FREE eBook

LEARNING d3.js

Free unaffiliated eBook created from
Stack Overflow contributors.

#d3.js

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: d3-js

It is an unofficial and free d3.js ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official d3.js.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with d3.js

Remarks

D3.js is a JavaScript library for manipulating documents based on data. D3 helps you bring data to life using HTML, SVG, and CSS. D3's emphasis on web standards gives you the full capabilities of modern browsers without tying yourself to a proprietary framework, combining powerful visualization components and a data-driven approach to DOM manipulation.

The official website: https://d3js.org/ A lot of examples here: http://bl.ocks.org/mbostock

Versions

Version	Release Date
v1.0.0	2011-02-11
2.0	2011-08-23
3.0 "Baja"	2012-12-21
v4.0.0	2016-06-28
v4.1.1	2016-07-11
v4.2.1	2016-08-03
v4.2.2	2016-08-16
v4.2.3	2016-09-13
v4.2.4	2016-09-19
v4.2.5	2016-09-20
v4.2.6	2016-09-22
v4.2.7	2016-10-11
v4.2.8	2016-10-20
v4.3.0	2016-10-27

Examples

Installation

There are a variety of ways to download and use D3.

Direct Script Download

1. Download and extract d3.zip
2. Copy the resulting folder to where you will keep your project's dependencies
3. Reference d3.js (for development) or d3.min.js (for production) in your HTML: `<script type="text/javascript" src="scripts/d3/d3.js"></script>`

NPM

1. Initialize NPM in your project if you have not done so already: `npm init`
2. NPM install D3: `npm install --save d3`
3. Reference d3.js (for development) or d3.min.js (for production) in your HTML: `<script type="text/javascript" src="node_modules/d3/build/d3.js"></script>`

CDN

1. Reference d3.js (for development) or d3.min.js (for production) in your HTML: `<script type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/d3/4.1.1/d3.js"></script>`

GITHUB

1. Get any version of d3.js (for development) or d3.min.js (for production) from Github : `<script type="text/javascript" src="https://raw.githubusercontent.com/d3/d3/v3.5.16/d3.js"></script>`

To link directly to the latest release, copy this snippet:

```
<script src="https://d3js.org/d3.v4.min.js"></script>
```

Simple Bar Chart

index.html

```
<!doctype html>
<html>
  <head>
    <title>D3 Sample</title>
  </head>
  <body>
    <!-- This will serve as a container for our chart. This does not have to be a div, and can
in fact, just be the body if you want. -->
    <div id="my-chart"></div>

    <!-- Include d3.js from a CDN. -->
    <script type="text/javascript"
src="https://cdnjs.cloudflare.com/ajax/libs/d3/4.1.1/d3.js"></script>

    <!-- Include our script that will make our bar chart. -->
```

```
    <script type="text/javascript" src="chart.js"></script>
  </body>
</html>
```

chart.js

```
// Sample dataset. In a real application, you will probably get this data from another source
such as AJAX.
var dataset = [5, 10, 15, 20, 25]

// Sizing variables for our chart. These are saved as variables as they will be used in
calculations.
var chartWidth = 300
var chartHeight = 100
var padding = 5

// We want our our bars to take up the full height of the chart, so, we will apply a scaling
factor to the height of every bar.
var heightScalingFactor = chartHeight / getMax(dataset)

// Here we are creating the SVG that will be our chart.
var svg = d3
  .select('#my-chart')           // I'm starting off by selecting the container.
    .append('svg')               // Appending an SVG element to that container.
    .attr('width', chartWidth)   // Setting the width of the SVG.
    .attr('height', chartHeight) // And setting the height of the SVG.

// The next step is to create the rectangles that will make up the bars in our bar chart.
svg
  .selectAll('rect')                                          // I'm selecting all of the
rectangles in the SVG (note that at this point, there actually aren't any, but we'll be
creating them in a couple of steps).
  .data(dataset)                                              // Then I'm mapping the dataset
to those rectangles.
  .enter()                                                    // This step is important in
that it allows us to dynamically create the rectangle elements that we selected previously.
    .append('rect')                                           // For each element in the
dataset, append a new rectangle.
      .attr('x', function (value, index) {                    // Set the X position of the
rectangle by taking the index of the current item we are creating, multiplying it by the
calculated width of each bar, and adding a padding value so we can see some space between
bars.
          return (index * (chartWidth / dataset.length)) + padding
        })
      .attr('y', function (value, index) {                    // Set the rectangle by
subtracting the scaled height from the height of the chart (this has to be done becuase SVG
coordinates start with 0,0 at their top left corner).
        return chartHeight - (value * heightScalingFactor)
      })
      .attr('width', (chartWidth / dataset.length) - padding) // The width is dynamically
calculated to have an even distribution of bars that take up the entire width of the chart.
      .attr('height', function (value, index) {               // The height is simply the
value of the item in the dataset multiplied by the height scaling factor.
        return value * heightScalingFactor
      })
      .attr('fill', 'pink')                                   // Sets the color of the bars.

/**
 *  Gets the maximum value in a collection of numbers.
```

```
 */
function getMax(collection) {
  var max = 0

  collection.forEach(function (element) {
    max = element > max ? element : max
  })

  return max
}
```

Sample code available at https://github.com/dcsinnovationlabs/D3-Bar-Chart-Example

Demo available at https://dcsinnovationlabs.github.io/D3-Bar-Chart-Example/

Hello, world!

Create an `.html` file containing this snippet:

```
<!DOCTYPE html>
<meta charset="utf-8">
<body>
<script src="//d3js.org/d3.v4.min.js"></script>
<script>

d3.select("body").append("span")
    .text("Hello, world!");

</script>
```

See this snippet in action at this JSFiddle.

What is D3? Data-Driven Documents.

We are so used to the name **D3.js** that it's possible to forget that D3 is actually **DDD** (**D**ata-**D**riven **D**ocuments). And that's what D3 does well, a data-driven approach to DOM (Document Object Model) manipulation: D3 binds data to DOM elements and manipulates those elements based on the bounded data.

Let's see a very basic feature of D3 in this example. Here, we won't append any SVG element. Instead, we'll use an SVG already present on the page, something like this:

```
<svg width="400" height="400">
    <circle cx="50" cy="50" r="10"></circle>
    <circle cx="150" cy="50" r="10"></circle>
    <circle cx="210" cy="320" r="10"></circle>
    <circle cx="210" cy="30" r="10"></circle>
    <circle cx="180" cy="200" r="10"></circle>
</svg>
```

This is a pretty basic SVG, with 5 circles. Right now, those circles are not "bound" to any data. Let's check this last allegation:

In our code, we write:

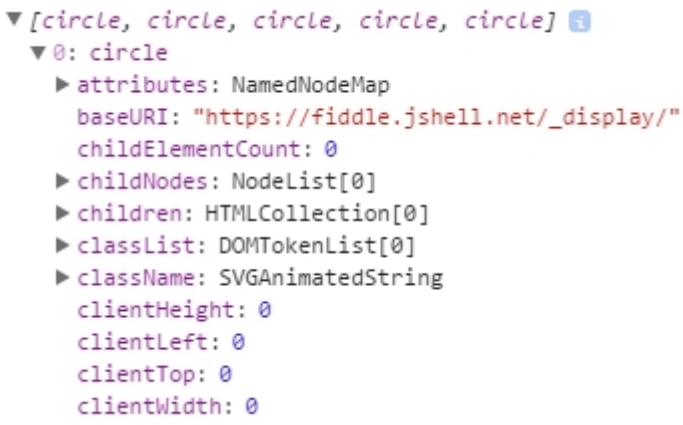
```
var svg = d3.select("svg");
var circles = svg.selectAll("circle");
console.log(circles.nodes());
```

Here, `d3.select("svg")` returns a d3 object containing `<svg width="400" height="400"></svg>` tag and all child tags, the `<circle>`s. Note that if multiple `svg` tags exist on the page, only the first is selected. If you do not want this, you can also select by tag id, like `d3.select("#my-svg")`. The d3 object has built in properties and methods that we will use a lot later.

`svg.selectAll("circle")` creates an object from all `<circle></circle>` elements from within the `<svg>` tag. It can search through multiple layers, so it does not matter if the tags are direct children.

`circles.nodes()` returns the circle tags with all of their properties.

If we look at the console and choose the first circle, we're gonna see something like this:



First, we have `attributes`, then `childNodes`, then `children`, and so on... but no data.

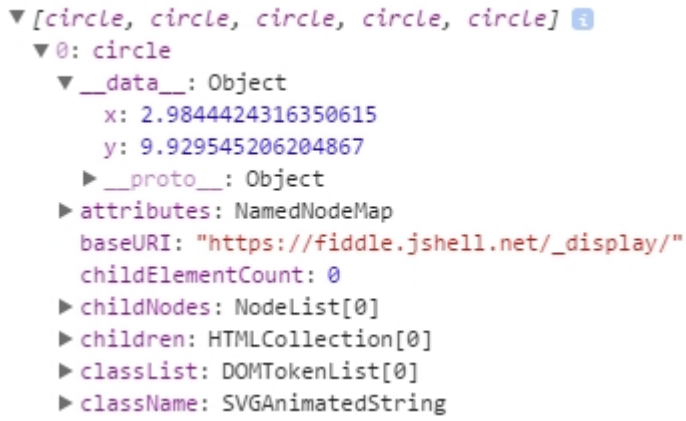
Let's bind some data

But, what happens if we bind *data* to these DOM elements?

In our code, there is a function that creates an object with two properties, `x` and `y`, with numeric values (this object is inside an array, check the fiddle below). If we bind this data to the circles...

```
circles.data(data);
```

This is what we are going to see if we inspect the console:



We have something new just before `attributes`! Something named `__data__`... and look: the values of `x` and `y` are there!

We can, for instance, change the position of the circles based on these data. Have a look at this fiddle.

This is what D3 does best: binding data to DOM elements and manipulating those DOM elements based on the bounded data.

Simple D3 Chart: Hello World!

Paste this code into an empty HTML file and run it in your browser.

```
<!DOCTYPE html>

<body>

<script src="https://d3js.org/d3.v4.js"></script>    <!-- This downloads d3 library -->

<script>
//This code will visualize a data set as a simple scatter chart using d3. I omit axes for
simplicity.
var data = [        //This is the data we want to visualize.
                    //In reality it usually comes from a file or database.
  {x: 10,    y: 10},
  {x: 10,    y: 20},
  {x: 10,    y: 30},
  {x: 10,    y: 40},
  {x: 10,    y: 50},
  {x: 10,    y: 80},
  {x: 10,    y: 90},
  {x: 10,    y: 100},
  {x: 10,    y: 110},
  {x: 20,    y: 30},
  {x: 20,    y: 120},
  {x: 30,    y: 10},
  {x: 30,    y: 20},
  {x: 30,    y: 30},
  {x: 30,    y: 40},
  {x: 30,    y: 50},
  {x: 30,    y: 80},
  {x: 30,    y: 90},
  {x: 30,    y: 100},
  {x: 30,    y: 110},
```

```
{x: 40,    y: 120},
{x: 50,    y: 10},
{x: 50,    y: 20},
{x: 50,    y: 30},
{x: 50,    y: 40},
{x: 50,    y: 50},
{x: 50,    y: 80},
{x: 50,    y: 90},
{x: 50,    y: 100},
{x: 50,    y: 110},
{x: 60,    y: 10},
{x: 60,    y: 30},
{x: 60,    y: 50},
{x: 70,    y: 10},
{x: 70,    y: 30},
{x: 70,    y: 50},
{x: 70,    y: 90},
{x: 70,    y: 100},
{x: 70,    y: 110},
{x: 80,    y: 80},
{x: 80,    y: 120},
{x: 90,    y: 10},
{x: 90,    y: 20},
{x: 90,    y: 30},
{x: 90,    y: 40},
{x: 90,    y: 50},
{x: 90,    y: 80},
{x: 90,    y: 120},
{x: 100,    y: 50},
{x: 100,    y: 90},
{x: 100,    y: 100},
{x: 100,    y: 110},
{x: 110,    y: 50},
{x: 120,    y: 80},
{x: 120,    y: 90},
{x: 120,    y: 100},
{x: 120,    y: 110},
{x: 120,    y: 120},
{x: 130,    y: 10},
{x: 130,    y: 20},
{x: 130,    y: 30},
{x: 130,    y: 40},
{x: 130,    y: 50},
{x: 130,    y: 80},
{x: 130,    y: 100},
{x: 140,    y: 50},
{x: 140,    y: 80},
{x: 140,    y: 100},
{x: 140,    y: 110},
{x: 150,    y: 50},
{x: 150,    y: 90},
{x: 150,    y: 120},
{x: 170,    y: 20},
{x: 170,    y: 30},
{x: 170,    y: 40},
{x: 170,    y: 80},
{x: 170,    y: 90},
{x: 170,    y: 100},
{x: 170,    y: 110},
{x: 170,    y: 120},
{x: 180,    y: 10},
```

```
  {x: 180,    y: 50},
  {x: 180,    y: 120},
  {x: 190,    y: 10},
  {x: 190,    y: 50},
  {x: 190,    y: 120},
  {x: 200,    y: 20},
  {x: 200,    y: 30},
  {x: 200,    y: 40},
  {x: 210,    y: 80},
  {x: 210,    y: 90},
  {x: 210,    y: 100},
  {x: 210,    y: 110},
  {x: 210,    y: 120},
  {x: 220,    y: 80},
  {x: 220,    y: 120},
  {x: 230,    y: 80},
  {x: 230,    y: 120},
  {x: 240,    y: 90},
  {x: 240,    y: 100},
  {x: 240,    y: 110},
  {x: 270,    y: 70},
  {x: 270,    y: 80},
  {x: 270,    y: 90},
  {x: 270,    y: 100},
  {x: 270,    y: 120}
];

//The following code chains a bunch of methods. Method chaining is what makes d3 very simple
and concise.
d3.select("body").append("svg").selectAll()  //'d3' calls the d3 library
                                             //'.select' selects the object (in this case the
body of HTML)
                                             //'.append' adds SVG element to the body
                                             //'.selectAll()' selects all SVG elements
    .data(data)                              //'.data' gets the data from the variable 'data'
  .enter().append("circle")                  //'.enter' enters the data into the SVG
                                             //the data enter as circles with
'.append("circle")'
    .attr("r", 3)                            //'.attr' adds/alters atributes of SVG,
                                             //such as radius ("r"), making it 3 pixels
    .attr("cx", function(d) { return d.x; }) //coordinates "cx" (circles' x coordinates)
    .attr("cy", function(d) { return d.y; }) //coordinates "cy" (circles' y coordinates)
    .style("fill", "darkblue");              //'.style' changes CSS of the SVG
                                             //in this case, fills circles with "darkblue"
color

</script>
```

Here is a JSFiddle of the chart.

