

Lab 8 | Introduction to D3

Outcome C: Build functional proficiency in openly available desktop and code-based geovisualization tools

This assignment is designed to give users hands on experience creating a map with [D3.js](#). D3 stands for “Data Driven Documents.” From their website:

D3.js is a JavaScript library for manipulating documents based on data. D3 helps you bring data to life using HTML, SVG, and CSS. D3’s emphasis on web standards gives you the full capabilities of modern browsers without tying yourself to a proprietary framework, combining powerful visualization components and a data-driven approach to DOM manipulation

Notably, D3 is not exclusively intended for mapping. But it is quite a popular framework for web-based visualizations, and naturally people wanted to make maps with it, so the developers have added support for mapping. This is good, but potentially a little challenging: It’s good because you’ll find TONS of examples on the web if you want to take the next steps and develop your final project (or some sort of more advanced interactive map) with this platform. You certainly can make some very complex mapping applications with it! It’s slightly more challenging because it isn’t like the other mapping APIs we’ve encountered in this class, where the library is designed exclusively to recognize and work with spatial data and has simplified functions built in. So while you could make a very simple interactive map with OpenLayers or Leaflet or ArcGIS API using just a few lines of code, you have to do a little more code cobbling here to get an equivalent application. You will see what I mean as we move through the following steps.

Please read the instructions carefully and do not simply copy and paste the code segments chronologically into a document. While you are *welcome* to copy code fragments directly out of this instructional document, you must ensure that you are putting them in their proper location and not duplicating any of the same sections – so pay attention to the context. Finally, publish your work to your GitHub Pages site as usual and submit to me through Canvas the URL to your *completed* work on your GitHub page.

Set up the Basics: A Street Map of Northeast Lincoln

Open a new blank page in your text editor. Let us start with a simple HTML file. Begin by writing the standard opening/closing HTML tags, head tags, and body tags.

- First, in the head, we give our document a title “Lab 8 | D3” and we also link out to the D3 library so that we can use features and functions from it.
- Also, create script tags within the body section; we’ll be adding our D3 code there!

```
<!DOCTYPE html>
<html>
  <head>
    <title>Lab 8 | D3</title>

    <script src="https://d3js.org/d3.v5.min.js" charset="utf-8"></script>
  </head>

  <body>

    <script>
      //we will be putting our D3 code here within body script tags

    </script>
  </body>
</html>
```

- Next, in the body’s `<script>` section, we’ll need to define the basic parameters of the map – how much room this graphic element should take up on the screen. We do this by adding variables to define the width and the height. We will append these attributes to our “SVG” (scalable vector graphic) canvas
- Essentially, we’re telling D3 that we want to create a graphical element in our page, and we want to set the “width” to our variable width, “height” to our variable “height”

- Finally, we'll tell D3 that we want to include geometry (appending "g" to our SVG set), and we'll use a variable called g to do this. Notably, this is just creating EMPTY geometry, because we haven't actually appended any data to it, yet. Just like we appended attributes to "svg" to set its definitions, we will need to eventually do this same with our data once we add it.

```
//width and height of the visualization
var width = 1000;
var height = 600;

//create the SVG
var svg = d3.select( "body" )
  .append( "svg" )
  .attr( "width", width )
  .attr( "height", height );

// Append empty placeholder g element to the SVG
// g will contain geometry elements
var g = svg.append( "g" );
```

- Although D3 doesn't natively understand a lot of GIS data, it can turn GeoJSON data into screen coordinates if it knows which projection to use for drawing these data. It has a handful of built-in projections, and there are plugins available for a wider variety of options. So you're not just stuck with Web Mercator! Yay!
- Let's set it to use an Albers conic projection – just because. Albers is one of the projections that doesn't require a special plugin, so we can call it up by using d3.geoAlbers() [see below]
- First, we create a variable for the projection and we want to center it on northeast Lincoln. Put this below the width and height definitions:

```
// Set Projection Parameters
var albersProjection = d3.geoAlbers()
  .scale( 2850000 )
  .rotate( [96.6327,0] )
  .center( [0, 40.8497493] )
  .translate( [width/2,height/2] );
```

