance when dealing with large amounts of those graphical elements. But you lose the flexibility of SVG.

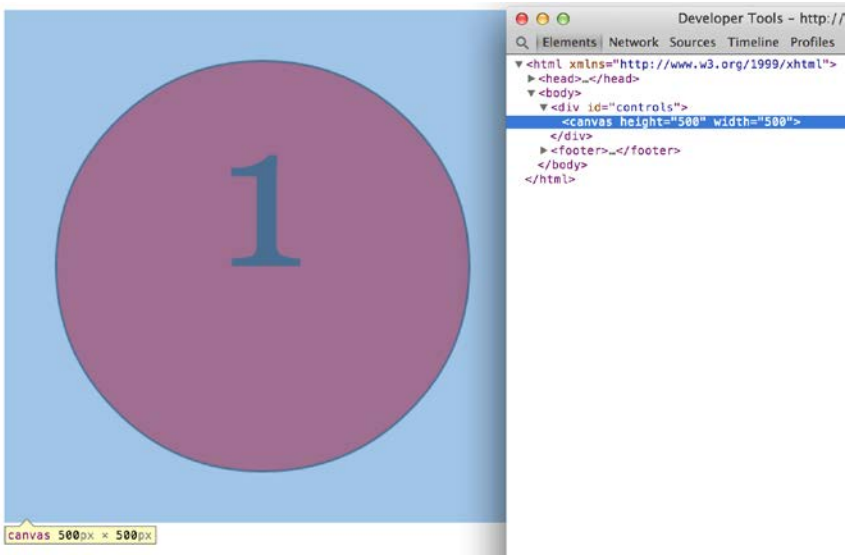


Figure 8.x Any graphics created in canvas are stored as a bitmap or raster image. Unlike in SVG, the individual shapes are no longer accessible or modifiable after being drawn.

8.3.2 Drawing and storing many images

We want images because our plan is to build an image gallery, but the canvas element in your DOM doesn't act like the kind of image that we're accustomed to dealing with in web development. You can't right-click and save it or open it in a new window in its current form, but the <canvas> element includes a .toDataURL() function that will provide you with a string designed to be the src attribute of an <img> element. You can see in Listing 8.x what the results of .toDataURL() looks like when applied to one of our drawn circles. This is only the first three lines—the actual value would go on like this for nine pages.

Listing 8.x Sample toDataURL() output

```

4Xu2dC3xV1ZX/171B1JJggNoSsSY+QrWiQnB4dCoEH7Tgg4dCoEH7Tgg4dVdNRCWg3SqQVm4Qg
9sNDW0KVWluFYCl0FIdAW99AALWUE2sCtWCgQQfkdz73+smV9TIfQmc7UPkbU
9sNDW0KVWluFYCl0FIdAW99AALWWUE2sCtWCgQQfkdz73+smV1NIyH3sc89ae//258MnKOf
```

In our new example in Listing 8.x, we'll create 100 circles of varying colors with varying borders and use .toDataURL to create an array of values that can be bound to <img> elements to create our first simple gallery of a hundred images.

Listing 8.x Drawing 100 circles with canvas

```
imageArray = [];
  d3.select("#traditional").append("canvas").attr("height",
500).attr("width", 500);
```

```
var context = d3.select("canvas").node().getContext("2d");
context.textAlign = "center";
context.font="200px Georgia";
colorScale = d3.scale.quantize().domain([0,1]).range(Reds[7]); #a

lineScale = d3.scale.quantize().domain([0,1]).range([10,40]); #a
for (var x=0;x<100;x++) {
context.clearRect(0,0,500,500);
context.strokeStyle = colorScale(Math.random());
context.lineWidth = lineScale(Math.random()); #b
context.fillStyle = colorScale(Math.random());
context.beginPath();
context.arc(250,250,200,0,2*Math.PI);
context.fill();
context.stroke();

context.fillStyle = "black";
context.fillText(x,250,280);
var dataURL = d3.select("canvas").node().toDataURL(); #c
imageArray.push({x: x, url: dataURL});
}
d3.select("#traditional")
.append("div").attr("class", "gallery")
.selectAll("img").data(imageArray) #d
.enter().append("img")
.attr("src", function(d) {return d.url}) #d
.style("height", "50px") #e
.style("float", "left")
```