You can also download the already created HTML file from GitHub.

The next step in learning d3 can be following Mike Bostock's (the d3's creator's) tutorial to create a bar chart from scratch.

Read Getting started with d3.js online: https://riptutorial.com/d3-js/topic/876/getting-started-with-d3-js

Chapter 2: Approaches to create responsive d3.js charts

Syntax

- var width = document.getElementById('chartArea').clientWidth;
- var height = width / 3.236;
- window.onresize = resizeFunctionCall;

Examples

Using bootstrap

One approach that is employed often is to use bootstrap's gridded framework in order to define the area that the chart will exist in. Using this in conjunction with `clientWidth` variable and the `window.onresize` event, it is very easy to create responsive d3 SVGs.

Let's first create a row and a column that our chart will be built in.

index.html

```
<!DOCTYPE html>
<html>
<head>
<link rel="stylesheet" type="text/css"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css">
</head>
<body>
  <div class="container">
    <div class="row">
      <div class="col-xs-12 col-lg-6" id="chartArea">
      </div>
    </div>
  </div>
<script src="https://cdnjs.cloudflare.com/ajax/libs/d3/4.1.1/d3.js"></script>
<script src="chart.js"></script>
</body>
</html>
```

This will create a colum that will be full screen on mobile device and half on a large screen.

chart.js

```
var width = document.getElementById('chartArea').clientWidth;
//this allows us to collect the width of the div where the SVG will go.
var height = width / 3.236;
//I like to use the golden rectangle ratio if they work for my charts.
```

```
var svg = d3.select('#chartArea').append('svg');
//We add our svg to the div area


//We will build a basic function to handle window resizing.
function resize() {
    width = document.getElementById('chartArea').clientWidth;
    height = width / 3.236;
    d3.select('#chartArea svg')
      .attr('width', width)
      .attr('height', height);
}

window.onresize = resize;
//Call our resize function if the window size is changed.
```

This is an extremely simplified example, but does cover the basic concepts of how to setup charts to be responsive. The resize function will need to make a call to your main update function that will redraw all paths, axis, and shapes just as if the underlying data had been updated. Most d3 users who are concerned with responsive visualizations will already know how to build their update events into functions that are easy to call as shown in the intro topic and this topic.

Read Approaches to create responsive d3.js charts online: https://riptutorial.com/d3-js/topic/4312/approaches-to-create-responsive-d3-js-charts

Chapter 3: Core SVG concepts used in D3.js visualization

Examples

Coordinate System

In a normal mathematical coordinate system, the point x=0, y=0 is at the lower left corner of the graph. But in the SVG coordinate system, this (0,0) point is at the top left corner of the 'canvas', it is sort of similar to CSS when you specify the position to absolute/fix and use top and left to control the exact point of the element.

It is essential to keep in mind that as y increases in SVG, the shapes move down.

Let's say we're going to create a scatterplot with each point correspondent to a x value and y value. To scale the value, we need to set the domain and the range like this:

```
d3.svg.scale()
  .range([0, height])
  .domain([0,max])
```

However, if you just keep the settings like this, the points will be based on the top horizontal edge instead of the bottom horizontal line as what we expected.

The nice thing about d3 is that you can easily change this by a simple adjustment in domain setting:

```
d3.scale.linear()
  .range([height, 0])
  .domain([0, max])
```

With above code, the zero point of domain is correspondent to the height of the SVG, which is the bottom line of the chart in the viewer's eyes, meanwhile, the max value of the source data will be correspondent to the zero point of the SVG coordinate system, which the max value for viewers.

The Element

`<rect>` represents rectangle, apart from aesthetic properties like stroke and fill, rectangle shall be defined by location and size.

As for the location, it is determined by the x and y attributes. The location is relative to the rectangle's parent. And if you don't specify the x or y attribute, the default will be 0 relative to the parent element.

After you specify the location, or rather the 'starting point' of the rect, the next thing is to specify the size, which is essential if you want to actually draw sth on the canvas, that is to say, if you

don’t specify the size attributes or the value is set as 0, you won’t see anything on the canvas.

Case: Bar Chart

Continue with the first scenario, the y axes, but this time, let’s try to draw a bar chart.

Assuming the y scale setting is the same, the y axis is properly set as well, the only difference between the scatterplot and this bar chart is that, we need to specify the width and the height, particularly the height. To be more specific, we’ve already got the ‘starting point’, the rest is to use things like for the height:

```
.attr("height", function(d){
  return (height - yScale(d.value))
})
```

The Element

`<svg>` element is the root element, or the canvas as we are drawing charts on it.

SVG elements can be nested inside each other, and in this way, SVG shapes can be grouped together, meanwhile, all shapes nested inside an element will be positioned relative to its enclosing element.

One thing might need mentioning is that, we can’t nest inside another , it won’t work.

Case: Multiple Charts

For example, this multiple donuts chart is made up by multiple elements, which contains a donut chart respectively. This can be achieved by using the element, but in this case where we only want to put donut chart one by one next to each other, is more convenient.

One thing to bear in mind is that we can’t use transform attribute on nevertheless we can use x, y for position.

The Element

SVG doesn’t currently support automatic line breaks or word wrapping, that’s when comes to rescue. element positions new lines of text in relation to the previous line of text. And by using dx or dy within each of these spans, we can position the word in relation to the word before it.

Case: Annotation on Axes

For example, when we want to add an annotation on y Axe:

```
svg.append("g")
  .attr("class", "y axis")
  .call(yAxis)
.append("text")
  .attr("transform", "rotate(-90)")
  .attr("y", 6)
  .attr("dy", ".71em")
```

```
  .style("text-anchor", "end")
  .text("Temperature (ºF)");
```

Correctly appending an SVG element

This is a relatively common mistake: You created an `rect` element, in a bar chart for instance, and you want to add a text label (let's say, the value of that bar). So, using the same variable that you used to append the `rect` and define its `x` and `y` position, you append your `text` element. Very logic, you may think. But this will not work.

How does this mistake occur?

Let's see a concrete example, a very basic code for creating a bar chart (fiddle here):

```
var data = [210, 36, 322, 59, 123, 350, 290];

var width = 400, height = 300;

var svg = d3.select("body")
    .append("svg")
    .attr("width", width)
    .attr("height", height);

var bars = svg.selectAll(".myBars")
    .data(data)
    .enter()
    .append("rect");

bars.attr("x", 10)
    .attr("y", function(d,i){ return 10 + i*40})
    .attr("width", function(d){ return d})
    .attr("height", 30);
```

Which gives us this result:

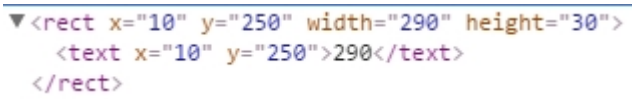
But you want to add some text elements, maybe a simple value to each bar. So, you do this:

```
bars.append("text")
   .attr("x", 10)
   .attr("y", function(d,i){ return 10 + i*40})
   .text(function(d){ return d});
```

And, voilà: nothing happens! If you doubt it, here is the fiddle.

"But I'm seeing the tag!"

If you inspect the SVG created by this last code, you're gonna see this:



And at this point a lot of people say: "But I'm seeing the text tag, it's appended!". Yes, it is, but this doesn't mean it will work. You can append *anything*! See this, for example:

```
svg.append("crazyTag");
```

It will give you this result:

```
<svg>
    <crazyTag></crazyTag>
</svg>
```

But you don't expect any result just because the tag is there, do you?

Append SVG elements the correct way

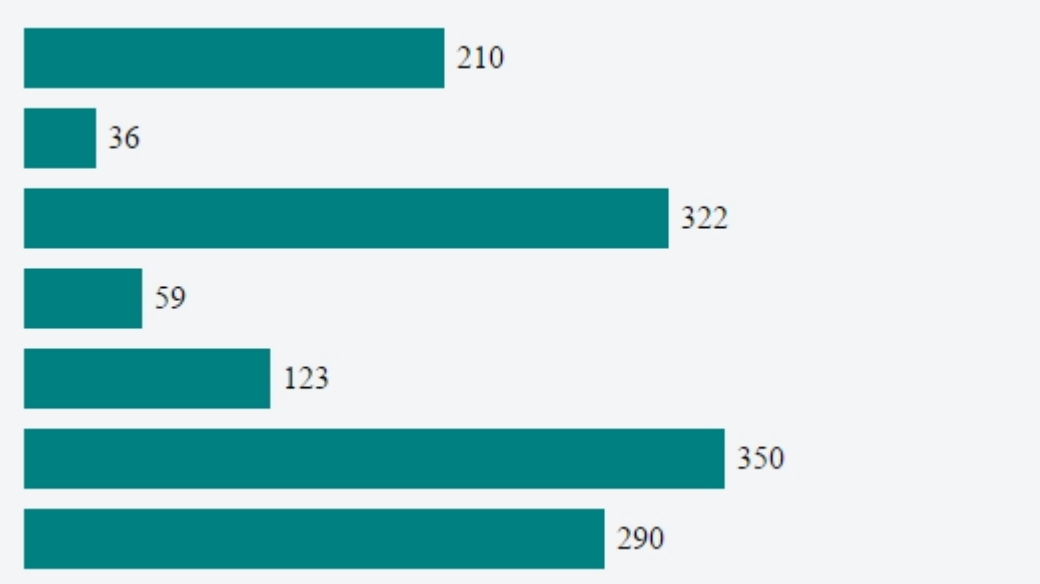
Learn what SVG elements can hold children, reading the specifications. In our last example, the code doesn't work because `rect` elements cannot contain `text` elements. So, how to display our texts?

Create another variable, and append the `text` to the SVG:

```
var texts = svg.selectAll(".myTexts")
    .data(data)
    .enter()
    .append("text");

texts.attr("x", function(d){ return d + 16})
    .attr("y", function(d,i){ return 30 + i*40})
    .text(function(d){ return d});
```

And this is the outcome:



And here is the fiddle.

SVG: the drawing order

This is something that can be frustrating: you make a visualisation using D3.js but the rectangle you want on top is hidden behind another rectangle, or the line you planned to be behind some circle is actually over it. You try to solve this using the *z-index* in your CSS, but it doesn't work (in SVG 1.1).

The explanation is simple: In an SVG, the order of the elements defines the order of the "painting", and the order of the painting defines who goes on top.

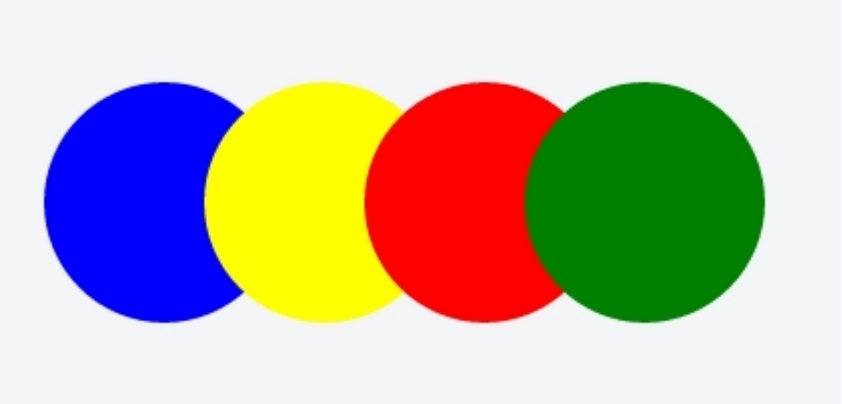
> Elements in an SVG document fragment have an implicit drawing order, with the first elements in the SVG document fragment getting "painted" first. Subsequent elements are painted on top of previously painted elements.

So, suppose that we have this SVG:

```
<svg width="400" height=200>
    <circle cy="100" cx="80" r="60" fill="blue"></circle>
    <circle cy="100" cx="160" r="60" fill="yellow"></circle>
    <circle cy="100" cx="240" r="60" fill="red"></circle>
    <circle cy="100" cx="320" r="60" fill="green" z-index="-1"></circle>
</svg>
```

He have four circles. The blue circle is the first one "painted", so it will be bellow all the others. Then we have the yellow one, then the red one, and finally the green one. The green one is the last one, and it will be on the top.

This is how it looks:



Changing the order of SVG elements with D3

So, is it possible to change the order of the elements? Can I make the red circle in front of the green circle?

Yes. The first approach that you need to have in mind is the order of the lines in your code: draw first the elements of the background, and later in the code the elements of the foreground.

But we can dynamically change the order of the elements, even after they were painted. There are several plain JavaScript functions that you can write to do this, but D3 has already 2 nice features, `selection.raise()` and `selection.lower()`.

According to the API:

> selection.raise(): Re-inserts each selected element, in order, as the last child of its parent. selection.lower(): Re-inserts each selected element, in order, as the first child of its parent.

So, to show how to manipulate the order of the elements in our previous SVG, here is a very small code:

```
d3.selectAll("circle").on("mouseover", function(){
    d3.select(this).raise();
});
```

What does it do? It selects all the circles and, when the user hover over one circle, it selects that particular circle and brings it to the front. Very simple!

And here is the JSFiddle with the live code.

