CSC 595 Lab 2

We will work on this in class, including a demonstration. The part at the end labeled "Assignment" must be completed and turned in on Dropbox by 07 October.

You should have the D3Demo.html downloaded from the D2L site.

In this lab we will briefly talk about using CSS to control style of SVG elements and then get on to making SVG elements programmatically.

Resources

JavaScript

Chapter 3 of Interactive Data Visualization (also other intro material)
JavaScript: The Good Parts (img from cscheid.net)
Effective JavaScript: 68 specific ways…
Mozilla Developer Network for really any SVG or JS reference you need (include MDN in google searches): https://developer.mozilla.org/en-US/

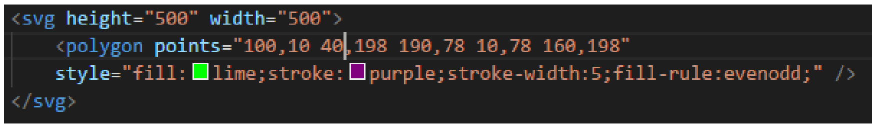
Using HTML and CSS to layout the page: https://www.w3schools.com/html/html_layout.asp

D3

D3 website: https://d3js.org/
Interactive Data Visualization
    Chapter 5: selection/insertion, method chaining and data binding
    Chapter 6: creating elements bound to data ('drawing with data')
Data Visualization with Python and JavaScript – Chapter 16 covers similar material

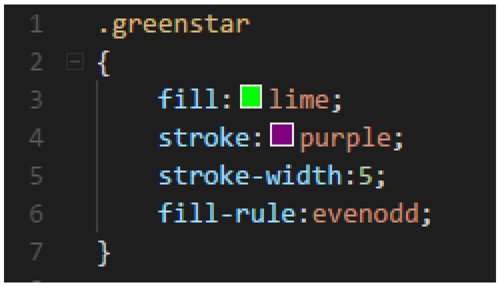
SVG with CSS

We start with a basic template that has a star made with an SVG path. Note that a lot of information about how this looks is in the style argument.



```
<svg height="500" width="500">
    <polygon points="100,10 40,198 190,78 10,78 160,198"
    style="fill:lime;stroke:purple;stroke-width:5;fill-rule:evenodd;" />
</svg>
```

Instead, we can make it possible to apply this style to lots of SVG items simultaneously (and therefore change them simultaneously if needed). Create 'D3Demo.css' and cut the text from the style argument and paste it into that file. Reformatting it a bit and enclosing it in *.greenstar*, we create a class called greenstar that we can apply to SVG elements by setting their class.

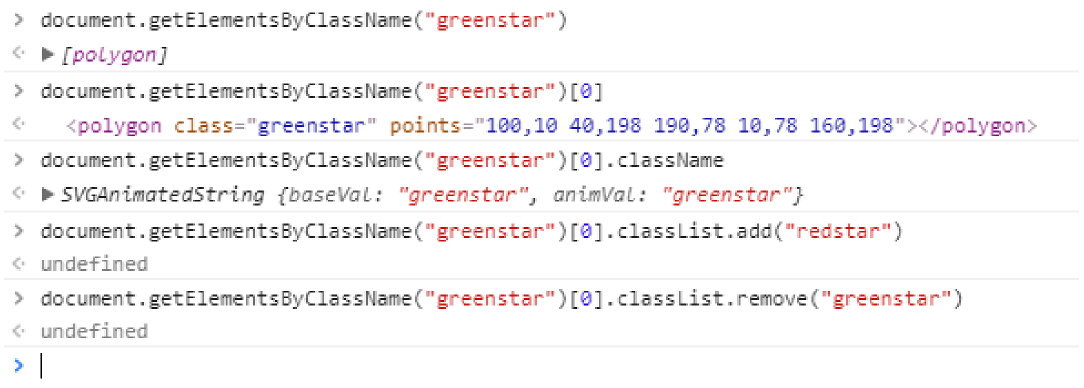


```
.greenstar
{
    fill:🟩lime;
    stroke:🟪purple;
    stroke-width:5;
    fill-rule:evenodd;
}
```

Create a new class that can be used for a red star and make another SVG canvas containing one star with this new class.

