Data visualization

COSC 480B

Reyan Ahmed

rahmed1@colgate.edu

Lecture 13

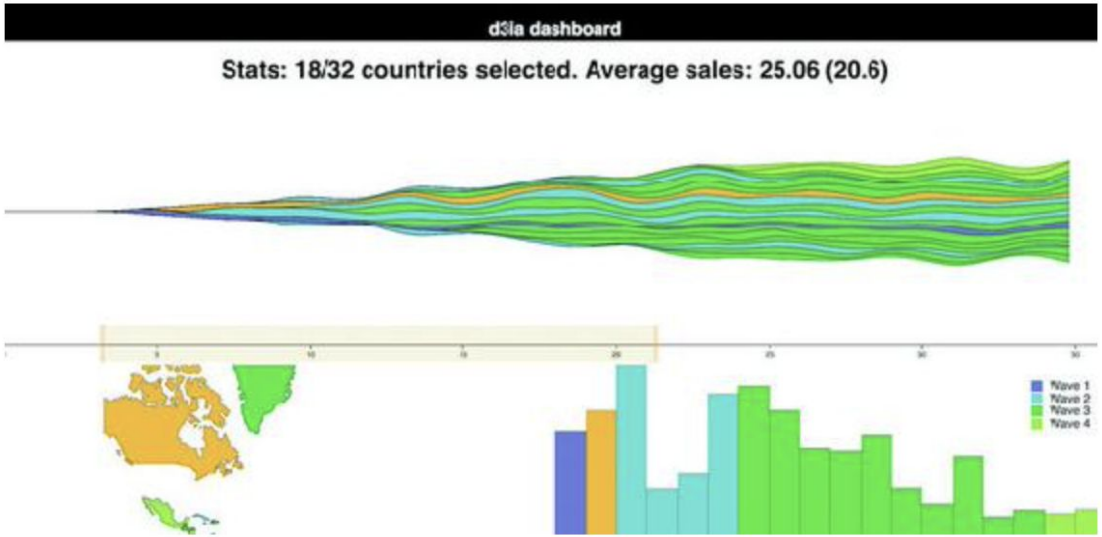
Interactive applications with React and D3

Overview

- Using D3 with React
- Linking multiple charts
- Automatically resizing graphics based on screen size change
- Creating and using brush controls

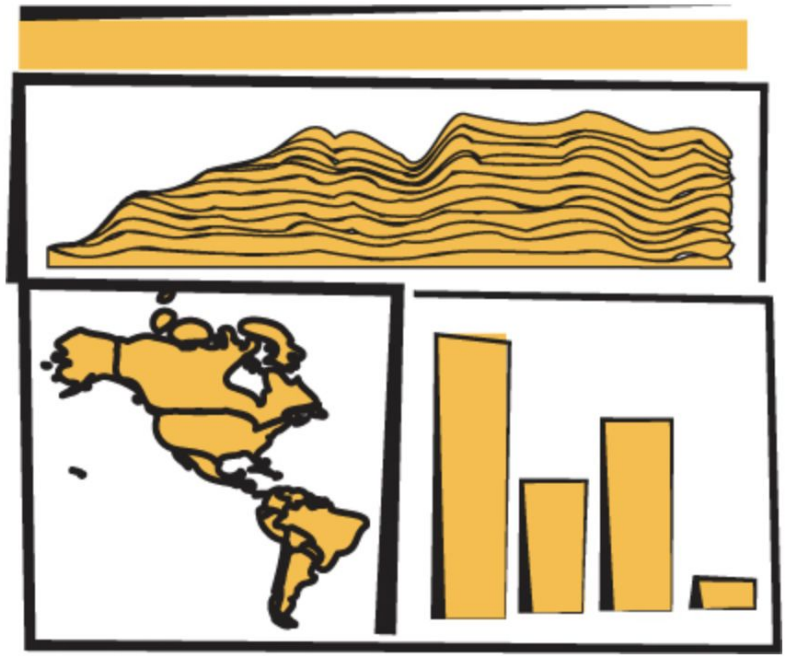
Overview

Throughout this chapter, we'll build toward this fully operational data dashboard, first creating the individual chart elements (section 9.1), then adding interactivity (section 9.2), and finally adding a brush to filter the data (section 9.3).