Read Core SVG concepts used in D3.js visualization online: https://riptutorial.com/d3-js/topic/2537/core-svg-concepts-used-in-d3-js-visualization

Chapter 4: D3 Projections

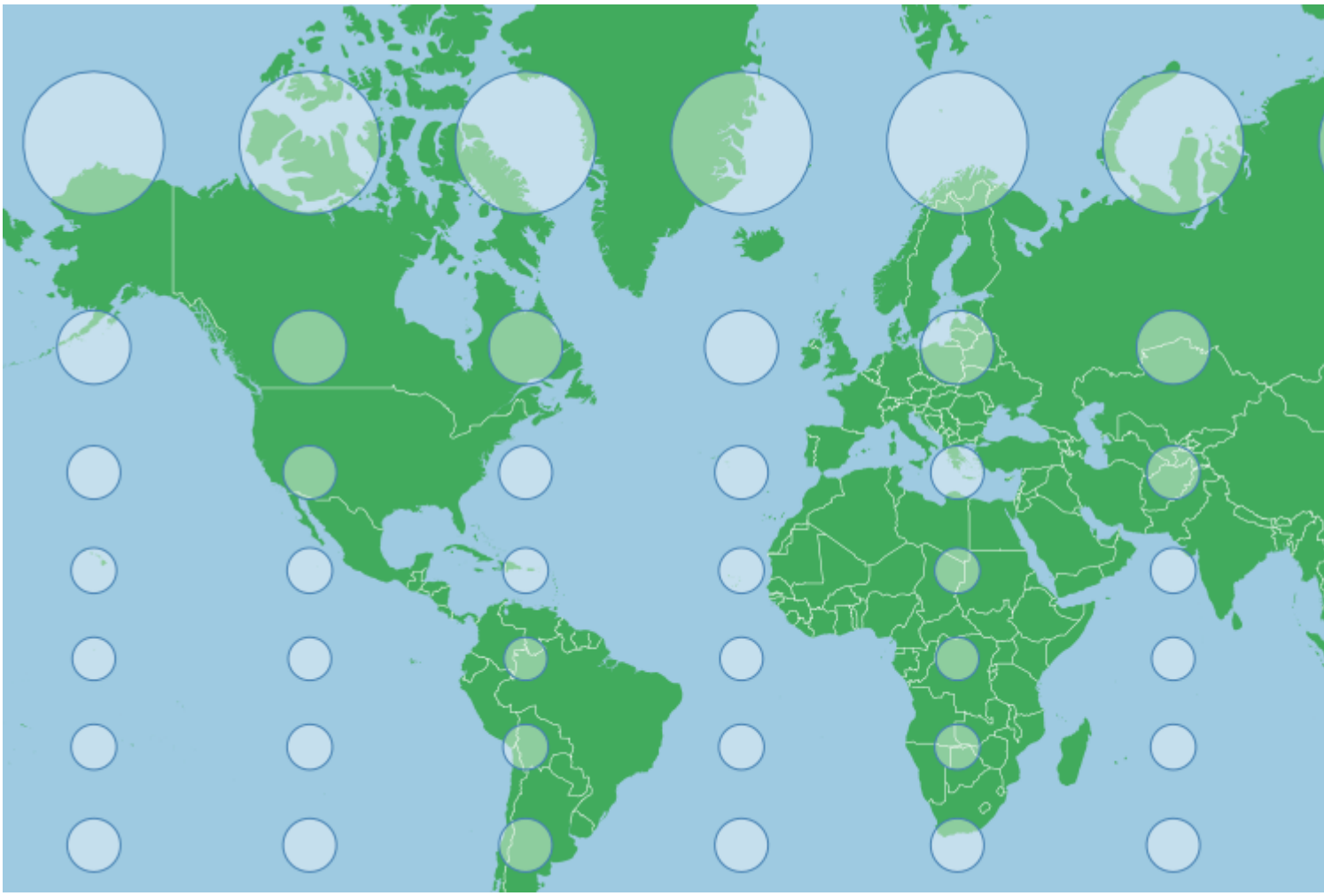
Examples

Mercator Projections

A Mercator projection is one of the most recognizable projections used in maps. Like all map projections it has distortion and for a Mercator, projection this is most noticeable in the polar regions. It is a cylindrical projection, meridians run vertically and latitudes run horizontally.

Scale depends on the size of your svg, for this example, all scales used are with a 960 pixel wide by 450 pixel high svg.

The map below shows a Tissot's Indicatrix for a Mercator projection, each circle is in reality the same size but the projection obviously shows some as larger than others:



This distortion is because the projection tries to avoid a one dimensional stretching of the map. As meridians begin to merge at the North and South Poles, the distance between them begins to approach zero, but the surface of the projection is rectangular (not the map, though it too is rectangular) and doesn't allow a change in the distance between meridians in the projection. This would stretch features along the x axis near the poles, distorting their shape. In order to counter this, a Mercator stretches the y axis as well as one approaches the poles, which makes the

indicators circular.

The projection for the map above is essentially the default Mercator projection shifted up a little:

```
var projection = d3.geoMercator()
    .scale(155)
    .center([0,40]) // Pan north 40 degrees
    .translate([width/2,height/2]);
```

To center the projection on a given point with a known latitude and a known longitude, you can pan to that point easily by specifying the center:

```
var projection = d3.geoMercator()
    .center([longitude,latitude])
```

This will pan to that feature (but not zoom) on the projected surface (which looks like the map above).

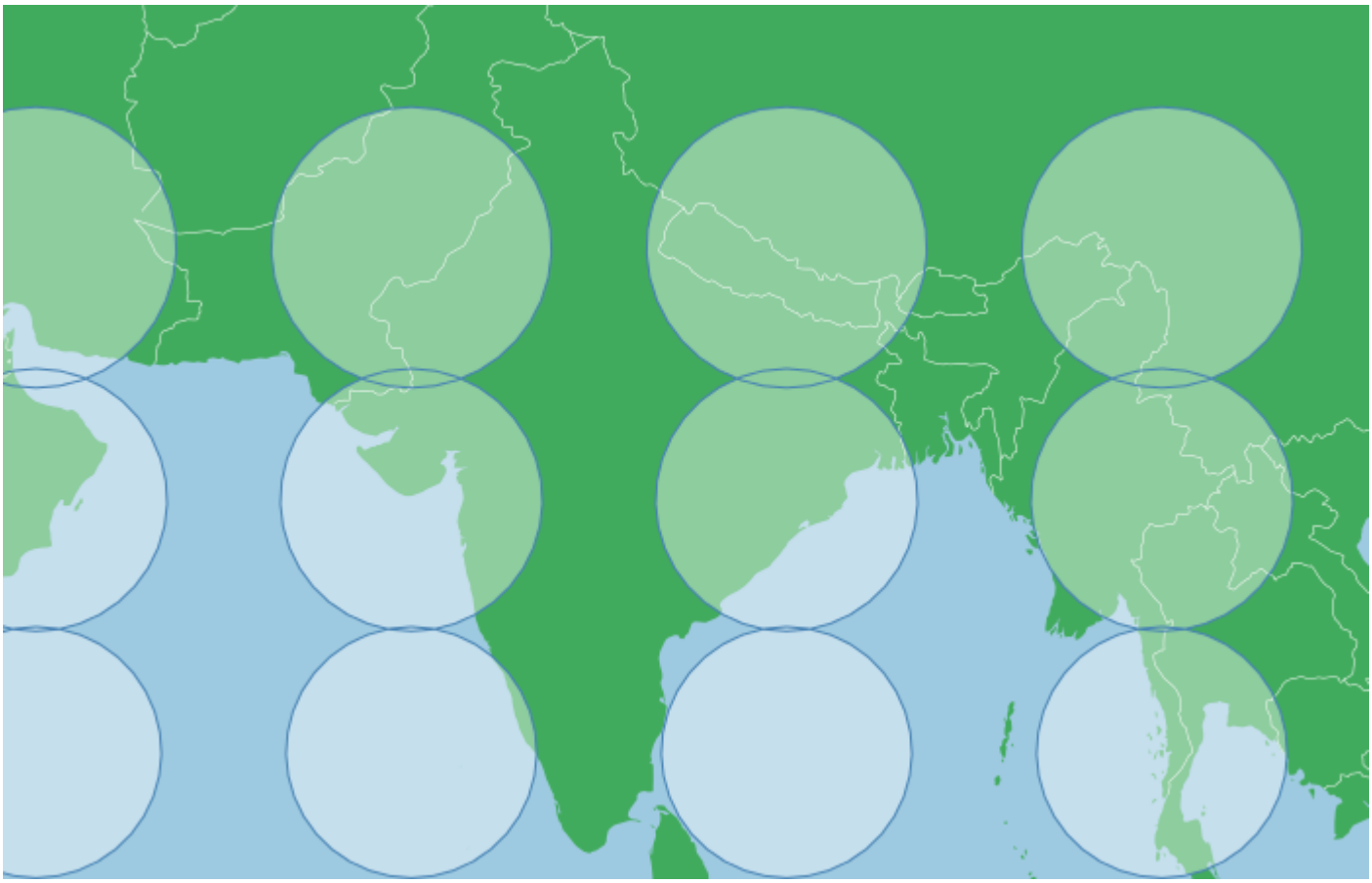
Scales will need to be tailored to the area of interest, larger numbers equal a larger features (greater degree of zooming in), smaller numbers the opposite. Zooming out can be a good way to get your bearings to see where your features are relative to the point you have centered on - if you are having trouble finding them.

Due to the nature of a Mercator projection, areas near the equator or at low latitudes will do best with this type of projection, while polar areas can be highly distorted. Distortion is even along any horizontal line, so areas that are wide but not tall may be good as well, while areas that have a large difference between their northern and southern extremes have more visual distortion.

For India, for example, we could use:

```
var projection = d3.geoMercator()
    .scale(800)
    .center([77,21])
    .translate([width/2,height/2]);
```

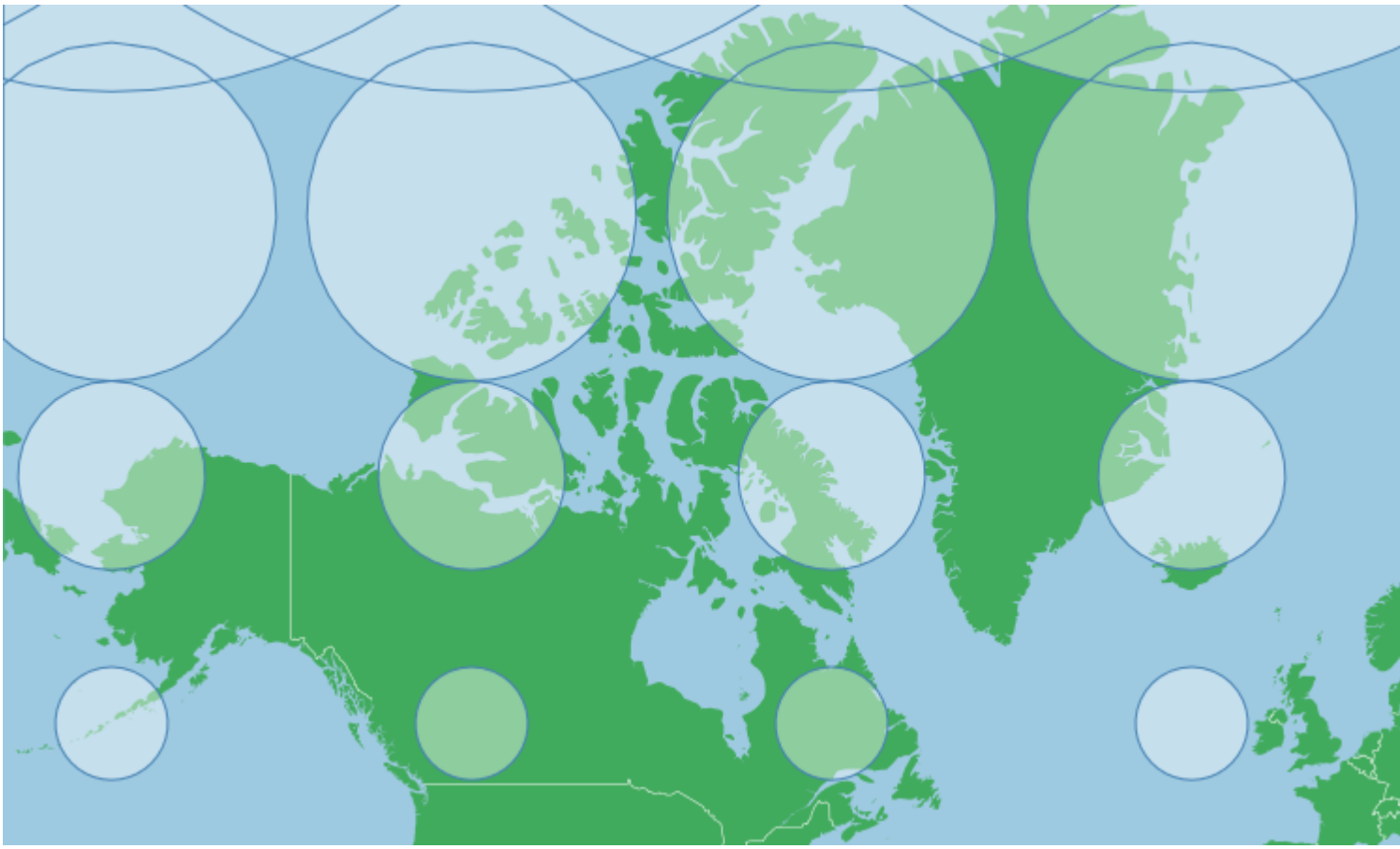
Which gives us (again with a Tissot's indicatrix to show distortion):



This has a low level of distortion, but the circles are largely the same size (you can see a greater overlap between the top two rows than the bottom two rows, so distortion is visible). Overall though, the map shows a familiar shape for India.