- Scale sets the scale of the map (in a frustrating opposition to all things mapping/GIS, smaller numbers are actually larger scales, so 1 is the smallest).
- Because this is a conic projection, rotate sets the longitude of origin for the projection.
- Center sets the standard parallel (latitude) – so you use rotate and center together to “center” the map rather than just “center” as you would in a cylindrical projection

Question 1: Why are cylindrical projections more common in web maps?

This answer is worth 5 points. Answer this in a separate text file or word document.

- Finally, translate is just a pixel offset, usually to ensure that the projection center aligns with the viewing area you want. *Just use the code I’ve provided for this.*
- Finally, we need to set up something called a “path generator” or “Geo Path.” In D3, we use `d3.geoPath` to display data – this is a function that looks to the latitude/longitude coordinates from a GeoJSON feature and then changes them into screen coordinates based on the projections you set (*for those interested: the more technical explanation here is that the “d” attribute of an SVG path element, which D3 uses for generating shapes, needs a particular type of data string before it can draw shapes, and this process algorithmically calculates a suitable data string and returns the feature objects as a series of vector drawings*)
- Because we already have a projection picked out and identified, declaring the “Geo Path” is quite simple. We just create a variable and then assign it the projection parameters we’ve already defined above:

```
var geoPath = d3.geoPath()
  .projection( albersProjection );
```

Adding layers

Now that you have the projection defined and all of your basic settings arranged, it is time to load in the data layers. But first, *confirm that you're caught up*. The script in your **body** section should look something like this:

```
<script>
var width = 1000;
var height = 600;

// Create SVG
var svg = d3.select( "body" )
    .append( "svg" )
    .attr( "width", width )
    .attr( "height", height );

// g will contain geometry elements
var g = svg.append( "g" );

// Width and Height of the whole visualization
// Set Projection Parameters
var albersProjection = d3.geoAlbers()
    .scale( 2850000 )
    .rotate( [96.6327,0] )
    .center( [0, 40.8497493] )
    .translate( [width/2,height/2] );

// Create GeoPath function to turn lat/lon into screen coordinates
var geoPath = d3.geoPath()
    .projection( albersProjection );

</script>
```

NOW, for data, we'll use GeoJson layers *stored as variables in .JS files*. Because we aren't able to meet in person to troubleshoot potential conversion issues, I'm just providing these files to you from my site. You load in these files just like you have in the previous assignments – by adding a script in the head with a link to each file.

- Add a new script tag and link to my roads file:
<https://rshepard2.github.io/Lab9/data/lnkrds.js>

(It should look like this in your head section):

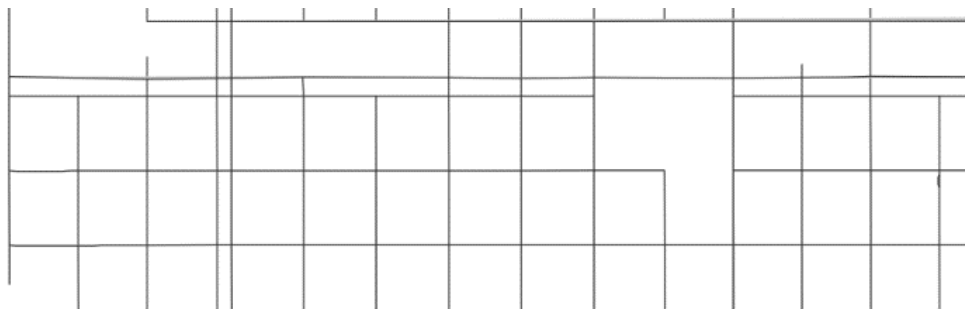
```
<script src="https://rshepard2.github.io/Lab9/data/lnkrds.js"></script>
```

If you click through to view the file I've provided, you will notice that the data are all already assigned to a variable, `var lnkrds`. Once you've added the script into your HTML file, you can use this variable just the same as if you wrote it all into your code.