One data source, many perspectives

A sketch of a dashboard, showing a map, bar chart, and stacked area chart that display our data



One data source, many perspectives

Dashboard CSS

```
rect.overlay {
  opacity: 0;
}

rect.selection {
  fill: #FE9922;
  opacity: 0.5;
}

rect.handle {
  fill: #FE9922;
  opacity: 0.25;
}

path.countries {
   stroke-width: 1;
   stroke: #75739F;
   fill: #5EAFC6;
}
```

Getting started with React

- Uses both javascript and HTML known as JSX
- In OS X you can open your terminal window and run the following commands:

```
npm install -g create-react-app
create-react-app d3ia
cd d3ia/
npm start
```

Getting started with React

The default page that create-react-app deploys with

Getting started with React

In your project directory run the following to install the d3-scale module:

```
npm i –SE d3-scale
```

Do the same thing with the following modules:

```
react-dom
d3-shape
d3-svg-legend
d3-array
d3-geo
d3-selection
d3-transition
d3-brush
d3-axis
```

Getting started with React

Simple react code:

```
const data = [ "one", "two", "three" ]
const divs = data.map((d,i) => <div key={i}>{d}</div>)
const wrap = <div style={{ marginLeft: "20px" }}
    className="wrapper">{divs}</div>
```

1. To put javascript we have to use curly braces
2. Use camel case for CSS (marginLeft instead of margin-left)
3. To set CSS class use className instead of class

Traditional D3 rendering with React

BarChart.js

```
import React, { Component } from 'react'
import './App.css'
import { scaleLinear } from 'd3-scale'                    1
import { max } from 'd3-array'                            1
import { select } from 'd3-selection'                     1

class BarChart extends Component {
  constructor(props){
    super(props)
    this.createBarChart = this.createBarChart.bind(this)     2
  }

  componentDidMount() {                                   3
    this.createBarChart()
  }

  componentDidUpdate() {                                  3
    this.createBarChart()
  }
```

1 Because we're importing these functions from the modules, they will not have the d3. prefix
2 You need to bind the component as the context to any new internal functions—this doesn't need to be done for any existing lifecycle functions
3 Fire your bar chart function whenever the component first mounts or receives new props/state

Traditional D3 rendering with React

```
createBarChart() {
  const node = this.node                          4
  const dataMax = max(this.props.data)                    5
  const yScale = scaleLinear()
    .domain([0, dataMax])
    .range([0, this.props.size[1]])               5

  select(node)
    .selectAll("rect")
    .data(this.props.data)
    .enter()
    .append("rect")

  select(node)
    .selectAll("rect")
    .data(this.props.data)
    .exit()
    .remove()
```

4 The element itself is referenced in the component when you render so you can use it to hand over to D3
5 Use the passed size and data to calculate your scale

Traditional D3 rendering with React

```
    select(node)
      .selectAll("rect")
      .data(this.props.data)
      .style("fill", "#fe9922")
      .attr("x", (d,i) => i * 25)
      .attr("y", d => this.props.size[1] - yScale(d))
      .attr("height", d => yScale(d))
      .attr("width", 25)
  }

  render() {                                              6
    return <svg ref={node => this.node = node}                    7
            width={500} height={500}>
    </svg>
  }
}

export default BarChart
```

