Data Visualisation and Analysis Report – Lab 2

Christopher Murdoch, cm151

Demonstrated to Amit on 25/02/22

Exercise 1

To create a reusable CSS animation with keyframes, I created a 'datapoint' class with an attached pulsar keyframe animation. Originally, when setting *transform-origin* to *50% 50%*, the dots moved in and out from the centre of the SVG. This was because *transform-origin: 50% 50%*, sets the centre of the **surrounding element** as the scale centre. To rectify this, I set *transform-box: fill-box*; and the element's own centre was used as the transform origin.

Exercise 2

To dynamically generate the page content, I created an array of strings, each representing a city. Using d3's *enter* function, I created images with their src attributes selected by name, and additional divs programmed to appear on image hover.

Exercise 3

I created a CSS class with keyframes set to change the border colour in a rainbow effect. I then used d3 to add event listeners designed to add and remove the rainbow border class, on mouseover and mouseout, respectively.

The event listeners chain calls just like d3's attribute functions so it was easy to add multiple listeners to the same element all at once.

Exercise 4

Very similarly to exercises 2 and 3, I created a CSS class that changes colour and grows, and used d3's event listeners to add and remove the class from elements when hovering.

Exercise 5

Unlike exercise 3, adding text that follows the mouse can't be done with *mouseover*. Instead, it must continuously update with *mousemove*. This is called every time the mouse moves and allows the added text to follow it.

Exercise 6

Chaining an additional transition is very easy with d3. Simply chaining one transition after another automatically starts the next transition after the first has completed. Attributes and duration can be added to the second transition with further chaining to achieve the desired effect. I created the transition sequence by chaining the transitions.

Exercise 7

Similarly to exercise 6, chaining multiple transitions allows a sequence of transitions to run, one after the other. I changed the colour and size in the first transition, and reverted it in the second.

Exercise 8

Adding transitions to an element based on a mouse event works perfectly – even when interrupting another transition. D3 automatically interpolates all active transitions to create a smooth and easy to use mixture of transitions.

I moved the colour change transition to the mouse event.

Exercise 9

Using bounce and sine easing methods, I was able to observe a clear difference in the way that d3 interpolates transitions. Not just in the size and movement – but also in the colour transitions.

Exercise 10

I added the bounce easing method to the shrink animation on *mouseout* and kept linear easing for the grow animation. This resulted in a visual effect similar to picking up the ball, then dropping it and watching it bounce.

Exercise 11

Adding size and colour transitions to the text element was easy with d3 and i added a similar bounce effect on mouseout.

Exercise 12

Chaining transitions to a third bar with duration and delay was simple, and worked in the same way as the previous two bars.

Exercise 13

Adding another delayed transition, I programmed the bars to all return to their previous positions after fully extending.

Exercise 14

Adding a colour transition to the bars was very simple as the colours can be chained right next to the height attribute. Both attributes are transitioned at the same time, using the same delay and duration.

Exercise 15

Adding the mouse events to the given example, I was able to see the chart gain interactivity when hovering over the bars. It helped highlight which bars were being displayed.

Exercise 16

Exercise 16 already had most of the code required to display the text above each bar, but there was a big in the code that stopped it working as intended. The *onMouseOut* function took in “d” and “i” parameters, treating them as data and iteration, but this isn’t what the function was being passed by d3. What the parameters should have been were “event” and “data”; no iteration argument is passed.

Fixing this issue and adding a transition for the text, the text was displayed over each bar when the bar was hovered over.

Exercise 17

Using the d3 scales to map the range to different colours, exercise 17 was very similar to those in Lab 1 where colour was interpolated based on value. Passing this gradient to the colour attribute, I was able to display the hover colours based on value.

Exercise 18

As the code is already quite well encapsulated, adding a third button with corresponding dataset was relatively simple.

Exercise 19

I changed each data variable from an array to an object. These objects then had a colour, and a data attribute, the data attribute having the original array.

With this extra colour attribute assigned to the data objects, I could then use d3 to automatically colour them. They also smoothly transition between each colour.

Exercise 20

Similarly to exercise 16, a text element can be appended to the SVG and centred above the bar. I copied over some of the code and added a transition on mouseover.

