F21DV: Lab 2

# Introduction

This report will serve as an accompaniment to html files containing the JavaScript code to solve lab 2 of the F21DV course. It will go through all 7 parts as described in the pdf overview and highlight key aspects of the code which led to the completion of each of the 32 exercises and mention other possible stylistic choices when appropriate. This report assumes some prior knowledge of JavaScript and will focus on describing the use of the d3.js v7 library.

I demonstrated to Ben Kenwright on the 25th of February 2022, showing exercises FILL IN .

A GitHub repository was created and showed to be helpful with code management. With the help of the desktop app for GitHub, I managed to smoothly operate on both my IDE of choice (WebStorm) and see the progress on my GitHub page. The repository is the following but may be set to private depending on when this link is clicked: <https://github.com/YoussefBonnaire/DataVisualisation>.

# Part 1

In this part we explored CSS effects and animations such as the CSS keyframes. We used our DrawWave function created for exercise 12 of Lab 1 in order to generate a line graph in an svg with circles elements for each point. We used the pulse class provided in the example which is defined as such in CSS:

.pulse {  
 border: 1px solid red;  
 fill: lightblue;  
 stroke: purple;  
 -webkit-transform: scale(1);  
 -webkit-transform-origin: 50% 50%;  
 transform: scale(1);  
 transform-origin: 50% 50%;  
}

It uses it specifies an action which happens when you hover over it with your mouse in the following code:

.pulse:hover {  
 -webkit-animation-name: pulsar;  
 -webkit-animation-duration: 0.2s;  
 -webkit-animation-iteration-count: infinite;  
 -webkit-animation-direction: alternate;  
 animation-name: pulsar;  
 animation-duration: 0.2s;  
 animation-iteration-count: infinite;  
 animation-direction: alternate;  
 -webkit-transform-origin: 50% 50%;  
 transform-origin: 50% 50%;  
}

The action called is the keyframes pulsar specified here:

@keyframes pulsar {  
 from {  
 fill: red;  
 }  
 to {  
 fill: red;  
 transform: scale(1.3, 1.3);  
 transform-origin: 2% 2%;  
 }  
}

With this information in the CSS, all that remains to do is the add the points specifying they are of class pulse:

// Add points  
svg.selectAll("dot")  
 .data(shape\_data)  
 .enter()  
 .append("circle")  
 .attr("cx", function (d) {  
 return x(d.x)  
 })  
 .attr("cy", function (d) {  
 return y(d.y)  
 })  
 .attr("r", 5)  
 .style("fill", "red")  
 .attr('class', 'pulse');

Another possible CSS effect which can be performed is simulating alt text. This can be done using a div and a span congruently with the following code forcing a divs to be displayed only when hovering over corresponding span:

<style>  
 div {  
 position: relative;  
 background-color: green;  
 padding: 20px;  
 display: none;  
 }  
 span:hover + div {  
 position: absolute;  
 display: inline-block;  
 }  
</style>

# Part 2

D3 also allows for events to occur in order to animate elements to your liking. Using the function “.on” after having created an element we are able to transform this element dynamically. For example, if we wish to have changes happen to an object once our mouse hovers said object as was the case in the previous part, we can use the keywords mouserover and mouseout in the following way:

d3.selectAll("div")  
 // Change svg when mouse hovers in the following ways  
 .on("mouseover", function () {  
 d3.select(this)

// insert desired changes on mouseover  
 .style("background-color", "orange")  
 .style('width', '150px')  
 .style('height', '30px')  
 .style('border', '5px outset orangered')  
 .style('display', 'block');  
 })  
 // Change svg after mouse hover in the following ways  
 .on("mouseout", function () {  
 d3.select(this)

// insert desired changes when mouse stops hovering  
 .style("background-color", "steelblue")  
 .style('width', '125px')  
 .style('height', '25px')  
 .style('border', '2px outset darkblue');  
 });

Another keyword which can be used is mousemove. This allows you see precisely where the user’s mouse is and have an action occur with said information. To move an already created text to follow the cursor we can use the following code:

svg.on('mousemove', function(event){  
 // Use event to get x and y positions of the mouse  
 Xpos = event.clientX  
 Ypos = event.clientY  
 // Move the text to mouse location.  
 mouse\_text.attr('x', Xpos-130)  
 .attr('y', Ypos - 50)  
 })

# Part 3

Transitions are another way to animate your webpage. They were briefly discussed in Lab 1 and can be easily chained as demonstrated in the following example:

d3.select('body')  
 .append("div")  
 .style('width', '100px')  
 .style('height', '100px')  
 .style('background-color', 'blue')  
 // transition to red  
 .transition()  
 .duration(1000)  
 .style("background-color", "red")  
 //Transition further to green  
 .transition()  
 .duration(2000)  
 .style('background-color','green');

However, they can be used with the .on function in order to have a smooth transition from one outcome to another unlike in Part 2 which had the change happen instantly. This can be done including the transition function and information within the on function like so:

d3.select('body')  
 .append("div")  
 .style('width', '100px')  
 .style('height', '100px')  
 .style('background-color', 'blue')  
 //transition on mouse over  
 .on('mouseover', function () {  
 // transition to small red div  
 d3.select(this).transition()  
 .duration(1000)  
 .style("background-color", "red");  
 })  
 // transition back after mouse over  
 .on('mouseout', function () {  
 //Transition further to large green div  
 d3.select(this).transition()  
 .duration(2000)  
 .style('background-color', 'blue');   
 })

These transitions can be further smoothened using the ease function. There are multiple types of ease functions, easeElastic, easeBounce, easeLinear, easeSin, easeQuad, easeCubic, easePolye, aseCircle, easeExp and easeBack. To use these, call the function with the desired type as an argument like this:

.transition()  
.duration(3000)  
.ease(d3.easeBounce)

Delays are another additional function which can be used with transitions, using delay is as evident as using duration and is called the same way:

.transition()  
 .ease(d3.easeLinear)  
 .duration(2000)  
 .delay(2000)

This example will delay the transition by 2 seconds compared to when it would be called otherwise. When another transition is called in the same chain, the delays stack and the second delay is compared to when the first transition ends, not when the first transition was called. Here is an example of how to stack these:

.transition()  
.ease(d3.easeLinear)  
.duration(2000)  
.delay(4000)  
// event to transition to   
.transition()  
.ease(d3.easeLinear)  
.duration(2000)

//event to transition to

# Part 4

In the case where the div elements do not exists and you decide to selectAll(“div”), you may still bind your data to this selection, the important difference then is that you then add the .enter() function which checks for corresponding div elements. If 1 element exist but we have 3 datapoints, only the first will be used, however, after the .enter() function we use the .append(div) function, it will add a div element for any datapoint missing one. An example of this is shown here below as we have not created any span objects prior, but the code will create it dynamically:

var myData = ['a', 4, 1, 'b', 6, 2, 8, 9, 'z'];  
console.log(typeof ('a'))  
var spans = d3.select("body")  
 .selectAll("span")  
 .data(myData)  
 .enter() // checks whether span element exist for data index  
 .append('span') // Adds non existing span elements  
 .text(function (d) {  
 return d;  
 })  
 .style('color', function (d) {  
 console.log(typeof (d))  
 if (typeof (d) == 'string') { // Selects objects of type string to change to blue  
 return 'blue'  
 } else {  
 return 'green'  
 }  
 });

Conversely, when a datapoint of the binded data is removed, calling the .exit() function on the spans variable above will remove the unused dom elements.

# Part 5

Being able to bind data from an array is very useful for a lot of basic activities, however, when we start having larger datasets they are usually provided in the form of external csv, json, tsv or xml files. Using these files in d3 is fairly simple, using for example a csv file, we can use it’s data in two ways:

This first way allows you to have a function acting on the entire data set gathered:

d3.csv(csvfile).then(data => {  
 *DoSomething*(data)  
 )

And this second way allows you to have a function working on each datapoint independently:

d3.csv(csvfile, function(data){  
 *DoSomething2*(data)  
 )

As an example, assume this data has a column for the age of each datapoint; DoSomething2 can count up the number of individuals above a certain age in the following way (having initialized age50 variable before similar to how it was done in exercise 10):

function *DoSomething2* (d) {  
 if (d.age >= 50) {  
 age50 += 1;  
 }

}

# Part 6

SVGs Scalable Vector Graphics (SVG) are a standard way of segmenting the screen to output visuals easily. This code below initialises the SVG and it’s size on the screen adding a border at the end to make it visible.

//Create SVG element  
var svg = d3.select("body")  
 .append("svg")  
 .attr("width", 400)  
 .attr("height", 400)  
 .style("border", '1px solid green');

Then within this svg we can easily append different shapes just like this line for example:

//Create blue line element inside SVG  
svg.append("line")  
 .attr("x1", 100)  
 .attr("x2", 200)  
 .attr("y1", 50)  
 .attr("y2", 50)  
 .attr("stroke", "blue")

Each shape has different attributes used to place it and determine it’s size, for example, a circle would be added in this way (this code is using data bound to the svg):

//Create circle element inside SVG

svg.append("circle")  
 .attr("cx", d.cx\_pos)  
 .attr("cy", d.cy\_pos)  
 .attr("r", d.width / 2)  
 .attr('fill', d.color);  
}

# Part 7

Bar charts use the idea presented here before considerably. After all, bar charts are simply rectangles placed along side each other (or above/under each other) with determined lengths. A crucial part of bar charts in SVGs are the scale which adjusts our rectangles appropriately. We will discuss that further in part 9. These are the simple parts of making a chart.

This part of the code creates the svg container we have our chart in:

var graph = d3.select("body")  
 .append("svg")  
 .attr("width", width)  
 .attr("height", barHeight \* age\_bins.length);

Then using data from a csv as can be seen more explicitly in exercise 14&15.html, we place the elements in which our bars will be located using the following code:

// Initialise g element to contain each bar at specified height  
var bar = graph.selectAll("g")  
 .data(age\_bins)  
 .enter()  
 .append("g")  
 .attr("transform", function (d, i) {  
 return "translate(0," + i \* barHeight + ")"; // Make sure containers are one below the other  
 });

Moreover, we can add the actual rectangles to each element g to show ours bars using the following code:

// Populate bars with data  
bar.append("rect")  
 .attr("width", function (d) {  
 return d \* scaleFactor;  
 })  
 .attr("height", barHeight - 1)  
 .style('fill', function (d) {  
 if (d > 100) {  
 return 'red'  
 } else {  
 return 'green'  
 }  
 });

Finally, we can add the label for each bar with the following code:

bar.append("text")  
 .attr("x", function (d) {  
 return (d \* scaleFactor);  
 })  
 .attr("y", barHeight / 2)  
 .attr("dy", ".5em")  
 .text(function (d) {  
 return d;  
 }).style('fill', function (d) {  
 if (d < 140) { // Specifies ranges for text color based on count  
 return 'yellow'  
 } else {  
 return 'purple'  
 }  
});

# Conclusion

With this report I have discussed all 16 parts of this course work supporting the 32 exercise files accompanying this report describing the basic functionality of D3 selection, data binding, DOM object manipulation, external file use, svg container use, bar, circle and line chart creation, scales for containing data in svgs, scales for data dependent colour schemes, additional markers on line charts, pie charts and adding images.