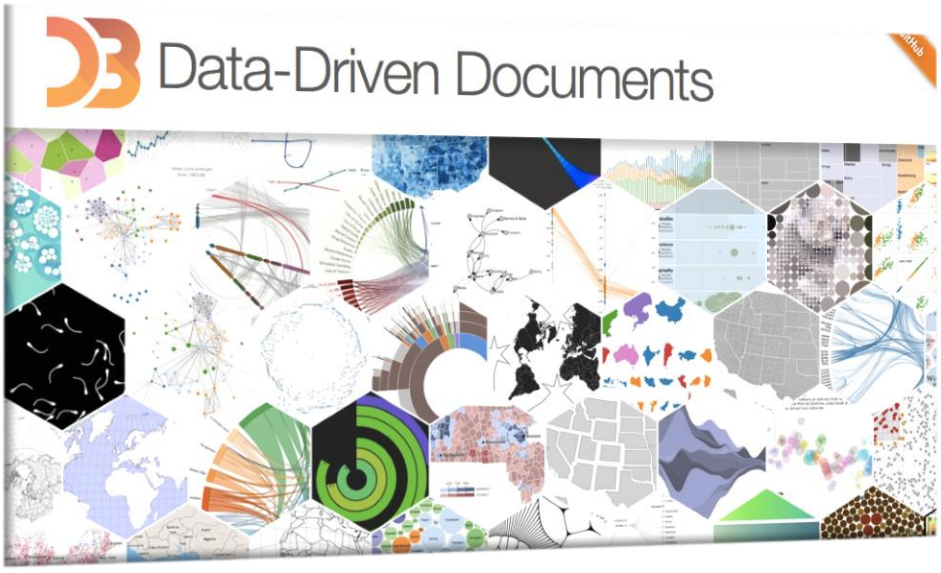


PORTFOLIO 2:

Creating with the d3 library

Group 8: Deeksha Juneja and Michael Rhodas

11/9/2015

Table of Contents

Introduction

The aim of our project is to demonstrate the functionality of the D3 library. The abbreviation D3 stands for Data-Driven Documents. This is a library of functionality that a programmer can import into their JavaScript project to visualize data. It has widely become known as "D3" and is referred to as "D3" by developers. Our project focuses on the data visualization capabilities of D3 to show how powerful this library is. Throughout the examples we created, each one demonstrates a different aspect of D3, and they turn out to be quite impressive. Additionally, we have created an animated home page to display our examples and organized the project as a website.

Background

D3 has the unique ability to bind data to Document Object Model, or DOM, elements. This provides all sorts of interesting implementations of the library. With the power to manipulate DOM elements through D3's advanced functions, you can create useful tools such as graphs, trees, animation, and other visual renderings. If you are in need of dynamic, and interactive data visualizations in your HTML files, D3 would be an extraordinarily useful tool for you.

Interaction with Materials

HTML

HTML, or HyperText Markup Language, was an essential part of our project. Starting with the overall organization of our project, HTML was used to separate out all of our examples into different webpages. It was also used for the beautifully animated homepage that is the face of the project and links together all of the examples.

An example of organizing our HTML files can be seen in our data_binding.html file.

```
<h1>Fundamentals of Data Binding in d3.</h1>
<h2>
    Example of the enter() method.<br><br>
    <p>
        <a href="enter.html">Open Enter Example</a>
    </p>
</h2>
```

In this file, there are three different headers that all label a different demonstration of a D3 function. From this file, you can click on the links and view the examples as separate pages so there is no need to scroll through a list of the examples. This reduces clutter and provides a clearer demonstration of the example's main point.

DOM

The Document Object Model is how a good portion of HTML is laid out. It is the tree structure of code and tags that defines a model. The DOM consists of tags to identify their main object or purpose. These are called DOM elements. For example, if you wanted to have a header in your webpage, you would specify a header DOM element "<h1>" and then put your text in with it while closing it with a "</h1>." This same format is used for any other element that exists. Put the element name in the carrots and end it with the forward slash version with the same element name.

Not all elements require a closing carrot structure because the tag itself can have a function. One example is the
 tag. This means to insert a line break at the current part of the DOM. We used this a lot in our HTML code to format the examples properly so the user could have a better experience.

