Lab 7: Interaction & Transition 1 (Pre Lab)

Wijaya, Mario edited this page 16 days ago · 1 revision

Learning Objectives

After completing this lab you will be able to:

- Create event listeners using D3 selections
- Write event callback methods
- Transition SVG elements with D3
- Understand how Enter, Update, Exit and transitions work together
- Understand the JS keyword this and the scope of execution

Prerequisites

- Update your *starter code repository* using one of the following methods:
  i. From the GitHub Desktop app click `Sync` on the top right
  ii. Open a command line prompt. Navigate to the repository directory, for example `cd ~\Development\CS4460-Spring2018\Labs` and run command `git pull`.
- You have **read Chapter 10** in D3 - Interactive Data Visualization for the Web by Scott Murray

Additional Reading

- Working with Transitions by Mike Bostock
- Animations and Transitions by Jerome Cukier
- Getting beyond hello world with d3 by Jerome Cukier
- W3 Schools - HTML Event Attributes
- General Update Pattern III by Mike Bostock
- D3 Easing Functions by Katsumi Takano

HTML Events

There are HTML events happening all the time. There are a lot of events out there other than `mouseover` and `click` though. Here's a list of some notable ones:

Mouse Events

- `click` - Fires on a mouse click on the element
- `dblclick` - Fires on a mouse double-click on the element
- `mouseover` - Fires when the mouse pointer moves over an element (start to hover event)
- `mouseout` - Fires when the mouse pointer moves out of an element (end to hover event)

Drag Events

- `drag` - Script to be run when an element is dragged
- `dragstart` - Script to be run at the start of a drag operation
- `dragend` - Script to be run at the end of a drag operation

Window Events

- `load` - Fires after the page is finished loading
- `resize` - Fires when the browser window is resized

Clone this wiki locally

`https://github.gatech.edu`

Keyboard Events

- `keydown` - Fires when a user is pressing a key
- `keyup` - Fires when a user releases a key

Basic Event Listeners

An event listener is an anonymous function that listens for a specific event on a specific element or elements. There a lot of different ways to specify an event listener.

Last lab we saw how the `change` event fires when a user *changes* the selected `option` of a `select` form element. We are going to start with the example from last week to learn how we can use other HTML events.

If you remember, we had a simple bar chart that updates based on applied filter `select` from element:

As a refresher, this bar chart is based on the following English letter frequency data:

letter	frequency
A	0.08167
B	0.01492
C	0.02782
D	0.04253

The `select` element has an attribute for an `onchange` listener. The listener listens for a change event. When that happens, the listener function is executed. So every time the user changes the selected option, `onCategoryChange` will get executed:

```
<select class="custom-select" id="categorySelect" onchange="onCategoryChanged()">
      <option selected value="all-letters">All Letters</option>
      <option value="only-consonants">Only Consonants</option>
      <option value="only-vowels">Only Vowels</option>
</select>
```

This gives us a nice interactive chart, but what if we want to move our filter controls into the `<svg>` itself. It would make sense to have the three filter options for the chart reside inside of the chart somewhere. So let's make some `.filter` elements within the svg:

```
<g transform="translate(10,20)">
    <text dy="-0.3em">Filters:</text>
    <g class="filter selected" value="all-letters">
        <rect height="20" width="65" rx="3" ry="3"/>
        <text x="4" dy="1.3em">All Letters</text>
    </g>
    <g class="filter" transform="translate(73)" value="only-consonants">
        <rect height="20" width="98" rx="3" ry="3"/>
        <text x="4" dy="1.3em">Only Consonants</text>
    </g>
    <g class="filter" transform="translate(177)" value="only-vowels">
        <rect height="20" width="75" rx="3" ry="3"/>
        <text x="4" dy="1.3em">Only Vowels</text>
    </g>
</g>
```

And with the following css styling:

```
.filter {
    cursor: pointer;
}

.filter text {
    font-size: 11px;
    font-weight: normal;
    fill: #333;
}

.filter rect{
    fill: #ddd;
}

.filter.selected text {
    fill: #fff;
}

.filter.selected rect {
    fill: #fe902b;
}
```

We get the desired look for our filters:

Now we are going to add `onclick` event listeners to each of these `.filter` groups. Using groups is a nice way to create a composite that jointly serves as an interactive element. Be careful though - group elements do not, by themselves, trigger any mouse events. The reason for this is that `g` elements have no pixels! Only their enclosed elements—like `rect`s, `circle`s, and `text` elements have pixels.

So we can specify event listeners on group elements as long as they have child elements with pixels. And we can do that two-different ways, we can use the standard HTML event listener attributes:

```
<g class="filter selected" onclick="updateChart(this.getAttribute('value'))"
value="all-letters">
```

Or we can use D3 to create a listener for a `click` event occurring on any `.filter` groups like so:

```
d3.selectAll('.filter')
    .on('click', function(){
        updateChart(d3.select(this).attr('value'));
    });
```

Either method allows us to filter the data on `click` events. But wait, what is `this` in the event listener context? `this` refers to the DOM element that the event was called on. In this specific case, `this` refers to the `g.filter` element that was clicked.