Using JavaScript to change the DOM (and thus SVG)

Now that we have an SVG element with a class, it's easy-ish to grab it and change it with JavaScript:



```
> document.getElementsByClassName("greenstar")
<· ▶ [polygon]
> document.getElementsByClassName("greenstar")[0]
<·   <polygon class="greenstar" points="100,10 40,198 190,78 10,78 160,198"></polygon>
> document.getElementsByClassName("greenstar")[0].className
<· ▶ SVGAnimatedString {baseVal: "greenstar", animVal: "greenstar"}
> document.getElementsByClassName("greenstar")[0].classList.add("redstar")
<· undefined
> document.getElementsByClassName("greenstar")[0].classList.remove("greenstar")
<· undefined
> |
```

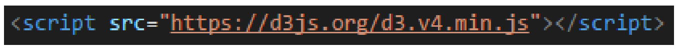
(Note that you can no longer select this item by its 'greenstar' class because it isn't there anymore.)

You get the feel for this. Similar commands can be used to add items to the DOM.

Using D3

There are resources above about D3 and you should use them to learn how to do this in much more detail. You will need to for the homework, in fact. For now, I give you a brief introduction so we can play with it. It does some lovely back-end things for you with respect to binding SVG elements directly to data (which is the point of Data Driven Documents (D3)), but first we're interested in generating elements with it and in that case it's basically just going to do what JS does, but more cleanly and conveniently.

First note that this HTML template we have includes an extra line at the top to import the D3 codebase from their site. You can also put a copy in your own folder to import and change the path accordingly. You want to use your own copy if you're worried they will make an update that breaks something you've coded.



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

Get an element and modify it

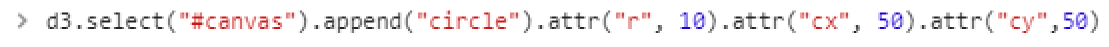
Now, just to make the point, here is the code to do what that JS did above:

```
d3.select(".greenstar").classed("redstar",true).classed("greenstar",false)
```

The methods are chained, so each call returns it result to be used again in the next modification. The *classed* call allows us to change the class of an object. There are a handful of these function, like *style* to change the CSS style properties and *attr* which works on general attributes.

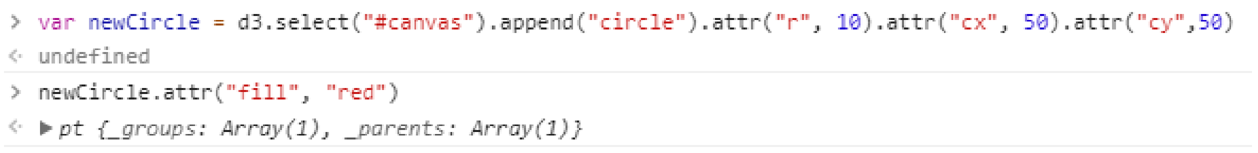
Add an element

Let’s add an element with D3. First, to simplify, let’s give our SVG canvas a unique identifier by adding the argument *id=”canvas”* to the SVG tag. Then we can refer to the SVG with selection strings as “#canvas”.



```
> d3.select("#canvas").append("circle").attr("r", 10).attr("cx", 50).attr("cy",50)
```

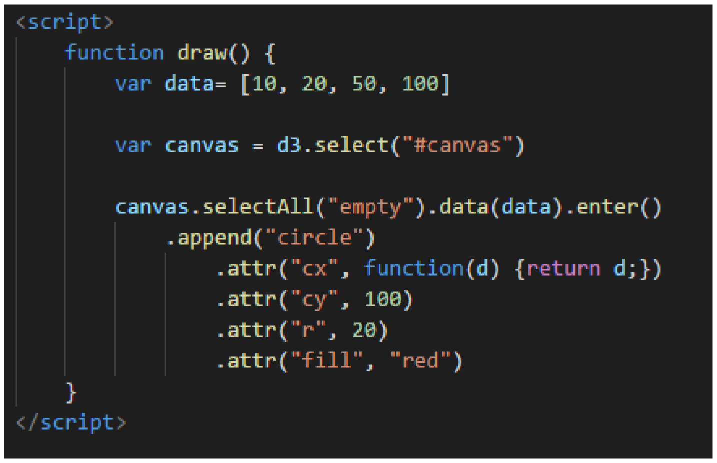
We can also save a reference to that and modify it later:



```
> var newCircle = d3.select("#canvas").append("circle").attr("r", 10).attr("cx", 50).attr("cy",50)
< undefined
> newCircle.attr("fill", "red")
< ▶pt {_groups: Array(1), _parents: Array(1)}
```

Add a set of elements bound to data

That’s all very well and good, but if we want to make this useful, we need to use data to create these. D3 has a way of letting us create a new element per data item. Since we are using a few lines at a time, we will put this inside a script tag in the header of the file. Also add *onload=”draw()”* to your body tag so that this new code gets run on startup.