Along with the type of DOM element you use, there are certain characteristics you can set. There are characteristics called attributes which define an aspect to the DOM element. For example, if you wanted the body of your html file to be black, you would type "<body bgcolor = "black">" in the tag. You can have multiple attributes in your DOM element that you can set that are added in a chain before the end carrot. Depending on the element, these attributes can have quite impacting effects on the DOM.

Scripts

The "script" DOM element allows us to run JavaScript files through the HTML file. In our project, we created many different JavaScript files that had to be run in certain locations in order to maintain organization and usability throughout our project. The D3 library is a JavaScript based file, so obviously there needed to be a way to run these functions.

To import the library, we use the DOM element in the following way.

```
<script type="text/javascript" src="d3/d3.js"></script>
```

This is in each example that uses the D3 library. By putting this into an html file, it loads the entire script from the specified "src" attribute. In this case, it is a disc location to the JavaScript file. In order to use the functions

in D3, this line had to be placed in the HTML document to define it's functions for use in other scripts later on.

The type attribute defines what sort of script to expect. JavaScript is the only type used in our project. We can also run our examples this way by placing our JavaScript source code location path into the "src" attribute in the same fashion.

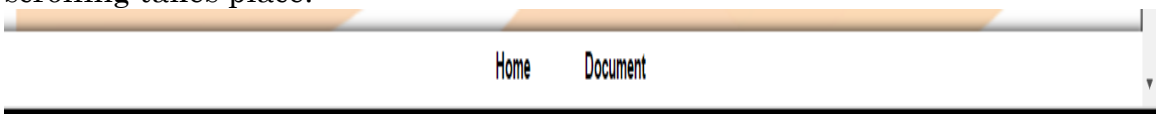
Lastly, our project uses smaller scripts that you can write in between the script tags as plain text. This way you do not need a separate, small JavaScript file in your web project directory. The script will be executed as written in the HTML file.

JavaScript

Manipulating DOM elements is a key way to implement a web project with some action in it. JavaScript is a scripting language that allows this sort of manipulation to be done quite easily. With JavaScript, you can search for DOM elements in your HTML file and change them to create a more interactive feel on your webpage. Maybe you want the background color to change every five seconds. JavaScript is what you are going to want to use. This is how we are able to show our visualizations in the project's JavaScript files.

Navigation Bar

The vision for the navigation bar was to have two things on it – home and the link to the document. We wanted our navigation bar to be consistent on all pages. It is placed at the bottom of the page and is placed in a way that it appears to be on top of the window screen due to the shading effect given in the CSS file. The navigation bar is consistent on all the pages of the webpage. The navigation bar editing can be found under the div navigation. The main.css file has a hover variable for navigation bar text. The color of the text changes on mouse hover. I have two classes nav-down and nav-up. This helps me decide which class to remove and which class to insert when the scrolling takes place.



```
if (st > lastScrollTop && st > navbarHeight){
    // Scroll Down
    $('.navigation').removeClass('nav-down').addClass('nav-up');
} else {
    // Scroll Up
    if(st + $(window).height() < $(document).height()) {
        $('.navigation').removeClass('nav-up').addClass('nav-down');
    }
}
```

jQuery

jQuery was not used a ton in our project, but it did come in handy for three of the examples, enter.js, exit.js, and update.js. All three examples used jQuery in the same way. Remember that when using jQuery, it is important to import the jQuery library with a script tag.

This is the code that uses jQuery in each file:

```
<script type="text/javascript">
    $(document).ready(function() {
        var count = 0;
        $("#enterCode").hide();

        $("#buttonEnterCode").click(function() {
            count++;
            if (count % 2 != 0) {
                $("#enterCode").show();
            } else {
                $("#enterCode").hide();
            }
        })
    });
</script>
```

This segment of code is used to toggle the JavaScript file as plane text for the user to analyze during the demonstration. It is connected to a button the will hide or show the tag with the specified id. In this case, it is tied to a paragraph DOM element given the id “enterCode.” There is also a counter that will determine if the element is to be hidden or shown on the click. jQuery allowed use to find the button and the paragraph and set a function to it to toggle the hiding action.

JSON

JSON notation, or JavaScript Object Notation, is a way to store an object in a unique way. This way, if you have a complicated object, you can send it around to your functions with ease. You can also define a separate .json file to use in multiple scripts if you do not want to hard code it.

You will have the prototype names defined in the JSON variable for the object, and then you can assign values to those object attributes. In the tree and cluster example, a JSON variable is used to represent a tree data structure. It is the first variable defined in the scripts. D3 then has built in methods to take these JSON variables, parse them, and perform operations on them.