#a These scales are designed for random numbers to create random graphics
#b Draw a randomly colored circle 100 times
#c Get the data URL for each drawing and push it into an array
#d Use that array to create 100 images
#e <img> elements have automatic resizing so the width of the image will automatically adjust to scale the image to this height without distorting

The results, as we see in Figure 8.x, are each of our slightly different circles turned into PNGs and assigned to <img> elements. As an aside, you can also use toDataURL() to create JPEGs by specifying that format, but by default it creates PNGs. Since they're <img> elements now, they resize automatically, and so even though we only specified the height of the images, the <img> element by default proportionately scaled the width of the image so that it wouldn't distory. Because of the float:left setting on those elements, they easily fill the div we created for them. And because it's an <img>, you can do anything with these that you normally could with an image on a web page, including save it or open it in a new tab.

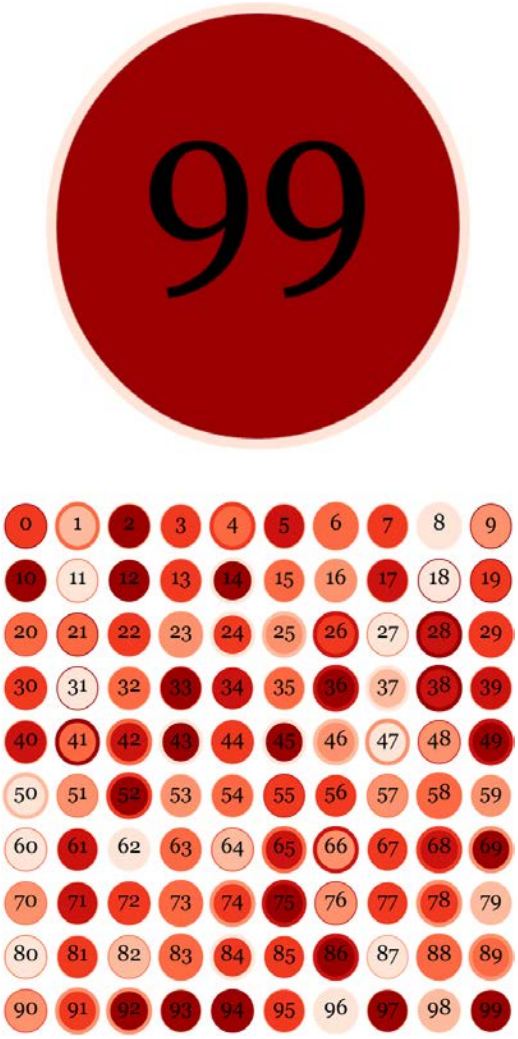


Figure 8.x The final canvas-drawn circle (top) remains in our <canvas> element and every variation according to the settings as an image in a div beneath it.

That's not much of an image gallery, though. We'll continue to expand on this code in the next section and also take advantage of the interaction and animation techniques we've already used to make something a bit more interesting.

8.4 Image Gallery

We spent that time learning canvas so that you could see how to make image elements for a gallery. When we think about spec'ing out an image gallery, there are a few basic features that

everyone wants. First, we'll want more control over where we place images, and so instead of using float, we'll do the same thing we did with the spreadsheet divs in section 8.2.2 and use "position:absolute" and use "top:" and "left:" to place them like we placed our div cells and rows or the SVG elements that we've used in previous chapters and which you're now more familiar with manipulating. We also want images to cleanly fit the space we provide, and we probably want those images to grow or shrink if the user changes the size of their window.

For all these examples, I'm going to use the same method described in Listing 8.x to create the imageArray dataset that we'll be using. Of course, the figures in this chapter will have slight variation from the results of running this code, since we're randomly generating some of the visual elements. We can create our first gallery with surprisingly little code, as seen in Listing 8.x.