Distortion in area is not linear, it is greatly exaggerated towards the poles, so Canada with fairly far apart northern and southern extremes and a position fairly close to the poles means distortion may be untenable:

```
var projection = d3.geoMercator()
    .scale(225)
    .center([-95,69.75])
    .translate([width/2,height/2]);
```

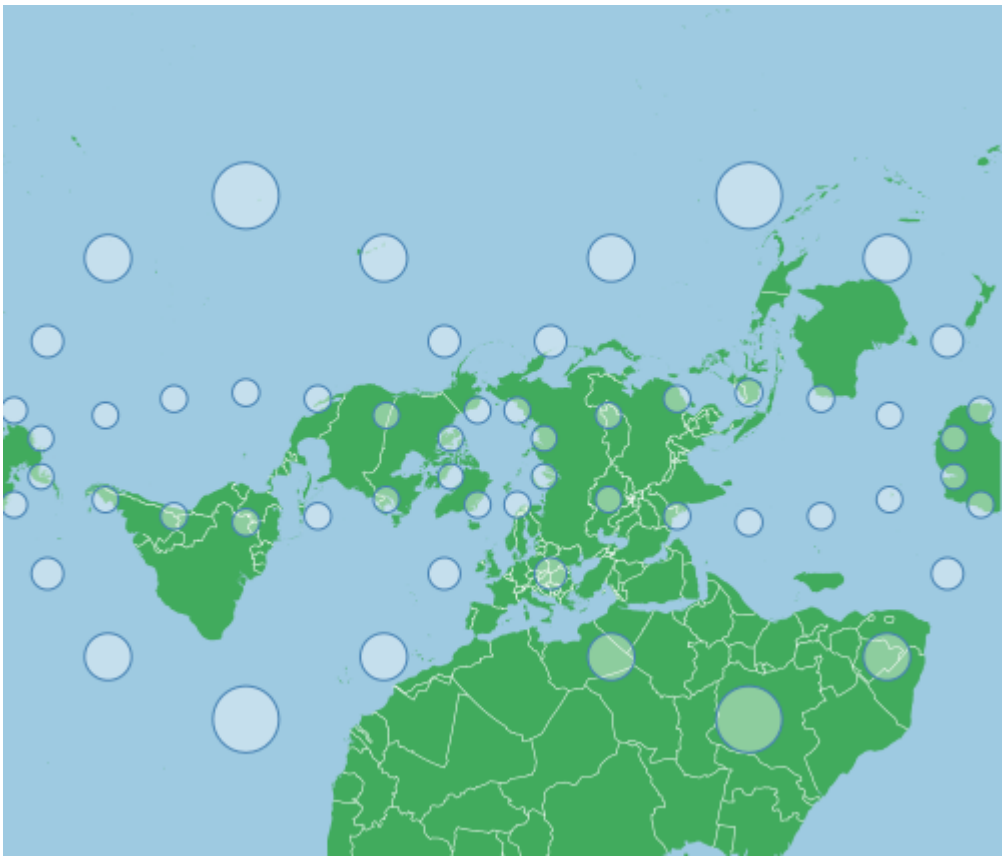


This projection makes Greenland look as big as Canada, when in reality Canada is nearly five times larger than Greenland. This is simply because Greenland is further north than the bulk of Canada (Sorry Ontario, I appear to have cut off some of your southern extremity).

Because the y axis is considerably stretched near the poles in a Mercator, this projection uses a point considerably north of the geographic center of Canada. If dealing with high latitude areas you may need to tailor your centering point to account for this stretching.

If you are in need of a Mercator Projection for polar areas there is a way that you can minimize distortion and still use a Mercator Projection. You can achive this by rotating the globe. If you rotate the x axis on the default Mercator you will appear to pan it left or right (you simply spin the globe in the cylinder that you are projecting on to), if however you change the y axis of the default Mercator, you can turn the earth sideways or to any other angle. Here is a Mercator with a y rotation of -90 degrees:

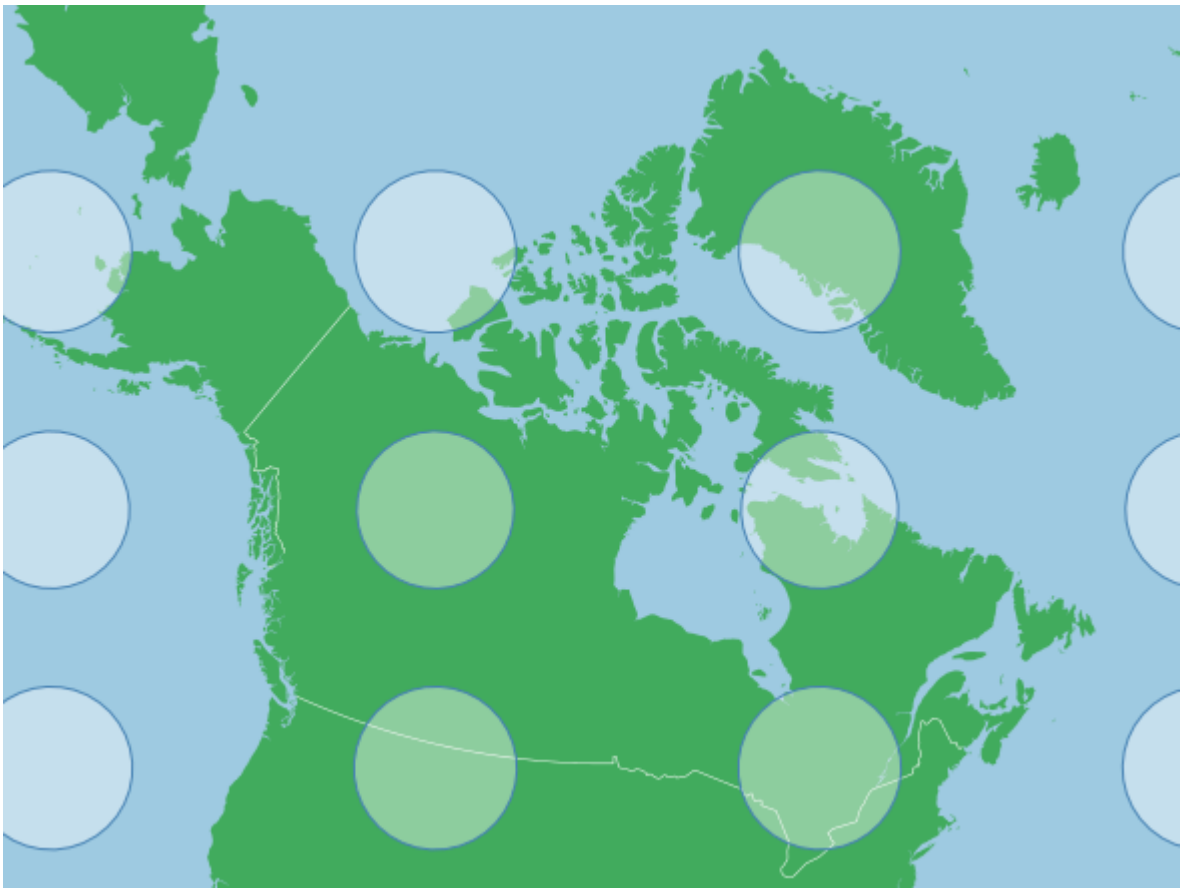
```
var projection = d3.geoMercator()
.scale(155)
.rotate([0,-90]);
.translate([width/2,height/2]);
```



The indicatrix points are in the same locations as the first map above. Distortion still increases as ones reaches for the top or bottom of the map. This is how a default Mercator map would appear if the earth rotated around a North Pole at [0,0] and a South Pole at [180,0], the rotation has turned the cylinder which we are projecting on to 90 degrees relative to the poles. Note how the poles no longer have untenable distortion, this presents an alternative method to project areas near the poles without too much distortion in area.

Using Canada as an example again, we can rotate the map to a center coordinate, and this will minimize distortion in area. To do so we can rotate to a centering point again, but this requires one extra step. With centering we pan to a feature, with rotation we move the earth under us, so we need the negative of our centering coordinate:

```
var projection = d3.geoMercator()
    .scale(500)
    .rotate([96,-64.15])
    .translate([width/2,height/2]);
```



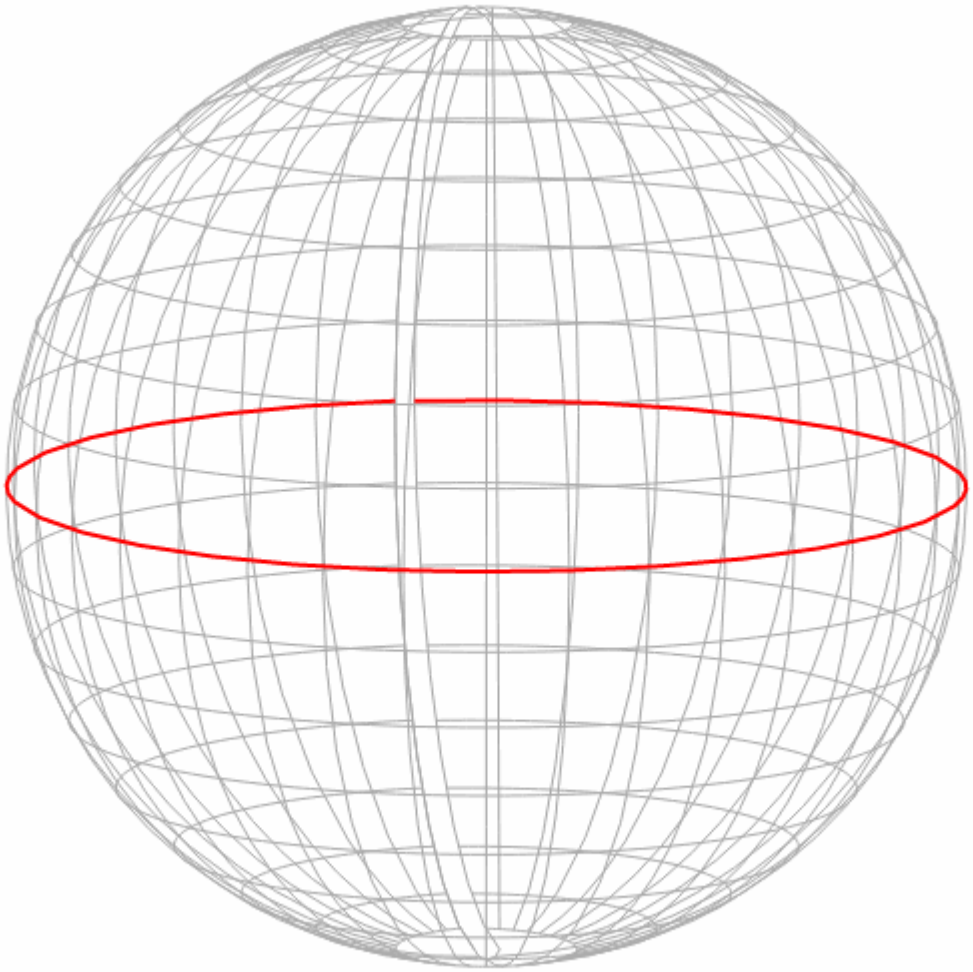
Note that the Tissot's indicatrix is showing low distortion in area now. The scale factor is also much larger than before, because this Canada is now at the origin of the projection, and along the middle of the map features are smaller than at the top or bottom (see first indicatrix above). We do not need to center because the center point or origin of this projection is at `[-96,64.15]`, centering would move us off this point.

Albers Projections

An Albers projection, or more properly, an Albers equal area conic projection, is a common conical projection and an official projeciton of a number of jurisdictions and organizations such as the US census bureau and the province of British Columbia in Canada. It preserves area at the expense of other aspects of the map like shape, angle, and distance.

General Properties

The general transformation is captured in the following gif:



(Based on Mike Bostock's block)

The Albers projection minimizes distortion around two standard parallels. These parallels represent where the conical projection intersects the earth's surface.

