

BONUS CHAPTER

Elijah Meeks



D3.js

IN ACTION



D3.js in Action

by Elijah Meeks

Chapter 12

Copyright 2015 Manning Publications

brief contents

PART 1 D3.JS FUNDAMENTALS.....1

- 1 ■ An introduction to D3.js 3
- 2 ■ Information visualization data flow 46
- 3 ■ Data-driven design and interaction 77

PART 2 THE PILLARS OF INFORMATION VISUALIZATION 105

- 4 ■ Chart components 107
- 5 ■ Layouts 139
- 6 ■ Network visualization 175
- 7 ■ Geospatial information visualization 204
- 8 ■ Traditional DOM manipulation with D3 240

PART 3 ADVANCED TECHNIQUES..... 259

- 9 ■ Composing interactive applications 261
- 10 ■ Writing layouts and components 283
- 11 ■ Big data visualization 303
- 12 ■ D3 on mobile (online only)

12

D3 on mobile

This chapter covers

- Creating your own zoom, rotate, and pan touch functions using `d3.touches`
- Building a dataviz application optimized for phone, tablet, or desktop display
- Using built-in, touch-enabled D3 functionality
- Learning how to use geolocation with D3 mapping

In this chapter you'll get a taste of the challenges and opportunities that come with working with D3 in a mobile environment. Responsive data visualization is still a new and changing field, but we'll cover a few lessons to keep in mind. Mobile—and by that I mean tablets and phones—brings with it many restrictions, including screen size and affordances, especially touch interaction. Later editions of this book may deal with D3 on watches, glasses, and 3D cameras, but this edition focuses on tablets and phones.

This chapter provides a general exploration of design restrictions in practice. But it doesn't deal with responsive design for different screen sizes or OS hiccups, nor does it deal with native app development. Instead, this chapter focuses on understanding touch interfaces and using built-in D3 functions and standard HTML5

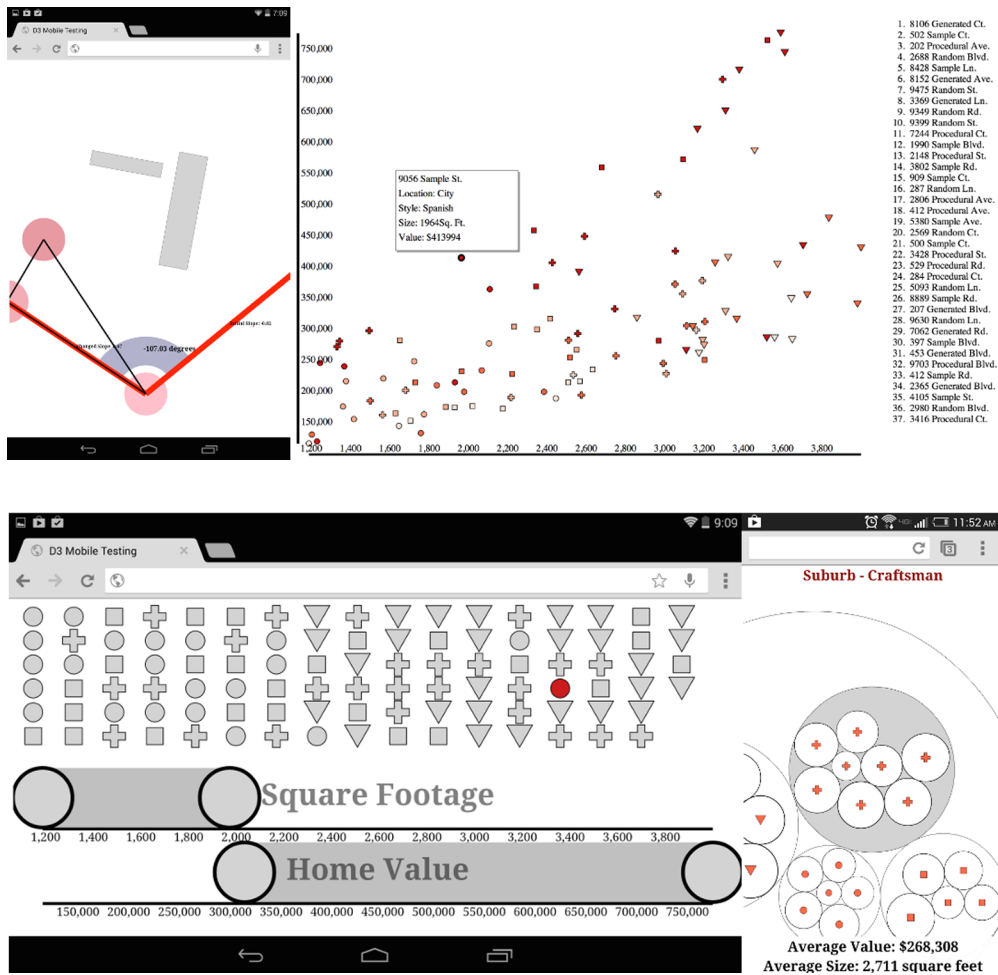


Figure 12.1 This chapter includes visualizing touch interaction (section 12.2), and creating a data visualization that's optimized first for the desktop (section 12.3.2), then for a tablet (section 12.3.3), and finally a phone (section 12.3.4).

functionality. This will help you visualize how a touch interface works, and show how touch can be tied to SVG transformation to get standard one-finger panning, pinch to zoom, and rotation. After that, we'll look at a data visualization web app that's designed to show data for desktops, tablets, and phones, screenshots of which can be seen in figure 12.1. Finally, a short demo shows how to access geolocation services to create a point on a D3 map based on a user's location.

Of the many different mobile platforms, I relied on Android phones and tablets running the most modern version of Android (4.4) as of this writing. The screenshots all come from a Nexus 7 tablet or an HTC One phone, because that's what I had access to, but the example code should work on any mobile browser.

12.1 Principles of data-driven mobile design

There aren't many examples of good mobile dataviz right now. For the most part, that's because the mobile web has been focused on traditional web content, like social media or news stories. Those kinds of media are mostly text-and-images games, and we've developed decent principles and frameworks to handle small screens and touch scrolling. In contrast, when you navigate to data visualization examples on a tablet or phone, you typically get the desktop version.

The one category where this isn't the case is geospatial information visualization. The best mapping apps and websites are styled and presented differently on a tablet or a phone than on the equivalent website. As you saw in chapter 7, mapping applications can use one of the oldest forms of information visualization with established patterns to create the most sophisticated applications. This is also the case with mobile, where mobile maps are common because maps are useful when you're in motion.

Transitioning a typical data dashboard from the browser to a limited screen size with a different interface isn't an easy or well-documented task. Two things are going on here. One is that data visualization, like web UIs, needs to be responsive—it needs to react to changes in the way the data visualization is being served to the user. Second, most of the mobile experience that you'll present to a user relies on a fundamentally different interface than that found on the web. Rather than a mouse and keyboard, you have to use touch, which presents new challenges and opportunities that are poorly served by treating touch like a mouse. Before you get into larger design and implementation issues, it makes sense to use D3 to understand touch interaction.

Emulation

You don't need to access the code here on an actual mobile device with a touch interface. You can use emulation, like that provided by Google Chrome Developer Tools in its Device Mode. This allows you to simulate various screen sizes as well as device position, geolocation, and touch input. You can read more about it here: <https://developer.chrome.com/devtools/docs/device-mode>.

12.2 Visualizing touch

Suppose you wanted to calculate a pinching or rotating motion so users can shrink an object or zoom, rotate, or otherwise interact with your data visualization in a way that isn't the default behavior on a touch interface. To create that interactivity, you need first to understand how touch works. This section focuses on touch and how to use it to create that interactivity.

Touch interaction can be confusing if you've only had experience working with mouse events. To help demystify touch, this section visualizes how touch works and what you can do with it. The first thing you want to do is see how a touch interface works at its most basic level. This follows the same advice I gave when we worked with

the Sankey layout in chapter 5: data visualization isn't just for datasets, it's also for understanding interaction and functionality. Before we get started, let's look at the HTML that we'll use for this chapter.

Listing 12.1 Basic HTML structure for chapter 12

```
<!DOCTYPE html>
<html>
<head>
  <title>D3 Mobile Testing</title>
  <meta charset="utf-8">
  <link type="text/css" rel="stylesheet" href="d3touch.css" />
</head>
<script src="http://d3js.org/d3.v3.min.js"></script>
<script src="http://d3js.org/colorbrewer.v1.min.js"></script>
<script src="d3touch.js"></script>
<body>
  <div id="vizcontainer">
    <svg></svg>
  </div>
</body>
</html>
```

We'll put all our
chapter 12 code in
this external file.

This doesn't look much different from earlier chapters, but when you look at the CSS for this chapter in the following listing, you'll see a difference.

Listing 12.2 CSS for chapter 12

```
body, html {
  width:100%;
  height:100%;
}
#vizcontainer {
  width:100%;
  height:100%;
}
  #touchStatus {
    position: absolute;
    top: 0px;
    left: 0px;
  }
svg {
  width: 100%;
  height: 100%;
}
li {
  font-size: 30px;
  font-weight: 900;
}
```

Whereas in the past we made most of our demos in a 500 x 500 pixel `<svg>` element, this time we'll set the SVG to fit the screen. In this section we'll start by looking at the touch interface data, and then show that data visually. After that, we'll work with

touch-based interactivity like panning and zooming to manipulate SVG elements that we typically work with in D3. Then we'll create a rotate function that's a bit more involved, and you'll see how to put all these functions in play at the same time. We can start exploring touch events by seeing how the `d3.touches` function works.

12.2.1 `d3.touches`

To get started, we need to add event listeners to our SVG for touch events. We use the event listeners to draw SVG elements representing the touch interaction. We'll collect touch data by getting an array of all the current xy positions of the various touch events by using `d3.touches`, a function that returns an xy array with each touch event.

