CSC 595 Lab 3

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 14 October.

In this lab, we will adopt a D3 example from the D3 Gallery. We will create a function allowing us to put the adopted visualization in multiple places with different data.

Resources

There are lots of great examples on the D3 Gallery by Mike Bostock: https://bl.ocks.org/mbostock

The documentation for D3 will come in handy: https://github.com/d3/d3/blob/master/API.md

Note that there was a recent change from D3 version 3 to 4. The update breaks some v3 things. It's important when looking at documentation, you look at the right version. We're using 4 and the link above goes to the docs for version 4.

Creating a Map in D3

This lab was original going to include making maps in D3 (as noted in the syllabus). However, we haven't covered maps yet and I think we could use this time well on the encapsulation of D3 examples. I recommend working through the map example from the Interactive Data Visualization for the Web book, which remember is available for free online. This book was written with D3 v3 (though a new edition of the book was release last month). Updated code is available in a repository, so you can check there was unsure why some things are going as expected.

Free online version of the book: http://chimera.labs.oreilly.com/books/1230000000345/index.html

Github repository of the companion code: https://github.com/alignedleft/d3-book

Adapting an example from the Gallery

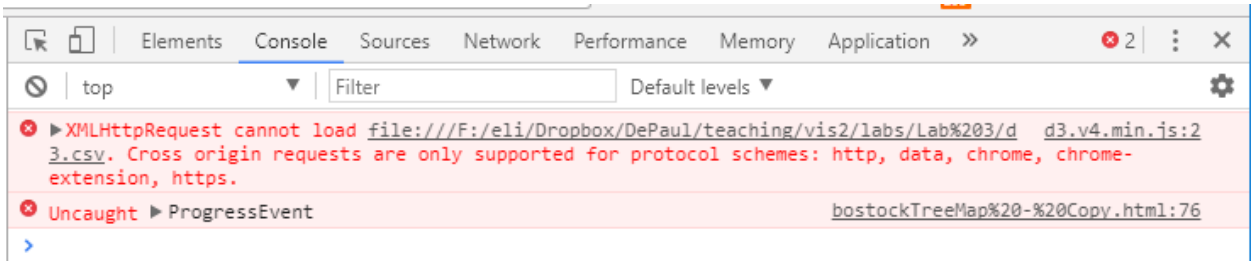
There is a great collection of examples on the D3 Gallery, many of the data visualizations we've discussed and a wide variety we haven't. The examples can serve as inspiration, and as a reference for how to do certain things. For the ones that are full visualizations of relevant types to our data, assuming licensing isn't an issue to your application, what we really want to be able to do is turn the visualization into an easily reusable component. Today, we'll take an example and turn it into a function that we'll use to make multiple copies of the vis on one page.

First let's just take the code from the page. There are two different files listed. We'll copy and paste them into text files and save them. These files don't actually use the full html template we discussed last time, so I took care of downloading the files and fixing them up that way. Download the html file and csv file from the D2L site. You will see I simply added in a header and moved content to the body tag, etc. Note that the body tag contains the script tag. In this configuration, it will just run all the code when it loads which is fine for us to start with. Since we are copying code from the web, we need to make sure the license is ok and that we understand everything in the code that we have copied, so I'll

walk us through this together (in the lab, not written here because it would be long and distract from the point of the lab).

Loading Data

Open this file in a browser and you will see nothing. Why? Take a look at the console in the developer tools to see the JavaScript errors:



In modern web pages and web apps, it is common practice to load data and content after the web page has loaded. This is done by sending another http request to the server ('another' meaning besides the one the browser sent to get the web page itself). This technology is generally called AJAX and typically the mechanism is the XMLHttpRequest (XHR) mentioned in the error message above. This is happening because in the code, they use *d3.csv* to get the data. While we have the csv file in our folder, it is not being served by a server and thus there is no response to the XHR request. We will create back-end servers later. For now, we choose a simple workaround: we will make a JavaScript file that includes the data as a variable declaration. Save a copy of the csv file and add a *.js* extension. Open that file and edit it to encompass the data in quotes and start the file with a *var d3CodeTreeCSV =* `

```
var d3CodeTreeCSVText = `size,path
,d3
,d3/d3-array
```

Note that is a back-tick, not a single quote. The button is in the upper left corner of the keyboard, shared with ~. This syntax, allowing multiple line quotes is new-ish in JavaScript, but for our purposes now we don't need to worry about compatibility with old browsers. Remember, this is a shortcut we won't need when using a back end.

To get access to the data now, we simple load this JS file as part of the header of our page. This JS code will be run, defining the variable in global scope. Here is the script tag and the console, verifying that the data string is loaded:

```
<svg width="960" height="960"></svg>
<script src="https://d3js.org/d3.v4.min.js"></script>
<script src="d3codetree.csv.js"></script>
<script>
```

```
> d3CodeTreeCSV
< "size,path
  ,d3
  ,d3/d3-array
  ,d3/d3-array/threshold
```

Drawing a TreeMap

Now it's time to focus on the code that draws the TreeMap.

```
d3.csv("d3.csv", function(d) {
  d.size = +d.size;
  return d;
}, function(error, data) {
  if (error) throw error;

  var root = d3.stratify()
      .id(function(d) { return d.path; })
      .parentId(function(d) { return d.path.substri
    (data)
      .sum(function(d) { return d.size; })
```

Note that it starts with that *d3.csv* code, and the code drawing the tree happens as the callback for the AJAX request to get the data. We just need to (1) factor the loading of the data out of the AJAX call and (2) wrap the whole thing in a function so we can use it multiple times.