- In D3, we need to “bind” the data layer to our geometry or “g” element here in order to get it to draw on the page. We do this by selecting and connecting to the features, appending them to the geo paths, then also defining the draw attributes (fill, colors, stroke color. Put this after your `geopath` variable section. It will look like this:

```
//select element; bind data; append attributes; apply
g.selectAll( "path" )
  .data( lnkrds.features )
  .enter()
  .append( "path" )
  .attr( "fill", "#ccc" )
  .attr( "stroke", "#333" )
  .attr( "d", geoPath );
```

At this point, you should be able to save and run (by opening with a web browser or double-clicking the HTML file in your directory to open a browser) your code to get a map. You'll see basic street lines with some strange gray areas (which are actually related to some topological issues from my roads dataset. Don't worry about it). It will just look, roughly, like this:



- Now that you've gone through the steps for adding a data layer, adding another layer is quite simple!

- For adding the second layer, we again just need to load in a .js file with GeoJson in it. Again, in the interest of time and simplicity, we'll just work from my own stash once again.
- Into the head section, add a link to my (heavily simplified) building shapefile layer. This is a GeoJson file of a bunch of residences:
 - https://rshepard2.github.io/Lab9/data/northeast_inkbldnsjs.js

Remember, you want to add `<script src="` and then the link, and then `></script>` just as you did above. I didn't write this out because I want you to be familiar with this process rather than just copying from the instructions.

You'll notice that I'm calling the variable (in the JS file) "Inkbldngsjson"

- SO, after you've linked out to the data to bring it into your HTML document, you again need to return to the script in the body, below your roads data definitions, and once again go through the process of identifying, selecting, and defining *how* you want those features to draw in your D3 geometry variable ("g"). This time, I'm also going to introduce a variable to "append" the buildings data to my list of geometry. I'm doing this because it's good practice, but also because I'm going to want to customize some user interactions on that layer specifically! First, add it:

```
var Ink_json = svg.append( "g" );

Ink_json.selectAll( "path" )
  .data( Inkbldngsjson.features )
  .enter()
  .append( "path" )
  .attr( "fill", "#900" )
  .attr( "stroke", "#999" )
  .attr( "d", geoPath );
```

Notice here that your variable name you use to refer to the dataset in your HTML document (here I use "Ink_json") does NOT have to match the GeoJSON variable name (which is "Inkbldngsjson") in the linked data file. HOWEVER, when you actually define the data for styling with D3 (in the .data part), you DO need to use the variable name that is used in your file. I chose two different names here to highlight this potential source of confusion.

Now at this point you'd have two different layers. Red buildings and gray streets, but they're not yet interactive for a prospective user – it's just a static map. And as you know, geovisualization is all about embracing the power of interactive mapping. There are potentially endless things you could do here, but I'll first introduce an interactive component that we've not yet used in previous labs, the "hover over." It's like a pop-up, but it is a user-triggered event that happens whenever a user's cursor hovers over a feature. In D3, as in most APIs, you can specify which layers are susceptible to this sort of event.

We will assign a hover over effect for the buildings layer only, so that users can see what year a house was built when they hover the cursor over a building footprint.

To do this, we FIRST create a class called "building data," and we tell D3 to populate this from the "RESYRBLT" field in the JSON data. If you are paying close attention to the code snippet, you'll also see that we're telling the script to put the data field into the text of "h2," an element on the page we haven't yet defined. Typically this is a "headline 2" page element, a subtitle, which should be larger than typical text but not the largest thing on a page. SO, **add the following script to your Ink_json section, just below the .attr("d", geoPath);** - *and make sure to eliminate the existing semicolon after .attr("d", geoPath);* See further down to confirm you have done this properly

```
.attr("class","buildingdata")
.on("mouseover", function(d){

//populate h2 with "built in" & the year built field from the json

d3.select("h2").text("Built in " + d.properties.RESYRBLT);
  d3.select(this).attr("class","buildingdata hover");
  })

//here, D3 to replace text interactively based on building data
  .on("mouseout", function(d){
    d3.select("h2").text("");
    d3.select(this).attr("class","buildingdata");
  });</script>
```