Listing 8.x Resizing 8-Image Gallery

```
imgPerLine = 8; #a
d3.select("canvas").remove(); #b
d3.select("#traditional")
.append("div").attr("class", "gallery")
.selectAll("img").data(imageArray).enter().append("img")
.attr("class", "infinite")
.attr("src", function(d) {return d.data})

redrawGallery(); #c

function redrawGallery() {
var newWidth = parseFloat(d3.select("div.gallery").node().clientWidth); #d
    var imageSize = newWidth / imgPerLine;
    function imgX(x) {
      return Math.floor(x / imgPerLine) * imageSize;
    } #e
    function imgY(x) {
      return Math.floor(x%imgPerLine * imageSize)
    }
    d3.selectAll("img")
    .style("width", newWidth / imgPerLine)
    .style("top", function(d) {return imgX(d.x)})
    .style("left", function(d) {return imgY(d.x)})
}

window.onresize = function(event) {
   redrawGallery(); #f
}
```

#a This code will automatically resize to fit any number of images per row
#b Delete the canvas element because it's not needed anymore
#c Placement code in a separate function for ease of use with dynamic updates
#d Size based on the parent div width
#e X and Y based on custom accessor functions
#f Resize the gallery whenever the page is resized

This results, as we see in Figure 8.x, in a simple, scrollable div with eight images per line that will not only scale to fit the div it's in, but rescale as you adjust your browser window. The

imgX and imgY functions, along with creating an object for each image that stores an x value, should remind you of D3 layout accessor functions. We'll build something more involved like this in Chapter 9 and dive into writing layouts in Chapter 10, but for this example we're not going to try to create an actual image gallery layout.

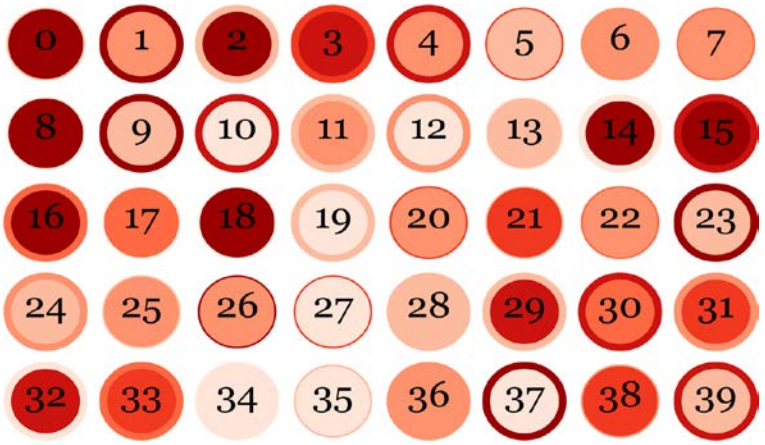


Figure 8.x Automatically scaled-to-fit images that pack 8 images per row.

8.4.1 Interactively highlighting DOM elements

From here, we can add some simple interactivity, such as making an image expand on mouseover. The code for it is in Listing 8.x, and it doesn't take much.

Listing 8.x Expand Image on Mouseover

```
    function highlightImage(d) {
        var newWidth = parseFloat(d3.select("div.gallery").node().clientWidth);
        var imageSize = newWidth / 8; #a
      d3.select(this).transition().duration(500).style("width", imageSize * 2)
      .style("background", "rgba(255,255,255,1)")
      .style("border-color", "rgba(0,0,0,1)");
      this.parentNode.appendChild(this) #b

    }

    function dehighlightImage(d) {
        var newWidth = parseFloat(d3.select("div.gallery").node().clientWidth);
        var imageSize = newWidth / 8;
      d3.select(this).transition().duration(500).style("width", imageSize)
      .style("background", "rgba(255,255,255,0)")
      .style("border-color", "rgba(0,0,0,0)") #c
    }

  d3.selectAll("img")
  .on("mouseover", highlightImage)
  .on("mouseout", dehighlightImage)
```