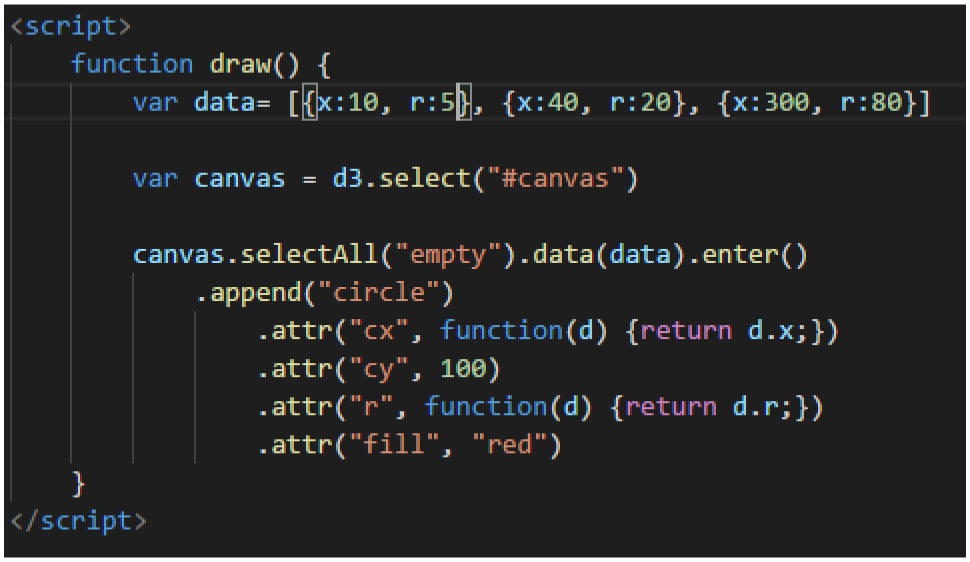


```
<script>
    function draw() {
        var data= [10, 20, 50, 100]

        var canvas = d3.select("#canvas")

        canvas.selectAll("empty").data(data).enter()
            .append("circle")
                .attr("cx", function(d) {return d;})
                .attr("cy", 100)
                .attr("r", 20)
                .attr("fill", "red")
    }
</script>
```

Believe it or not, that’s really the crux of it right there – you just used data to influence the properties of SVG elements, and that’s how we build visualizations. The *canvas.selectAll* starts us at the canvas element and makes an empty selection because there are no elements called ‘empty’. This tells D3 that we are strictly adding data, not modifying. The *.data* then specifies the data and *.enter()* tells D3 we’re ready to add elements based on this data. The *.append* tells it for each data element what to do –

append a circle. Then we use *.attr* calls to edit those added circles. The code will be applied to each one. To make the circles different, we make them depend on the data. Note that when we assign a value, the value can be supplied as a function. The function argument is bound to the current data item being processed, so we can use its information to change properties of the visualization.

Check this example out for use of slightly more complicated data. The data now specify the x location and radius of each circle.



```
<script>
    function draw() {
        var data= [{x:10, r:5}, {x:40, r:20}, {x:300, r:80}]

        var canvas = d3.select("#canvas")

        canvas.selectAll("empty").data(data).enter()
            .append("circle")
                .attr("cx", function(d) {return d.x;})
                .attr("cy", 100)
                .attr("r", function(d) {return d.r;})
                .attr("fill", "red")
    }
</script>
```

The only thing missing to use this to make, say, a bar graph, is that you need to calculate the locations of the bar in SVG space based on the data values. Basically you're doing the same math we did to decide element locations by hand in the last lab, but writing functions to do it for you. I leave actually doing this for the assignment. You'll want to look at the resources I pointed to because they run you through your first bar graph pretty well.

Assignment

The assignment component is straightforward – you will create a working implementation of bar, scatter and line graphs. This mostly follows chapters 6 and 7 in Interactive Data Visualization. As before, when drawing anything with code, I recommend drawing on paper first and thinking about how to calculate the coordinates.

1. Create a function that draws a bar chart given a set of data. The data will be an array of floating point values and can vary in length. This means you will need to make sure your graph automatically scales correctly.
2. Modify your bar graph code to draw a line instead of bars. Before you start doing it, think about what the difference actually has to be. The change in code could be quite small.
3. Finally, create a scatterplot version of the function. Now the data need to include both an x and a y value and you will need to figure out the right bounds in both directions. You'll find the code is still pretty similar, but it's a worthwhile exercise in thinking about how these work.