- Finally, even though we've added interactivity to the mapped area, we still need to set these styles in the head (for example, the H2 element is not yet defined anywhere in our code, so the hover over is writing to an object that we haven't even created). You can create styles with standalone CSS pages, or you can define the styles in the head. We are going to do the latter here to keep things simple.
- Put this style into the head section (where the style tags are currently empty)-

```
<style>
body {
  position: absolute;
  font-family: "Proxima Nova", "Montserrat", sans-serif;
}
h1, h2 {
  position: absolute;
  background: white;
  left: 10px;
  font-size: 1.3em;
  font-weight: 100;
}
h2 {
  top: 30px;
  font-size: 1em;
}
.hover {
  fill: yellow;
}
</style>
```

- Then, follow up in the BODY section with an actual H2 section so that this text can GO somewhere in the map. Put this right after the BODY tag -

```
<h1>Aging Northeast Lincoln Homes</h1>
<h2></h2>
```

Ultimately, not including the style tag text and the lines immediately above (defining where the H2 section goes), your Lincoln building section should now look like this (*new addition from last page is in darker text*) -

```
var lnk_json = svg.append( "g" );
```

```

lnk_json.selectAll( "path" )
    .data( lnkbldngsjson.features )
    .enter()
    .append( "path" )
    .attr( "fill", "#900" )
    .attr( "stroke", "#999" )
    .attr( "d", geoPath )
    .attr("class","buildingdata")
.on("mouseover", function(d){

//populate h2 with "built in" & the year built field from the json

d3.select("h2").text("Built in " + d.properties.RESYRBLT);
    d3.select(this).attr("class","buildingdata hover");
    })
//here, D3 to replace text interactively based on building data
    .on("mouseout", function(d){
        d3.select("h2").text("");
        d3.select(this).attr("class","buildingdata");
    });

```

This above section tells D3 to populate the subtitle (called H2 here) with information taken from the RESYRBLT field, when a user hovers over an object in the buildings layer. It also defines what to do when users move off an object (mouseout), which is to set it all back to blank `d3.select("h2").text("");`

Colorizing the Buildings – an Interactive Choropleth

- Now we will apply some sort of color style to all the buildings, based on the year of construction as well.
- To do this, we will tell D3 that we want to use a color ramp based on the RESYRBLT field that we've used to populate the hover over. This shows us that the same variable in the GeoJSON layer can be used to stylize objects as well as populate hover over effects (or pop ups).
- The first step here is determining what colors to use. Instead of determining breaks, we will just pick a "high color" and a "low color" for this ramp and let D3 determine the colors in between. We'll make a range from a very light pink hex `f9f9f9` for newer buildings to a strong raspberry color `bc2a66` for older buildings to keep it in the same spectrum. We first add these hex colors as

variables. Put this below the var width and var height declarations, within the script but near the top:

```
var lowColor = '#bc2a66'  
var highColor = '#f9f9f9'
```

Question 2: In a monochromatic choropleth such as this, how are audiences likely to interpret the darker shades? Why? How would you modify colors and/or shades if the title were “brand new lincoln homes” instead? Why?

This answer is worth 5 points. Answer this in a separate text file or word document.

- Then we will need to tell D3 what the minimum and maximum VALUES are for this ramp, also declaring these as variables. Remember that we’re using years, so we don’t put these in quotes, these are numbers. Put these variables anywhere in the BODY script. I just put them at the top with the others to keep the primary variables organized.

```
var minVal = 1900  
var maxVal = 2020
```

- Now, just below these, we’ll define the color ramp. You’re drawing here on D3’s ability to create a “linear scale” based on values, and we’re using the values (and colors) we just set to define that range:

```
var ramp = d3.scaleLinear().domain([minVal,maxVal]).range([lowColor,highColor]);
```

We just need to make one final change: we’ve got to go back to the attributes of the buildings layer and change the “fill” pattern.