#a We have to recalculate the width because that value isn't accessible in this function
#b Move the image up the DOM to ensure it is drawn above the images around it
#c We don't move the image back in the DOM because it cannot overlap when its reduced in size

If you're a savvy web developer, you've probably spotted an artifact from working with SVG in the code above. It's the appendChild trick that we need to use to make SVG elements draw above each other, and because we're using relative and absolute positioned DOM elements, we don't need this, because CSS has a "z-index" that allows elements to be drawn above each other. But I wanted to keep appendChild to remind you that there are different affordances in working with traditional DOM elements that aren't present in SVG elements.

There's another reason, and that's to highlight the array position value seen in the accessor functions in D3. You might think that the array position corresponds to the array position of a datapoint in the original JavaScript array that you bound to the selection, but it doesn't. It is the array position of the DOM element in the selection. So when we start to use appendChild to shift elements up and down the DOM, we change that array value. That's why when we first created imageArray we set the x value equal to the original array position, and did not use array position to place the individual gallery images. This is why redrawGallery keeps drawing images in the right place, even after you start shifting images around in the DOM by mousing over them.

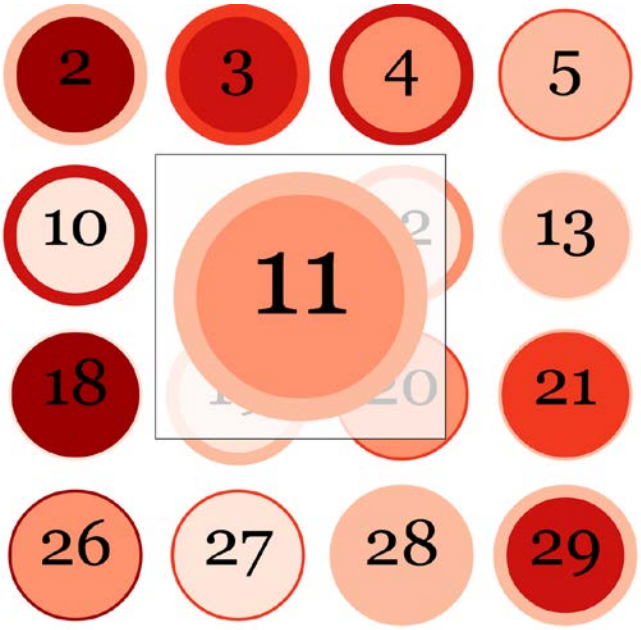


Figure 8.x One of our gallery images in mid-transition. A border and background are added for UX purposes--the transparent regions of a PNG will still register mouse events and so the user should be reminded of the actual region for mouseover events.

As you can see when you run the code in Listing 8.x and in Figure 8.x, D3's transitions are smart enough to process the rgb string designating a transparent background. In some cases, like the next example, you may have to use D3's tweening capabilities to make sure that a DOM element interpolates properly, since it probably doesn't follow the simple rules that make shape and color transitions work so easily. Still, with color and simple size transitions you can use exactly the same code for elements that you have been with <rect> and <circle> elements, unless you're trying to transition to "height:auto" or some other non-numerical value.

8.4.2 Selecting

Our final example is going to be the addition of a dropdown list that lets you select a particular image and scroll the gallery to the row that holds that image. To do so, we need to populate the <select> element with choices that correspond to our images, and write a function that scrolls the gallery to the correct line. By now, if you know how a <select> element works (it has a bunch of <option> elements nested underneath it in the DOM), you should guess how to do this with D3 using imageArray as your data. Creating the scrolling function, however, is a bit more involved because we need to write a custom tween to scroll the element that contains our gallery.