6 Render is returning an SVG element waiting for your D3 code
7 Pass a reference to the node for D3 to use

Traditional D3 rendering with React

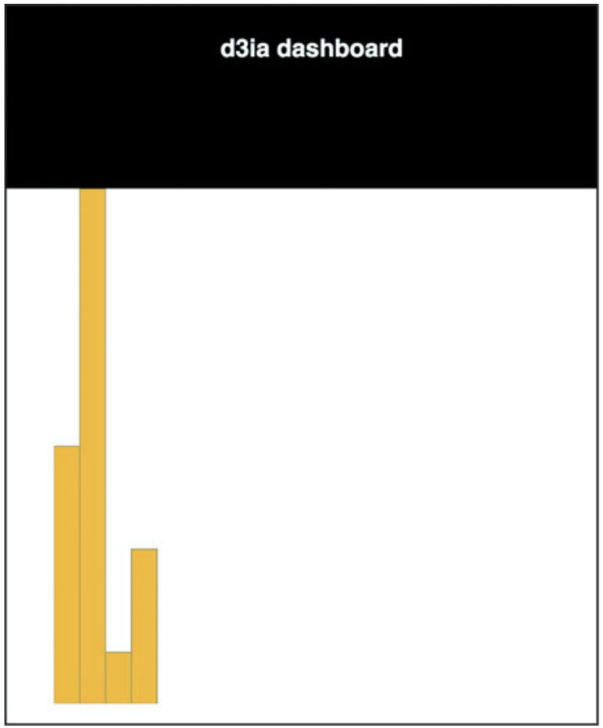
Referencing BarChart.js in App.js

```
import React, { Component } from 'react'
import './App.css'
import BarChart from './BarChart'                          1

class App extends Component {
  render() {
    return (
      <div className="App">
        <div className="App-header">
          <h2>d3ia dashboard</h2>
        </div>
        <div>
        <BarChart data={[5,10,1,3]} size={[500,500]} />        2
        </div>
      </div>
    )
  }
}

export default App
```

1 We need to import our newly created component
2 This is how we can get those props.data and props.size to use to render our bar chart in BarChart.js

Traditional D3 rendering with React

Your first React + D3 app, with a simple bar chart rendered in your app



React for element creation, D3 as the visualization kernel

WorldMap.js and associated world.js

```
import React, { Component } from 'react'
import './App.css'
import worlddata from './world'                          1
import { geoMercator, geoPath } from 'd3-geo'
class WorldMap extends Component {
  render() {
    const projection = geoMercator()
    const pathGenerator = geoPath().projection(projection)
    const countries = worlddata.features
      .map((d,i) => <path                                2
        key={"path" + i}                                 3
        d={pathGenerator(d)}
        className="countries"                            4
      />)
    return <svg width={500} height={500}>
      {countries}                                        5
    </svg>
  }
}
export default WorldMap
```

1 Rather than fiddling with asynch calls, we can import the map data because it won't be changing
2 Map the arrays to svg:path elements
3 Make sure they each have a unique
4 Remember className not class with JSX
5 Nest the array of paths within the svg element

React for element creation, D3 as the visualization kernel

world.js

```
export default {"type":"FeatureCollection","features":[
{"type":"Feature","id":"AFG","properties":{"name":"Afghanistan"},"geometry":{
    "type":"Polygon","coordinates":[[[61.210817,35.650072],...
```

React for element creation, D3 as the visualization kernel

The basic map we saw in previous chapter but now rendered via React and JSX with D3 providing the drawing instructions

Data dashboard basics

Updated App.js with sample data

```
...import the existing app.js imports...
  import WorldMap from './WorldMap'
   mport worlddata from './world'
  import { range, sum } from 'd3-array'                    1
  import { scaleThreshold } from 'd3-scale'                  1
  import { geoCentroid } from 'd3-geo'                     1

  const appdata = worlddata.features
  .filter(d => geoCentroid(d)[0] < -20)                  2

  appdata
  .forEach((d,i) => {
    const offset = Math.random()
    d.launchday = i
    d.data = range(30).map((p,q) =>
      q < i ? 0 : Math.random() * 2 + offset) })          3
```