12.2.2 Logging touch events with a list

Let's get started by using `d3.touches` to pass the raw array to a list in the browser. We'll be able to see the xy coordinates of all the touch events as we make them in our app. The following listing shows how to make this happen. We also use `d3.event` to get the touch event itself to find out what type of touch event it is and display it in the interface.

Listing 12.3 Logging touch events to a list

```
d3.select("#vizcontainer")
  .append("div")
  .attr("id", "touchStatus")
  .append("p")
  .html("Touch Status:")
  .append("ol");

d3.select("svg").on("touchstart", touchStatus);
d3.select("svg").on("touchmove", touchStatus);

function touchStatus() {
  d3.event.preventDefault();
  d3.event.stopPropagation();
  d = d3.touches(this);
  d3.select("#touchStatus")
    .select("ol")
    .selectAll("li")
    .data(d)
    .enter()
    .append("li");

  d3.select("#touchStatus")
    .select("ol")
    .selectAll("li")
    .data(d)
    .exit()
    .remove();

  d3.select("#touchStatus")
    .select("ol")
    .selectAll("li")
    .html(function(d) {return d3.event.type + d;});
};
```

Creates a list to hold our elements

Runs touchStatus both when a touch starts and on every move of a touch

Prevents the default from interrupting browser events and prevents propagation so touch isn't treated like a click

Adds a new line for every touch in the array

Removes a line if a touch leaves

Updates the text of the line to be the xy coordinates of the touch and the event type

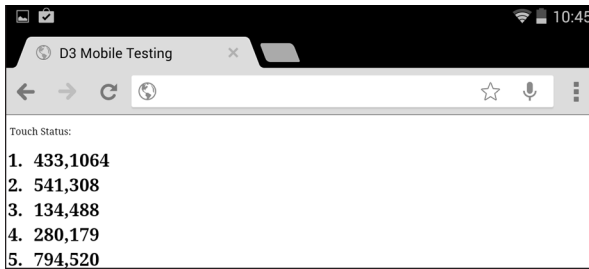


Figure 12.2 The raw output of `d3.touches` is an array listing the xy coordinates of each active touch. Here we use that array to populate an HTML list with those coordinates. 📱

With that in place, you can put your fingers on your screen and see the xy coordinates of those fingers, as shown in figure 12.2. Experimentation will show you that touch events happen whenever you move, add, or remove a fingertip from the surface of a touchscreen. They happen anytime a finger touches or stops touching the screen, and not just when you first touch the screen. Touch events track the position of those fingertips in the order in which they touch the screen.

Now that we have the data stored for each touch, we can represent it in a more graphically rich way.

12.2.3 Visualizing touch events

Instead of producing a list of xy coordinates, let's put a circle onto the canvas each time the user touches the screen at the location of that touch. By seeing where touch events are in relation to each other, you can better understand how you might build more complex functions based on gestures. By now, you should realize that adding those circles is practically the same as adding the list elements in listing 12.3. In the following listing, you can see how easy it is to visualize touch.

Listing 12.4 Circles representing the position of each touch

```
d3.select("svg")
  .on("touchstart", touchStatus)
  .on("touchmove", touchStatus);

function touchStatus() {
  d3.event.preventDefault();
  d3.event.stopPropagation();
  d = d3.touches(this);
  visualizeTouches(d);
};

function visualizeTouches(d) {
  var touchColor = d3.scale.linear()
    .domain([0,10]).range(["pink", "darkred"]);

  d3.select("svg").selectAll("circle")
    .data(d)
    .enter()
    .append("circle")
    .attr("r", 75)
    .style("fill", function(d, i) {return touchColor(i);});
```

We'll have a color scale to distinguish the first fingertip from the last.

Creates a circle for each touch event

```

d3.select("svg").selectAll("circle")
  .data(d)
  .exit()
  .remove();

d3.select("svg").selectAll("circle")
  .attr("cx", function(d) {return d[0];})
  .attr("cy", function(d) {return d[1];});
};

```

Places each circle
in a position
corresponding to
its touch event

Notice how large a circle has to be to correspond to a fingertip, and compare that to the size of typical symbols you might mouse over or click. This code creates a circle for each fingertip, as shown in figure 12.3. That's an important difference between touch- and mouse-based interactivity: the size of an interactive element in a touch interface is much larger than one you access with a mouse. The minimum size of a touch-accessible element isn't specifically defined, but the iPhone Human Interface Guidelines recommend 44ppi (<https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/MobileHIG/LayoutandAppearance.html>) and Android recommends 38dpi (<http://developer.android.com/design/style/metrics-grids.html>).

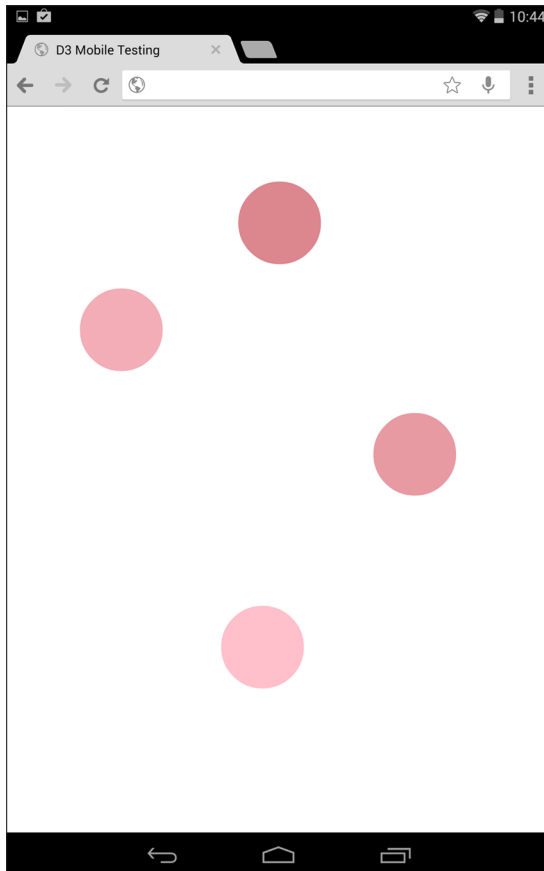


Figure 12.3 A pink-to-dark-red circle is placed at a position corresponding to each fingertip on a touchscreen. 🌈

12.2.4 Single-touch panning

Now that we can see where a touch is on the page, we can easily use that position, and the change in its position, to enact single-finger panning of our data visualization. In listing 12.5 you can see how to accomplish this. We need to add two new variables: `initialPosition` and `initialD`. `initialPosition` finds the current translate value of the transform attribute of the `<g>` element that contains all our graphical elements, and `initialD` stores all the touch points after the first touch event, so we can calculate how far away the current touch is. We do all this only when the screen detects a single touch. To give you a sense of what we're doing, we'll also draw a couple of rectangles onscreen so you can see them move.

Listing 12.5 Single-touch panning

```
var initialD;
var initialPosition;

d3.select("svg").on("touchstart", touchBegin);
d3.select("svg").on("touchmove", touchStatus);

var graphicsG = d3.select("svg").append("g").attr("id", "graphics");
graphicsG.append("rect")
  .attr("width", 250)
  .attr("height", 50)
  .attr("x", 50)
  .attr("y", 50)
  .style("fill", "red")
  .style("stroke", "gray")
  .style("stroke-width", "1px");
graphicsG.append("rect")
  .attr("width", 100)
  .attr("height", 400)
  .attr("x", 350)
  .attr("cy", 400)
  .style("fill", "gray")
  .style("stroke", "black")
  .style("stroke-width", "1px");
```

← We need variables to store the initial `<g>` position and initial touch values.

← The group container that we'll move

← A couple of rectangles, so we can see if we're moving the `<g>`

With all that in place, we modify the two touch events, one keyed to when touch starts and one keyed to whenever touch moves or otherwise changes. We calculate the change in the xy position of the first touch event, and apply that change to the `<g>` that holds the graphics, as shown in the following listing. This activates only when a single touch is active, so our panning takes place only when one finger is onscreen.

Listing 12.6 `touchBegin` and `touchStatus` functions for panning

```
function touchBegin() {
  d3.event.preventDefault();
  d3.event.stopPropagation();

  d = d3.touches(this);
  if (d.length == 1) {
```

← We only want to pan when one finger is on the screen.

```

    initialD = d;
    initialPosition = d3.transform(d3.select("#graphics")
                                  .attr("transform")).translate;
  }
};
function touchStatus() {
  d3.event.preventDefault();
  d3.event.stopPropagation();
  d = d3.touches(this);
  visualizeTouches(d);

  if (d.length == 1) {
    var newX = initialD[0][0] - d[0][0] - initialPosition[0];
    var newY = initialD[0][1] - d[0][1] - initialPosition[1];
    d3.select("#graphics")
      .attr("transform", "translate(" + (-newX) + "," + (-newY) + ")");
  }
};

```

Applies the new position to the <g>

Still visualizing touch with circles

Uses d3.transform to get the current transform values of the <g>

Calculates the change in touch position relative to the original position of the <g>

When you put one finger on the screen and move it around, the code produces something like that shown in figure 12.4.