Listing 8.x Zoom to a specific image from a select input

```
function zoomTo() {
      var selectValue = d3.select("select").node().value; #a
      var newWidth = parseFloat(d3.select("div.gallery").node().clientWidth);
      var imageSize = newWidth / 8;
  var scrollTarget = Math.floor(selectValue / 8) * imageSize; #b

  d3.selectAll("img").filter(function(d) {return d.x == selectValue})
  .transition().duration(2000).style("width", imageSize * 2)
    .style("background", "rgba(255,255,255,1)")
    .style("border-color", "rgba(0,0,0,1)"); #c

  var selectedNode = d3.selectAll("img")
.filter(function(d) {return d.x == selectValue}).node();
  selectedNode.parentNode.appendChild(selectedNode); #d

    d3.select("div.gallery").transition().duration(2000)
        .tween("scrollTween", scrollTopTween(scrollTarget)); #e

function scrollTopTween(scrollTo) {
    return function() {
        var i = d3.interpolateNumber(this.scrollTop, scrollTo);
        return function(t) { this.scrollTop = i(t); }; #f
    };
}
}

d3.select("div.gallery").style("height",
"50%").style("overflow","scroll").style("border", "2px black solid")

d3.select("#traditional").append("select")
```

```
.on("change", zoomTo)
.selectAll("option").data(d3.selectAll("img").data()).enter().append("option"
).attr("value", function(d) {return d.x}).html(function(d) {return "Image #"
+d.x})
```

#a Get the selected image value
#b Calculate where that image is
#c Zoom that image in the same way as we did with mouseover earlier
#d Bring that image forward in the DOM
#e Scroll the div with a tween
#f Continuously update the div's scrollTop attribute to be a value between its current scrollTop and the calculated location of the selected image

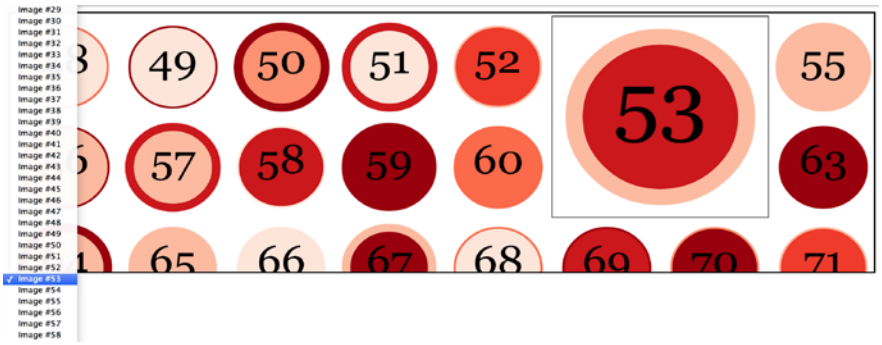


Figure 8.x Selecting an image from the list scrolls the div to the proper location and increases its size.

For a gallery that you'll want to deploy, this would need some cleaning up, but the purpose of this chapter was to demonstrate how you can use D3 functionality to work with bitmaps and divs and other traditional materials of web design. Notice that you'll need to adjust some of your workflows and syntax and also integrate CSS more, but that ultimately buttons and images and paragraphs can be data-driven and have the same kind of graphical and interactive sophistication as the geometric shapes you're going to work with more often.

I tend to use D3 for my traditional DOM elements not only because of the flexibility, but because it uses the same syntax and abstractions. As a result, it's easy to do things like create a view of your data as a bulleted list to go along with your map.

8.5 Summary

In this chapter, you saw some of the potential of using D3 to create dynamic content with traditional DOM elements. By doing this, you can embed your more traditional SVG-based data visualization in a web page that is equally dynamic, or create dynamic web pages that don't have any SVG at all. Specifically, this chapter focused on:

- Using D3 to create and manipulate traditional DOM elements like <table>, <select>, and <img>
- Creating interactive and dynamic spreadsheets and galleries

- Getting a taste of the HTML5 canvas API
- Taking a closer look at tweening and transitions

In Chapter 9 we'll see how we can tie together multiple visualizations and traditional DOM elements with custom events to create your first interactive data-driven application that will examine tweets using multiple views into the data. While we haven't dealt with combining traditional DOM elements and SVG data visualization in this chapter, we'll see that in the next chapter as we put sparklines in our spreadsheets and divs in our tree diagrams.