EXERCISE: Introduction to the D3 JavaScript Library for Interactive Graphs and Maps

Barend Köbben

Version 3.1
October 14, 2016

Contents

Key points

In this exercise you will discover the basics of using the JavaScript D3 library to build dynamic web pages. You will learn:

1. the principles of using the D3 library
2. D3 selections for DOM access
3. D3 data binding to change the DOM based on data
4. loading external data
5. using D3 scales and axes
6. using D3 for maps

! → In many cases during these exercises, you will have to type code (HTML, Python, JavaScript or MapServer configuration code). It's very easy to make mistakes in such code. HTML code and MapServer map files are not case–sensitive, but JavaScript is: the variable `mySomething` is different than the variable `MySomeThing`! Also take care of the *special character* (→) in the code examples we provide:

→ this character means the line should be typed **without interruption**, the move to the next line in our example is only because it would not fit otherwise. So do **not** type a `return` or `enter` in this place!

Typing the code of longer listings is usually not necessary: You can copy the code from the `filefragments` folder in the exercise data. In this folder you will find all code fragments from the exercises in text files with the same names as the listing. Do **not copy** from the PDF exercise description!

There are several software tools that can help you: Use a text–editor that is more intelligent than your basic text editor, e.g. on MacOSX and Linux use *TextWrangler* or *medit*, on Windows *Notepad++*.This will provide you with line numbers, automatic highlighting of recognised HTML and JavaScript keywords, etcetera.

Use a modern web–browser: a recent FireFox, Chrome or Opera, or Internet Explore version 9 or higher. These are HTML5 compatible and have built-in web developer tools. This provides error messages, code views and a JavaScript console, network traffic monitoring, etc...

For students that intend to do more work on developing interactive web applications (using HTML5, CSS, Javascript, etcetera) there is an Integrated Development Environment available in the ITC software Manager called *Webstorm*.

1 Introduction

The javascript library `D3.js` was created to fill a need for web-accessible, sophisticated data visualisation. Its creator is Mike Bostock, who started it as a research project for his PhD work and is now its main developer as well as a Graphics Editor for the New York Times.

Until recently, you couldn't build high-performance, rich internet applications in the browser unless you built them in Flash or as a Java applet. Flash and Java are still around on the internet, and especially for internal web apps, for this reason. D3.js provides the same performance, but integrated into web standards and the Document Object Model (DOM) at the core of HTML. D3 provides developers with the ability to create rich interactive and animated content based on data and tie that content to existing web page elements. It gives you the tools to create high-performance data dashboards and sophisticated data visualization, and to dynamically update traditional web content.

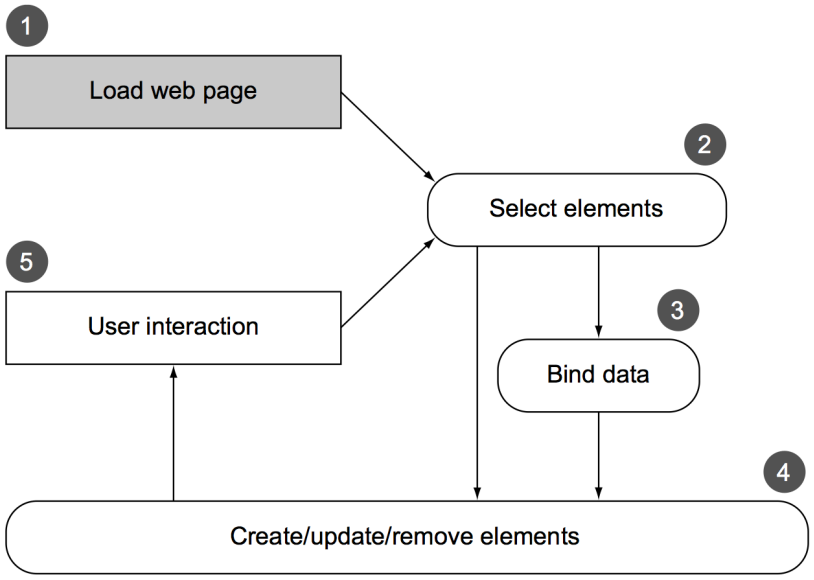


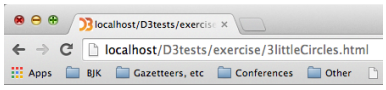
Figure 1: Principle of the use of the D3 library in HTML pages (from Meeks, 2015).

