CSC 595 Lab 4

We will work on this in class, including a demonstration. The part at the end labeled “Assignment” must be completed and turned in on D2L by the posted deadline.

In this lab, we will make an example of coordinated multiple views. We start with an implementation of TreeMap (adapted from a previous lab) and an implementation of a bar chart (that I adapted from a D3 gallery example). Each one follows our previous structure for visualization code: a function draws a visualization to the screen given a location and data. In this lab, we will start from that point and wrap the functions into objects. Those objects will also have functions that expose message passing.

We will use the d3.dispatch capability to pass messages between visualizations. The key is that the object representing a visualization will have exposed functions that you wire together. You connect the ‘new selection’ output event from one to the ‘new data’ input on another. Whenever the first event triggers, the latter behavior happens.

Resources

Here is the documentation on d3-dispatch: https://github.com/d3/d3-dispatch

Here is a classic example of multiple coordinated views (scatterplot matrix) using D3:
https://bl.ocks.org/mbostock/4063663

And another D3 interactive visualization explicitly using *dispatch*:
https://bl.ocks.org/mbostock/5872848

You may want to take advantage of D3’s brushing library to handle selection from one visualization:
https://github.com/d3/d3-brush

A simple review of the core JavaScript ideas that enable this modularization and a few more bits like prototypes to fake inheritance (start about halfway down) (again thanks to Carlos Scheidegger):
https://cscheid.net/courses/spr15/cs444/lectures/week3.html

Starting Point

We have the TreeMap we made before and a bar chart I adapted from the D3 gallery. Note that each is in its own file and both files are imported by the main lab HTML file. The style information has also been pulled out into CSS files that are loaded by the main lab HTML file.

Modularizing

Because we want to be able to associate multiple functions with a given visualization, we need to associate them with objects. Right now we have a function to generate the visualization, but that is it. We will wrap those functions as the draw functions of objects.

TreeMap Data

The first step is actually to deal with the data mismatch between the bar chart and the TreeMap. Since we are sending messages from the TreeMap to the bar chart, we want the communication to be in a language they can both understand. Rather than make a bar chart that knows how to work with the TreeMap data, we should put that logic in the code that connects the two. In our previous implementation, though, the code for translating CSV data to a tree structure is actually part of the TreeMap code itself. We will pull that out.

In the main lab4 code, we create a function that does the data adaptation, pulling the code from the current TreeMap implementation (note change of variable *data* to *csvData*):

```
function csvToTree(csvData) {
    var root = d3.stratify()
        .id(function(d) { return d.path; })
        .parentId(function(d) { return d.path.substring(0, d.path.lastIndexOf("/")); })
       (csvData)
        .sum(function(d) { return d.size; })
        .sort(function(a, b) { return b.height - a.height || b.value - a.value; });

    return root;
}
```

The call to draw the TreeMap now changes:

```
var treeData = csvToTree(treeCSVData);
drawTreemap(treeSvg, treeData);
```

The TreeMap will now take a different type of data (need to change the variable name in two other places):

```
1   var drawTreemap = function(svg, treeroot) {
2       var width = +svg.attr("width"),
```

And finally, we will need this function to get the sibling data for a given node (copy/paste it):

```
function getSiblingData(node, searchStr) {

            // separate the piece of the search path we're looking for from the rest
            var firstSep = searchStr.indexOf("/");
            var remainder = searchStr.substring(firstSep + 1);
            var searchChild = searchStr.split("/")[1];

            // look at all the children to find the search string and recurse
            for (var iChild = 0; iChild < node.children.length; iChild++) {
                var child = node.children[iChild];
                var lastChildSplit = child.data.path.split("/").slice(-1)[0];
                if (lastChildSplit == searchChild) {
                    if (child.children) { return getSiblingData(child, remainder); }
                    else {
                         return node.children.map( (child) => ({label:child.id, value:child.data.size}) )
                    }
                }
            }
        }
```

Wrapping in Objects

Wrapping the drawing functionalities in objects is straightforward. We create a function (for scope reasons) that creates an object with the methods we want, and returns it.

```
var TreeMapVis = function() {
    var newTM = {
        drawTreemap: function(svg, treeroot) {
            var width = +svg.attr("width"),
                height = +svg.attr("height");

            var color = d3.scaleOrdinal(d3.schemeC
```

…

```
  };

return newTM;
```

Now we can run *T = TreeMapVis(); T.drawTreemap(…).* We want the same for bar charts.

We'll do the same for the bar chart.

```
TM = TreeMapVis();
TM.drawTreemap(treeSvg, treeData);

BC = BarChartVis();
BC.drawBarChart(barSvg, [{label:"a", value:1},
                         {label:"b", value:2},
```

Making Connections

Let's take a look at d3.dispatch together… (see docs)

I am going to have the TreeMapVis object have a dispatch object with a 'selected' event that can be connected to.

```
        cell.append("title")
            .text(function(d) { return
    },

    dispatch: d3.dispatch("clicked")
};
```

In the main code, I will then connect to this dispatch. When the signal is given, the main code will then redraw the bar chart.

```
BC = BarChartVis();
TM = TreeMapVis();
TM.drawTreemap(treeSvg, treeData);
TM.dispatch.on("selected",
               function(selectedPath) { BC.drawBarChart(barSvg,
                                                        getSiblingData(treeData,
                                                                       selectedPath)) }
```

The on click code for a given square in the TreeMap will call the function linked to in that dispatch.

```
var cell = svg.selectAll("a")
.data(treeroot.leaves())
.enter().append("a")
    .attr("target", "_blank")
    // removing links .attr("xlink:href", function(d) { var p = d.data.path.split("/
    .attr("transform", function(d) { return "translate(" + d.x0 + "," + d.y0 + ")";
    .on("click", function(d) {newTM.dispatch.call("selected", {}, d.data.path); })
```

We will also need to get rid of existing bars each time in the draw bar graph function:

```
// remove previous bars
svg.selectAll("g").remove();

var bar = svg.selectAll("g")
    .data(data)
    .enter().append("g")
```

Addendum

JavaScript is always being updated and it's a bit of a moving target because all the different browser companies can update it however they like for the implementation in their browser. However, there is a standard called ECMAScript that these companies can comply with for compatibility. The ECMAScript 6 specification (aka ECMAScript 2015) introduced *classes* and has been adapted by the major browsers. It's been a few years so it should be safe to use them. Here is a good overview of the syntax from W3Schools, a website with web development tutorials and references: https://www.w3schools.cn/js/js_classes.asp .

Here is an example of how this would affect our TreeMapVis class:

```
class TreeMapVis {
    constructor() {
        this.dispatch = d3.dispatch("selected");
    }

    drawTreemap(svg, treeroot) {
        ...
    }
}
```

Assignment

The assignment component is again straightforward – you will modularize and connect at least two visualizations. Before you start coding, think about how the two will connect to each other. Specifically, what interaction will happen in one and what will that cause in the other? What information needs to be passed from one to the other to make that happen? Feel free to use the visualizations you've made previously and turn them into modules. You can also adapt something new for it.