- Find this section of code where you defined the color of the buildings:

```
lnk_json.selectAll( "path" )
```

```
.data( lnkbldngsjson.features )  
.enter()  
.append( "path" )  
.attr( "fill", "#900" )  
.attr( "stroke", "#999" )
```

- We don't want it all to be set to hex color #900, we want that color to come in dynamically from our dataset instead, based on the color ramp we created. So we will rewrite this line to draw upon the ramp, using the RESYRBLT field from our dataset to pick colors from that within the range of the color ramp:

```
lnk_json.selectAll( "path" )  
  .data( lnkbldngsjson.features )  
  .enter()  
  .append( "path" )  
  .attr( "fill", function(d) { return ramp(d.properties.RESYRBLT) } )  
  .attr( "stroke", "#999" )
```

Now save the HTML file and upload to your GitHub page. Save the link in your text file with the question answers so that you can include it with your submission to Canvas. This will give you a working example from which you can build future geovisualizations, and your successfully working code will count for 20 points of the assignment's 50 points (see rubric at the end of this document for the scoring breakdown and instructions for submitting your assignment).

Part II: Making your own map

For 20 points, you will create your own map in HTML with D3, using the code from this lab (and/or online examples in the module gallery) as a reference point. In your work, you will pick any topic of interest, drawing from data that you create or transform using QGIS. You should upload your data to your GitHub Pages account as .JS or .JSON files to consume them in your application, or send them to me in advance so that I can host them for you. You should use at least two layers, style them with colors or symbols of your choice, and make the layers interactive in some capacity (hover over, pop up, sliders, et cetera). Follow the rubric on the final page of this assignment. Upload your finished work to your GitHub Pages account and include the link to your webpage in your assignment submission.

Submitting Your Assignment and Grading

- Through Canvas, submit to me a word file or text response answering the questions posed in this assignment. Include links to your completed maps.

Rubric (50 points)

- 10 points** – Both assignment questions answered. Rubric:
 - Question 1.
 - Response identifies technical challenges of computer cartography influencing projection choices (2 points)
 - Response applies principles from projections unit (1 point)
 - Response answer situates cylindrical projections within the landscape of popular opinion and expectations in mapping (2 points)
 - Question 2.
 - Response applies principles from colors and map interpretations units (5 points)
 - 20 points** – Your Own D3 Map Application for Lab 8. Rubric:

| Score | Design | Code | Data | Interactive Features | Productivity |
|--------------------------|---|---|---|--|---|
| 0 (not shown) | No serious attempt to put map together in a legible format | No code provided/shared | No data used | No interactivity with layers | Application not completed |
| 1 (attempted) | Colors/symbology are extremely confusing and distort legibility, message has not been considered at all | Not close to anything functional; map does not load | Data are invalid or not loaded in properly | One interactive feature attempted | Application attempted, but highly incomplete by due date |
| 2 (basic) | Map is lacking refinements such as consideration of colors relative to subject, difficult to understand | Application loads, but has too many issues to be used as intended | Data linked from third parties, nothing unique | One interactive layer is present and works properly | Application incomplete, but nearly functional by due date |
| 3 (acceptable) | Some minor issues, such as opaque overlapping symbols; clashing layers | Some issues that impact some of the intended functionality | At least 1 Valid, unique dataset used | One functional interactive feature; a second attempted | Application mostly complete, with only minor errors or issues |
| 4 (advanced) | Symbology is appropriate, map is quickly understood by most audiences | Few/no errors or issues that impact functionality | All data are original or transformed by student | At least two layers are interactive in some capacity | Application fully completed on time, good use of time |

- 20 points** - Student submits guided code example outlined in exercise