1 We'll need these functions to build our sample data
2 Constrain our map to only North and South America for simplicity's sake
3 Generate some fake data with relatively interesting patterns—the "launch day" of each country is its array position

Data dashboard basics

```
class App extends Component {

    render() {

const colorScale = scaleThreshold().domain([5,10,20,30,50])
   range(["#75739F", "#5EAFC6", "#41A368", "#93C464", "#FE9922"])     4
      return (
        <div className="App">
          <div className="App-header">
            <h2>d3ia dashboard</h2>
          </div>
          <div>
            <WorldMap colorScale={colorScale} data={appdata} size={[500,400]}
/>
          </div>
      </div>
    )
  }
}

export default App
```

4 Color each country by its launch date

Data dashboard basics

Our rendered WorldMap component, with countries colored by launch day

Data dashboard basics

Updated WorldMap.js getting data and color scale from parent

```
import React, { Component } from 'react'
import './App.css'
import { geoMercator, geoPath } from 'd3-geo'

class WorldMap extends Component {
  render() {
    const projection = geoMercator()
      .scale(120)
      .translate([430,250])                                          1
    const pathGenerator = geoPath().projection(projection)
    const countries = this.props.data
      .map((d,i) => <path
        key={"path" + i}
        d={pathGenerator(d)}
        style={{fill: this.props.colorScale(d.launchday),                2
            stroke: "black", strokeOpacity: 0.5 }}
        className="countries"
      />)
    return <svg width={this.props.size[0]} height={this.props.size[1]}>    3
      {countries}
    </svg>
  }
}

export default WorldMap
```

1 Updated translate and scale because we've constrained our geodata
2 Use the color scale passed via props
3 Base height and width on size prop so that we can make the chart responsive later

Data dashboard basics

App.js updates for adding the bar chart

```
import BarChart from './BarChart'
...
        <BarChart colorScale={colorScale} data={appdata}
size={[500,400]} />
...
```

Data dashboard basics

Updated BarChart.js

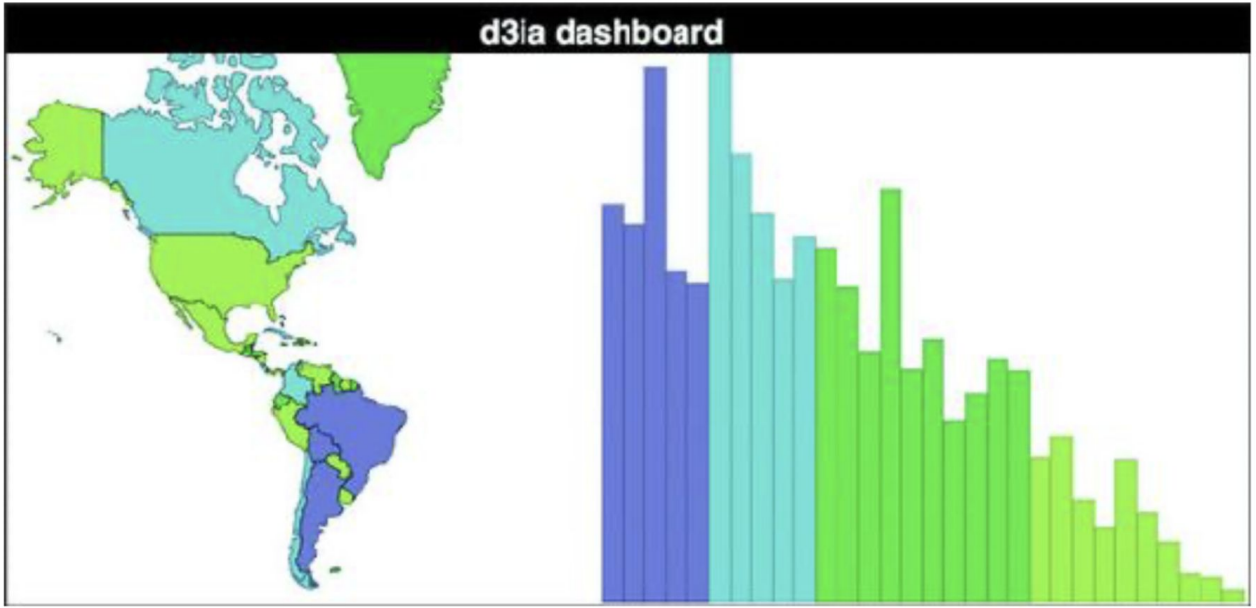
```
createBarChart() {
    const node = this.node
    const dataMax = max(this.props.data.map(d => sum(d.data)))
    const barWidth = this.props.size[0] / this.props.data.length      1
    const yScale = scaleLinear()
      .domain([0, dataMax])
      .range([0, this.props.size[1]])

...nothing else changed in createBarChart until we create rectangles...

    select(node)
      .selectAll("rect")
      .data(this.props.data)
      .attr("x", (d,i) => i * barWidth)
      .attr("y", d => this.props.size[1] - yScale(sum(d.data)))
      .attr("height", d => yScale(sum(d.data)))
      .attr("width", barWidth)
      .style("fill", (d,i) => this.props.colorScale(d.launchday))
      .style("stroke", "black")
      .style("stroke-opacity", 0.25)

    }
```

```
   render() {
     return <svg ref={node => this.node = node}
           width={this.props.size[0]}
height={this.props.size[1]}>
     </svg>
  }
```