Exercise 21

I used d3 to add additional axes to the top and right. Similarly to Lab 1, this involved creating groups and translating them by SVG height and width.

Exercise 22

Using d3's *enter* and *exit* functions, I was able to create transitions for incoming and outgoing bars. The bars slide in from above and the other bars resize to accommodate the new ones. Conversely, bars grow to fill the space if other leave.

The axes also transition and adjust to new data. When changing the X or Y ranges of the displayed data, both axes slowly fade out with an opacity transition, followed by *remove*. Once the old axes have faded out, new ones are created and faded in.

Exercise 23

Using similar code to the bar chart example, I created a graph with adjustable transitioning axes based on the data ranges. I then used the transition function to change the d attribute of a path placed on the graph.

Transition automatically interpolates data arrays passed to the d attribute of paths.

Exercise 24

The output for the code is [16.2, 34.4, 5.2]. This is because, like numbers and colours, d3 can interpolate arrays. Interpolating between the first and second array with an interpolant of 0.2 returned a mixture of these elements, with high weighting towards the first array's values.

This works for any kind of array, as long as d3 knows how to interpolate the array's individual elements.

Exercise 25

The output for the code is brown. This is because, like previously mentioned, colours can be interpolated. I created elements with d3 to demonstrate this.

Exercise 26

I interpolated JS 'Date' objects in d3 the same way as I would colours, numbers or arrays. D3 has a large selection of values it's able to interpolate.

Exercise 27

In order to smoothly interpolate between the different data arrays for the pie chart, I created a variable to store the pie chart's last value. Each time the pie chart updates, it begins a transition interpolating between the arcs made with the last data, and the arcs made with the new data. This utilised the attrTween function to properly calculate and interpolate the curves.

If a regular attr transition is used instead of attrTween, each point of the curves linearly-interpolates between its last and new positions. This results in odd behaviour where arcs try to cut across the chart, instead of moving around it.

Another necessary addition was to pad the old and new data arrays so that they had the same number of elements. This is so that new elements are transitioning for 0 to X, not undefined to X.

Exercise 28

To create spheres of different colours, I first changed the data object from an array of numbers to an array of objects: each object containing a numerical radius, and string colour.

Using d3 to access the data array, I was able to colour in the circles according to their relevant colour.

Exercise 29

For this exercise, I decided to load in the data from an external CSV file, hosted on GitHub. As I've used this in many exercises before, loading the data and using it in the *.then* callback was easy to do.

Exercise 30

To change the sphere colours on hover, I added a CSS class, selecting *circle:hover*. I used a filter to lighten the colour, and a drop shadow to add glow behind each sphere.

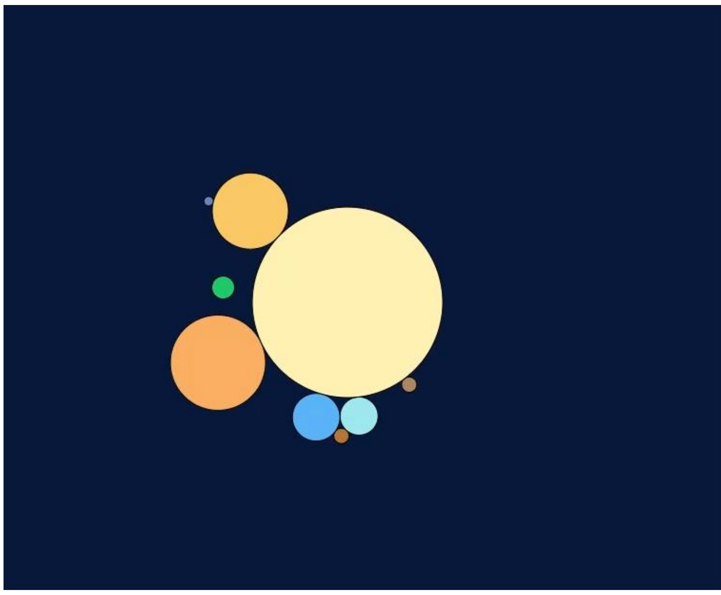
Exercise 31

As the spheres constantly move, it's not enough to update the text position in the mousemove event, as the mouse may stay still while the sphere moves.