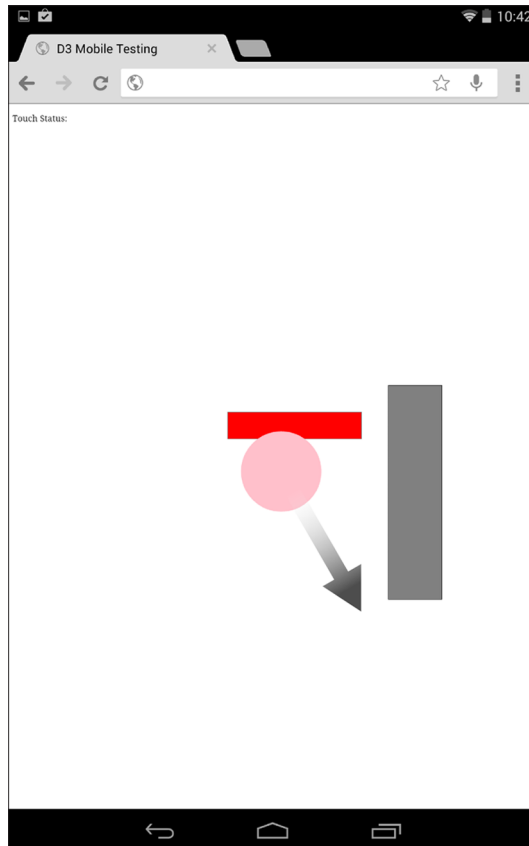


Figure 12.4 Using CSS `translate` transformation, we can move a `<g>` element (which is invisible but contains the red and gray rectangles) in relation to the position of a touch event to implement single-touch panning. The arrow is only to show the direction of the panning in this figure, and won't appear on your screen. 🎨

12.2.5 Visualizing touch extrapolation

In order to implement complex touch interactions, you need to understand how touch events are mapped in relation to each other to discover more complex gestures like pinching. As with a mouse, the position of these touch events can be keyed to interactivity. But unlike a mouse, the interrelation between the various touch events produces more complex touch functionality. For your users to use touch to zoom or rotate, you'll need to add code to know where each finger is in relation to the others.

Complex touch events are the result of extrapolating multiple positions of touches to derive an abstract function. We'll see how each touch event is connected to the others by drawing SVG lines between all the touches. Later, we'll remove this code, because it clutters the screen while we're working on more complicated functionality, but it's important to see it to understand touch events better. The following listing shows the code, which loses the complexity necessary for panning.

Listing 12.7 Touch extrapolation base code

```
d3.select("svg").on("touchstart", touchStatus);
d3.select("svg").on("touchmove", touchStatus);

function touchStatus() {
  d3.event.preventDefault();
  d3.event.stopPropagation();
  d = d3.touches(this);

  visualizeTouches(d);

  var lines = [];
  if (d.length > 1) {
    for (x in d) {
      for (y in d) {
        if (y != x) {
          var lineObj = {
            source: d[x],
            target: d[y]
          };
          lines.push(lineObj);
        }
      }
    }
  }

  d3.select("svg").selectAll("line")
    .data(lines)
    .enter().append("line")
    .style("stroke", "black").style("stroke-width", "3px");

  d3.select("svg").selectAll("line")
    .attr("x1", function(d) {
      return d.source[0];
    }).attr("y1", function(d) {
      return d.source[1];
    }).attr("x2", function(d) {
```

← We'll populate this array with our line data.

← For every touch event, creates a datapoint that links it and every other touch event.

← Creates a line for each datapoint

← Draws that line from the source touch to the target touch

```

    return d.target[0];
  }).attr("y2", function(d) {
    return d.target[1];
  });
};

d3.select("svg").selectAll("line")
  .data(lines).exit().remove();
};

```

← Removes any lines
without a corresponding
touch event

You can see how each touch exists in a spatial relationship with all other touches. Figure 12.5 visualizes this spiderweb of relations between touches. This may sound abstract, but you need to understand that spatial relationship if you want to calculate a pinching motion, or a rotating motion, which we'll do in the following sections.

Every pair of touch events now has a measurable line between them, and we can tie the length of that line to the scale of our elements to implement pinch zooming.

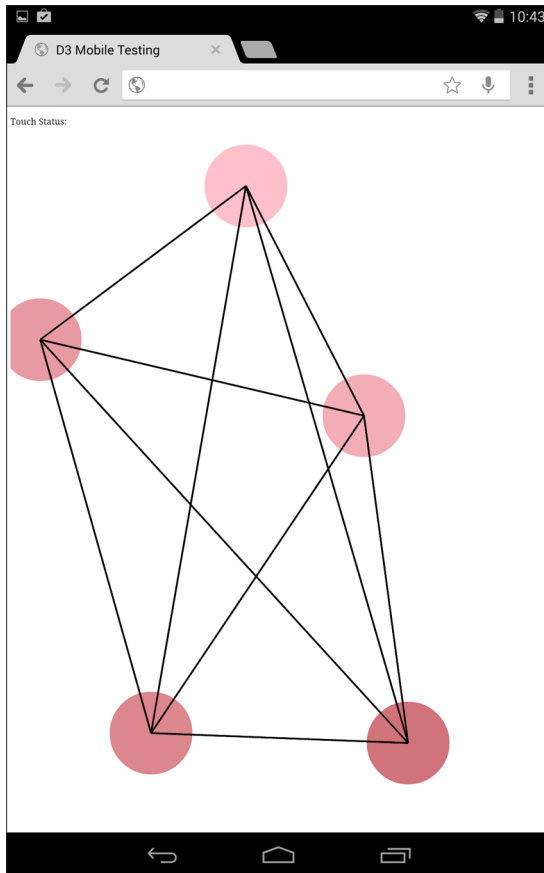


Figure 12.5 Drawing lines from every touch to all the other touches allows us to visualize the possible relations between touch that we may want to use as complex touch events. 🎨

12.2.6 Pinch zooming

The pinch gesture associated with zoom has become a touch interface standard. You want pinch zoom to work when you have exactly two touch elements. If you have only one, then you want to pan. For more than two, then another kind of touch interaction might be taking place, like rotating as in section 12.2.7. Earlier, we set panning or rotating to a single-finger event. Later we'll tie rotation to a three-finger event. In the following listing we calculate the change in the pinch motion and check to see if the number of touch events equals two.

Listing 12.8 Simple pinch zooming

```
var initialLength = 0;
var initialScale = 1;
d3.select("svg").on("touchstart", touchBegin);
d3.select("svg").on("touchmove", touchStatus);

var graphicsG =
    d3.select("svg").append("g").attr("id", "graphics");

graphicsG.append("rect").attr("width", 250)
    .attr("height", 50).attr("x", 50).attr("y", 50)
    .style("fill", "red").style("stroke", "gray")
    .style("stroke-width", "1px");

graphicsG.append("rect").attr("width", 100)
    .attr("height", 400).attr("x", 350).attr("cy", 400)
    .style("fill", "gray").style("stroke", "black")
    .style("stroke-width", "1px");
```

We need to store both the initial scale of the `<g>` we'll zoom and the initial length between the first two touches.

The rest is the same as before.

The only thing that's changed in that initial piece of code is the creation of the `initialLength` and `initialScale` variables. They store the distance that the pinch starts at, and the current zoom factor of the `<g>` element that we'll modify with that pinch. This modification is accomplished in the two touch events in listing 12.9. We use a little math in this listing, but it's only the Pythagorean theorem, which helps us calculate the sides of a right triangle and is perfect for calculating the length between two pinching fingertips.

Listing 12.9 Touch functions for pinch zooming

```
function touchBegin() {
    d3.event.preventDefault();
    d3.event.stopPropagation();

    d = d3.touches(this);
    if (d.length == 2) {
        initialLength =
            Math.sqrt(Math.abs(d[0][0] - d[1][0])
                + Math.abs(d[0][1] - d[1][1]));
        initialScale = d3.transform(d3.select("#graphics")
            .attr("transform")).scale[0];
    }
```

Make sure zoom only happens on a two-finger touch.

This is the Pythagorean theorem.

Stores the current scale of the `<g>`

```

    }
  };

  function touchStatus() {
    d3.event.preventDefault();
    d3.event.stopPropagation();
    d = d3.touches(this);

    visualizeTouches(d);

    if (d.length == 2) {
      var currentLength = Math.sqrt(Math.abs(d[0][0] - d[1][0])  

        + Math.abs(d[0][1] - d[1][1]));
      var zoom = currentLength / initialLength;
      var newScale = zoom * initialScale;
      d3.select("#graphics").attr("transform", "scale(" + newScale + ")")
    }
  };

```

Pythagorean theorem, again →

Makes zoom a direct ratio of initial compared to new pinch size ←

New scale is initial scale times zoom. →

Applies new scale ←

When you pinch your screen, the <g> element and the rectangles inside it grow and shrink. Figure 12.6 shows the shapes increasing in size as you expand your pinch.

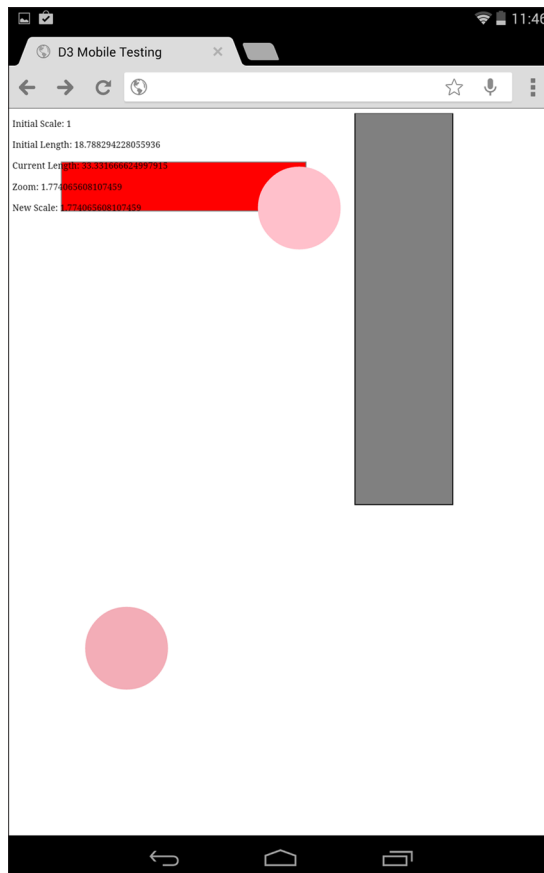


Figure 12.6 By tying the distance between two touch events to the scale of a <g> element, we can implement pinch zooming by using CSS scale transformation. 🎨

Now we can move on to a more involved complex touch event: rotation.