1 Dynamically calculate bar width to make the chart more responsive in the future

Data dashboard basics

A rudimentary dashboard with two views into the data. The bars are ordered by our fake “Launch Day,” and sometimes the randomized data shows interesting patterns like the dark green bar showing a higher total sales than countries that launched 10 days earlier.



Data dashboard basics

A React streamgraph

```
import React, { Component } from 'react'
import './App.css'
import { stack, area, curveBasis, stackOrderInsideOut, stackOffsetSilhouette }
from 'd3-shape'
import { range } from 'd3-array'
import { scaleLinear } from 'd3-scale'

class StreamGraph extends Component {
  render() {

    const stackData = range(30).map(() => ({}))                    1
    for (let x = 0; x<30; x++) {
      this.props.data.forEach(country => {
        stackData[x][country.id] = country.data[x]                 2
      })
    }
    const xScale = scaleLinear().domain([0, 30])
      .range([0, this.props.size[0]])

    const yScale = scaleLinear().domain([0, 60])
      .range([this.props.size[1], 0])
```

1 This will be our blank array of objects for our reprocessed streamgraph data
2 Transform our original data into a format amenable for stack

Data dashboard basics

```
    const stackLayout = stack()
      .offset(stackOffsetSilhouette)
      .order(stackOrderInsideOut)
      .keys(Object.keys(stackData[0]))                    3
    const stackArea = area()
      .x((d, i) => xScale(i))
      .y0(d => yScale(d[0]))
      .y1(d => yScale(d[1]))
      .curve(curveBasis)

    const stacks = stackLayout(stackData).map((d, i) => <path
        key={"stack" + i}
        d={stackArea(d)}
        style={{ fill: this.props.colorScale(this.props.data[i].launchday),
            stroke: "black", strokeOpacity: 0.25 }}
    />)

    return <svg width={this.props.size[0]} height={this.props.size[1]}>
      <g transform={"translate(0," + (-this.props.size[1] / 2) + ")"}>   4
      {stacks}
      </g>
    </svg>
  }
}
export default StreamGraph
```

3 Each key maps to a country from our data
4 We need to do this offset because our streamgraph runs along the 0 axis—if you're drawing a regular stacked area or line graph, it's not necessary