To have the text always update correctly, I used a recursive function to 'tick' every few milliseconds, calling itself with *setTimeout*. The function checks to see if the mouse is over an SVG element and, if it is, places the text above said element.

The tick speed is configurable but works best when it's running at least 30 times per second.

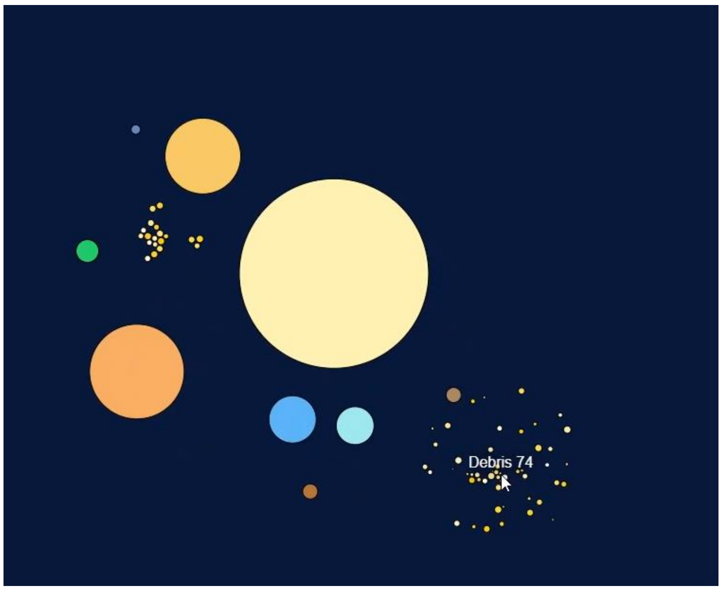
Exercise 32



For exercise 32, I decided to implement the ability to create small explosions with clicks of the mouse. The explosions create particle debris of varying size and colour, which slowly dissipate over time. They also generate a repelling force for a short period of time.

The SVG has a click event listener that algorithmically generates 50 particles of random, varying attributes. Each of the particles are appended to the data array with radii, colours, and positions. The Force graph is then updated with this new data using the *nodes* function.

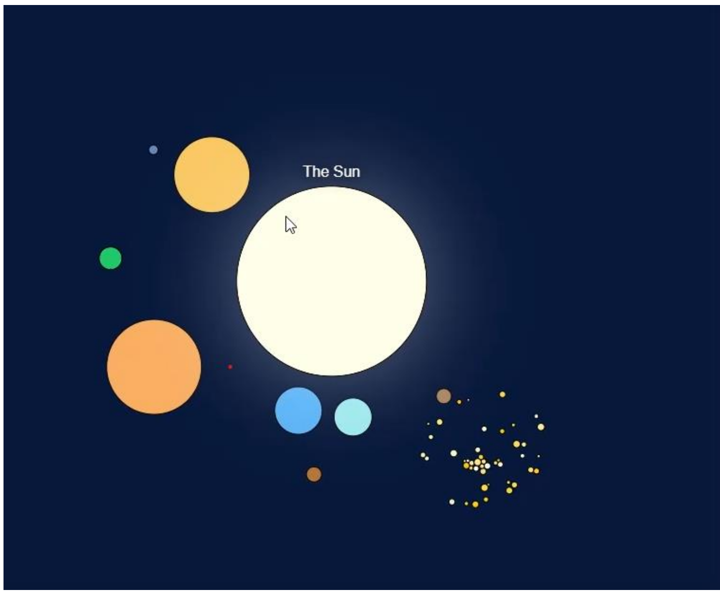
Once the data has been added to the array and the force graph updated, the new particles will be visually constructed using the *enter* function.



The debris particles each invoke a *setTimeout* which removes them from the data array a few seconds after the explosion.

Alongside creating the debris particles, the click event creates an X and Y force at the click origin. This is added to the force graph and the strength is gradually decreased by the tick function until it goes back to zero.

In order to have the simulation run indefinitely, I had to set the force graph's alpha decay to 0. This just means that the particles never slow down.



Like the other spheres, each of the debris have names. You have hover over them to see the names of each particle.

Because the particles decay, there are never too many on screen. This keeps the simulation's framerate high.