12.2.7 Three-finger rotating

When you move beyond two-finger touch events, you get into uncharted territory. Complex touch gestures aren't formalized. What a three-finger event is, and how you deal with it, are up to you. For this example we'll assume that whenever three fingers are on the screen, the user is trying to rotate the screen. To be clear, we don't use all three fingers to calculate rotation, but instead we use the *presence* of three fingers to distinguish between zooming and rotation. We could easily use two-finger events to do rotation and zooming. In listing 12.10 you see the new touch events, which assume we created two new variables in our base code: `initialD` to store the touch events as an interaction begins and `initialRotate` to store the current rotation of the `<g>` element we intend to modify.

Listing 12.10 Touch events for rotation

```
function touchBegin() {
  d3.event.preventDefault();
  d3.event.stopPropagation();
  initialRotate = d3.transform(d3.select("#graphics")
                              .attr("transform")).rotate;

  d = d3.touches(this);
  if (d.length == 3) {
    initialD = d;
  }
};

function touchStatus() {
  d3.event.preventDefault();
  d3.event.stopPropagation();
  d = d3.touches(this);

  visualizeTouches(d);

  if (d.length == 3) {
    var slope1 = (initialD[0][1] - initialD[1][1]) /
                  (initialD[0][0] - initialD[1][0]);

    var slope2 = (d[0][1] - d[1][1]) / (d[0][0] - d[1][0]);

    var angle = Math.atan((slope1 - slope2) / (1 + slope1*slope2))
                  * 180/Math.PI;

    var newRotate = initialRotate - angle;

    d3.select("#graphics")
      .attr("transform", "rotate(" + newRotate + ")");
  }
};
```

Stores all the touch positions from the initial touch positioning

Rotates only when three fingers are on the screen

Calculates the slopes of the lines

Calculates the angle of change between the two slopes

Applies new rotate

Determines new rotate value

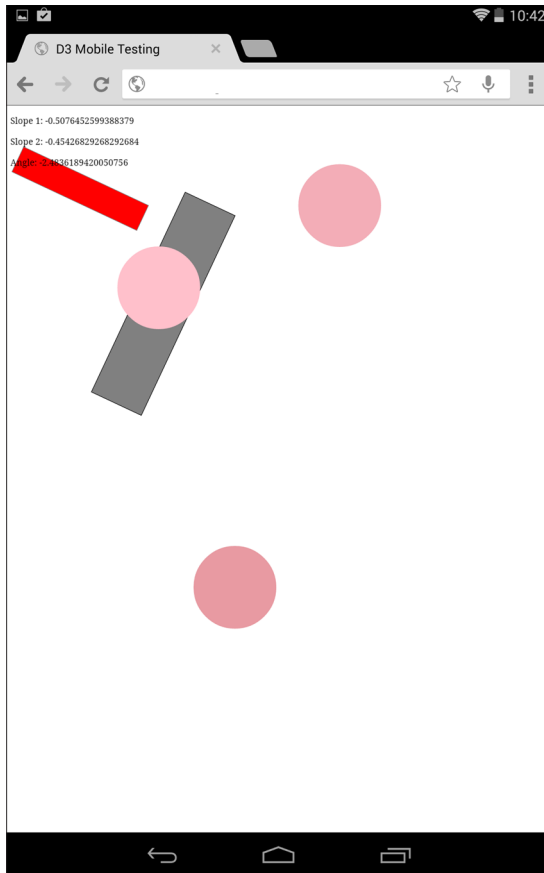


Figure 12.7 Calculating the rotation of a three-finger touch gesture by comparing the slopes of two of the lines of the touch gesture allows us to determine an angle suitable for a CSS rotate transformation. 🌈

This rotates the graphics smoothly with an effect similar to that shown in figure 12.7. You can see in our code that we don't use the third touch other than to distinguish this interaction from panning or zooming.

Figure 12.8 shows a screenshot of a more involved example, which you can find at <http://bl.ocks.org/emeeeks/840a3aff6e48196de718a>. This example uses `d3.svg.arc` and labels for the slope values to better visualize touch rotate.

Now that we've built each complex touch function separately, we can streamline the code to provide all three touch functions simultaneously.

12.2.8 Tying it all together

Up until this point, we've hardwired many of our changes into the functions. In the following listing we genericize the touchstart and touch events to deal with all three of our defined complex touch actions: panning, zooming, and rotating.

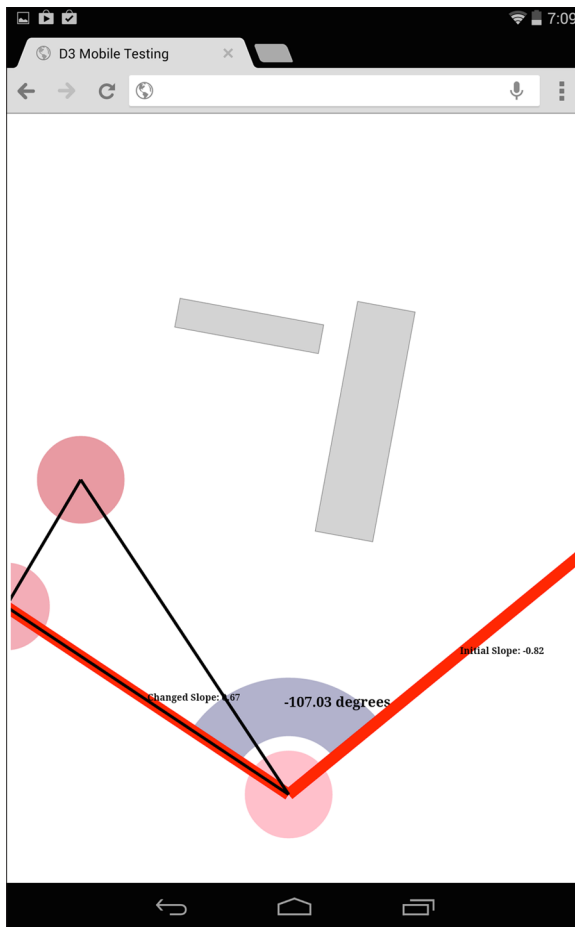


Figure 12.8 Screenshot from the Visualize Touch Rotate example at <http://bl.ocks.org/emeeeks/840a3af6e48196de718a>.

Listing 12.11 Integrated interactivity

```
var initialD = [];
var initialTransform;
```

We need to store all the transform data and all the initial touch positions because we're changing so much.

```
d3.select("svg").on("touchstart", touchBegin);
d3.select("svg").on("touchend", touchBegin);
d3.select("svg").on("touchmove", touchUpdate);

var graphicsG = d3.select("svg").append("g").attr("id", "graphics");

var sampleData = d3.range(10).map(function(d) {
  var datapoint = {};
  datapoint.id = "Sample " + d;
  datapoint.x = Math.random() * 500;
  datapoint.y = Math.random() * 500;
  return datapoint;
});
```

Ten randomly placed datapoints

```

var samples = graphicsG.selectAll("g")
    .data(sampleData)
    .enter()
    .append("g")
    .attr("transform", function(d) {return "translate("+d.x+","+d.y+")"});

var sampleSubG = samples.append("g").attr("class", "sample");

sampleSubG.append("rect")
    .attr("width", 100).attr("height", 100)
    .style("fill", "red").style("stroke", "gray")
    .style("stroke-width", "1px");

sampleSubG.append("text").text(function (d) {return d.id}).attr("y", 20);

```

Each is represented with a square and a label.

In the following listing we add a randomly placed set of 10 labeled squares to see the effect of rotate on graphical objects while keeping their labels readable.

Listing 12.12 Touch functions for integrating all complex touch actions

```

function touchBegin() {
    d3.event.preventDefault();
    d3.event.stopPropagation();

    d = d3.touches(this);
    initialD = d;
    initialTransform = d3.transform(d3.select("#graphics")
        .attr("transform"));
};

function touchUpdate() {
    d3.event.preventDefault();
    d3.event.stopPropagation();
    d = d3.touches(this);

    visualizeTouches(d);

    var newX = initialTransform.translate[0];
    var newY = initialTransform.translate[1];
    var newRotate = initialTransform.rotate;
    var newScale = initialTransform.scale[0];

    if (d.length == 1) {
        newX = -(initialD[0][0] - d[0][0] - initialTransform.translate[0]);
        newY = -(initialD[0][1] - d[0][1] - initialTransform.translate[1]);
    }
    else if (d.length == 2) {
        var initialLength = Math.sqrt(Math.abs(initialD[0][0]
            - initialD[1][0]) + Math.abs(initialD[0][1] - initialD[1][1]));

        var currentLength = Math.sqrt(Math.abs(d[0][0] - d[1][0])
            + Math.abs(d[0][1] - d[1][1]));

        var zoom = currentLength / initialLength;
        newScale = zoom * initialTransform.scale[0];
    }
    else if (d.length == 3) {
        var slope1 = (initialD[0][1] - initialD[1][1])
            / (initialD[0][0] - initialD[1][0]);
    }
}

```

Stores initial touch and position information

Gets individual transform details

Pans if one finger

Pinch-zooms if two fingers

Rotates if three fingers (we could also have this as part of the two-finger handling)

Counter-rotates labels

```

var slope2 = (d[0][1] - d[1][1]) / (d[0][0] - d[1][0]);
var angle = Math.atan((slope1 - slope2) /
    (1 + slope1*slope2)) * 180/Math.PI;
var newRotate = initialTransform.rotate - angle;
d3.selectAll("g.sample > text")
    .attr("transform", "rotate(" + (-newRotate) + ")")
}
d3.select("#graphics")
    .attr("transform", "translate(" + (newX) + "," + (newY) + ") "
        + "scale(" + newScale + ") rotate(" + newRotate + ")")
};

```

Applies new transform based on interaction

You can see that the labels could become hard to read when the screen rotates. Here we implemented an easy way to keep the labels readable by counter-rotating the labels. That's why figure 12.9 shows zoomed, panned, and rotated squares, but the labels on each <g> holding the <rect> elements are still easily readable.