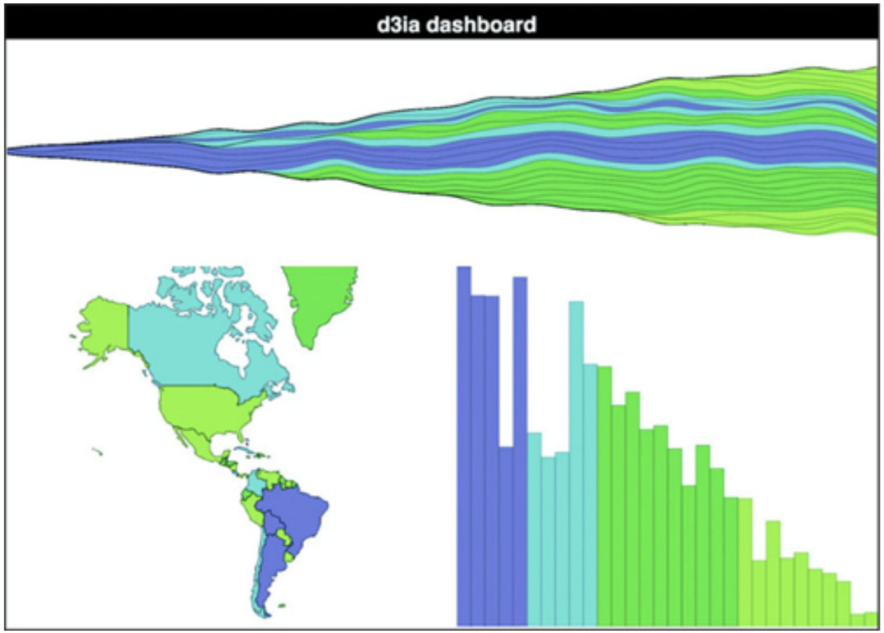
Data dashboard basics

Also reference it in our App.js

```
...other imports...
import StreamGraph from './StreamGraph'
...the rest of your existing app.js behavior...
   <StreamGraph colorScale={colorScale} data={appdata} size={[1000,250]} />
...
```

Data dashboard basics

A typical dashboard, with three views into the same dataset. In this case, it shows MatFlicks launch waves geographically while summing up sales in the bar chart and showing sales over time in the streamgraph.



Dashboard upgrades

1. Make it responsive.
2. Add a legend.
3. Cross-highlight on bars, countries, and trends.
4. Brush based on launch day.
5. Show numbers.

Dashboard upgrades

App.js state and resize listener

```
...import necessary modules...
class App extends Component {
  constructor(props){
    super(props)
    this.onResize = this.onResize.bind(this)
    this.state = { screenWidth: 1000, screenHeight: 500 }                1

  }

  componentDidMount() {
    window.addEventListener('resize', this.onResize, false)              2
    this.onResize()
  }

  onResize() {
    this.setState({ screenWidth: window.innerWidth,
    screenHeight: window.innerHeight - 70 })                             3
  }
```

1 Initialize state with some sensible defaults
2 Register a listener for the resize event
3 Update state with the new window width and height (minus the size of the header)