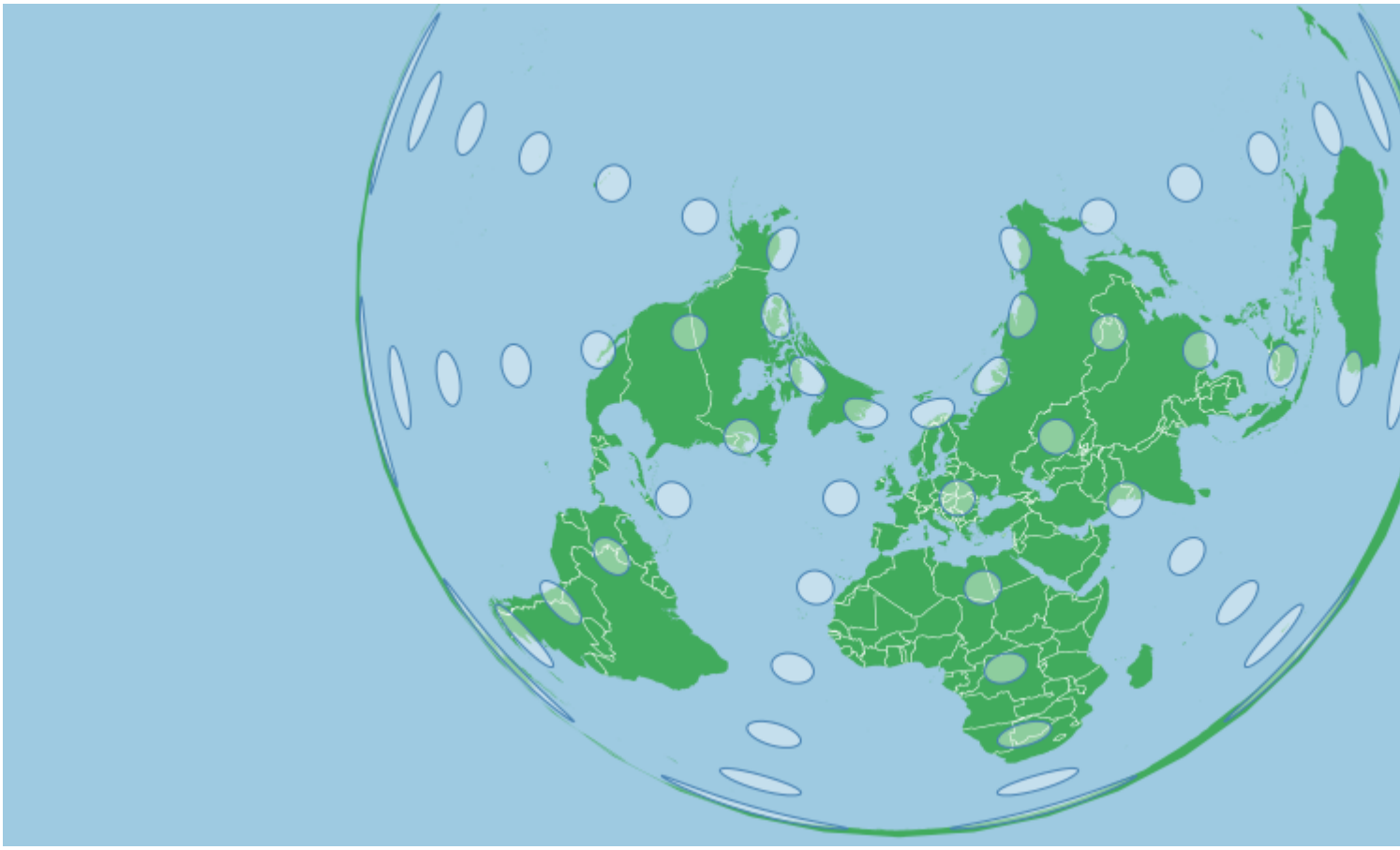
For this example, all scales are used with 960 pixel wide by 450 pixel high svg dimensions, scale will change with these dimensions

The map below shows a Tissot's Indicatrix for an Albers projection with standard parallels of 10 and 20 degrees north. Each circle is in reality the same size and shape, but the map projection will distort these in shape (not area). Notice how at around roughly 10 to 20 degrees north, the indicators are rounder than elsewhere:

This was created with the following projection:

```
var projection = d3.geoAlbers()
    .scale(120)
    .center([0,0])
    .rotate([0,0])
    .parallels([10,20])
    .translate([width/2,height/2]);
```

If we use parallels that in the higher altitudes, the degree of arcing in the projection increases. The following images uses the parallels of 50 and 60 degrees north:



```
var projection = d3.geoAlbers()
    .scale(120)
    .center([0,70]) // shifted up so that the projection is more visible
    .rotate([0,0])
    .parallels([40,50])
    .translate([width/2,height/2]);
```

If we had negative (Southern) parallels, the map be concave down instead of up. If one parallel is north and one south, the map will be concave towards the higher/more extreme parallel, if they are the same distance from the equator then the map won't be concave in any direction.

Choosing Parallels

As parallels mark the areas with the least distortion, they should be chosen based on your area of interest. If you area of interest stretches from 10 degrees north to 20 degrees north, then choosing parallels of 13 and 17 will minimize distortion throughout your map (as distortion is minimized on either side of these parallels).

Parallels shouldn't be the extreme northern and southern boundaries of your area of interest. *Parallels can both be the same value if you only want the projection to intersect the surface of the earth once.*

Projection references and definitions include parallel data that you can use to recreate standardized projections.

Centering and Rotating

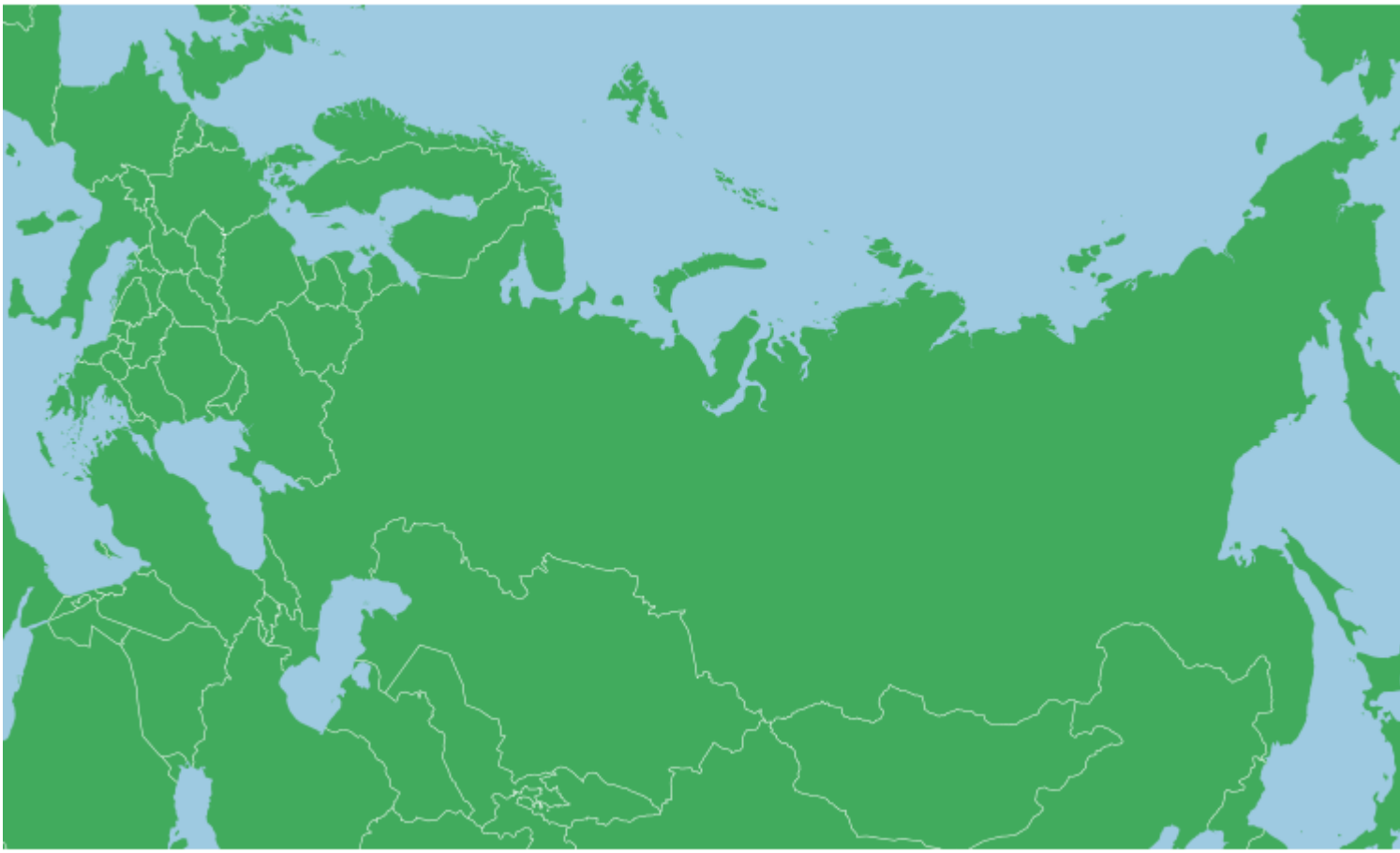
Once parallels are selected, the map must be positioned so that the area of interest is aligned properly. If using just `projection.center([x,y])`, the map will simply be panned to the selected point and no other transformation will take place. If the target area is Russia, panning might not be ideal:

```
var projection = d3.geoAlbers()
    .scale(120)
    .center([0,50]) // Shifted up so the projection is more visible
    .rotate([0,0])
    .parallels([50,60])
    .translate([width/2,height/2]);
```

The central meridian of an Albers projection is vertical, and we need to rotate the earth under the projection to change the central meridian. Rotation for an Alber's projection is the method for centering a projection on the x axis (or by longitude). And as the earth is spinning underneath the projection, we use the negative of the longitude we want to be centered. For Russia, this could be about 100 degrees East, so we'll spin the globe 100 degrees the other way.

```
var projection = d3.geoAlbers()
    .scale(120)
    .center([0,60])
    .rotate([-100,0])
    .parallels([50,60])
```

Now we can pan up and down and the features along and near the central meridian will be upright. *If you `.center()` on the x axis, your centering will be relative to the central meridian set by the rotation*. For Russia we might want to pan a fair bit North and zoom in a bit:



```
var projection = d3.geoAlbers()
    .scale(500)
    .center([0,65])
    .rotate([-100,0])
    .parallels([50,60])
```

For a feature like Russia, the arch of the map means that the far edges of the country will be stretch up around the pole, which means that the centering point might not be the centroid of your feature as you may need to pan more to the north or south than usual.

With the Tissots Indicatrix, we can see some flattening near the pole itself, but that shape is fairly true throughout the area of interest (Remember that for the size of Russia, distortion is fairly minimal, it would be much less for smaller features):

Default Parameters

Unlike most other projections, the d3.geoAlbers projection comes with default parameters that are not .rotate([0,0]) and .center([0,0]), the default projection is centered and rotated for the United States. This is also true of `.parallels()`. So if any of these are not set, they will default to non zero values.

Summary

An Albers projection is generally set with the following parameters:

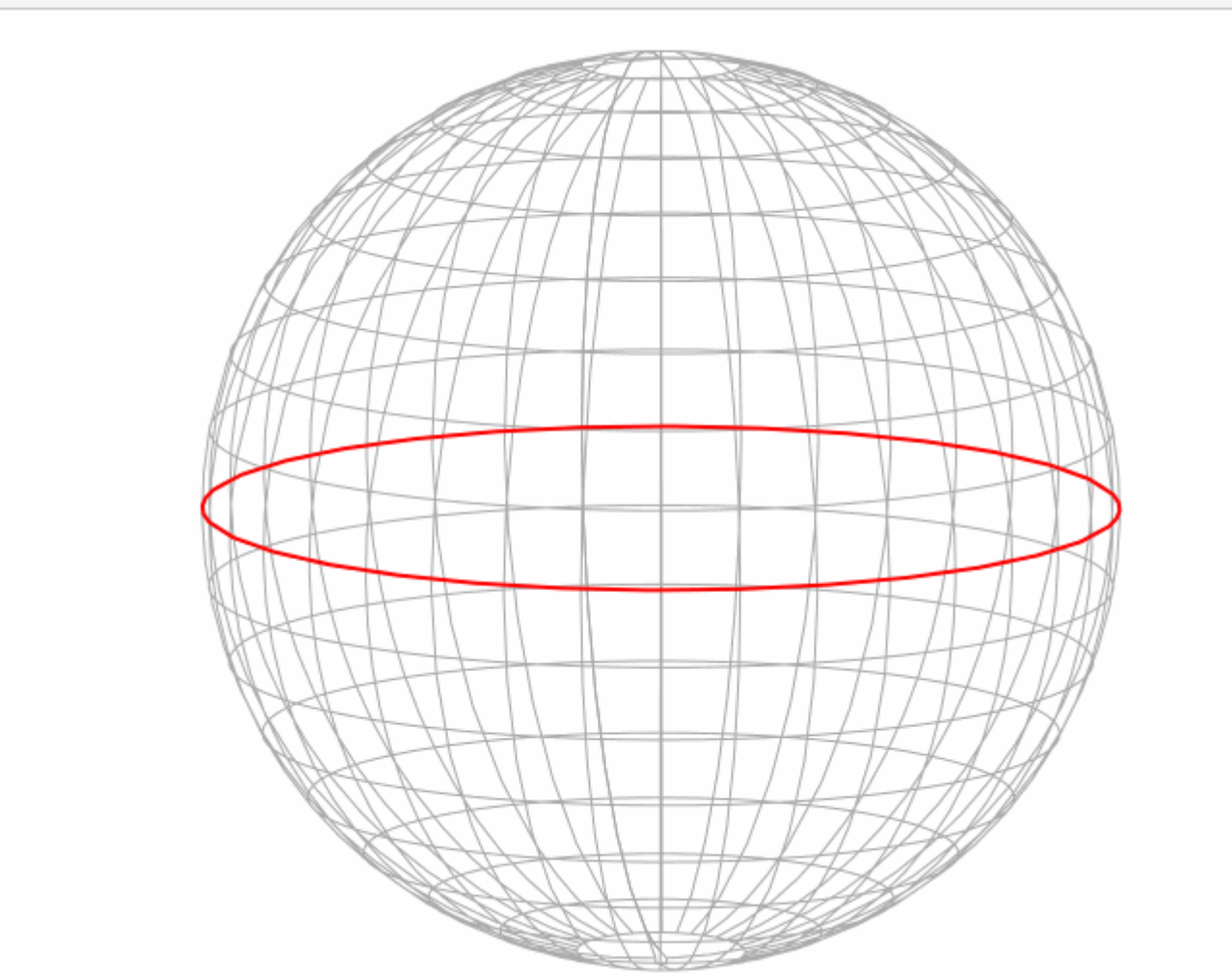
```
var projection = d3.geoAlbers()
    .rotate([-x,0])
    .center([0,y])
    .parallels([a,b]);
```

Where a and b equal the two parallels.

Azimuthal Equidistant Projections

General Properties:

An Azimuthal Equidistant projection is best recognized when used in polar areas. It is used in the UN's emblem. From the center point, angle and distance are preserved. But the projection will distort shape and area to achieve this, especially as one moves further from the center. Likewise, distance and angle are not true in locations other than the center. The projection falls within the azimuthal category (rather than cylindrical (Mercator) or conic (Albers). This projection projects the earth as a disc:



(Based on Mike Bostock's block. *Centered on North Pole, Ignore the triangular artifact on top of the image once folded out*)

Scale depends on the size of your svg, for this example, all scales used are within a 960 pixel wide by 450 pixel high svg (and screen clipped for a square where needed) - unless otherwise specified.