In figure 1 you see the common use of D3: A page is typically built in such a way that the page loads with styles, data, and content as defined in traditional HTML development [1] with its initial display using D3 selections of HTML elements [2], either with data-binding [3] or without it. The selections and data-binding can be used to modify the structure and appearance of the page [4]. The changes

in structure prompt user interaction [5], which causes new selections with and without data- binding to further alter the page. Step 1 is shown differently because it only happens once (when you load the page), whereas every other step may happen multiple times, depending on user interaction.

Learning D3 is not all that easy, because it is **not** a high-level charting or mapping library, instead it is a low-level API to bind data to graphics elements and transform them based on that data. For example, D3 doesn't have one single function to create a pie chart. Rather, it has a function that processes your dataset with the necessary angles so that, if you pass the dataset to D3's arc function, you get the drawing code necessary to represent those angles. And you need to use yet another function to create the paths necessary for that code. It's a much longer process than using dedicated charting libraries, but the D3 process is also its strength. Although other charting libraries conveniently allow you to make line graphs and pie charts, they quickly break down when you want to make something more complicated than that. D3 allows you to build whatever data-driven graphics and interactivity you can think of.

2 Three little circles – a first attempt at D3



● ● ●

Task 1 : Create an HTML webpage called `3LittleCircles.html`, and make sure its contents match the code in listing 1. As explained in page 1, you can use the text file in the `filefragments` folder to avoid having to type it all.

Open the file in a web browser. You will see three circles. We will use these three SVG circles to show you how **selections** and **data-binding** in D3 work. •

Listing 1: filefragments/3LittleCircles.html.txt

```
<!DOCTYPE html>
<html>
<head>
    <script type="text/javascript" src="lib/d3.v4.min.js">
    </script>
</head>
<body>
<svg id="svg" height="100%" width="100%">
    <circle cx="40" cy="60" r="10"></circle>
    <circle cx="80" cy="60" r="10"></circle>
    <circle cx="120" cy="60" r="10"></circle>
```

```
</svg>
<script>
    // our D3 code goes here...
</script>
</body></html>
```

There are various ways to run D3 code. A very easy way, with immediate results and feedback, is to interactively type the code in the Web browsers' javascript console (as demonstrated in class). The disadvantage is that your changes are not saved.

Therefore, in this exercise you will type the code within the (still empty) `script` tag in the html file. All the code fragments we show in the coming pages should be entered in there, and then tried out by saving the file and reloading it in the browser. We will **not** always tell you the exact place where to enter the code, this is for you to figure out...!

2.1 Using SVG for drawing

The D3 library can be used to manipulate **any** element within the webpage, through the Domain Object Model (DOM). In these exercise, we use it to draw graphics, therefore we need to use the graphics elements within a DOM. We use the *Scalable Vector Graphics* language (SVG) for this.

The SVG element can be thought of as a viewport — things within the SVG Viewport's dimensions are visible, things outside of the dimensions are not. The SVG element dimensions are defined using the attribute value pairs of "height" and "width". Note that the top-left corner is 0,0, the x-axis is drawn left-to-right, but the y-axis is drawn top-to-bottom.

2.2 Selecting Elements for script access to the DOM

The `d3.selectAll` method takes a selector string, such as "circle", and returns a selection representing all elements that match the selector:

```
var circle = d3.selectAll("circle");
```

The selector string will select all elements in the HTML DOM that match it. There are several types of **selector strings**, that select:

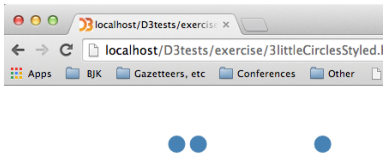
- by **element** name; As in the case above, where we select all elements of type "circle": `d3.selectAll("div");`
- by CSS **class**: `d3.selectAll(".shaded");`
- by HTML **id**: `d3.selectAll("#myID");`

- by combinations, e.g. selecting all `div` elements of class `shaded`: `d3.selectAll("div.shaded");`

You can use `selectAll()` to select all elements or simply `select()` to select only the first one encountered. Once we have a selection, we can make various changes to the selected elements. For example, we might change the fill color using `selection.style` and the radius using `selection.attr`:

```
circle.style("fill", "steelblue");
circle.attr("r", 30);
```

If you use the three lines of code shown above, the D3 library sets styles and attributes for **all** selected elements to the same values. If you refresh the file in the browser, you will see the changes, and if you would look at the HTML DOM in its active state (e.g. by using the browser developer tools), you will see the SVG part has been changed to include the style elements (`style="fill:steelblue;"`).