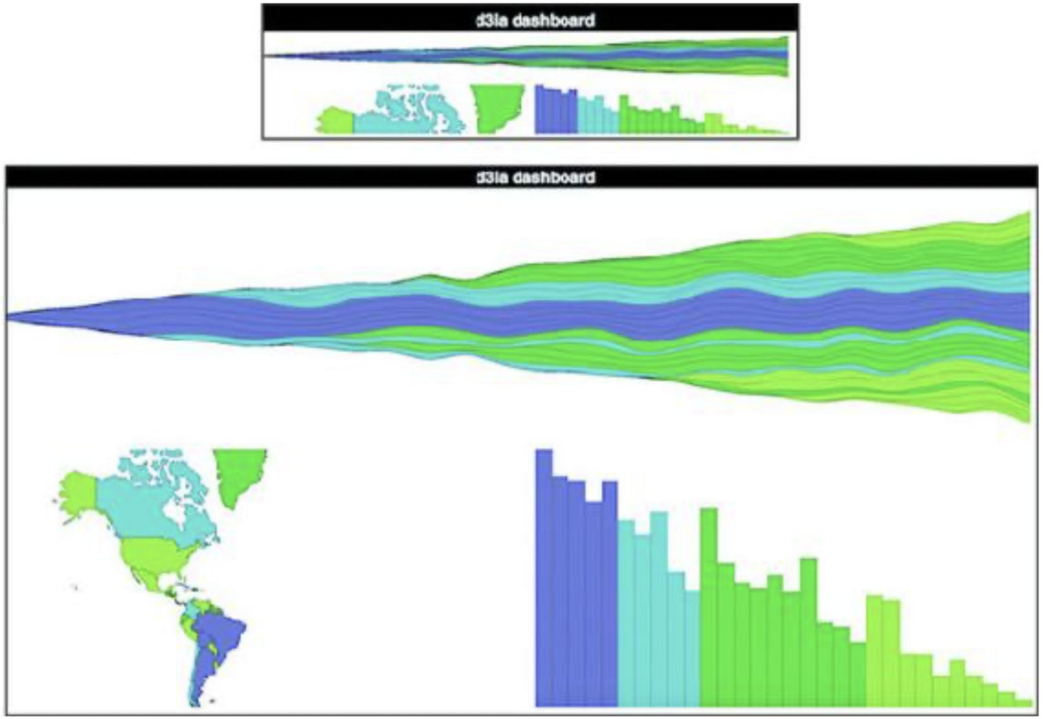
Dashboard upgrades

```
  render() {
...existing render behavior...
<StreamGraph colorScale={colorScale} data={appdata}
    size={[this.state.screenWidth, this.state.screenHeight / 2]} />      4

<WorldMap colorScale={colorScale} data={appdata}
    size={[this.state.screenWidth / 2, this.state.screenHeight / 2]} />
<BarChart colorScale={colorScale} data={appdata}
    size={[this.state.screenWidth / 2, this.state.screenHeight / 2]} />
```

4 Send height and width from the component state

Dashboard upgrades

The same dashboard on a large screen and a small screen



Dashboard upgrades

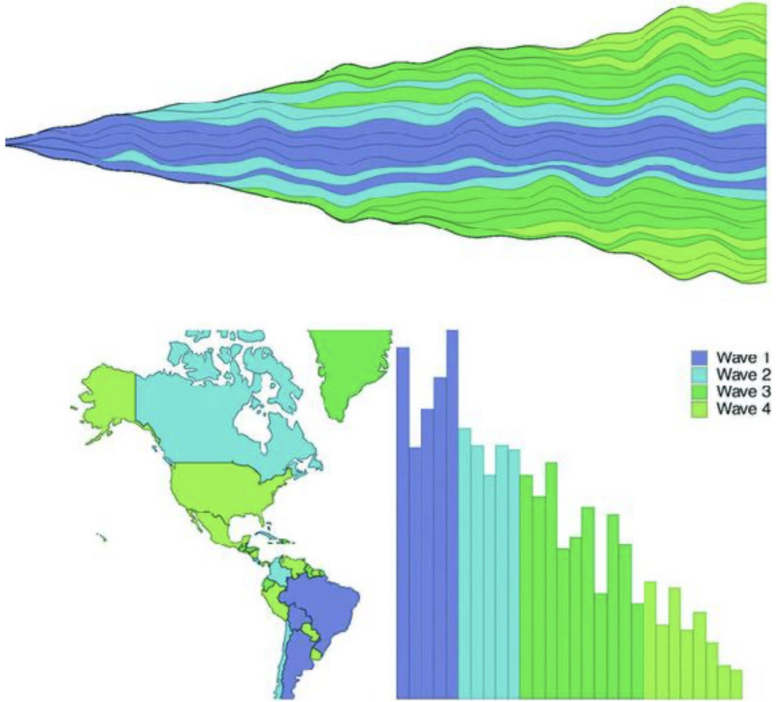
Adding a legend

```
import { legendColor } from 'd3-svg-legend'
import { transition } from 'd3-transition'                              1
...
  createBarChart() {
    const dataMax = max(this.props.data.map(d => sum(d.data)))
    const barWidth = this.props.size[0] / this.props.data.length
    const node = this.node

    const legend = legendColor()
      .scale(this.props.colorScale)
      .labels(["Wave 1", "Wave 2", "Wave 3", "Wave 4"])                 2

    select(node)
      .selectAll("g.legend")
      .data([0])                                                        3
      .enter()
      .append("g")
      .attr("class", "legend")
      .call(legend)

    select(node)
      .select("g.legend")
      .attr("transform", "translate(" + (this.props.size[0] - 100) + ", 20)"4
...
```