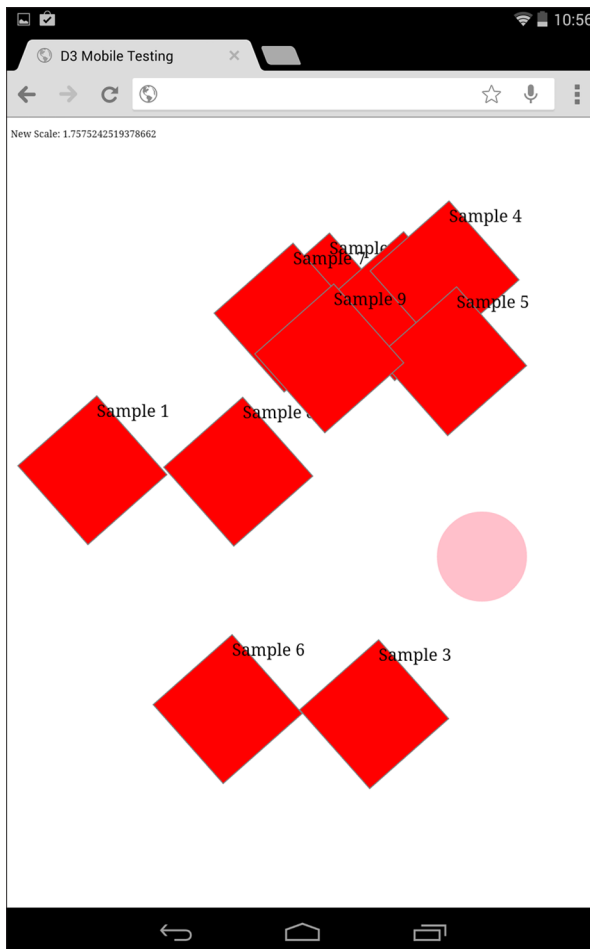


Figure 12.9 The results of panning, zooming, and rotating create larger squares with labels that are counter-rotated to maintain readability. 📱

Now that you understand how touch works, I want to zoom out and discuss how to design data visualization for touch when you have the screen size restrictions of tablets and phones, in contrast to traditional desktop websites. This next section introduces a lot of code and doesn't spend much time explaining the basics. That's because the details of the code—the particular layouts, techniques, components, or behaviors—are explained in more detail in the appropriate chapters.

12.3 Responsive data visualization

Responsive web design adjusts the experience of a website to the different affordances provided by computers, tablets, and phones. If you google “responsive data visualization,” you'll get lots of results. But the content of those results isn't about modifying data visualization but rather about organizing individual components of data visualization to fit different screen sizes. That's a straightforward concept, and figure 12.10 shows how a data dashboard like the kind we dealt with in chapter 9 might be pared down or otherwise served differently on a desktop, tablet, or phone.

Responsive data visualization isn't just a matter of making your dataviz smaller. You need to take into account several things: some data visualization layouts or components



Figure 12.10 Screen size and aspect ratio differences between devices mean that a dashboard on a desktop (top left) may need to be pared down for a tablet (top right) or turned into individual slide views for a phone (bottom).

aren't amenable to touch interaction; the data visualization layouts need to be appropriate to the screen size; and views into the data need to be at a scale appropriate to the graphical display and interaction that a user can have.

12.3.1 Creating responsive data visualization

The concept of creating websites that are properly formatted for various screen sizes and interaction methods is known as *responsive web design*. Applying the same perspective to data visualization can be thought of as *responsive data visualization*. I found little written on the subject of responsive data visualization, and what I did find focused on moving the individual components of, say, a data dashboard so that it could be presented in a long scrolling window, as illustrated in figure 12.10.

The problem with this approach is that, unlike text, different types of data visualization—and any graphical display of information, more generally—are more or less suitable for different-size canvases. Charts that are perfectly legible on a 15" screen can end up being useless on the screen of a 4" phone.

Interactivity compounds this problem. The interaction available in a website when accessed from the desktop has fundamental differences when accessed in a touch interface. Clickable icons can be much smaller than touchable icons, and well-established modes of interaction like panning or zooming sometimes have very different interaction expectations on touch compared to a mouse or keyboard.

We'll deploy a data visualization app based on a desktop dataviz app. We'll develop it into a tablet-oriented dataviz app, and then a focused app for the phone. Along the way, I'll highlight the key differences at each scale, and interesting touch-based interactive techniques you could incorporate.

12.3.2 Making a basic app

By the end of the section we'll have created a fully functional mobile real estate application with 100 houses plotted by price and square footage. Each one will show attributes about the house when you click it in the desktop version. We'll be able to access that data in different ways with different interactivity in the phone and tablet versions. Our data dashboard won't be too involved—a scatterplot alongside a list of elements in that scatterplot. Each of the 100 datapoints is a house with attributes like these:

```
Address: We'll use this as a simple name
Price: 250,000 - 1,000,000
Style: Spanish, Craftsman, Ranch, McMansion
Location: Suburb, City, Coast, Country
Size: 1100 - 4000
```

This is a good example, because it deals with quantitative, categorical, and nominal data simultaneously. The dataset consists of a 100 entries like this in JSON format, and you can find it at <http://emeeks.github.io/d3mobile/realestate.json>. Likewise, you

can experience a fully functional version of this app at <http://emeeks.github.io/d3mobile/>. If you take a look at this page on a phone and compare it to what you see on a desktop or a tablet, you'll see different views into the data.

To make this work, we need to add to our existing CSS some reference to the new elements we'll create.

Listing 12.13 CSS additions for desktop app

```
#popup {
  position: absolute;
  height: 130px;
  width: 200px;
  background: white;
  box-shadow: 2px 2px 0px #888888;
  border: 1px #888888 solid;
  left: -300px;
  top: -300px;
}
#popup > p {
  margin: 2px;
  padding: 2px;
}
div.list {
  float: left;
  height: 90%;
  width: 18%;
  overflow: auto;
}
```

← Keeps the pop-up
offscreen until the
first mouseover event

Like earlier examples, this one is missing pieces that would make it a successful data visualization. One major missing piece is a legend, to explain what the symbols and colors mean. Also, the user has no good explanation of how to interact with the data visualization, so you'll have to settle for the explanation that I give here. For this to be a finished product, it would need a legend, more context, and tutorial elements.

In listings 12.13 and 12.14 you see the code to get started with a data visualization that's oriented toward desktop web browsers.

Listing 12.14 The desktop app

```
d3.json("realestate.json", function(data) {realEstate(data)});

function realEstate(data) {
  var svg = d3.select("svg");
  var svgNode = d3.select("svg").node();

  svg.style("width", "80%").style("float", "left");
  d3.select("#vizcontainer").append("div").attr("id", "list");
  d3.select("#vizcontainer").append("div").attr("id", "popup");

  d3.select("#list").append("ol").selectAll("li")
    .data(data).enter().append("li")
    .attr("class", "datapoint")
}
```

Creates and
adjusts the
divs to share
the screen


```

    .html(function(d) {return d.name})
    .on("mouseover", highlightDatapoint);

var screenHeight = parseFloat(svgNode.clientHeight ||
    svgNode.parentNode.clientHeight);

var screenWidth = parseFloat(svgNode.clientWidth ||
    svgNode.parentNode.clientWidth);

var sizeExtent = d3.extent(data, function(d) {return d.size});
var valueExtent = d3.extent(data, function(d) {return d.value});
var xScale = d3.scale.linear()
    .domain(sizeExtent).range([40, screenWidth-40]);
var yScale = d3.scale.linear()
    .domain(valueExtent).range([screenHeight-40, 40]);

svg.append("g").attr("id", "dataG")
    .selectAll("g.datapoint")
    .data(data, function(d) {return d.name}).enter()
    .append("g").attr("class", "datapoint");

var locationScale = d3.scale.ordinal()
    .domain(["Rural", "Coastal", "Suburb", "City"])
    .range(colorbrewer.Red[4]);

var typeShape = {"Spanish": "circle", "Craftsman": "cross", "Ranch": "square",
    "McMansion": "triangle-down"};

dataG = d3.selectAll("g.datapoint")
    .attr("transform", function(d) {
        return "translate(" + xScale(d.size) + "," + yScale(d.value) + ")";
    })
    .each(function(d) {
        houseSymbol = d3.svg.symbol().type(typeShape[d.type]).size(64);
        d3.select(this).append("path")
            .attr("d", houseSymbol)
            .style("fill", locationScale(d.location))
            .style("stroke", "black")
            .style("stroke-width", "1px")
            .on("mouseover", highlightDatapoint);
    });

var xAxis = d3.svg.axis().scale(xScale).orient("top").tickSize(4);
var yAxis = d3.svg.axis().scale(yScale).orient("right").tickSize(4);

svg.append("g")
    .attr("id", "xAxisG").attr("class", "axis")
    .attr("transform", "translate(0, "+(screenHeight - 20)+")")
    .call(xAxis);

svg.append("g")
    .attr("id", "yAxisG").attr("class", "axis")
    .attr("transform", "translate(20, 0)")
    .call(yAxis);
};

```

Creates a list from our data

Calculates the size of the screen and the extent of the numerical attributes to create scales

Creates <g> elements for all datapoints

Color code by location

Symbolizes by type

Creates symbols in each <g>

Creates axes

This uses `d3.svg.symbol`, which is a new function that you haven't seen before. This function generates `<path>` drawing code corresponding to commonly used symbols

based on keywords like *circle*, *cross*, or *triangle-down*. We reference a single function, `highlightDatapoint`, shown in the following listing, which moves the pop-up, populates it with details of the datapoint, and highlights the symbol and list item that correspond to that datapoint.