CSS

Cascading stylesheets or CSS is the most important formatting tool in our project. In total we have four css files. Three D3 examples use the format.css file whereas the other two examples use main.css file. The main.css, demo.css and style.css are used for creating the website.

The format file uses the borders and header one. The main file does most of the formatting for the webpage and the other two examples. It defines the style of the headers, body, divisions, sections, the background image in the first section, it adds the test shadowing to the headers, it formats the navigation bar for which it uses webkit, and it also formats the ninjascroll.

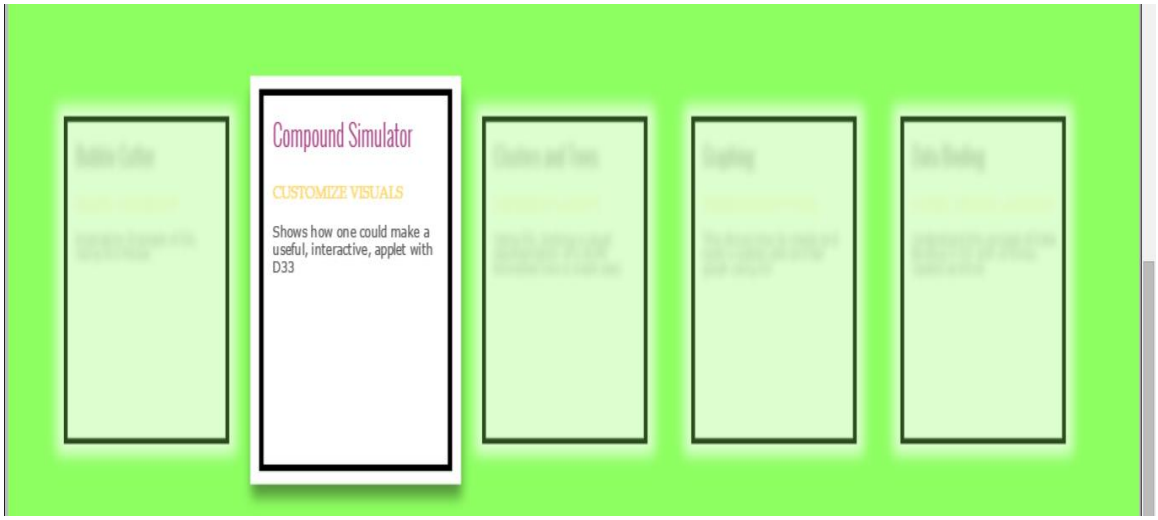
The demo.css and style.css file is majorly for the second section of the webpage. It contains .ib-container and each of the elements of it. All these elements are formatted in these two files. These two files are used to create the blur effect seen in webpage section two. The article.blur and article.active sections of the style.css file are used to highlight the element we hover on and blur all the other elements. Additionally, box-shadow, webkit, moz and o transitions were used to enhance the look of webpage. It is added to each of the container article element of style.css.

Complex Topics

CSS and JavaScript Blurring

The blur effect on the second section of the web page was created by .blur filter in CSS library. The blur effect was applied to all the components of the section starting with the container, heading, header and paragraph consecutively. To aid the blur effect on the container, box shadowing was used along with WebKit transform and moz transform to add two dimensional and 3-dimensional effect. Additionally to facilitate the movement

of the box when hovered upon, we used o transform facility. It helped box move forward when hovered upon by the user. Once we set how the boxes would individually react to the mouse hover we moved on to the article. We added text shadows and opacity to all the text as necessary. There were different styling options used to create the shadow and on hover - blur - effect. For instance color and opacity. Opacity was used to add to blur effect of the background as and when necessary.



This was particularly difficult due to the lack of examples available for it. Most examples just showed a picture of it and not how to implement the transformation. Moreover, when we just implemented the blur on a certain part then either nothing would get blurred or the whole document would get blurred. It would also override the active example and make it blur. Then I had to separately specify the opacity of the active elements as 1.

```
.ib-container article.active h3 a,
.ib-container article.active header span,
.ib-container article.active p{
    opacity: 1;
`
```

It seems like an extremely small two lines of code. Writing the code was not difficult, understanding that this code was required was the complex part of the topic.