We can also set all values in a selection seperately (on a per-element basis) by using **anonymous functions**. Such a function (without a name, hence the title 'anonymous') is evaluated once for every selected element. Anonymous functions are used extensively in D3 to compute attribute values. To set each circle's x-coordinate to a random value, **add** a line that sets the x-position (cx):

```
circle.attr("cx", function() { return Math.random() * 500; });
```

2.3 Binding Data

Of course, changing things at random is not really useful. More commonly, we would want to use existing **data** to drive the appearance of our circles. Let's say we want these circles represent the numbers 500, 150, and 1000. The `selection.data` method binds the numbers to the circles:

```
circle.data([500,150,1000]);
```

Data is specified as an array of values; this mirrors the concept of a selection, which is an array of selected elements. In the code above, the first number (the first datum, 500) is bound to the first circle (the first element, based on the order in which they are defined in the DOM), the second number is bound to the second circle, and so on.

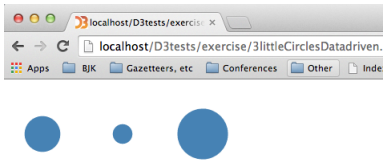
After data is bound, it is accessible as the first argument to the anonymous data functions. By convention, we typically use the

name `d` to refer to bound data. To set the area of the circle according to the data proportions, use the square root of the data to set the radius:

```
circle.attr("r", function(d) { return Math.sqrt(d); });
```

There's a second optional argument to each function you can also use: the **index** of the element within its selection. The index is often useful for positioning elements sequentially. Again, by convention, this is often referred to as `i`. For example:

```
circle.attr("cx", function(d, i) {return i * 100 + 40; });
```



Task 2 : Now use this mechanism to change the radius and position of the circles based on the data. •

2.4 Entering Elements automatically

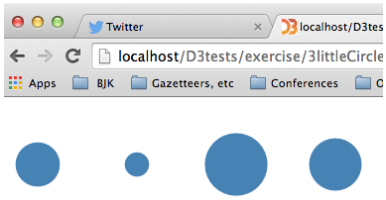
What if we had four numbers to display, rather than three? We wouldn't have enough circles, and we would need to create more elements to represent our data. You can append new nodes manually, but a more powerful alternative is the `enter` selection computed by a data join.

When joining data to elements, D3 puts any leftover data — or equivalently "missing" elements — in the `enter` selection. With only three circles, a fourth number would be put in the `enter` selection, while the other three numbers are returned directly by `selection.data`. By appending to the `enter` selection, we can create new circles for any missing data.

Taking this to the logical extreme, then, what if we have no existing elements, such as with an empty page? Then we're joining data to an empty selection, and all data ends up in `enter`.

This pattern is very common in D3 applications. You willl often see the `selectAll + data + enter + append` methods called sequentially, one immediately after the other, chained together using javascript's so-called **method chaining** (the chaining together of methods by using the `.` notation).

Task 3 : Change your webpage to match listing 2. As you can see, we now start with an empty svg and create the circles fully from the data. You now should get four circles! •



Listing 2: filefragments/4LittleCirclesDatadriven.html.txt

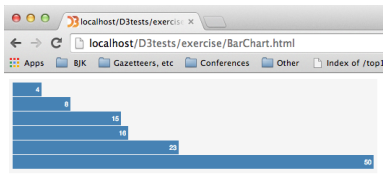
```
<!DOCTYPE html>
<html>
<head>
  <script type="text/javascript" src="lib/d3.v4.min.js">
  </script>
</head>
<body>
<svg id="svg" height="100%" width="100%">
</svg>
<script>
  var svg = d3.select("svg");
  svg.selectAll("circle")
    .data([500, 150, 1000, 750])
    .enter().append("circle")
    .attr("fill", "steelblue")
    .attr("cy", 60)
    .attr("cx", function(d, i) { return i * 100 + 30; })
    .attr("r", function(d) { return Math.sqrt(d); })
  ;
</script>
</body>
</html>
```

Note the first line of the script is now `var svg = d3.select("svg");` and the `d3.selectAll` was replaced by `svg.selectAll`. This is done to make sure the newly created circles do not just get appended to the end of the file, but are "attached" to the (until then) empty `svg` element.