Listing 12.15 Populating pop-up with information

```
function highlightDatapoint(d) {
  d3.selectAll("li.datapoint")
    .style("font-weight", function(p) {
      return p == d ? 900 : 100;
    });

  d3.selectAll("g.datapoint").select("path")
    .style("stroke-width", function(p) {
      return p == d ? "3px" : "1px";
    });

  var popup = d3.select("#popup")
    .style("top", yScale(d.value) - 135)
    .style("left", xScale(d.size) - 100);

  popup.selectAll("*").remove();
  popup.append("p").html(d.name);
  popup.append("p").html("Location: " + d.location);
  popup.append("p").html("Style: " + d.type);
  popup.append("p").html("Size: " + d.size + "Sq. Ft.");
  popup.append("p").html("Value: $" + d.value);
};
```

← Highlights the corresponding line

← Highlights the corresponding symbol

← Moves the pop-up over the corresponding symbol

← Populates the pop-up with data from this datapoint

This code produces a scatterplot using SVG based on the dataset and a list of the names of the datapoints using traditional HTML. If you mouse over a symbol or list item, then the corresponding entry is highlighted on the list, and a pop-up presents details for that datapoint above the symbol. An example of this interaction is shown in figure 12.11.

If you access the site from a desktop browser, you'll see this scatterplot and a list of the datapoints represented on the scatterplot. It includes visual and interactive features that are optimized for the desktop, such as axes and mouseover. The axes represent price on the y-axis and size of the houses on the x-axis. They're rudimentary, and in a final version the axes and the aforementioned more general elements would be improved.

This view maximizes the fine control and large screen of a desktop browser, allowing you to mouseover small elements on the scatterplot, or scroll through a list with a hundred entries. Neither of those is useful on a smaller screen that doesn't have a mouse interface.

12.3.3 Tablet scale

As you saw in section 12.2, a touch-based application requires icons of a certain size. Before we deal with this application on a tablet, let's see how it would look if we made the icons large enough to be accessible via touch. Figure 12.12 shows the results and

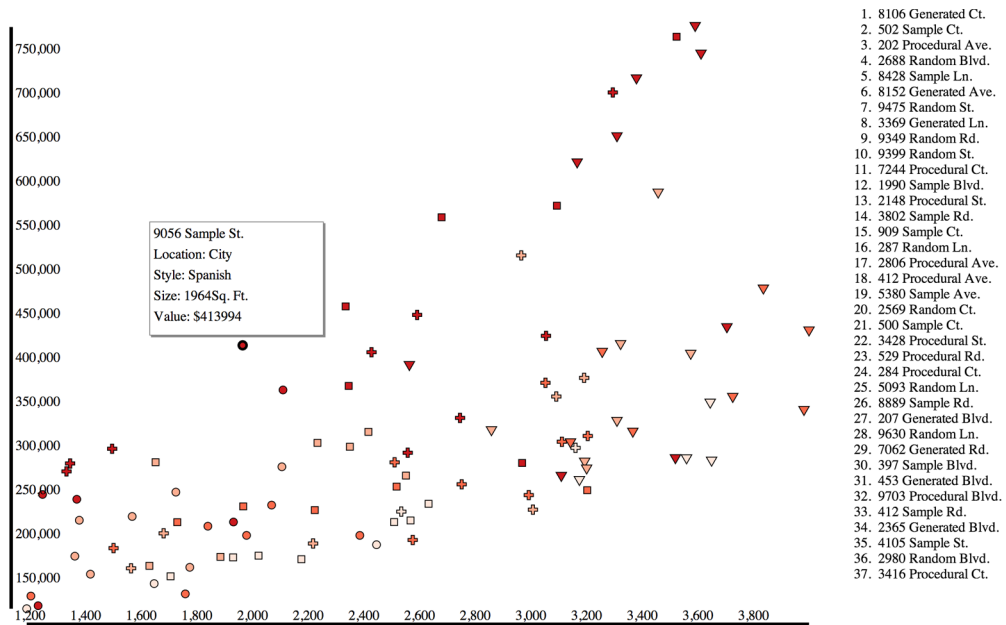


Figure 12.11 A view of the dataset designed for the desktop, using a scatterplot and a list 

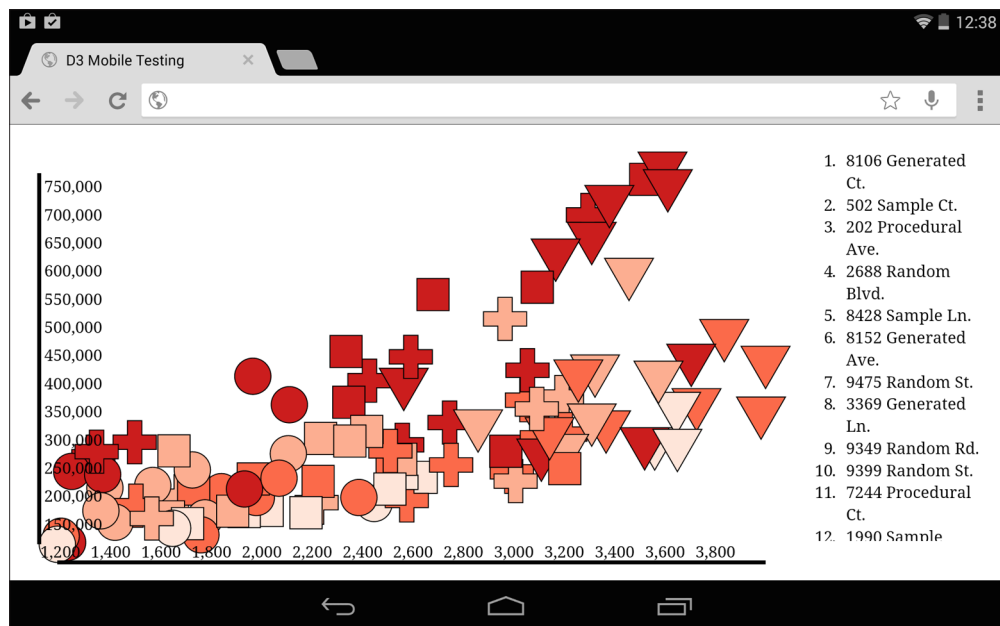


Figure 12.12 Our scatterplot with icons large enough to allow meaningful touch interaction on a tablet. The larger icons cover each other and can't easily be located along either axis.

demonstrates that this isn't a viable solution. Icons obscure each other at this scale, and interaction is iffy, at best. Using native zoom and the desktop resolution of the icons is no fix, because as soon as you zoom in to click a symbol, you lose the axes necessary to see where that symbol is.

Instead, we need a different view into the data that both enables touch interaction and allows users to explore the patterns in the data important for the initial scatter-plot view. Despite the popularity of tablets, many applications don't have a tablet-optimized version, and this is doubly true with data visualization. This partly stems from the belief (or perhaps hope) that tablets are large enough surfaces so that the same interaction can take place as on the desktop. Instead of trying to re-create the same data visualization on a smaller screen, we'll adjust the existing graphical elements and add new interactivity optimized for touch.

First, though, we need to add to our CSS the contents of the following listing.

Listing 12.16 CSS additions for the tablet app

```
rect.extent {
  opacity: .25;
}
g.resize > circle {
  fill: lightgray;
  stroke: black;
  stroke-width: 5px;
}

text.brushLabel {
  font-size: 40px;
  font-weight: 900;
  text-anchor: middle;
  opacity: .5;
  pointer-events: none;
}
```

This requires rather significant changes in the code, which take place in the `tabletView()` function. This function assumes access to all the variables in the initial `responsiveDataVisualization()` function. In the following listing you can see the `tabletView` function.

Listing 12.17 Functions to optimize our data visualization for a tablet

```
function tabletView() {
  d3.select("svg").style("width", "100%");
  d3.select("#list").remove();

  var screenWidth = parseFloat(d3.select("svg").node().clientWidth ||
    d3.select("svg").node().parentNode.clientWidth);

  var cellWidth = screenWidth / 18;
  var cellHeight = screenHeight / 14;
  sortedData = data.sort(function(a,b) {
```

Removes the list div and makes the SVG div take up the entire screen size

```

    if (a.value > b.value) {
        return 1;
    }
    if (a.value < b.value) {
        return -1;
        return 0;
    }
});

d3.selectAll("g.datapoint")
    .data(sortedData, function(d) {return d.name})
    .transition()
    .duration(1000)
    .attr("transform", function(d,i) {
        return "translate("+((Math.floor(i/6) + .5) *
        cellWidth)+", "+((i%6 + .5)*cellHeight)+") "
    });

d3.selectAll("g.datapoint").select("path")
    .on("mouseover", null)
    .each(function(d) {
        var houseSymbol = d3.svg.symbol()
            .type(typeShape[d.type]).size(512);
        d3.select(this).transition().duration(1000).attr("d", houseSymbol);
    });

xScale.range([40,screenWidth-40]);
xAxis.orient("bottom").scale(xScale);
yScale.range([40,screenWidth-40])
yAxis.orient("bottom").scale(yScale);

var sizeBrush = d3.svg.brush()
    .x(xScale)
    .extent(sizeExtent)
    .on("brush", brushed);

var valueBrush = d3.svg.brush()
    .x(yScale)
    .extent(valueExtent)
    .on("brush", brushed);

d3.select("#xAxisG")
    .transition()
    .duration(1000)
    .attr("transform", "translate(0,"+(screenHeight - 150)+")")
    .call(xAxis);

d3.select("#yAxisG")
    .transition()
    .duration(1000)
    .attr("transform", "translate(0,"+(screenHeight - 50)+")")
    .call(yAxis);

d3.select("#xAxisG").append("g")
    .attr("class", "brushG")
    .attr("transform", "translate(0,-80)")
    .call(sizeBrush)

```

Sorts the data into a grid

Increases the symbol size

Changes scales so both are associated with screen width

Creates brushes on the numeric attributes

Assigns axes to the new brushes

```

    .insert("text", "rect")
    .attr("class", "brushLabel")
    .attr("y", 50)
    .attr("x", screenWidth / 2)
    .text("Square Footage");

d3.select("#yAxisG").append("g")
  .attr("class", "brushG")
  .attr("transform", "translate(0,-80)")
  .call(valueBrush).insert("text", "rect")
  .attr("class", "brushLabel")
  .attr("y", 50)
  .attr("x", screenWidth / 2)
  .text("Home Value");

d3.selectAll(".brushG").selectAll("rect").attr("height", 80);

d3.selectAll(".brushG").selectAll(".resize")
  .append("circle").attr("r", 40).attr("cy", 40);
};

```

Creates and labels brushes

Brush handles

The brushed function in the following listing is the only interaction function.