The documentation for D3 tells us how d3.csv actually works, giving us what we need to write an equivalent bit of code starting from a string (like we have). Note that *d3.csv* actually take two arguments in this usage. The first is a function to restructure data, which is applied before calling the second argument, a function specifying what to do when the data is loaded (a callback). https://github.com/d3/d3-request/blob/master/README.md#csv

What we want to do is unfold that and use the CSV parsing on our string, with the same transformation function:

```
var treeCSVData = d3.csvParse(d3CodeTreeCSVText,
                              function(d) {
                                d.size = +d.size;
                                return d;
                              })
```

Reload the page and type *treeCSVData* in the console and you will see this:

```
> treeCSVData
< ▼ (463)
  [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…},
  ▼ [0 … 99]
    ▼ 0:
        path: "d3"
        size: 0
      ▶ __proto__: Object
    ▼ 1:
        path: "d3/d3-array"
        size: 0
      ▶ __proto__: Object
    ▶ 2: {size: 0, path: "d3/d3-array/threshold"}
    ▶ 3: {size: 0, path: "d3/d3-axis"}
    ▶ 4: {size: 0, path: "d3/d3-brush"}
```

Now we'll actually create the function that draws the visualization by wrapping the code that was the callback into a new function:

```
var drawTreemap = function(data) {
  var root = d3.stratify()
      .id(function(d) { return d.path; })
```

Note that we remove references to *error* which contained error flags from the XHR process, which we're no longer doing. If we were concerned with robustness, we would need to check if the CSV parser emits and errors and handle them.

Reload the page and type `> drawTreemap(treeCSVData)` into the console… pretty cool!

Making it Modular

We have a function that draws a TreeMap, but to use it modularly, we want to be able to draw multiple visualizations with different data at different places. We need to add a function parameter for the location to draw. A good balance of simple and flexible comes from providing a link to the DOM representation of the SVG where we want to put this.

```
var drawTreemap = function(svg, data) {
```

Note that the variable name *svg* is the variable used to represent the SVG object within this code already, so all we have to do is supply a new SVG as an argument and this should work fine. If you look at where the variable *svg* was defined before, you'll see it above.

```
var svg = d3.select("svg"),
    width = +svg.attr("width"),
    height = +svg.attr("height");
```

There is more than one way to replace this. The code is getting a reference to the one SVG tag that exists on the page based on the original HTML code. We could write code that inserts new SVG elements into the DOM (i.e. selecting *body* or some *div* and then *appending* an SVG tag; make note, you will do this in the assignment portion). The return value from doing that would give us the variable we need. We could also create the spaces for those SVG tag in our HTML, i.e. using two SVG tags with different IDs:

```
<svg id="d3tree" width="960" height="960"></svg>
<svg id="madeup" width="200" height="200"></svg>
```

Then getting the *svg* variable is:

```
var d3Svg = d3.select("#d3tree"),
    madeupSvg = d3.select('#madeup');
```

We can use either of those as an argument to our *drawTreeMap* function, and if you try it, you'll see that we actually can fill in two TreeMaps on the screen. The problem is that while we are passing in data, this function still only works on one dataset because it is using variables from outside the function's immediate scope.

Looking in the code we see the following variables are defined above *drawTreeMap*: *width*, *height*, *color*, *format*, *treemap* and *version*. The first two are used to calculate the sizes of TreeMap components. They depend on the SVG object being used, and thus must be moved into our function

and modified to use the *svg* variable. The *color* and *format* items are defining functions for how to choose colors and format numbers. These are not dependent on the data or SVG, but they are directly a part of the drawing functionality and are not shared for any other purpose so they should be copied directly into the function. Similarly *treemap* is defining a function that is used in drawing the TreeMap. It depends on *height* and *width*, but since those are coming into our function, this should be a simple cut and paste into the function. The *version* variable is the tough one. It is actually holding data used to draw the TreeMap, i.e. it is breaking our desire to keep the data separated out. The use of it is actually just in constructing the URL that these rectangles point to, so we will just remove it since it isn't a necessary part of our generic TreeMap. If we wanted to use this information, we could make a separate argument to the function that gave links for each cell, or change our data format to allow specifying the URLs.

I've included a fake dataset I created by hand. I encourage you to try the same and make this run on that data. A final html file with the product of this lab is included in case you have troubles and want to check against it.

Assignment

The assignment component is again straightforward – you will choose a couple (at least 2) visualizations from the D3 Gallery, or anywhere else you see D3 used and can get the code. Create a function for each one that draws the visualization with data and location supplied as arguments. Put these in separate files.

Create an HTML page that loads those two JavaScript files and uses those functions to draw multiple copies of each visualization on the page. Generate the SVG tags with code, allowing you to make a loop that adds a whole bunch of visualizations. You will need to make up at least a couple data files, or write code to automatically generate fake data. Just pay close attention to the format and refer to the visualization's code for any questions about what will work in the format.

You can do all sorts of fun things at this point. For example, you could load a dataset of plain, numeric CSV data and create a histogram of every variable and a scatterplot of every pair.