The map below shows a Tissot's Indicatrix for an Azimuthal Equidistant projection:

This was created with the following projection:

```
var projection = d3.geoAzimuthalEquidistant()
   .scale(70)
   .center([0,0])
   .rotate([0,0])
   .translate([width/2,height/2]);
```

Centering and Rotating:

Centering will simply pan a map but not change its overall composition. Centering an azimuthal equidistant on the North Pole while leaving other parameters unchanged or at zero will move the north pole to the center of the screen - but otherwise will not change the map above.

To properly center an area, you need to rotate it. As with any rotation in d3, it is best to think of it as moving the earth under the projection, so rotating the earth -90 degrees underneath the map on the y axis will actually place the north pole in the middle:

```
var projection = d3.geoAzimuthalEquidistant()
   .scale(70)
   .center([0,0])
   .rotate([0,-90])
   .translate([width/2,height/2]);
```

Likewise, rotation on the x axis behaves similarly. For example, to rotate the map such that the Canadian arctic is upright, while centering on the north pole, we can use a projection such as this:

```
var projection = d3.geoAzimuthalEquidistant()
    .scale(400)
    .center([0,0])
    .rotate([100,-90])
    .translate([width/2,height/2]);
```

This map used a 600x600 svg

Overall, this simplicity makes a azimuthal equidistant an easier projection to set, just use rotation. A typical projection will look like:

```
var projection = d3.geoProjection()
   .center([0,0])
   .rotate([-x,-y])
   .scale(k)
   .translate([width/2,height/2]);
```

Read D3 Projections online: https://riptutorial.com/d3-js/topic/9001/d3-projections

Chapter 5: Dispatching Events with d3.dispatch

Syntax

- d3.**dispatch** - create a custom event dispatcher.
- dispatch.**on** - register or unregister an event listener.
- dispatch.**copy** - create a copy of a dispatcher.
- dispatch.**call** - dispatch an event to registered listeners.
- dispatch.**apply** - dispatch an event to registered listeners.

Remarks

Dispatching is a convenient mechanism for separating concerns with loosely-coupled code: register named callbacks and then call them with arbitrary arguments. A variety of D3 components, such as d3-request, use this mechanism to emit events to listeners. Think of this like Node's EventEmitter, except every listener has a well-defined name so it's easy to remove or replace them.

Related Readings

- Dispatching Events by Mike Bostock
- d3.dispatch Documentation
- Dispatch Event in NPM

Examples

simple usage

```
var dispatch = d3.dispatch("statechange");

dispatch.on('statechange', function(e){ console.log(e) })

setTimeout(function(){dispatch.statechange('Hello, world!')}, 3000)
```

Read Dispatching Events with d3.dispatch online: https://riptutorial.com/d3-js/topic/3399/dispatching-events-with-d3-dispatch

Chapter 6: Making Robust, Responsive and Reusable (r3) for d3

Introduction

d3 is a powerful library for creating interactive charts; however, that power stems from users having to work at a lower level than other interactive libraries. Consequently many of the examples for d3 charts are designed to demonstrate how to produce a particular thing - e.g. whiskers for a box and whisker plot - while often hard coding in parameters thereby making the code inflexible. The purpose of this documentation is demonstrate how to make more reusable code to save time in the future.

Examples

Scatter Plot

This example contains over 1000 lines of code in total (too much to be embedded here). For that reason all code is accessible on http://blockbuilder.org/SumNeuron/956772481d4e625eec9a59fdb9fbe5b2 (alternatively hosted at https://bl.ocks.org/SumNeuron/956772481d4e625eec9a59fdb9fbe5b2). Note the bl.ocks.org uses iframe so to see the resizing you will need to click the button open (lower right corner of the iframe). Since there is a lot of code, it has been broken up into multiple files and the relevant code segment will be reference both by file name and line number. Please open this example as we go through it.

What makes a chart?

There are several core components that go into any complete chart; namely these include:

- title
- axes
- axes labels
- the data

There are other aspects that might be included depending on the chart - e.g. a chart legend. However, many of these elements can be circumvented with tooltip. For that reason there are interactive chart specific elements - e.g. buttons to switch between data.

Since our chart's content will be interactive, it would be appropriate for the chart itself to be dynamic - e.g. resize when the window's size changes. SVG is scalable, so you could just allow your chart to be scaled maintaining the current perspective. However, depending on the set perspective, the chart may become too small to be readable even if there is still sufficient space

for the chart (e.g. if the width is greater than the height). Therefore it may be preferable to just redraw the chart in the remaining size.

This example will cover how to dynamically calculate the placement of the buttons, title, axes, axes labels, as well as handle datasets of variant amounts of data

---

Set up

Configuration

Since we are aiming for code reuse, we should make a configuration file to contain global options for aspects of our chart. An example of such a configuration file is `charts_configuration.json`.

If we look at this file we can see I have included several elements which should have already have clear use for when we make our chart:

- files (stores the string for where our chart data is held)
- document_state (which button is currently selected for our chart)
- chart_ids (html ids for the charts we will make)
- svg (options for the svg, e.g. size)
- plot_attributes
    - title (set various font attributes)
    - tooltip (set various tooltip style properties)
    - axes (set various font attributes)
    - buttons (set various font and style attributes)
- plots
    - scatter (set various aspects of our scatter plot, e.g. point radius)
- colors (a specific color palette to use)

Helper functions

In addition to setting up these global aspects we need to define some helper functions. These can be found under `helpers.js`

- `ajax_json` (load json files either synchronously or asynchronously)
- `keys` (returns keys of the given object - equivalent to d3.keys())
- `parseNumber` (a general number parse in case we do not know what type or number is)
- `typeofNumber` (return the number type)

index.html

Lastly we should set up our html file. For the purpose of this example we will put our chart in a `section` tag where the `id` matches the id provided in the configuration file (line 37). Since percentages only work if they can be calculated from their parent member, we also include some

basic styling (lines 19-35)

Making our scatter plot

Let's open `make_scatter_chart.js`. Now let's pay close attention to line 2, where many of the most important variables are predefined:

- svg - d3 selection of the chart's svg
- chart_group - d3 selection of the group inside the svg in which the data will be placed
- canvas - core aspects of the svg extract for convenience
- margins - the margins we need to take into consideration
- maxi_draw_space the largest x and y values in which we can draw our data
- doc_state - the current state of the document if we are using buttons (in this example we are)

You may have noticed that we did not include the svg in the html. Therefore before we can do anything with our chart, we need to add the svg to`index.html` if it doesn't yet exist. This is achieved in the file `make_svg.js` by the function `make_chart_svg`. Looking at `make_svg.js` we see that we use the helper function `parseNumber` on the chart configuration for the svg width and height. If the number is a float, it makes the svg's width and height proportional its section's width and height. If the number is an integer, it will just set it to those integers.

Lines 6 - 11 tests to see - in effect - if this is the first call or not and sets the `chart_group` (and document state if it is the first call).

Line 14 - 15 extracts the currently selected data by the clicked button; line 16 sets `data_extent`. While d3 has a function for extracting the data extent, it is *my* preference to store the data extent in this variable.

Lines 27 - 38 contains the magic which sets up our chart by making the margins, the buttons, the title, and the axes. These are all dynamically determined and might seem a bit complex, so we will look at each in turn.

make_margins (in make_margins.js)

We can see that the margins object takes into account some space on the left, right, top and bottom of the chart (x.left, x.right, y.top, y.bottom respectively), the title, the buttons, and the axes.

We also see that the axes margins are updated in line 21.

Why do we do this? Well unless we specify the number of ticks, the tick labels the tick size, and the tick label font size, we could not calculate the size the axes need. Even then we still would have to guesstimate the space between the tick labels and the ticks. Therefore it is easier to make some dummy axes using our data, see how large the corresponding svg elements are, and then return the size.

We actually only need the width of the y axis and the height of the x axis, which is what is stored in

axes.y and axes.x.

With our default margins set, we then calculate the `max_drawing_space` (lines 29-34 in make_margins.js)

make_buttons (in make_buttons.js)

The function makes a group for all of the buttons, and then a group for each button, which in turn stores a circle and text element. Line 37 - 85 calculates the position of the buttons. It does this by seeing if the text right of each button's length is longer than space allowed for us to draw in (line 75). If so, it drops the button down a line and updates the margins.

make_title (in make_title.js)

`make_title` is similar to make_buttons in that it will automatically break up the title of your chart into multiple lines, and hyphenate if need be. It is a bit hacky since it doesn't have the sophistication of TeX's hyphenation scheme, but it works sufficiently well. If we need more lines than one the margins are updated.

With the buttons, title, and margins set, we can make our axes.

make_axes (in make_axes.js)

The logic of make_axes mirrors that for calculating the space needed by the dummy axes. Here, however, it adds transitions to change between axes.

Finally our scatter plot

With all of that set up done, we can finally make our scatter plot. Since our datasets may have a different number of points we need to take this into account and leverage d3's enter and exit events accordingly. Getting the number of already existing points is done in line 40. The if statement in line 45 - 59 adds more circle elements if we have more data, or transitions the extra elements to a corner and then removes them if there is too many.

Once we know we have the right number of elements, we can transition all of the remaining elements to their correct position (line 64)

Lastly we add tooltip in line 67 and 68. The tooltip function is in `make_tooltip.js`

Box and Whisker Chart

To show the value of making generalized functions like those in the previous example (make_title, make_axes, make_buttons, etc), consider this box and whisker chart: https://bl.ocks.org/SumNeuron/262e37e2f932cf4b693f241c52a410ff

While the code for making the boxes and whiskers is more intensive than just placing the points, we see that the same functions work perfectly.

Bar Chart

https://bl.ocks.org/SumNeuron/7989abb1749fc70b39f7b1e8dd192248

Read Making Robust, Responsive and Reusable (r3) for d3 online: https://riptutorial.com/d3-js/topic/9849/making-robust--responsive-and-reusable--r3--for-d3

Chapter 7: On Events

Syntax

- .on('mouseover', function)
- .on('mouseout', function)
- .on('click', function)
- .on('mouseenter', function)
- .on('mouseleave', function)

Remarks

For a more in depth example where custom events are defined refer here.

Examples

Attaching basic events on selections

Often times you will want to have events for your objects.

```
function spanOver(d,i){
  var span = d3.select(this);
  span.classed("spanOver",true);
}

function spanOut(d,i){
  var span = d3.select(this);
  span.classed("spanOver", false);
}

var div = d3.select('#divID');

div.selectAll('span')
  .on('mouseover' spanOver)
  .on('mouseout' spanOut)
```

This example will add the class `spanOver` when hovering over a span inside the div with the id `divID` and remove it when the mouse exits the span.

By default d3 will pass the datum of the current span and the index. It's really handy that `this`'s context is the current object as well so that we can do operations on it, like add or remove classes.

You can also just use an anonymous function on the event as well.

```
div.selectAll('span')
  .on('click', function(d,i){ console.log(d); });
```

Data elements can also be added to the current selected object as well.

```
div.selectAll('path')
  .on('click', clickPath);

function clickPath(d,i) {
  if(!d.active) {
    d.active = true;
    d3.select(this).classed("active", true);
  }
  else {
    d.active = false;
    d3.select(this).classed("active", false);
  }
}
```

In this example active is not defined on the selection before the click event is fired. If you were to go back over the path selection though all clicked objects would contain the `active` key.

Remove event listener

d3.js doesn't have a `.off()` method to detatch existent event listeners. In order to remove an event handler, you have to set it to `null`:

```
d3.select('span').on('click', null)
```

Read On Events online: https://riptutorial.com/d3-js/topic/2722/on-events

Chapter 8: Selections

Syntax

- d3.**select**(selector)
- d3.**selectAll**(selector)
- *selection*.**select**(selector)
- *selection*.**selectAll**(selector)
- *selection*.**filter**(filter)
- *selection*.**merge**(other)

Remarks

Related Readings:

- How Selections Work - Mike Bostock
- d3-selection README

Examples

Basic selection and modifications

If you are familiar with jQuery and Sizzle syntax, d3 selections should not be much different. d3 mimics the W3C Selectors API to make interacting with elements easier.

For a basic example, to select all `<p>` and add a change to each of them:

```
d3.selectAll('p')
  .attr('class','textClass')
  .style('color', 'white');
```

In a nutshell this is relatively the same as doing in jQuery

```
$('p')
  .attr('class','textClass')
  .css('color, 'white')
```

Generally you will start with a single select to your container div to add an SVG element which will be assigned to a variable (most commonly called svg).

```
var svg = d3.select('#divID').append('svg');
```

From here we can call on `svg` to do our sub-selections of multiple objects (even if they don't yet exist).

```
svg.selectAll('path')
```

Different selectors

You can select elements with different selectors :

- by tag : `"div"`
- by class : `".class"`
- by id : `"#id"`
- by attribute : `"[color=blue]"`
- multiple selectors (OR): `"div1, div2, class1"`
- multiple selectors (AND): `"div1 div2 class1"`

Simple data bounded selection

```
var myData = [
    { name: "test1", value: "ok" },
    { name: "test2", value: "nok" }
]

// We have to select elements (here div.samples)
// and assign data. The second parameter of data() is really important,
// it will determine the "key" to identify part of data (datum) attached to an
// element.
var mySelection = d3.select(document.body).selectAll("div.samples") // <- a selection
                    .data(myData, function(d){ return d.name; }); // <- data binding

// The "update" state is when a datum of data array has already
// an existing element associated.
mySelection.attr("class", "samples update")

// A datum is in the "enter" state when it's not assigned
// to an existing element (based on the key param of data())
// i.e. new elements in data array with a new key (here "name")
mySelection.enter().append("div")
    .attr("class", "samples enter")
    .text(function(d){ return d.name; });

// The "exit" state is when the bounded datum of an element
// is not in the data array
// i.e. removal of a row (or modifying "name" attribute)
// if we change "test1" to "test3", "test1" bounded
//          element will figure in exit() selection
// "test3" bounded element will be created in the enter() selection
mySelection.exit().attr("class", "samples remove");
```

The role of placeholders in "enter" selections

What is an enter selection?