Listing 12.18 Tablet brush function

```

function brushed() {
  d3.selectAll("g.datapoint").each(function(d) {
    var color = locationScale(d.location);
    if (
      d.value < valueBrush.extent()[0] || d.value > valueBrush.extent()[1] ||
      d.size < sizeBrush.extent()[0] || d.size > sizeBrush.extent()[1]
    ) {
      color = "lightgray";
    }
    d3.select(this).select("path").style("fill", color);
  })
}

```

If a datapoint isn't within the extent of both brushes, then color it gray.

We have the full application, shown in action in figure 12.13. Were we to put this into production, we'd want to integrate a method to allow the user to get details on various symbols like we did with mouseover and the pop-up in the desktop version. I'd go with an information panel that slides out with various details, but you may prefer another method. The details of how to implement that information box aren't something I can cover here.

The final tablet view has a different perspective of the data. The list is gone because it would take up too much room on a small canvas size. Instead of a scatterplot, which would be hard to read and difficult to interact with, you get a grid of all the datapoints and two brushes, allowing you to cross-brush by the same attributes that ordered the scatterplot: value and size. What's missing here is on-touch functionality for the symbols (which have increased in size to be touchable) that would bring up the same details as the pop-up in the desktop version. An interesting aspect of this

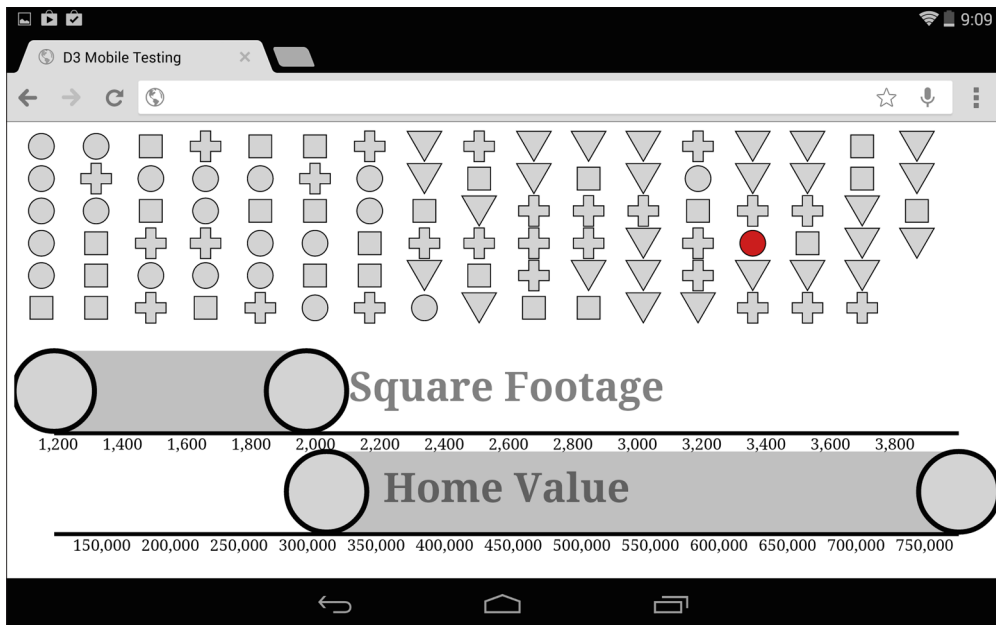


Figure 12.13 Our finished tablet-focused view on the data, which uses a pair of brushes to allow the user to find houses with a particular square footage and home value. Here we see the only house worth more than \$300,000 that's smaller than 2000 square feet. If you're interested, it's 9056 Sample St., a 1964-sq-ft Spanish-style house in the city valued at \$413,994.

particular implementation is that the cross-brushing is fundamentally a simple implementation of parallel coordinates, and could serve as a gentle introduction to that powerful information visualization method.

Notice that we never had to make any changes for the brush component to work on a tablet. That's because by default `d3.svg.brush` is touch aware. A two-finger gesture allows you to set the brush extent with the first finger fixed and the second finger setting the width, whereas a single-finger gesture moves the extent or each edge.

D3'S BUILT-IN TOUCH The `d3.svg.brush` function has touch-based functionality, and `d3.behavior.zoom` and `d3.behavior.drag` also have touch interactivity out of the box. In the case of zoom, the zoom object automatically recognizes pinch zooming and panning. For drag, you can touch an element with that behavior and drag it around without adding any new code.

Now we'll move to the most challenging frame in which to explore our data: the smartphone.

12.3.4 Phone scale

Here we'll switch approaches entirely and deploy a nested, circle pack layout. This data isn't nested, but you know from working with nesting in chapters 3 and 5 that you

can nest according to whatever attributes you want. We have two categorical attributes (location and type) that we could use, but we can also nest by the quantized price or size. First, take a look at the CSS additions we'll need for this application.

Listing 12.19 CSS additions for the phone app

```
div.viewTitle {
  font-size: 54px;
  font-weight: 900;
  color: darkred;
  position: fixed;
  top: 0;
  width: 100%;
  height: 140px;
  background: rgba(255,255,255,.95);
  text-align: center;
}

div.viewStats {
  font-size: 54px;
  font-weight: 900;
  position: fixed;
  bottom: 0;
  width: 100%;
  height: 140px;
  background: rgba(255,255,255,.95);
  text-align: center;
}

div.viewStats > div {
  width: 100%;
}
```

Holds our generated title for the current view; notice the extremely big font size

Holds the statistics of the current view

The code to produce a phone view is more involved than that for the tablet view, because we're making a more radical reformulation of our data visualization for such a significantly smaller screen size. In the following listing you see the initial code for the new view.

Listing 12.20 Transforming our dataviz for use on a phone

```
function phoneView() {
  nestPhoneData();

  screenWidth = parseFloat(d3.select("svg").node().clientWidth ||
    d3.select("svg").node().parentNode.clientWidth);
  d3.select("#list").remove();
  d3.selectAll("g.axis").remove();

  circleSize = d3.scale.linear().domain(sizeExtent).range([2,10]);
  circleStroke = d3.scale.linear().domain(valueExtent).range([1,5]);

  packChart = d3.layout.pack();
  packChart.size([screenWidth,screenHeight-200])
}
```

Removes the list like we did with tablet view

Scales for the leaf node display


```

        .children(function(d) {return d.values})
        .value(function(d) {return circleSize(d.size)});

d3.selectAll("g.datapoint").select("path")
    .style("pointer-events", "none");

d3.select("#dataG")
    .attr("transform", "translate(0,100)")
    .selectAll("circle")
    .data(packChart(packableData))
    .enter()
    .insert("circle", "g")
    .attr("class", "pack")
    .style("fill", "white")
    .style("stroke", "black")
    .style("stroke-width", function(d) {return circleStroke(d.oValue)})
    .on("touchmove", changeView)
    .on("click", changeView);

d3.select("#vizcontainer").append("div")
    .attr("class", "viewTitle").html("Current View");

var viewStats =
d3.select("#vizcontainer").append("div").attr("class", "viewStats");

viewStats.append("div").attr("id", "viewValue").html("Average Value");
viewStats.append("div").attr("id", "viewSize").html("Average Size");

changeView(packableData)
};

```

Interaction is with the circles, not the symbols

Adds label divs

Initializes at the zoomed-out level

The next listing shows the first new function we call, `nestPhoneData()`, which creates the `packableData` dataset that we need for circle packing.

Listing 12.21 Multipart nesting

```

function nestPhoneData() {
  for (x in data) {
    data[x].oValue = data[x].value;
  }

  nestedData = d3.nest()
    .key(function (d) {return d.location})
    .key(function (d) {return d.type})
    .entries(data);

  packableData =
    {id: "root", key: "All Real Estate", values: nestedData}
}

```

Nesting overwrites the value attribute, so we need to move it to a new attribute.

Creates a new set of nested data based on location and type

Assigns the nested data to a root node for circle packing

This code refers to a few new functions, detailed in listings 12.22 and 12.23, that provide the functionality necessary for the app to work. This includes zooming in on a circle when a chart is touched and summarizing the data onscreen in an easily digestible manner.