This blurring was accomplished perfectly through the use of javascript. I removed the blur class and added the active class on mouseenter. When the mouse left the element then it would exit class active blur.

```
$articles.on( 'mouseenter', function( event ) {

    var $article    = $(this);
    clearTimeout( timeout );
    timeout = setTimeout( function() {

        if( $article.hasClass('active') ) return false;

        $articles.not( $article.removeClass('blur').addClass('active') )
                 .removeClass('active')
                 .addClass('blur');

    }, 65 );

});

$container.on( 'mouseleave', function( event ) {

    clearTimeout( timeout );
    $articles.removeClass('active blur');

});
```

SVG

SVG or scalable vector graphics are used in all the D3 examples. What they are exactly are two dimensional graphics with the ability to animate and provide more interaction for visual elements. The D3 library uses SVG elements heavily because you can add, remove, and edit these elements in real time with its functions. To begin creating an SVG based plane, you set a canvas to work with and for there you can append other visual elements to be displayed through the HTML DOM.

In our project in the chemistry compound problem we used append to add an SVG element to the body. We also added the “mousemove” function through SVG to the body. Here d3.mouse(this) finds the current coordinate of mouse and if it is in the area we wish to apply attribute to and applies the attribute to it when reasonable.

In the collision program, SVG for circle is then linked with slice method. The slice method allows to extract a string and create a new string. Therefore, this helps us add new circle at the start of collision program. Next the attribute function returns radius of the total structure after respective addition of individual circles by calling function and then style function is used to add colors according to radius. The color is taken in percent gradient and is returned respectively to style attribute.

Elements

There are quite a variety of elements that can be added to the SVG canvas. All of these elements possess attributes to style the element to how you need it to look. For example, the x and y coordinates, the fill color, width, height and other stylistic characteristics for shapes are all attributes that can be set to change the view of the element.

Here is a list of the elements we used in our examples in the SVG canvas:

<rect> - This defines a rectangle.

<circle> - This defines a circle.

<g> - defines a group of elements that can be added to in order to perform operations of all its members.

<text> - This defines a text element to be placed on the plane.

<defs> - This is used embed definitions to use later on in the code.

<linearGradient> - This is used to define a type of linear gradient.

<stop> - In our project, this is used to define gradient colors.

<path> - This is used to create paths connecting two items, in our case, nodes.

Groups

Groups are an essential part of manipulating these SVG canvas elements. The tag used to make a group is <g>. With a group, you can add other elements into the group. After the group is created, you can perform actions on all of the elements in the group at once. It is essentially like creating a mini plane on the SVG plane. In our tree example, a group is made to place all the visual elements in a group to ensure the correct placement of the elements.

```
var canvas = d3.select("body").append("svg")
    .attr("width", width)
    .attr("height", height)
    .style("border", "solid 2px #000000")
        .append("g")
            .attr("transform", "translate(" + (width + pad) / 8 + ", 0)");
```

By doing this, we prevent a lot of offsetting of positions later on in the creation of the visual representation of the tree.

The D3 Library

When you start working with D3, you immediately notice some similarities with jQuery. The magic of D3 is being able to create functions using data that can set attributes and styles to the HTML DOM instead of having them be constant. Interjecting HTML text into the document dynamically is a powerful way to create some very interesting visuals.

The D3 library is filled with extremely powerful functions. Every time you call functions from the library, it is doing a whole lot of work for you. This makes it easy to perform complex animations and actions because the format of D3 helps organize these operations into nice looking chained methods.

Attribute and Style Manipulation

D3 allows you to edit attribute and style information related to the DOM element being inserted. This is a fundamental action in D3 because you are able to manipulate these attributes built into more complicated algorithms. All you have to do to set an attribute is to use the .attr(attribute, value).

For example, if you wanted to change a selected circle's radius, you would type a line like .attr("r", 5). This will either set the circles radius to 5 or dynamically change an existing circle to the radius. If the element already exists, then the user will see the radius change on the screen.

Data Binding

Probably the most useful and cool aspect of D3 is the ability to bind data to these visual producing functions. Let us say that there is an array of data that needs to be displayed as a bar graph. D3 has the power to take all of your data, and create SVG DOM elements to be inserted onto the webpage.

Here is a code segment from our bar graph example:

```
//Labels
svg.selectAll("text")
     .data(data)
          .enter()
               .append("text")
                    .text(function(d) {
                         return d;
                    })
```

In this snippet, the selectAll() method will determine what type of DOM element will be inserted. In this case, it is the text for the bars on the bar graph. After selecting the elements we want to manipulate, we want to

create an algorithm that will apply the data from the array to the manipulation.