3 A Bar Chart of Population Data

In this section we will make a more elaborate data visualisation: A bar chart of the population of the municipalities of Overijssel province. We will start by looking at some D3 code that creates a simple bar chart from data that is included in the code, in the same way you did in the previous section:

Task 4 : Create a new webpage called `BarChart.html` from the listing 3. Study the code carefully, try to understand how it works: How the data is used to set the position of the bars, their width, the text in the bars, and so on... On the next page we explain the new elements. Experiment with changing key numbers (such as the `xScale` variable) and other parts of the code to see how it influences the visualisation. •



Listing 3: filefragments/BarChart.html.txt

```
<!DOCTYPE html>
<meta charset="utf-8">
<html>
<head>
    <style>
        .chart {
            background-color: rgb(245,245,245);
            padding: 5px;
        }
        .chart rect {
            fill: steelblue;
        }
        .chart text {
            fill: white;
            font: 9px sans-serif;
            text-anchor: end;
        }
    </style>
    <script src="lib/d3.v4.min.js"></script>
</head><body>
<svg class="chart"></svg>
<script>
    var data = [4, 8, 15, 16, 23, 50];
    var svgwidth = 500,
            barHeight = 20;
    svgheight = barHeight * data.length;
    var xScale = d3.scaleLinear()
            .domain([0, d3.max(data)])
            .range([0, svgwidth]);
    var chart = d3.select("svg")
                    .attr("width", svgwidth)
                    .attr("height", svgheight)
            ;
    var bar = chart.selectAll("g")
                    .data(data)
                    .enter().append("g")
                    .attr("transform", function(d, i) { →
                        return "translate(0," + i * →
                        barHeight + ")"; })
            ;
    bar.append("rect")
            .attr("width", function(d) { return xScale(d);}→
                 )
            .attr("height", barHeight - 1)
    ;
    bar.append("text")
            .attr("x", function(d) { return xScale(d) - 3; →
                })
            .attr("y", barHeight / 2)
```

```
            .attr("dy", ".35em")
            .text(function(d) { return d; })
    ;
</script>
</body>
</html>
```

Some elements we have not seen before are:

- the svg `g` element (a placeholder to group an arbitrary amount of elements together);
- the `rect` and `text` elements, that not surprisingly create rectangles and texts, respectively;
- the `transform` attribute. This applies to a (group of) element(s), the content of the `transform` string sets the actual transformation: In this case a `translate`, other possibilities are `rotate` and `scale`. In our code, the transform is used to draw each bar at a different position, by translating along the y-axis based on the data index.
- the code at the top (in the `<style>` tag). This is using the CSS or Cascading Style Sheets language. We will not into details on CSS in this exercise, suffice to say that it sets the styling of the `.chart` class.

3.1 Using D3 data-dependent scaling

A weakness of the code above is the *"'magic"* number 10 to set the `xScale` variable. We use it to scale width of the bars, depending on the data value. This number depends on the domain of the data (the minimum and maximum value, 0 and 50 respectively), and the desired width of the chart (500 = the svg width), but of course these dependencies are only implicit in the value 10, and if the data changes, this numbers should also be changed.

We can make these dependencies explicit and eliminate the magic number by using a linear scale dependent on the actual data. D3's `linear scale` specifies a mapping from data space (`domain`) to display space (`range`):

```
var xScale = d3.scaleLinear()
    .domain([0, d3.max(data)])
    .range([0, svgwidth]);
```

Although xScale as before looks like a simple variable, it is now actually a **function**, that returns the scaled display value in the range for a given data value in the domain. For example, an input

value of 4 returns 40, and an input value of 16 returns 160. To use the new scale, simply replace the hard-coded multiplication (`xScale * d)` by a call to the `scale` function in two locations:

```
bar.append("rect")
    .attr("width", function(d) { return xScale(d)} )
```

and

```
bar.append("text")
    .attr("x", function(d) { return xScale(d)- 3; })
```

Task 5 : To understand how this makes scaling flexible, implement the changes mentioned above and experiment with changing the width of the SVG by changing the number in the line:

```
var svgwidth = 500,
```

Notice how the chart adapts to the width available... •

D3 scales can also be used to interpolate many other types of display-space values, such as paths, color spaces and geometric transforms.

3.2 Loading external data

Another weakness is that the data values are hard-coded in the `data array`. In real-life use, we often need the data come from some external source (a downloaded file, or a webservice, or even a sensor device).

To use external data in a web browser, we need to download the file from a web server and then parse it, which converts the text of the file into usable JavaScript `objects`. Fortunately, D3 has several built-in methods that do both of these things in a single function, for several types of data. Here we will use the `d3.csv` function. This is used for CSV files, that are plain text files containing comma-separated values, tab-separated values or other arbitrary delimiter-separated values. These tabular formats are popular with spreadsheet programs such as Microsoft Excel. Each line represents a table row, where each row consists of multiple columns separated by tabs. The first line is the header row and specifies the column names.

! → Do **not** open the file from your file–system, but always from your actual web site, e.g.:
`http://win371.ad.utwente.nl/...` or `http://localhost/...`
This is the only way to properly test your web site!