However, be careful of how you use `this`. The reserved keyword `this` points to many different objects depending on where you use it:

> In JavaScript, as in most object-oriented programming languages, `this` is a special keyword that is used within methods to refer to the `object` on which a method is being invoked. The value of `this` is determined using a simple series of steps:

> 1. If the function is invoked using `Function.call` or `Function.apply`, `this` will be set to the first argument passed to call/apply. If the first argument passed to call/apply is null or undefined, `this` will refer to the global object (which is the window object in Web browsers).
> 2. If the function being invoked was created using `Function.bind`, `this` will be the first argument that was passed to bind at the time the function was created.
> 3. If the function is being invoked as a method of an `object`, `this` will refer to that `object`
> 4. Otherwise, the function is being invoked as a standalone function not attached to any object, and `this` will refer to the `global object`.

> *Source: JavaScript Basics by Rebecca Murphey*

We also want to change the class of our `g.filter` elements to be `selected` or not `selected` whenever a click occurs to change the styling of the element. We can achieve this with the D3 `.classed(<classname|string>, <add|boolean>)` method:

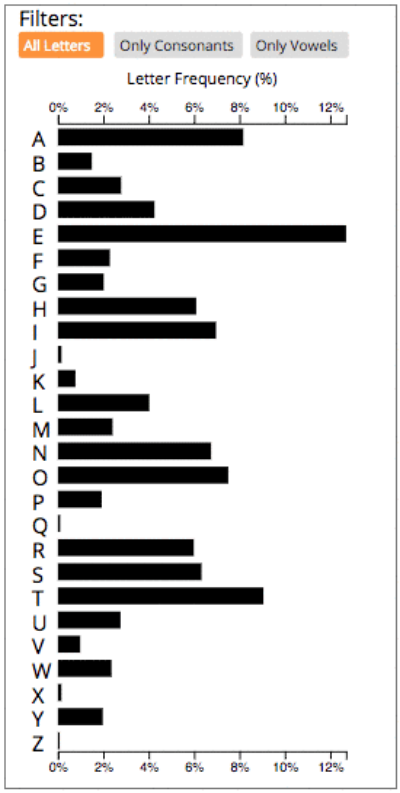
```
d3.selectAll('.filter')
    .on('click', function(){
        // Remove the currently selected classname from that element
        d3.select('.filter.selected').classed('selected', false);

        var clicked = d3.select(this);

        // Add the selected classname to element that was just clicked
        clicked.classed('selected', true);

        updateChart(clicked.attr('value'));
    });
```

With this added block of code we now have our desired interactive filters:



Responsive and Interactive Charts

Binding Event Listeners

JavaScript uses an event model in which events are triggered by things happening, such as new input from the user, provided via a keyboard, mouse, or touch screen. To make our pieces interactive, we define chunks of code that listen for specific events being triggered on specific DOM elements. You saw previously how we used the `.on()` method to create an event listener:

```
d3.selectAll('.filter')
    .on('click', function(){
        updateChart(d3.select(this).attr('value'));
    });
```

This binds an event listener to the `g.filter` elements. The listener happens to be listening for the click event, which is the JavaScript event triggered when the user clicks the mouse on any of the `g.filter` elements.

Its important to note that JavaScript's events are always called on a specific element. So the code just shown isn't activated whenever any click occurs; it is run just when a click occurs on any of the `g.filter` elements.

You could achieve all this with raw JavaScript (as we showed above), but D3's `on()` method is a handy way to quickly bind event listeners to D3 selections.

Making your visualization interactive is a simple, two-step process that includes:

- Binding event listeners
- Defining the behavior

Using our letters bar chart example, we will go over how to bind event listeners to your d3-selections and use data callbacks when defining the behavior.

Defining behaviors with data

Similar to how we used `.selectAll('.filter').on(...)` to bind click events for elements already in the SVG, we can use `.on()` when appending data-joined elements. For example, here is our existing code that creates our bars, to which I've simply tacked on `.on()`:

```
var barsEnter = bars.enter()
    .append('g')
    .attr('class', 'bar')
    .on('mouseover', function(d, i) {
        // This will run every time the user starts to hover over the bar
    })
    .on('mouseout', function(d, i) {
        // This will run every time the user stops hovering over the bar
    });
```

When defining the anonymous function, you can reference `d`, or `d` and `i`, or neither, just as you've seen throughout D3.