The variable d seems to appear out of nowhere. This variable is what represents the data that the selection is bound to. By creating an anonymous function after data is bound with the parameter d, within the D3 library, the correct value of d is passed up through the functions.

Now we need a way to apply these visualizations depending on the existing selected DOM elements and the one's that do not yet exist.

Look at the following code from the enter.js example:

```
//Declare a data array with two elements.
var data = [10, 20];

//Make a svg canvas.
var canvas = d3.select("body")
    .append("svg")
        .attr("width", 700)
        .attr("height", 400);

//The explicitly defined circle.
var explicitCircle = canvas
    .append("circle")
        .attr("cx", 250)
        .attr("cy", 250)
        .attr("r", 100);

//After the delay, the data is bound to the circles in the DOM and
//enter creates more circles because of the data the circles represent.
setTimeout(function(){
    var bindCircles = canvas.selectAll("circle")
        .data(data)
        .enter()
            .append("circle")
                .attr("cx", 500)
                .attr("cy", 250)
                .attr("r", 100)
                .attr("fill", "red");
        }, 2000);
```

The data array declared has two elements. There is one explicitly defined circle named explicitCircle. After two seconds pass, the data binding is initiated. When it is initiated, D3 selects all circles that exist which is only one. When the enter() method is applied after data binding, D3 knows that when it runs out of existing elements, to use the method chain after enter() occurs. So the second element causes that second circle to be appended. The original black circle is there and then a red circle appears after the data bind.

What if you want to update the circle that already exists? It is very simple to do this. Here is the data binding method from the update.js example:

```
setTimeout(function(){
    var bindCircles = canvas.selectAll("circle")
        .data(data)
        //This is where the current elements are updated.
        .attr("fill","blue")
        .enter()
            .append("circle")
                .attr("cx", 500)
                .attr("cy", 250)
                .attr("r", 100)
                .attr("fill", "red");
        }, 2000);
```

As you can see, the data binding code is almost identical to the enter.js function. There is one added line before the enter() method is executed. This is the area where you can update existing DOM elements. If you look at the example run, you will see that after two seconds, the explicitly defined circle will turn from black to blue. This is because of the update. Then, the enter() method finishes with creating the second circle.

The last situation is when you have more existing DOM elements than the number of data elements in the array. Refer to the exit.js example, here is the data binding function from it:

```
setTimeout(function(){
    var bindCircles = canvas.selectAll("circle")
        .data(data)
        .exit()
            .attr("fill", "green");
        }, 2000);
```

Keep in mind that in this example, we have two explicitly defined circles before this data bind as well as only one element in the data array. First, the existing DOM elements are matched to the number of data items and the update section is executed. After the data runs out, D3 checks if there are more existing elements you would like to edit, make a call to the exit() method. This tells the D3 engine to execute the method chain following this exit method. This is why you see the second circle change to green.

Gradients

Making a gradient of color is a good way to bring some extra features to your D3 project. Examine the following code snippet:

```
//Insert a gradient definition.
var gradient = svg.append("defs")
    .append("linearGradient")
        .attr("id", "gradient")
        .attr("x1", "0%")
        .attr("y1", "0%")
        .attr("x2", "0%")
        .attr("y2", "100%")
        .attr("spreadMethod", "pad");

//Style the colors for the gradient.
gradient.append("stop")
    .attr("offset", "0%")
    .attr("stop-color", "red")
    .attr("stop-opacity", 1);

gradient.append("stop")
    .attr("offset", "100%")
    .attr("stop-color", "blue")
    .attr("stop-opacity", 1);
```

To create a gradient using D3, we simply append the correct DOM elements to define the color gradient you would like to use. First, we need to make a definition to define the name of the gradient. We append the <defs> DOM element to do this and then add a <linearGradient> to that DOM element. By setting the "class" attribute of this linearGradient, you define the name to be called later to use the gradient. The x1, y1, x2, y2 attributes are used to set the direction of the gradient. In this case, the gradient is vertical.

To set up the colors of the gradient, you append the <stop> tags. The gradient will know to transition from the first color defined, to the second color defined.

This is the legend created using the gradient to label the color's meaning:

Scales

D3 lets developers create a scale that will be able to remap data into the proper value needed for proper visual display. These scales can be used with numbers, colors, and other scalable variables.