The reason is that browsers are strict on so–called *JavaScript cross–domain security*, meaning it won't run JavaScript that is not coming from the same server as the html. And from the browser viewpoint, a file coming from `file:///` is from another domain than one from `http://win371.ad.utwente.nl/`, even if these point to the same file...! Also, the security mechanism usually does not allow loading from harddisk (such as `file:///C:`) anyway, although sometimes you can override this, e.g. in FireFox. So to do the latter part of the exercises, you need to serve the files through a webserver. All this is also explained at the D3 site (with some tips on how to install a localhost server) at: https://github.com/d3/d3/wiki#local-development An alternative is using local server in Python see: http://stackoverflow.com/questions/27977972/how-do-i-setup-a-local-http-server-using-python.

Loading data introduces a new complexity: downloads are **asynchronous**. After you call the `d3.csv` function, it returns immediately while the file downloads in the background. At some point in the future when the download finishes, the so-called **callback function** is invoked: That function is defined by the second argument of the `d3.csv` function, in most cases by creating another one of those anonymous function calls we used before:

```
d3.csv("myDatFile.csv", function(error, data) {
  // Code here is the callback function...!
  // inside it you have access to the loaded data object
  // or the error codes when unsuccesful
  });
```

Therefore we need to be sure that any code that needs the data (which in our case is actually all of the code), is inside the callback function, so that it waits to run until after all data has been loaded. In order to achieve that, let's "wrap" our existing code inside the `d3.csv` function:

Task 6 : Create a copy of the HTML webpage `BarChart.html` you made before, and call it `BarChartOverijssel.html`.

Replace the line:
`var data = [4, 8, 15, 16, 23, 50];`
with the lines:

```
d3.csv("overijssel_population.csv",
  function(error, data) {
  //start of callback function:
```

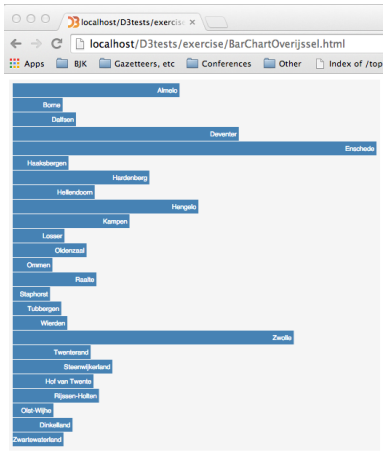
And after the last line of the code (just before the `</script>` statement), add the proper closing of the callback function and `d3.csv`

method:

```
}); //end of callback function
```

•

If you now run the code in your browser, you will get no result, and several error messages in the Javascript console, such as: "Invalid value for <rect> attribute width='NaN"'. So, what is wrong? Now that our dataset contains more than just a simple array of numbers, each data instance is a complex **object** rather than a single **number**. The equivalent representation in JavaScript would look not like a simple array, but like this:




```
var data = [
  {gm_code: "GM0141", gm_naam: "Almelo", aant_inw: "72730"},
  {gm_code: "GM0180", gm_naam: "Staphorst", aant_inw: "16275"},
  ...etc..
  ];
```

If you study this, you can see the `data` object is an array (delimited by `[ ]`) of several objects (delimited by `{ }`) , in each of which `gm_code` is the code for the municipality (gemeente), `gm_naam` is its name and `aant_inw` is the number of inhabitants.

Thus, if we want to read a data value for number of inhabitants, we must refer to the value as `d.aant_inw` rather than just `d`. So, whereas before we could pass the scale `xScale(d)` to compute the width of the bar, we must now specify the correct data value to the scale: `xScale(+d.aant_inw)`

Likewise, when computing the maximum value from our dataset in our `xScale` function, we must pass an anonymous function to `d3.max` that tells it which data variable to use to determine the maximum:

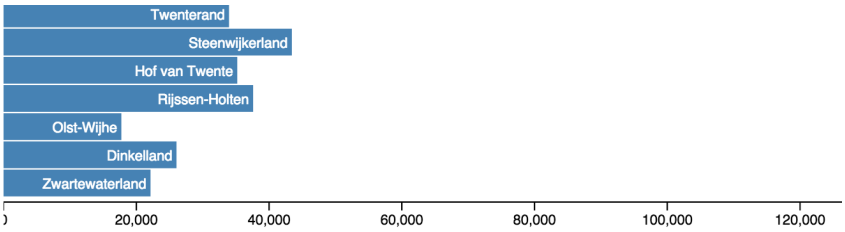
```
var xScale = d3.scaleLinear()
    .domain( [0, d3.max(data, function(d) {
      return +d.aant_inw;
      } ) ] )
    .range( [0, svgwidth] )
    ;
```

! → In the code examples above, note the use of `xScale(+d.aant_inw)`, instead of just `xScale(d.aant_inw)`. This a way to overcome the problem that Javascript has no strict **typing**. Thus, if the variable `d.aant_inw` is read, we can not be sure it is treated as a number, or maybe as a string. To enforce the treatment of the variable as a number, we "add it to nothing" (by prefixing it with the plus sign).