This is a quick and easy way to verify your data values, for example:

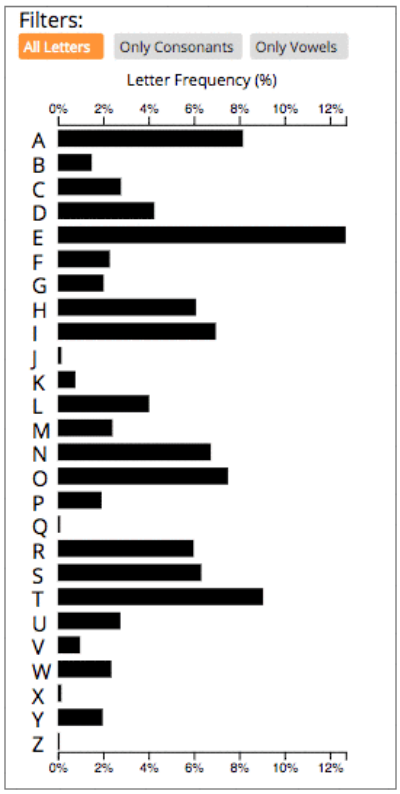
```
    .on('mouseover', function(d) {
        console.log(d);
    });
```

Hover to Highlight

Highlighting elements in response to mouse interaction is a common way to make your visualization feel more responsive, and it can help users navigate and focus on the data of interest.

A simple hover effect can be achieved with CSS alone—no JavaScript required! The CSS pseudoclass selector :hover can be used in combination with any other selector to select, well, that same thing, but when the mouse is hovering over the element. Here, we select all `rect` s and set the fill to orange on hover:

```
rect:hover{
    fill: orange;
}
```



CSS hover styling is fast and easy, but limited. There's only so much you can achieve with :hover.

Using the `.on()` methods we created before for `mouseover` and `mouseout` we can add more functionality to hover events. In the below code, we will change the class of the group to also include `hovered` - which will highlight the `text` and `rect` of the hovered `g.bar` element. We will also append a new text element with the `frequency` value for that bar.

```
.on('mouseover', function(d) {
    // Use this to select the hovered element
    var hovered = d3.select(this);
    // add hovered class to style the group
    hovered.classed('hovered', true);
    // add a new text value element to the group
    hovered.append('text')
        .attr('class', 'value')
        .attr('x', xScale(d.frequency) + 10)
        .attr('dy', '0.7em')
        .text(formatPercent(d.frequency));
})
.on('mouseout', function(d) {
    // Clean up the actions that happened in mouseover
    var hovered = d3.select(this);
    hovered.classed('hovered', false);
    hovered.select('text.value').remove();
});
```

And we've added the following CSS styling:

```
.bar.hovered rect {
    fill: #fe902b;
};

.bar.hovered text {
    font-weight: bold;
};
```

Which gives us the ability to inspect the details of each letter frequency:

Transitions

When coupled with interaction, it's a very useful way to give feedback to the user. What has changed since their last command? If what's on screen animates from one state to another, it's obvious, it stands out and it makes sense. Or, when showing any form of real-time data, animation is pretty much required.

Animation can bring focus on the important things as a chart loads. Our vision is very sensitive to movement, so using these introduction transitions sensibly helps a lot to ease the effort required to get the right information off a chart.

Transitions are a limited form of key frame animation with only two key frames: start and end. The starting key frame is typically the current state of the DOM, and the ending key frame is a set of attributes, styles and other properties you specify. Transitions are thus well-suited for transitioning to a new view without complicated code that depends on the starting view.

Given start and end key frames, how do we get from here to there? To perform a smooth animated transition, D3 needs to know how to interpolate—or blend—from a given starting value to its corresponding ending value. The d3.interpolate method determines an appropriate interpolator by inferring a type for each pair of starting and ending values.

Let's use the bar chart example to create transitions when the bars are re-positioned. To first transition the `transform` property when the bars are repositioned we will add the following:

```
bars.merge(barsEnter)
    .transition()
    .duration(600)
    .attr('transform', function(d,i){
        return 'translate('+[0, i * barBand + 4]+')';
    });
```

Which gives us the following transition:

Making a nice, super smooth, animated transition is as simple as adding one line of code:

```
.transition()
```

Without `transition()`, D3 evaluates every `attr()` statement immediately, so the changes in `transform` happen right away. When you add `transition()`, D3 introduces the element of time. Rather than applying new values all at once, D3 interpolates between the old values and the new values, meaning it normalizes the beginning and ending values, and calculates all their in-between states.

duration(), or How Long Is This Going to Take?

So the attr() values are interpolated over time, but how much time? It turns out the default is 250 milliseconds.

Fortunately, you can control how much time is spent on any transition by adding:

```
.duration(1000)
```

The `duration()` must be specified after the `transition()`, and durations are always specified in milliseconds, so duration(1000) is a one-second duration.

The actual durations you choose will depend on the context of your design and what triggers the transition.

Home