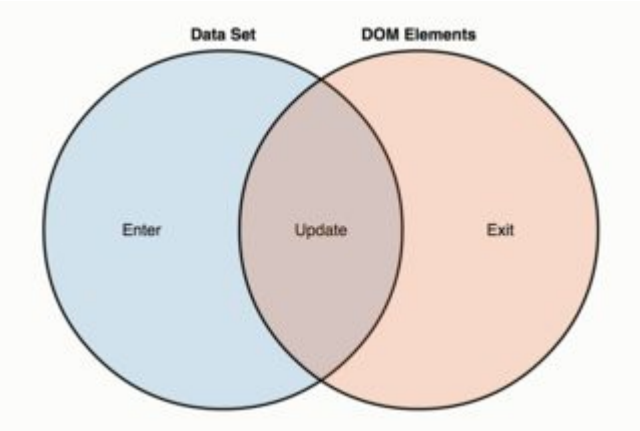
In D3.js, when one binds data to DOM elements, three situations are possible:

1. The number of elements and the number of data points are the same;
2. There are more elements than data points;

3. There are more data points than elements;

In the situation #3, all the data points without a corresponding DOM element belong to the *enter* selection. Thus, In D3.js, *enter* selections are selections that, after joining elements to the data, contains all the data that don't match any DOM element. If we use an `append` function in an *enter* selection, D3 will create new elements binding that data for us.

This is a Venn diagram explaining the possible situations regarding number of data points/number of DOM elements:



As we can see, the *enter* selection is the blue area at the left: data points without corresponding DOM elements.

The structure of the enter selection

Typically, an *enter* selection has these 4 steps:

1. `selectAll`: Select elements in the DOM;
2. `data`: Counts and parses the data;
3. `enter`: Comparing the selection with the data, creates new elements;
4. `append`: Append the actual elements in the DOM;

This is a very basic example (look at the 4 steps in the `var divs`):

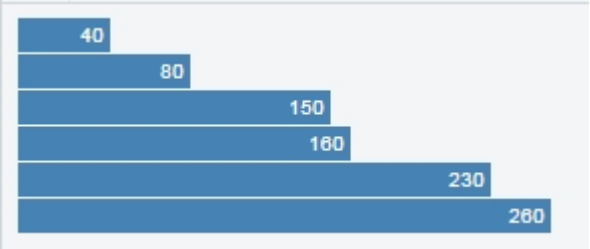
```
var data = [40, 80, 150, 160, 230, 260];

var body = d3.select("body");

var divs = body.selectAll("div")
    .data(data)
    .enter()
    .append("div");

divs.style("width", function(d) { return d + "px"; })
    .attr("class", "divchart")
    .text(function(d) { return d; });
```

And this is the result (jsfiddle here):



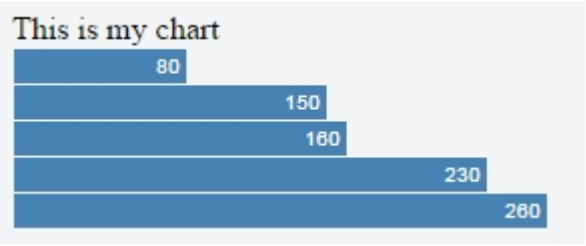
Notice that, in this case, we used `selectAll("div")` as the first line in our "enter" selection variable. We have a dataset with 6 values, and D3 created 6 divs for us.

The role of placeholders

But suppose that we already have a div in our document, something like `<div>This is my chart</div>` at the top. In that case, when we write:

```
body.selectAll("div")
```

we are selecting that existent div. So, our enter selection will have only 5 datum without matching elements. For instance, in this jsfiddle, where there is already a div in the HTML ("This is my chart"), this will be the outcome:



We don't see the value "40" anymore: our first "bar" disappeared, and the reason for that is that our "enter" selection now has only 5 elements.

What we have to understand here is that in the first line of our enter selection variable, `selectAll("div")`, those divs are just *placeholders*. We don't have to select all the `divs` if we are appending `divs`, or all the `circle` if we are appending `circle`. We can select different things. And, if we don't plan to have an "update" or an "exit" selection, we can select *anything*:

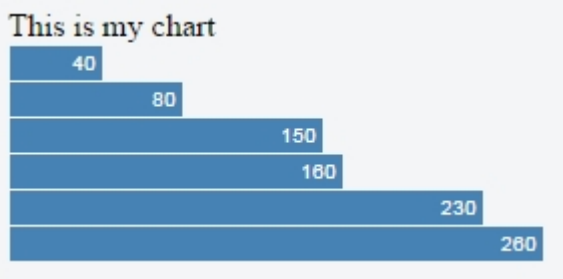
```
var divs = body.selectAll(".foo")//this class doesn't exist, and never will!
    .data(data)
    .enter()
    .append("div");
```

Doing this way, we are selecting all the ".foo". Here, "foo" is a class that not only doesn't exist, but also it's never created anywhere else in the code! But it doesn't matter, this is only a placeholder. The logic is this:

> If in your "enter" selection you select something that doesn't exist, your "enter" selection will *always* contain all your data.

Now, selecting `.foo`, our "enter" selection have 6 elements, even if we already have a div in the

document:



And here is the corresponding jsfiddle.

Selecting `null`

By far, the best way to guarantee that you are selecting nothing is selecting `null`. Not only that, but this alternative is way faster than any other.

Thus, for an enter selection, just do:

```
selection.selectAll(null)
    .data(data)
    .enter()
    .append(element);
```

Here is a demo fiddle: https://jsfiddle.net/gerardofurtado/th6s160p/

Conclusion

When dealing with "enter" selections, take extra care to do not select something that already exists. You can use anything in your `selectAll`, even things that don't exist and will never exist (if you don't plan to have an "update" or an "exit" selection).

The code in the examples is based on this code by Mike Bostock:
https://bl.ocks.org/mbostock/7322386

Using "this" with an arrow function

Most of functions in D3.js accept an anonymous function as an argument. The common examples are `.attr`, `.style`, `.text`, `.on` and `.data`, but the list is way bigger than that.

In such cases, the anonymous function is evaluated for each selected element, in order, being passed:

1. The current datum (`d`)
2. The current index (`i`)
3. The current group (`nodes`)
4. `this` as the current DOM element.

The datum, the index and the current group are passed as arguments, the famous first, second and third argument in D3.js (whose parameters are traditionally named `d`, `i` and `p` in D3 v3.x). For

using `this`, however, one doesn't need to use any argument:

```
.on("mouseover", function(){
    d3.select(this);
});
```

The above code will select `this` when the mouse is over the element. Check it working in this fiddle: https://jsfiddle.net/y5fwgopx/

The arrow function

As a new ES6 syntax, an arrow function has a shorter syntax when compared to function expression. However, for a D3 programmer who uses `this` constantly, there is a pitfall: an arrow function doesn't create its own `this` context. That means that, in an arrow function, `this` has its original meaning from the enclosing context.

This can be useful in several circumstances, but it is a problem for a coder accustomed to use `this` in D3. For instance, using the same example in the fiddle above, this will not work:

```
.on("mouseover", ()=>{
    d3.select(this);
});
```

If you doubt it, here is the fiddle: https://jsfiddle.net/tfxLsv9u/

Well, that's not a big problem: one can simply use a regular, old fashioned function expression when needed. But what if you want to write all your code using arrow functions? Is it possible to have a code with arrow functions **and** still properly use `this` in D3?

The second and third arguments combined

The answer is **yes**, because `this` is the same of `nodes[i]`. The hint is actually present all over the D3 API, when it describes this:

> ...with `this` as the current DOM element (`nodes[i]`)

The explanation is simple: since `nodes` is the current group of elements in the DOM and `i` is the index of each element, `nodes[i]` refer to the current DOM element itself. That is, `this`.

Therefore, one can use:

```
.on("mouseover", (d, i, nodes) => {
    d3.select(nodes[i]);
});
```

And here is the corresponding fiddle: https://jsfiddle.net/2p2ux38s/

Read Selections online: https://riptutorial.com/d3-js/topic/2135/selections

Chapter 9: SVG charts using D3 js

Examples

Using D3 js for creating SVG elements

Although D3 is not specific for handling SVG elements, it is widely used for creating and manipulating complex SVG based data visualizations. D3 provides many powerful methods which helps to create various geometrical SVG structures with ease.

It is recommended to understand basic concepts of SVG specifications first, then using extensive D3 js examples to create visualizations.

D3 js examples

Basics of SVG

Read SVG charts using D3 js online: https://riptutorial.com/d3-js/topic/1538/svg-charts-using-d3-js

Chapter 10: update pattern

Syntax

- selection.enter()
- selection.exit()
- selection.merge()

Examples

Updating the data: a basic example of enter, update and exit selections

Creating a chart that displays a static dataset is relatively simple. For example, if we have this array of objects as the data:

```
var data = [
    {title: "A", value: 53},
    {title: "B", value: 12},
    {title: "C", value: 91},
    {title: "D", value: 24},
    {title: "E", value: 59}
];
```

We can create a bar chart where each bar represents a measure, named "title", and its width represents the value of that measure. As this dataset doesn't change, our bar chart has only an "enter" selection:

```
var bars = svg.selectAll(".bars")
    .data(data);

bars.enter()
    .append("rect")
    .attr("class", "bars")
    .attr("x", xScale(0))
    .attr("y", function(d){ return yScale(d.title)})
    .attr("width", 0)
    .attr("height", yScale.bandwidth())
    .transition()
    .duration(1000)
    .delay(function(d,i){ return i*200})
    .attr("width", function(d){ return xScale(d.value) - margin.left});
```

Here, we are setting the width of each bar to 0 and, after the transition, to its final value.

This enter selection, alone, is enough to create our chart, that you can see in this fiddle.

But what if my data changes?

In this case, we have to dynamically change our chart. The best way to do it is creating an "enter", an "update" and an "exit" selections. But, before that, we have to do some changes in the code.

First, we'll move the changing parts inside a function named `draw()`:

```
function draw(){
    //changing parts
};
```

These "changing parts" include:

1. The enter, update and exit selections;
2. The domain of each scale;
3. The transition of the axis;

Inside that `draw()` function, we call another function, that creates our data. Here, it's just a function that returns an array of 5 objects, choosing randomly 5 letters out of 10 (sorting alphabetically) and, for each one, a value between 0 and 99:

```
function getData(){
    var title = "ABCDEFGHIJ".split("");
    var data = [];
    for(var i = 0; i < 5; i++){
        var index = Math.floor(Math.random()*title.length);
        data.push({title: title[index],
            value: Math.floor(Math.random()*100)});
        title.splice(index,1);
    }
    data = data.sort(function(a,b){ return d3.ascending(a.title,b.title)});
    return data;
};
```

And now, let's move to our selections. But before that, a word of caution: to maintain what we call *object constancy*, we have to specify a key function as the second argument to selection.data:

```
var bars = svg.selectAll(".bars")
    .data(data, function(d){ return d.title});
```

Without that, our bars won't transition smoothly, and it'd be difficult to follow the changes in the axis (you can see that removing the second argument in the fiddle below).

So, after correctly setting our `var bars`, we can deal with our selections. This is the exit selection:

```
bars.exit()
    .transition()
    .duration(1000)
    .attr("width", 0)
    .remove();
```

And these are the enter and the update selections (in D3 v4.x, the update selection is merged with the enter selection using `merge`):

```
bars.enter()//this is the enter selection
    .append("rect")
    .attr("class", "bars")
```

```
    .attr("x", xScale(0) + 1)
    .attr("y", function(d){ return yScale(d.title)})
    .attr("width", 0)
    .attr("height", yScale.bandwidth())
    .attr("fill", function(d){ return color(letters.indexOf(d.title)+1)})
    .merge(bars)//and from now on, both the enter and the update selections
    .transition()
    .duration(1000)
    .delay(1000)
    .attr("y", function(d){ return yScale(d.title)})
    .attr("width", function(d){ return xScale(d.value) - margin.left});
```

Finally, we call the `draw()` function every time the button is clicked:

```
d3.select("#myButton").on("click", draw);
```

And this is the fiddle showing all these 3 selections in action.

Merging selections

The update pattern in D3 version 3

A correct understanding of how the “enter”, “update” and “exit” selections work is fundamental for properly changing the dataviz using D3.

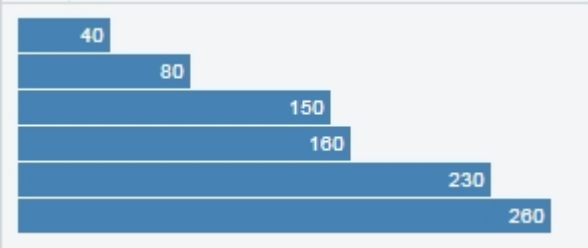
Since D3 version 3 (actually, since version 2), this snippet could set the transitions for both “enter” and “update” selections (live demo here):

```
var divs = body.selectAll("div")
    .data(data);//binding the data

divs.enter()//enter selection
    .append("div")
    .style("width", "0px");

divs.transition()
    .duration(1000)
    .style("width", function(d) { return d + "px"; })
    .attr("class", "divchart")
    .text(function(d) { return d; });
```

Giving this result after the transition:



But what happen with the exactly same code if we use D3 version 4? You can see it in this live demo: *nothing*!

Why?

Changes in the update pattern in D3 version 4

Let's check the code. First, we have `divs`. This selection binds the data to the `<div>`.

```
var divs = body.selectAll("div")
    .data(data);
```

Then, we have `divs.enter()`, which is a selection that contains all the data with unmatched elements. This selection contains all the divs in the first time we call the function `draw`, and we set their widths to zero.

```
divs.enter()
    .append("div")
    .style("width", "0px");
```

Then we have `divs.transition()`, and here we have the interesting behaviour: In D3 version 3, `divs.transition()` makes all the `<div>` in the "enter" selection changing to their final width. But that makes no sense! `divs` doesn't contain the "enter" selection, and should not modify any DOM element.