Let's look at the scatter plot example and how it uses scales.

```
//Set a linear scale for the x values using the d3 scaler.
var xScale = d3.scale.linear()
     .domain([0, d3.max(data, function(d) { return d[0]; })])
     .range([pad, width - pad * 2]);
```

In this example, we needed the x and y coordinates of the points to be correct, correlating to the size of the SVG canvas. We make a variable to hold the scale and set its domain and range to be mapped to. Based on the scaling of the SVG plane, a scale will be made to adjust positions of the points.

```
//Make a x axis.
var xAxis = d3.svg.axis()
     .scale(xScale)
     .orient("bottom");
     //.ticks(6);
```

After defining the domain and range, we make the axis and set the scale and orientation to it.

```
//Add X axis
svg.append("g")
    .attr("class", "axis")
    //Moves the axis into place.
    .attr("transform", "translate(0," + (height - pad) + ")")
    //This adds all the elements to the group.
    .call(xAxis)
    //This will add a label to the x axis.
    .append("text")
        .attr("class", "label")
        .attr("x", width-pad)
        .attr("y", -6)
        .style("text-anchor", "end")
        .text("Students");
```

After there is a defined axis, we can append it to the graph with the correct data labels because of the call function.

Attaching Handlers

If you want to add more interactive elements to the web page, D3 makes attaching handlers quite easy. By using the on(event, action) method, you can attach handlers to your DOM elements.

For example, in the scatter plot example, you can move your mouse over the points and they will enlarge to show your mouse is over it. It will return to normal size when you move the mouse off the element. Then, if you click the circle element, it will be selected by changing the color to blue. This is all done by attaching a handler with on() when creating all the points. It is shown in the following code snippet:

```
.on("mouseover", function() {
    d3.select(this)
    .attr("r", 9);
})
.on("mousedown", function() {
    d3.select(this)
    .attr("fill", "blue");
})
.on("mouseout", function(d, i) {
    d3.select(this)
    .attr("r", 7);
});
```

Tree and Radial Cluster Layout

Another powerful tool in D3 is the ability to use different layouts. For the tree example, we use a tree layout and set its size.

```
// Start a d3 tree layout.
var structure = d3.layout.tree()
    .size([width, (height / 2) + 100]);
```

To gain the positions of the nodes, D3 has a nodes(JSON) method that will return a list of all nodes in the given JSON file. This includes all name and children information. This is an incredibly powerful method because it would be very tedious to parse the JSON file yourself.

After defining the node positions, we need to create paths linking the nodes in the correct way. Here is how you define the type of paths to be used.

```
// Set the path type to link the tree nodes.
var diagonalPath = d3.svg
    .diagonal()
        .projection(function(d) {
            return [d.y, d.x];
        });
```

This will use diagonal paths that are defined by the nodes positions. Now that we know what type of paths we want to use, we need to get a list of all the paths needed. Just like the nodes, D3 has a wonderful function called links(nodes) that we can use. This will return an array of all the paths needed from the JSON nodes. Then all you have to do is use this array of links to append all the path DOM elements to the SVG canvas. By just binding the path array with data(paths), that data will be applied to all the appending paths until there are no more paths to place.

```
//Get a list of positions as a node and append them to a group.
var nodePositions = canvas.selectAll("g.node")
    .data(nodes)
    .enter()
    //Make a group to be able to rotate the plane depending on the point.
    .append("g")
        .attr("transform", function(d) {
            return "translate(" + d.y + "," + d.x + ")";
        });
```

This code segment will take all the nodes and append a group to them. Then, using the data from the JSON variable, the positions are translated to the correct spots. All that is left to do is use this position data to place circles and text at all the nodes.

Now, converting this tree into a radial cluster is not an arduous task. There are only a few changes needed. Instead of a tree layout, we use a cluster layout. Next, the type of path needs to be declared radial by adding a radial() method call to the chain defining the path like so.

```
//This will set the type of paths to link the nodes in a radial fashion.
var diagonalPathRadial = d3.svg.diagonal
    .radial()
        .projection(function(d) {
            return [d.y, d.x / 180 * Math.PI];
        });
```

Last, the math to position the nodes and paths needs to be correctly converted from a "linear" approach to a "radial" approach. Instead of translating positions, you rotate them. Instead of using an x-y coordinate system, think of it more like a polar graph with some parts simplified.