Listing 12.22 Phone app change view function

```

function changeView(d) {
  newScale = (screenHeight / 2) / (d.r + 100);

  d3.select("#dataG").selectAll("circle")
    .style("fill", function(p) {return p == d ? "lightgray" : "white"})

  d3.select("#dataG").selectAll("circle")
    .style("pointer-events", function(p) {
      return (p.depth == d.depth || p.parent == d) &&
        p != d ? "auto" : "none"
    });

  d3.select("#dataG")
    .transition()
    .duration(1000)
    .attr("transform",
      "translate(" + ((screenWidth/2)-(d.x * newScale)) + "," +
        ((screenHeight/2) - (d.y * newScale)) + ") "
    );

  d3.selectAll("circle.pack")
    .transition()
    .duration(1000)
    .attr("r", function(d) {return d.r * newScale})
    .attr("cx", function(d) {return d.x * newScale})
    .attr("cy", function(d) {return d.y * newScale});

  symbolSize = d3.scale.linear()
    .domain(sizeExtent)
    .range([100 * newScale, 180 * newScale]);

  d3.selectAll("g.datapoint")
    .transition()
    .duration(1000)
    .attr("transform", function(d) {return "translate(" +
      (d.x * newScale) + "," + (d.y * newScale) + ")";
    })
    .select("path")
    .each(function(d) {

  houseSymbol = d3.svg.symbol().type(typeShape[d.type])
    .size(symbolSize(d.size));
  d3.select(this).transition().duration(1000)
    .attr("d", houseSymbol);
});

  calculateStatistics(d);
};

```

Highlights the currently selected circle by filling it with gray

Sizes the view to the size of the current clicked circle

Activates events only for children of this circle or circles at the same level that aren't this circle

Centers on this circle

Increases circle size and adjusts position to zoom in

Zooms symbol size

Finally, the `calculateStatistics` function does a recursive search to calculate the average of all the datapoints inside the current circle, or the exact value if the currently clicked circle represents a real estate datapoint.

Listing 12.23 Phone app statistical summary

```

function calculateStatistics(d) {
  if (d.name) {
    d3.select("div.viewTitle")
      .html(d.parent.parent.key + " - " + d.parent.key + "<br>" + d.name);
    d3.select("#viewValue").html("Value: $" + d.oValue);
    d3.select("#viewSize").html("Size: " + d.size + " square feet");
  }
  else {
    var allDatapoints = allChildren(d);
    var averageValue =
      d3.mean(allDatapoints, function(d) {return d.oValue});
    var averageSize =
      d3.mean(allDatapoints, function(d) {return d.size});
    d3.select("div.viewTitle")
      .html(d.depth == 2 ? d.parent.key + " - " + d.key : d.key);
    d3.select("#viewValue")
      .html("Average Value: $" +
        d3.format("0,000")(Math.floor(averageValue)));
    d3.select("#viewSize")
      .html("Average Size: " + d3.format("0,000")(Math.floor(averageSize))
        + " square feet");
  }
};

function allChildren(d) {
  var childArray = [];
  for (x in d.values) {
    if (d.values[x].name) {
      childArray.push(d.values[x]);
    }
    else {
      childArray =
        allChildren(d.values[x]);
    }
  }
  return childArray;
}

```

Only real estate datapoints have name attributes, so we can use this to identify them.

Sets labels to be values from the datapoint

Collects all leaf nodes from this node

Averages their numerical attributes

Sets labels

Recursively finds all leaf nodes

If you navigate to our page with a phone, you'll see something like figure 12.14.

Our new phone view into the data gives a different user experience. The datapoints are nested by their location and type attributes. All the circles are touch sensitive so that you can touch a circle and zoom to that grouping. Because you need large touchable areas to have accurate touch

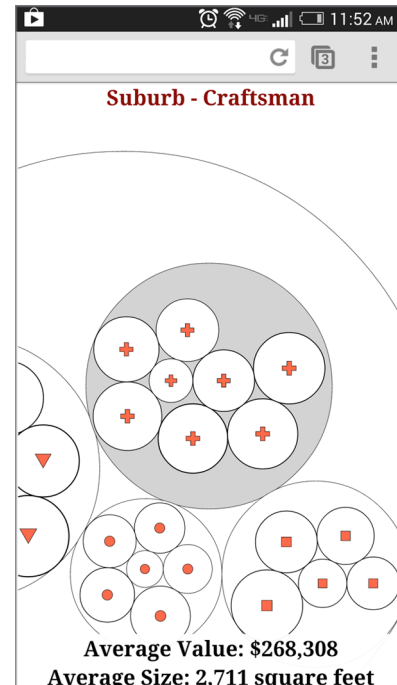


Figure 12.14 The final phone-optimized data visualization app

on a small screen, you can only touch on circles that are one level deeper or at the same depth that you're currently at. This lets you drill down in the groups, getting averages of size and value. When you've zoomed in enough, the individual datapoints are large enough to be selected and provide the attributes of that datapoint. In a full version, there would be a mechanism to let you zoom out, and a mechanism to change the nesting priorities. The former could be accomplished with the pinch gesture and the latter with the rotate gesture, both of which are now relatively intuitive to touch users.

12.3.5 Automatically recognizing different screen sizes

You can rely on screen width to detect whether a browser is on a desktop, tablet, or phone. But be aware that you may get a phone view from your tablet or a tablet view from your phone, depending on the aspect ratio and the size of the device. In the following listing is the code to let you do that in a rough way. For a production application, you'll want something more exhaustive.

Listing 12.24 Device recognition based on screen size

```
var screenSize = screen.width;
if (screenSize < 480) {
  phoneView();
}
else if (screenSize < 1000) {
  tabletView();
}
```

These examples lack much functionality, but they should provide you with a good sense of where you can go with mobile.

12.3.6 General principles of responsive data visualization

Non-desktop data visualization views are hard to design, even when you're cutting corners on a simple example like this. I think it's a great challenge to ask yourself what your data would look like on a phone or tablet, and what new affordances and limitations you need to account for to present the most valuable view into complex datasets like these. Responsive data visualization can make your data more accessible and present new views and methods for data in any form. You'll notice that each of these views has embedded in it different capabilities and different visual and interaction rhetorics. The goal isn't to provide the same view into the data, but rather allied views that let users understand a dataset from multiple points of interaction.

12.4 Geolocation

If you're creating a D3 mapping application, you can easily use HTML5 geolocation to create a point representing the user's current location on a D3 map. The following

listing takes the most basic map code from chapter 7 and puts a circle on it for your current position.

Listing 12.25 Device geolocation example

```
function geolocateMap() {
  d3.json("world.geojson", function(data) {createMap(data)});

  function createMap(countries) {
    projection = d3.geo.mercator();
    geoPath = d3.geo.path().projection(projection);
    d3.select("svg").selectAll("path")
      .data(countries.features)
      .enter()
      .append("path")
      .attr("d", geoPath)
      .style("fill", "red")
      .style("stroke-width", 1)
      .style("stroke", "black")
      .style("opacity", .5)

    if (navigator.geolocation) {
      navigator.geolocation.getCurrentPosition(placeMarker);
    }

    function placeMarker(point) {
      d3.select("svg").selectAll("circle")
        .data([point])
        .enter()
        .append("circle")
        .style("fill", "white")
        .style("stroke", "black")
        .style("stroke-width", 3)
        .attr("r", 5)
        .attr("cx", function(d) {return
          projection([d.coords.longitude,d.coords.latitude])[0]})
        .attr("cy", function(d) {return
          projection([d.coords.longitude,d.coords.latitude])[1]})
    }
  }
}
```

Makes sure the code doesn't run if the browser doesn't support geolocation

Note the formatting of the data that geolocation provides

First, you place the geolocation call inside a test to make sure that the device supports geolocation. Unless your user's browser is set to always allow geolocation, it will prompt for the location. If the user chooses to submit the location, then this code will produce results similar to those in figure 12.15, with the circle placed at the user's location.

Given the default position of the Mercator projection, if you're in Asia you won't see your circle, so take that into account. If you want to dig into mapping, make sure you first take a look at chapter 7, "Geospatial information visualization."



Figure 12.15 A white circle with a black border created in the San Francisco Bay Area, where I took this screenshot.

12.5 Summary

If you've made it this far, you've learned how touch works, how you can visualize touch, and how to build responsive data visualization. We've covered the following topics:

- How to collect touch interactions using `d3.touches`
- Creating graphics to indicate touches
- Touch panning
- Visualizing relations between touch events using SVG lines
- Pinch zoom
- Three-finger rotate
- Tablet-optimized data visualization
- Phone-optimized data visualization

Mobile data visualization is still in its infancy, with the best dataviz available on mobile still dominated by mapping apps and video games. Taking advantage of touchscreen interfaces while accounting for the smaller screen size is a challenge, but a rewarding one. The techniques and functionality from this chapter should help you get started in developing for mobile using D3.

D3.js IN ACTION

Elijah Meeks

D3.js is a JavaScript library that allows data to be represented graphically on a web page. Because it uses the broadly supported SVG standard, D3 allows you to create scalable graphs for any modern browser. You start with a structure, dataset, or algorithm and programmatically generate static, interactive, or animated images that responsively scale to any screen.

D3.js in Action introduces you to the most powerful web data visualization library available and how to use it to build interactive graphics and data-driven applications. You'll start with dozens of practical use cases that align with different types of charts, networks, and maps using D3's out-of-the-box layouts. Then, you'll explore practical techniques for content design, animation, and representation of dynamic data—including interactive graphics and live streaming data.

What's Inside

- Interacting with vector graphics
- Expressive data visualization
- Creating rich mapping applications
- Prepping your data
- Complete data-driven web apps in D3

Readers need basic HTML, CSS, and JavaScript skills. No experience with D3 or SVG is required.

Elijah Meeks is a senior data visualization engineer at Netflix. His D3.js portfolio includes work at Stanford University and with well-known companies worldwide.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/D3.jsinAction

“A mandatory introduction to a very complex and powerful library.”

—Stephen Wakely
Thomson Reuters

“Quickly gets you coding amazing visualizations.”

—Ntino Krampis, PhD
City University of New York

“A remarkable exploration of the world of dataviz possibilities with D3.”

—Arun Noronha, Directworks Inc.

“A must-have book.”

—Arif Shaikh
Sony Pictures Entertainment

“One of the most comprehensive books about data visualization I have ever read.”

—Andrea Mostosi, The Fool s.r.l.



ISBN 13: 978-1-617292-11-8
ISBN 10: 1-617292-11-7