There is a reason why this strange behaviour have been introduced in D3 version 2 and 3 (source here), and it was "corrected" in D3 version 4. Thus, in the live demo above, nothing happens, and this is expected! Furthermore, if you click the button, all the previous six bars appear, because they are now in the "update" selection, not anymore in the "enter" selection.

For the transition acting over the "enter" selection, we have to create separate variables or, an easier approach, merging the selections:

```
divs.enter()//enter selection
    .append("div")
    .style("width", "0px")
    .merge(divs)//from now on, enter + update selections
    .transition().duration(1000).style("width", function(d) { return d + "px"; })
    .attr("class", "divchart")
    .text(function(d) { return d; });
```

Now, we merged the "enter" and the "update" selections. See how it works in this live demo.

Read update pattern online: https://riptutorial.com/d3-js/topic/5749/update-pattern

Chapter 11: Using D3 with JSON and CSV

Syntax

- d3.csv(url[[, row], callback])
- d3.tsv(url[[, row], callback])
- d3.html(url[, callback])
- d3.json(url[, callback])
- d3.text(url[, callback])
- d3.xml(url[, callback])

Examples

Loading data from CSV files

There are several ways of getting the data that you will bind to the DOM elements. The simpler one is having your data in your script as an array...

```
var data = [ ... ];
```

But **D3.js** allows us to load data from an external file. In this example, we will see how to properly load and deal with data from an CSV file.

CSV files are *comma-separated values*. In this kind of file, each line is a data record, each record consisting of one or more fields, separated by commas. Its important to know that the function we are about to use, `d3.csv`, uses the first line of the CSV as the **header**, that is, the line that contains the names of the fields.

So, consider this CSV, named "data.csv":

```
city,population,area
New York,3400,210
Melbourne,1200,350
Tokyo,5200,125
Paris,800,70
```

To load "data.csv", we use the function `d3.csv`. To make it easier, suppose that "data.csv" is in the same directory of our script, and its relative path is simply "data.csv". So, we write:

```
d3.csv("data.csv", function(data){
    //code dealing with data here
});
```

Notice that, in the callback, we used `data` as an argument. That's a common practice in D3, but you can use any other name.

What `d3.csv` does with our CSV? It converts the CSV in an array of objects. If, for instance, we `console.log` our data:

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

This is what we are going to see:

```
[
    {
        "city": "New York",
        "population": "3400",
        "area": "210"
    },{
        "city": "Melbourne",
        "population": "1200",
        "area": "350"
    },{
        "city": "Tokyo",
        "population": "5200",
        "area": "125"
    },{
        "city": "Paris",
        "population": "800",
        "area": "70"
    }
]
```

Now we can bind that data to our DOM elements.

Notice that, in this example, `population` and `area` are strings. But, probably, you want to deal with them as numbers. You can change them in a function inside the callback (as a `forEach`), but in `d3.csv` you can use an "accessor" function:

```
d3.csv("data.csv", conversor, function(data){
    //code here
});

function conversor(d){
    d.population = +d.population;
    d.area = +d.area;
    return d;
}
```

You can also use accessors in `d3.tsv`, but not in `d3.json`.

Note: `d3.csv` is an asynchronous function, meaning that the code after it will execute immediately, even before the CSV file is loaded. So, special attention for using your `data` inside the callback.

One or two parameters in callback—error handling in d3.request()

When using `d3.request()` or one of the convenience constructors (d3.json, d3.csv, d3.tsv, d3.html and d3.xml) there are many sources for error. There may be problems with the issued request or

its response due to network errors, or the parsing might fail if the content is not well-formed.

Within the callbacks passed to any of the above mentioned methods it is, therefore, desirable to implement some error handling. For this purpose the callbacks may accept two arguments, the first being the error, if any, the second being the data. If any error occurred during loading or parsing information about the error will be passed as the first argument `error` with the `data` being `null`.

```
d3.json{"some_file.json", function(error, data) {
  if (error) throw error;   // Exceptions handling
  // Further processing if successful
});
```

You are, whatsoever, not obliged to provide two parameters. It is perfectly fine to use the request methods with a callback featuring only one parameter. To handle these kinds of callbacks there is a private function `fixCallback()` in request.js, which adjusts the way information is passed to the method's single argument.

```
function fixCallback(callback) {
  return function(error, xhr) {
    callback(error == null ? xhr : null);
  };
}
```

This will be invoked by D3 for all callbacks having only one parameter, which by definition is the data.

No matter how many parameters are supplied to the request method's callback, the rule for the `data` parameter is:

- if the request fails, `data` will be `null`
- if the request succeeds, `data` will contain the loaded (and parsed) contents

The only difference between the one-parameter vs. the two-parameter version is in the way information is provided about the error which may occur. If the `error` parameter is omitted, the method will fail silently leaving `data` as `null`. If, on the other hand, the callback is defined to have two arguments, information about an error during loading or parsing will be passed to the first parameter enable you to handle it appropriately.

The following four calls to `d3.json` demonstrate the scenarios possible for existing/non-existing files vs. one paramter/two parameter callbacks:

```
// FAIL: resource not available or content not parsable
// error contains information about the error
// data will be null because of the error
d3.json("non_existing_file.json", function(error, data) {
  console.log("Fail, 2 parameters: ", error, data);
});

// FAIL: resource not available or content not parsable
// no information about the error
```

```
// data will be null because of the error
d3.csv("non_existing_file.json", function(data) {
  console.log("Fail, 1 parameter: ", data);
});

// OK: resource loaded successfully
// error is null
// data contains the JSON loaded from the resource
d3.json("existing_file.json", function(error, data) {
  console.log("OK, 2 parameters: ", error, data);
});

// OK: resource loaded successfully
// no information about the error; this fails silently on error
// data contains the JSON loaded from the resource
d3.json("existing_file.json", function(data) {
  console.log("OK, 1 parameter: ", data);
});
```

Read Using D3 with JSON and CSV online: https://riptutorial.com/d3-js/topic/5201/using-d3-with-json-and-csv

Chapter 12: Using D3 with other frameworks

Examples

D3.js component with ReactJS

This example is based on a blog post by Nicolas Hery. It utilizes ES6 classes and ReactJS's lifecycle methods to keep the D3 component updated

---

d3_react.html

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>Hello, d3React!</title>
  <style>
    .d3Component {
      width: 720px;
      height: 120px;
    }
  </style>
</head>
<script src="https://fb.me/react-15.2.1.min.js"></script>
<script src="https://fb.me/react-dom-15.2.1.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/babel-
core/5.8.34/browser.min.js"></script>
<script src="https://d3js.org/d3.v4.min.js"></script>

<body>
  <div id="app" />
  <script type="text/babel" src="d3_react.js"></script>
</body>

</html>
```

d3_react.js

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      d3React: new d3React()
    };
    this.getd3ReactState = this.getd3ReactState.bind(this);
  }

  getd3ReactState() {
    // Using props and state, calculate the d3React state
    return ({
```

```
      data: {
        x: 0,
        y: 0,
        width: 42,
        height: 17,
        fill: 'red'
      }
    });
  }

  componentDidMount() {
    var props = {
      width: this._d3Div.clientWidth,
      height: this._d3Div.clientHeight
    };
    var state = this.getd3ReactState();
    this.state.d3React.create(this._d3Div, props, state);
  }

  componentDidUpdate(prevProps, prevState) {
    var state = this.getd3ReactState();
    this.state.d3React.update(this._d3Div, state);
  }

  componentWillUnmount() {
    this.state.d3React.destroy(this._d3Div);
  }

  render() {
    return (
      <div>
        <h1>{this.props.message}</h1>
        <div className="d3Component" ref={(component) => { this._d3Div = component; } } />
      </div>
    );
  }
}

class d3React {
  constructor() {
    this.create = this.create.bind(this);
    this.update = this.update.bind(this);
    this.destroy  = this.destroy.bind(this);
    this._drawComponent = this._drawComponent.bind(this);
  }

  create(element, props, state) {
    console.log('d3React create');
    var svg = d3.select(element).append('svg')
      .attr('width', props.width)
      .attr('height', props.height);

    this.update(element, state);
  }

  update(element, state) {
    console.log('d3React update');
    this._drawComponent(element, state.data);
  }

  destroy(element) {
```

```
    console.log('d3React destroy');
  }

  _drawComponent(element, data) {
    // perform all drawing on the element here
    var svg = d3.select(element).select('svg');

    svg.append('rect')
      .attr('x', data.x)
      .attr('y', data.y)
      .attr('width', data.width)
      .attr('height', data.height)
      .attr('fill', data.fill);
  }
}

ReactDOM.render(<App message="Hello, D3.js and React!"/>, document.getElementById('app'));
```

Place the contents of `d3_react.html` and `d3_react.js` in the same directory and navigate a web browser to the d3React.html file. If all goes well, you will see a header stating `Hello, D3.js and React!` rendered from the React component and a red rectangle below from the custom D3 component.

React uses refs to "reach out" to the component instance. The lifecycle methods of the `d3React` class require this ref to append, modify, and remove DOM elements. The `d3React` class can be extended to create more custom components and inserted anywhere a `div.d3Component` is created by React.

D3js with Angular

Using D3js with Angular can open up new fronts of possibilities such as live updation of charts as soon as data is updated. We can encapsulate complete chart functionality within an Angular directive, which makes it easily reusable.

index.html >>

```
<!DOCTYPE html>
<html ng-app="myApp">
<head>
  <script src="https://d3js.org/d3.v4.min.js"></script>
  <script data-require="angular.js@1.4.1" data-semver="1.4.1"
src="https://code.angularjs.org/1.4.1/angular.js"></script>
  <script src="app.js"></script>
  <script src="bar-chart.js"></script>
</head>

<body>

  <div ng-controller="MyCtrl">
    <!-- reusable d3js bar-chart directive, data is sent using isolated scope -->
    <bar-chart data="data"></bar-chart>
  </div>

</body>
</html>
```

We can pass the data to the chart using controller, and watch for any changes in the data to enable live updation of chart in the directive:

app.js >>

```
angular.module('myApp', [])
  .controller('MyCtrl', function($scope) {
    $scope.data = [50, 40, 30];
    $scope.$watch('data', function(newVal, oldVal) {
      $scope.data = newVal;
    }, true);
  });
```

Finally, the directive definition. The code we write to create and manipulate the chart will sit in the link function of the directive.

Note that we have put a scope.$watch in the directive too, to update as soon as the controller passes new data. We are re-assigning new data to our data variable if there is any data change and then calling the repaintChart() function, which performs the chart re-rendering.

bar-chart.js >>

```
angular.module('myApp').directive('barChart', function($window) {
  return {
    restrict: 'E',
    replace: true,
    scope: {
      data: '='
    },
    template: '<div id="bar-chart"></div>',
    link: function(scope, element, attrs, fn) {

      var data = scope.data;
      var d3 = $window.d3;
      var rawSvg = element;

      var colors = d3.scale.category10();

      var canvas = d3.select(rawSvg[0])
        .append('svg')
        .attr("width", 300)
        .attr("height", 150);

      // watching for any changes in the data
      // if new data is detected, the chart repaint code is run
      scope.$watch('data', function(newVal, oldVal) {
        data = newVal;
        repaintChart();
      }, true);

      var xscale = d3.scale.linear()
        .domain([0, 100])
        .range([0, 240]);

      var yscale = d3.scale.linear()
        .domain([0, data.length])
        .range([0, 120]);
```

```
      var bar = canvas.append('g')
        .attr("id", "bar-group")
        .attr("transform", "translate(10,20)")
        .selectAll('rect')
        .data(data)
        .enter()
        .append('rect')
        .attr("class", "bar")
        .attr("height", 15)
        .attr("x", 0)
        .attr("y", function(d, i) {
          return yscale(i);
        })
        .style("fill", function(d, i) {
          return colors(i);
        })
        .attr("width", function(d) {
          return xscale(d);
        });

      // changing the bar widths according to the changes in data
      function repaintChart() {
        canvas.selectAll('rect')
          .data(data)
          .transition()
          .duration(800)
          .attr("width", function(d) {
            return xscale(d);
          })
      }
    }
  }
});
```

Here is the working JSFiddle.

D3.js chart with Angular v1

HTML:

```
<div ng-app="myApp" ng-controller="Controller">
    <some-chart data="data"></some-chart>
</div>
```

Javascript:

```
angular.module('myApp', [])
.directive('someChart', function() {
    return {
        restrict: 'E',
        scope: {data: '=data'},
        link: function (scope, element, attrs) {
        var chartElement = d3.select(element[0]);
            // here you have scope.data and chartElement
            // so you may do what you want
        }
```

```
    };
});

function Controller($scope) {
    $scope.data = [1,2,3,4,5]; // useful data
}
```

Read Using D3 with other frameworks online: https://riptutorial.com/d3-js/topic/3733/using-d3-with-other-frameworks

Credits

S. No	Chapters	Contributors
1	Getting started with d3.js	4444, Adam, altocumulus, Arthur Tarasov, Community, davinov, Fawzan, Gerardo Furtado, Ian H, jmdarling, kashesandr, Marek Skiba, Maximilian Kohl, Rachel Gallen, Ram Visagan, RamenChef, Ruben Helsloot, TheMcMurder, Tim B, winseybash
2	Approaches to create responsive d3.js charts	Ian H
3	Core SVG concepts used in D3.js visualization	fengshuo, Gerardo Furtado
4	D3 Projections	Andrew Reid
5	Dispatching Events with d3.dispatch	aug, kashesandr, Ram Visagan
6	Making Robust, Responsive and Reusable (r3) for d3	SumNeuron
7	On Events	aug, Ian H, Kemi, Tim B
8	Selections	Ashitaka, aug, Carl Manaster, Gerardo Furtado, Ian, Ian H, JulCh, Tim B
9	SVG charts using D3 js	rajarshig
10	update pattern	Gerardo Furtado
11	Using D3 with JSON and CSV	altocumulus, Gerardo Furtado
12	Using D3 with other frameworks	Adam, kashesandr, Rishabh