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 Shuangjiang Xue on the 25th of February 2022, showing exercises 2, 4, 17, 9, 22, 23, 27 and 33.

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

By combining the transitions and mouse over functions from the previous sections, we are able to create charts which enable a lot of interaction from the user and animate in satisfying manners. For example, a bar chart can be animated by enlarging and changing the color of each bar when the mouse hovers over it. To change the color. It may be useful to have different classes in the css style section to differentiate the bar when it is being moused over and when it is not:

<style>  
 .bar {  
 fill: steelblue;  
 }  
  
 .highlight {  
 fill: orange;  
 }  
</style>

Used with:

//mouseover event handler function  
function *onMouseOver*(d, i) {  
 d3.select(this).attr('class', 'highlight');

And

//mouseout event handler function  
function *onMouseOut*(d, i) {  
 // use the text label class to remove label on mouseout  
 d3.select(this).attr('class', 'bar');

See exercises 15-17 for the full code.

When adding label text while hovering over the rectangles I decided to explore adding text directly to the graph element and using the event handler instead of the rectangles themselves. I discovered, in order to find the values required for this example there are two different ways:

d.target.x.animVal.value

which will give me the x position of the animation,

d.target.\_\_data\_\_.value

Which will give me the value of the data for which we can pass through the x scale to get the x position.

# Part 5

Updating data used in a visual is most commonly done using a button which calls an update function with a different dataset.

<button onclick="*update*(data1)">Variable 1</button>  
<button onclick="*update*(data2)">Variable 2</button>

To transition smoothly between these, it is important to remember the x and y scales may change and including the domains of these scales in the function is necessary. Similarly, as data can have different lengths it is important to remove extra data with a smooth transition i.e.:

u.exit()  
 .transition().duration(1000).attr('x', 600)  
 .remove()

This code segment is what causes extra rectangles to slide out of the view when the data changes. With both axis and the data itself transitioning to fit your new data set, you have a smooth bar chart which updates. The same can be done with a line graph however when using categorical data it is useful to remember the rectangles have a certain bandwidth and the x parameter must be adjusted accordingly otherwise the point of the line is placed at the start of each container. This is done in the following section of the code for exercise 23:

.attr("d", d3.line()  
 .x(function (d) {  
 return x(d.group) + x.bandwidth() / 2;  
 })

# Part 6

Interpolate calculates intermediate values between two given values, similar to a scale between the two lists given. With colors this means the interpolation will be a list of values of the rgb values between the colors selected and it can also be used with a date to find the dates at between two dates.

Using interpolate you can find all the positions an arc would need to be to transition from one arc to another. This is useful for an animated graph where we desire a change in data to be done with a smooth transition. To make an arc transition in from where it is meant to start the following code segment is used:

.attrTween("d", function (d) {  
 var i = d3.interpolate(d.endAngle, d.startAngle);  
 return function (t) {  
 d.startAngle = i(t);  
 return arc(d);  
 }

This causes the arc to smoothly move from the start of the angle to the end. Inversing the endAngle and startAngle causes the arc to disappear and was used in exercise 27 to remove point in one data set no longer in a new one in the following way:

function *removeArcTween*(a) {  
 var i = d3.interpolate(a.startAngle, a.endAngle);  
 return function (t) {  
 a.startAngle = i(t);  
 return arc(a);  
 };  
}

Finally in order to move a datapoint from one dataset value to another the following code was used which moves the current angle to the desired end angle:

function *transitionArc*(a) {  
 var i = d3.interpolate(this.\_current, a);  
 this.\_current = i(0);  
 return function (t) {  
 return arc(i(t));  
 };  
}

# Part 7

The d3 force layout uses the same general logic used throughout the entire library. The way it function is instead of working directly form an object in the svg, we create an svg object within the force simulation using the .on(‘tick’,…) function and key word. Ie:

var simulation = d3.forceSimulation(data)  
 .force('charge', d3.forceManyBody().strength(5))  
 .force('center', d3.forceCenter(width / 2, height / 2))  
 .force('collision', d3.forceCollide().radius(function (d) {  
 return d.radius + 1  
 }))  
 .on('tick', ticked);

Within the .on, the ticked function we created looks the exact same as every single other object creation we have done in the labs:

function ticked() {  
 var u = d3.select('svg')  
 .selectAll('circle')  
 .data(data)  
 .join('circle')  
 // specify color of circle based on data  
 .attr('fill', function (d) {  
 return Color(d.radius)  
 })  
 .attr('r', function (d) {  
 return d.radius  
 })  
 .attr('cx', function (d) {  
 return d.x  
 })  
 .attr('cy', function (d) {  
 return d.y  
 })  
 .on('mouseover', *displayRadius*)  
 .on('mouseout', *removeRadius*)  
}

Allowing creation of data dependent objects, actions which can be observed hovering over these and changes which can be observed after.

There are 6 different types of forces which can be applied to the simulation: many body which makes elements attract or repel each other, center which acts a center of gravity pulling all objects to it, collide which prevent or allows overlapping of the objects, Force X which attracts object in the x direction, Force Y which attracts object in the y direction and link which creates a fixed distance between connected elements. Here is an example of how to use 4 of these in a simulation:

var simulation = d3.forceSimulation(data)  
 .force('charge', d3.forceManyBody().strength(10))  
 .force('center', d3.forceCenter(width / 2, height / 2))  
 .force('y', d3.forceY().y(function () {  
 return 5;  
 }))  
 .force('collision', d3.forceCollide().radius(function (d) {  
 return d.radius + 3  
 }))  
 .on('tick', ticked);

# Conclusion

With this report I have discussed all 7 parts of this course work supporting the 32 exercise files accompanying this report describing the basic functionality of CSS based effects and animations, d3 events and the .on function, smooth transitions using d3, creating an animated chart, transitioning between changing datasets for bar, line and pie charts and using the d3 force layout.