Task 7 : Repair the use of the d value to match the data-structure, as explained above. Test again to see if you now get the correct bar chart for all Overijssel municipalities. •

3.3 Adding an Axis

Task 8 : The next task is to create a D3 axis. Below we mention the building blocks needed, you should figure out yourself where these should be placed, and test the results in your browser. The final axis should look like the figure below... •



D3 has a powerful mechanism to add axes to graphs, charts and maps. We define an axis by telling it to use a scale and declaring one of the four orientations. If your axis should appear below the bars, you use the `axisBottom(scale)` function (the other possibilities are, not surprisingly, `axisTop`, `axisLeft` and `axisRight`).

```
var theAxis = d3.axisBottom(xScale);
```

The resulting object (here named `theAxis`) can be used to render multiple axes by repeated application using `selection.call()`. Think of it as a rubber stamp which can print out axes wherever they are needed. The axis elements are positioned relative to a local origin, so to transform into the desired position we set the "transform" attribute on a containing g element.

```
chart.append("g")
    .attr("class", "axis")
    .attr("transform", "translate(x, y)"
    .call(theAxis);
```

The axis container should also have a class name so that we can apply styles. The name `"axis"` is arbitrary; call it whatever you like. The axis component consists of a path element which displays the domain, and multiple g ".tick" elements for each tick mark. A tick in turn contains a text label and a line mark. Most of D3's examples therefore use the following minimalist style in CSS:

```
.axis text {
  font: 9px sans-serif;
  fill: black;
}
.axis path,
.axis line {
  fill: none;
  stroke: black;
}
```

You will also need to make room for the axis below the graph. To do that, you'll have to make the `svg height` bigger then the height needed for the chart bars alone!

4 Using D3 for maps

In many ways, a map is just another data-driven graphic. So, of course D3 can also be used to create maps. We will use a `geosjon` file to load a map of the same Overijssel municipalities we have used above. The data was created using the `export to geojson` option in QGIS, such possibilities are available in most GIS software, or from web services such as the one at http://www.mapshaper.org/

4.1 Visualising Geographic Data

Task 9 : Create a webpage using the code in listing 4. Test it, you should see a basic map, of the municipalities of Overijssel province.

Study the code to see how the map was made using techniques very similar to the BarChart examples before... •

Listing 4: filefragments/Map.html.txt

```
<!DOCTYPE html>
<meta charset="utf-8">
<html><head><title>The Graphic Web Locations</title>
<script src="./lib/d3.v4.min.js"></script>
<style>
.mapSVG {
  background-color: rgb(245,245,245);
}
.municipality {
  fill: rgb(255,240,214);
  stroke: rgb(255,159,227);
}
</style>
```

```
</head><body>
<div id="mapDiv"></div>
<script>
var mapWidth = 400, mapHeight = 350;
var mapDiv, mapSVG, svgpath;
// select the mapDiv by id:
mapDiv = d3.select("#mapDiv");
// create an svg of fixed size:
mapSVG = mapDiv.append("svg")
       .attr("class", "mapSVG")
       .attr("width", mapWidth)
       .attr("height", mapHeight)
;
// Define Mercator proj to center at data (lon-lat):
var myProj = d3.geoMercator()
       .center([6.0 , 52.5])
       .scale(10000)
       .translate([mapWidth / 2, mapHeight / 2])
;
//Define svg path generator using the projection
svgpath = d3.geoPath().projection(myProj);
// asynchronously load geojson:
d3.json("overijssel.json", function (error, myGeoJson) {
    // create new svg paths:
    mapSVG.selectAll("path")
        // bind the data:
            .data(myGeoJson.features).enter()
        // for each d create a path:
            .append("path")
            .attr("class", "municipality")
            .attr("d", svgpath)
    ;
});
</script>
</body></html>
```

You should have encountered only a few D3 functions and methods that we did not use before. First of all a projection object:

```
var myProj = d3.geoMercator()
       .center([6.0 , 52.5])
       .scale(10000)
       .translate([mapWidth / 2, mapHeight / 2])
;
```

D3 has a lot of functionality to use geographic data, i.e. data that is expressed as coordinate pairs in longitude and latitude.