Force Layout

The force layout in the compound simulator program is used to create a force to hold the structure on the webpage. It works in the following way. We first define the dimension of the visualization. Next we define the data or nodes. Nodes helps us decide where to place the node before the force starts showing its effect on the webpage. Since we have no node, it means we are using a default position which is the center of the webpage dimension we have defined earlier as a variable. The link distance helps us give program the link between two nodes. The charge property creates a predefined charge on the node. This helps the program to decide how apart relatively each node will be from all nodes. Negative charge value indicate repulsion and vice versa.

Bloom's Taxonomy

Creating

The idea was to create a web platform to portray some visualizations using D3. Our first idea was to just have some examples of D3. Being unfamiliar with D3 and having limited exposure to JavaScript posed a great challenge. We decided to do five examples in total and were successfully able to complete those. Once we had that accomplished, we came back to the creating phase and created a webpage to have all the examples and the write up document easily navigable.

Evaluating

We had to evaluate each step we took in the design of the examples carefully. We spend a day to understand the working of svg and then implementing it. We assessed of how the charge and the force function of D3 worked and how they interacted with the other components. D3 has 4 categories with different colors set up. After looking at all 4 of those we decided to create our own category called the category 100. So we modified the D3 file to create a category and use it in our code. In addition to that, we had to evaluate the workings of a tree format and how that works with reading a JSON file.

The second round of evaluation started when we decided to put everything on a webpage. We first talked about a basic layout of it. We decided to have a fancy navigation bar and scroll bar. Moreover, we wanted to explore some cool formatting features using CSS as well. We decided to try those with headings and blur.

Analyzing

The analyses came in when we had to decide how to divide the work amongst our self. We both were very pumped up about using D3. But we decided that one person should do three D3 examples and the other person should work on 2 examples and the website. This decision was based on the respective interests in the fields and worked very evenly.

The first step was create the D3 examples. A lot of brainstorming went into that. There are a lot of resources available on the internet for D3. We looked at plenty of those resources and first understood how to use D3. The second step was to create basic D3 examples to experiment how well we understood D3. The third step was to decide which examples of D3 we wanted to implement for portfolio 2. We came up with 5 examples we thought were not only interesting but could also we used by people in real life. For example, the graphs are a tool used by everyone today to represent data. Also, the compound creator could be implemented for chemistry students.

After our D3 implementation the major task to do was create a webpage. The glowing effect used in our headings was quite tricky to come up with. Another major set that we faced was in blurring done on the second scroll. While implementing the blurring, it had to be done separately for everything. If blurring was applied for just one thing, it would blur everything. A detail on that is given in the complex topics of blurring with CSS. We also decided to implement a navigation bar. The special thing about navigation bar is that it independent of the scrolling window. We also wanted to make a one sweep scroll or a ninja scroll.

Applying

Application of all aspects of the project put together took us much more time than we thought it would take us. We messed up a lot of times while we were doing the D3 examples. A lot of times it was hard to figure out the mistakes we had made. But other than that, it was smooth sailing. This project gave us a chance to use our knowledge of JavaScript and HTML, and in turn it extended our knowledge of JavaScript, D3, CSS and HTML.

One of the most tricky applications was to make the navigation bar stay put when we scrolled that is, making the navigation bar present for every window.

Implementing the blur with the CSS was also extremely hard. We had to implement the blurring feature for each and every component. Each component required a different blurring configuration as well. For some the blurring needed to be turned on and for others it needed to be turned off.

In D3, a tricky implementation was the one of mouse function. We made a new D3 category called D3.scale.category100 to have our own set of colors for

our example. Another cool thing we implemented with D3 was the example of data binding using enter, update and exit. It is a 3-in-1 example.

Since D3 is such a powerful library, it took a lot of analysis of the library to figure out exactly what was going on behind the sense. Having to revisit and analyze examples was a part of the learning process. Then to build the code up our selves took applying all that knowledge to our creative process.

Conclusion

In conclusion, we have a pretty good website running and working. It has a few components to it. First being D3, we have five data visualization examples using D3. Second is the webpage itself. The webpage is again divided in three components, first of which being the navigation bar. The navigation bar takes you “Home” and to the “Write up (this document)”. The second being the scrollbar, which is a smooth one swipe scroll. The third being the CSS component of the website design. Most of the formatting on the webpage was done using CSS for example the blurring and the highlighting. Overall, we have been able to accomplish our purpose of portraying the abilities of D3 and learning D3. It is clear that this is a very powerful library that makes very complicated procedures easy.