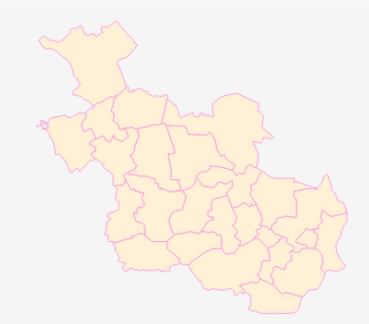
Here we use the `d3.geoMercator()` function that is a so-called *factory* to turn lon-lat coordinate pairs into cartesian projected

coordinates, using the Mercator projection. To specify the projection, here we use the `.center`, `.scale` and `.translate` methods of this projection object to zoom into the Overijssel area. There are a multitude of available projections, see https://github.com/d3/d3/blob/master/API.md#geographies-d3-geo for examples and explanation.

The projection factory is used in turn in another very useful method called `d3.geoPath()`. This takes a stream of real-world coordinates and transforms them into an SVG drawing `path` that uses screen coordinates to fit into an SVG object on the webpage:

```
svgpath = d3.geoPath().projection(myProj);
```

This is then used in the statement `.attr("d", svgpath)` to load, project and draw the coordinates of each municipality from the dataset.



Task 10 : The map does not really fit nicely in the div. **Experiment** with the projection settings to make the map fit better (as in the figure left). •

The dataset was loaded in the line

```
d3.json("overijssel.json", function (error, myGeoJson) {
```

which works exactly as the `d3.csv` method we used earlier, but now for `geoJSON` data. This is a special version of the general JSON format (JavaScript Object Notation). It can be created by exporting data from many GIS systems, as well as using on-line tools such as http://www.mapshaper.org/.

The format of geoJSON is standardised (see http://geojson.org/), but the standard allows for several ways to structure the data, therefore it is sometimes tricky to find the proper objects to address. In most cases there is a top-level GeoJSON object with the type `FeatureCollection` , that has a member with the name `features`. The value of `features` is a JSON array, and each element of the array is a `Feature` object which in turn will have `properties`, of which one is a `geometry`.

In the case of our Overijssel data this looks as follows:

```
{"type":"FeatureCollection",
  "features":
    [ {"type":"Feature", "properties":
      {"gid":7, "gm_naam":"Almelo","aant_inw":72730 ...etc... },
      "geometry":{"type":"Polygon","coordinates":
```

```
[[[6.69863498963163,52.39374126373987], ...etc...
```

The array of features is similar to the array of data objects we used in the CSV solution, therefore to bind to it we use the line:

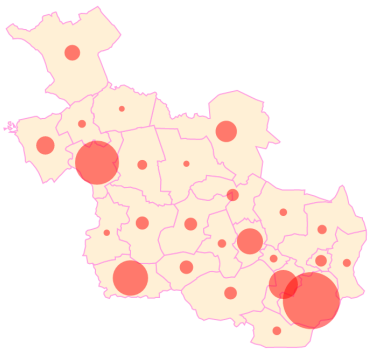
`.data(myGeoJson.features).enter()`

The d3.geoPath function is smart enough to be able to extract the needed geometry objects from this array by itself...

To get to the other elements of the data, you address sub-objects of the `features` elements, such as `d.properties.aant_inw` for the number of inhabitants data.

4.2 Thematic Maps

Just like in the bar chart example, you can of course use the properties in the data files to create data-driven visualisations using the SVG drawing possibilities. For example, you can create an extra layer on top of the municipalities to draw circles of different sizes, depending on a data property. Thus, you create a so-called *proportional point symbol map* with the code fragment below:



Listing 5: filefragments/MapProportionalFragment.html.txt

```
//add this to the CSS style section:

    .propCircle {
        fill: rgb(255,0,0);
        opacity: 0.5;
    }

// create new svg circles:
  mapSVG.selectAll("circle")
     // bind the data:
     .data(myGeoJson.features).enter()
     // for each d create a circle:
     .append("circle")
     .attr("class", "propCircle")
     .attr("cx", function (d) {
        // calculates centroid X of geo path :
        return Math.round(svgpath.centroid(d)[0]);
     })
     .attr("cy", function (d) {
        // calculates centroid Y of geo path:
        return Math.round(svgpath.centroid(d)[1]);
     })
     .attr("r", function (d) {
      // set r to be using some data value instead of 50:
        return 50;
     })
```

;

Task 11 : Insert the code in listing 5 into the code you used in the previous task. You should figure out yourself:

- where to insert the code fragments, and
- how to use the data value for "number of inhabitants" (`aant_inw`) to properly scale the circles;

To refine your visualisation, experiment with the CSS settings and with the ratio used in the code that determines the circle radius. . . .

•

---

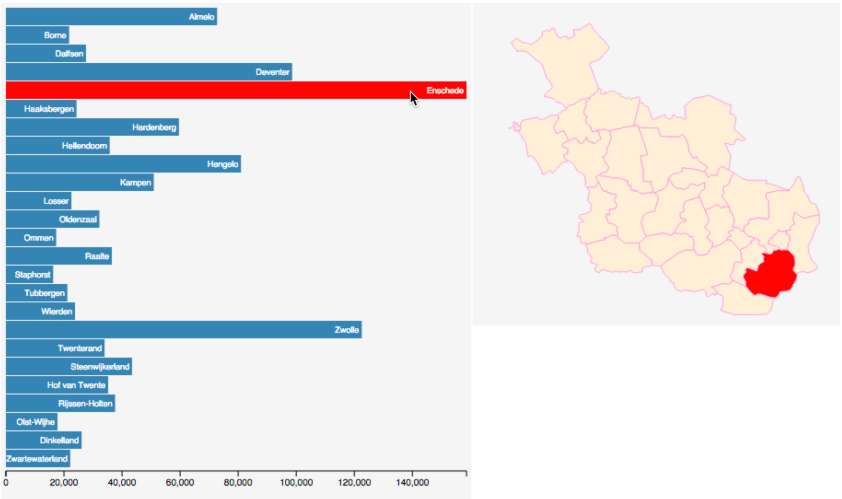
5 CHALLENGE: Interactively Linking Map and Chart

As an **optional challenge**, we ask you to create some interactivity for the bar chart:

Task 12 : The optional challenge task is to create the following:

- A webpage that holds both the Overijssel Bar chart and the municipal Proportional Point Symbol map;
- If the user moves the mouse over a bar in the graph, the bar colour changes to red, and the corresponding municipality in the map is highlighted.

Below we show some of the building blocks needed, you should figure the rest out yourself, and test the results in your browser. . . •



In order to place the graph and map next to each other, you can use different lay-out possibilities. The simplest way of doing that is styling the HTML `<div>` element we use to hold the map. You can change the creation of the mapDiv we used in the map page, from:

```
mapDiv = d3.select("#mapDiv");
```

to:

```
mapDiv = d3.select("#mapDiv")
       .style("position", position type)
       .style("left", X in pixels)
       .style("top", Y in pixels)
;
```

The style properties are expressed using **CSS** (Cascading Style Sheets), the styling language standardised by the W3C, the World Wide Web Consortium. The meaning of the properties used is:

position: this lets you define positioning *absolute* or *relative* to other divs.

left & top: the (absolute or relative) location with respect to the upper left corner of the window

width & height: the size of the element (in pixels)

overflow: if the content is larger than fits the div, it will *not* be shown if this is set to `hidden`. Other settings are `visible` (will overflow), `scroll` (will make scrollbars) and `auto` (let browser decide).

border: the border look (width, type and colour). You can also set the `fill` in a similar way.

As an alternative to creating the style using D3 selectors, you can of course also use a CSS definition for this and create a classed div using `.attr("class", "cssname")`...

Interactivity is triggered by an **event** connected to the appropriate selection. D3 supports interactivity of all its selections, by the use of `selection.on`. This adds an event listener to each element in the current selection, for the specified type. The type is a string event type name, such as `"click"`, `"mouseover"`, `"mouseout"`, or `"submit"`. Any DOM event type supported by your browser may be used. The specified listener is invoked in the same manner as any other operator functions, being passed the current datum `d` and index `i`, with the `this` context as the current DOM element, in such a way:

```
selectedObject
    .on("mouseover", function(d, i) {
      ... code run per data instance ...
      ... e.g. to show information ...
      ... and to change bar colour ...
     })
    ;
```

The interactivity can of course be improved by having the reverse highlighting also working (mouse in map highlights bar), by having the data displayed, etcetera. We will add a working example to your learning platform (e.g. Blackboard) towards the end of the exercise.

Happy puzzling...!

6 Links & References

For this exercise we used existing materials from various sources, that we also recommend to you for learning D3:

- The book "D3.js in Action" (2015) – Elijah Meeks, Manning Publication co. (New York). ISBN: 9781617292118 (in ITC library);
- The D3 website at http://d3js.org/, with the official API documentation and many D3 Examples and tutorials;
- The first section of this exercise was based on the 3 Little Circles tutorial at http://bost.ocks.org/mike/circles/.
- A useful reference for the JavaScript Language in general can be found at http://www.w3schools.com/jsref/default.asp.