1 You need to import transition so that d3-svg-legend can use it
2 Although we could use threshold values, it's better to use semantically meaningful names for your categories
3 We bind a single value array of data to ensure that we append one <g> element only and then during later refreshes we update it
4 Make the transform change happen during every refresh, so the legend is responsive in its placement

Dashboard upgrades

Our MatFlicks table mat rollout dashboard, now with a legend to show your users which countries are in which wave of launches



Dashboard upgrades

App.js updates

```
this.onHover = this.onHover.bind(this)                                    1
this.state = { screenWidth: 1000, screenHeight: 500, hover: "none" }
...
onHover(d) {
    this.setState({ hover: d.id })                                        2
}
...
<StreamGraph hoverElement={this.state.hover} onHover={this.onHover}      3
     colorScale={colorScale} data={appdata} size={[this.state.screenWidth,
this.state.screenHeight / 2]} />
<WorldMap hoverElement={this.state.hover} onHover={this.onHover}         3
     colorScale={colorScale} data={appdata}
size={[this.state.screenWidth / 2, this.state.screenHeight / 2]} />
<BarChart hoverElement={this.state.hover} onHover={this.onHover}         3
     colorScale={colorScale} data={appdata}
size={[this.state.screenWidth / 2, this.state.screenHeight / 2]} />
```

1 We're going to store the currently hovered on element in state so we need to initialize our state with a hover property
2 The hover function will expect us to send the data object
3 We need to send to the components both the hover function and the current hover state for them to interact properly

Dashboard upgrades

BarChart.js updates

```
...
    select(node)
      .selectAll("rect")
      .data(this.props.data)
      .enter()
      .append("rect")
      .attr("class", "bar")
      .on("mouseover", this.props.onHover)                          1
...
    select(node)
      .selectAll("rect.bar")
      .data(this.props.data)
      .attr("x", (d,i) => i * barWidth)
      .attr("y", d => this.props.size[1] - yScale(sum(d.data)))
      .attr("height", d => yScale(sum(d.data)))
      .attr("width", barWidth)
      .style("fill", (d,i) => this.props.hoverElement === d.id ?     2
            "#FCBC34" : this.props.colorScale(i))
...
```

1 Bind the hover function we've passed down like we would any other function in a D3 chart
2 Fill with the usual colorScale value unless we're hovering on this element, in this case orange

Dashboard upgrades

WorldMap.js Updates

```
.map((d,i) => <path
    key={"path" + i}
    d={pathGenerator(d)}
    onMouseEnter={() => {this.props.onHover(d)}}                    1
    style={{fill: this.props.hoverElement === d.id ? "#FCBC34" :
        this.props.colorScale(i), stroke: "black",
        strokeOpacity: 0.5 }}                                       2
    className="countries"
/>)
```

1 In React, it’s called onMouseEnter and it doesn’t automatically send the bound data
2 Same as with the D3 method, except in React syntax

Dashboard upgrades

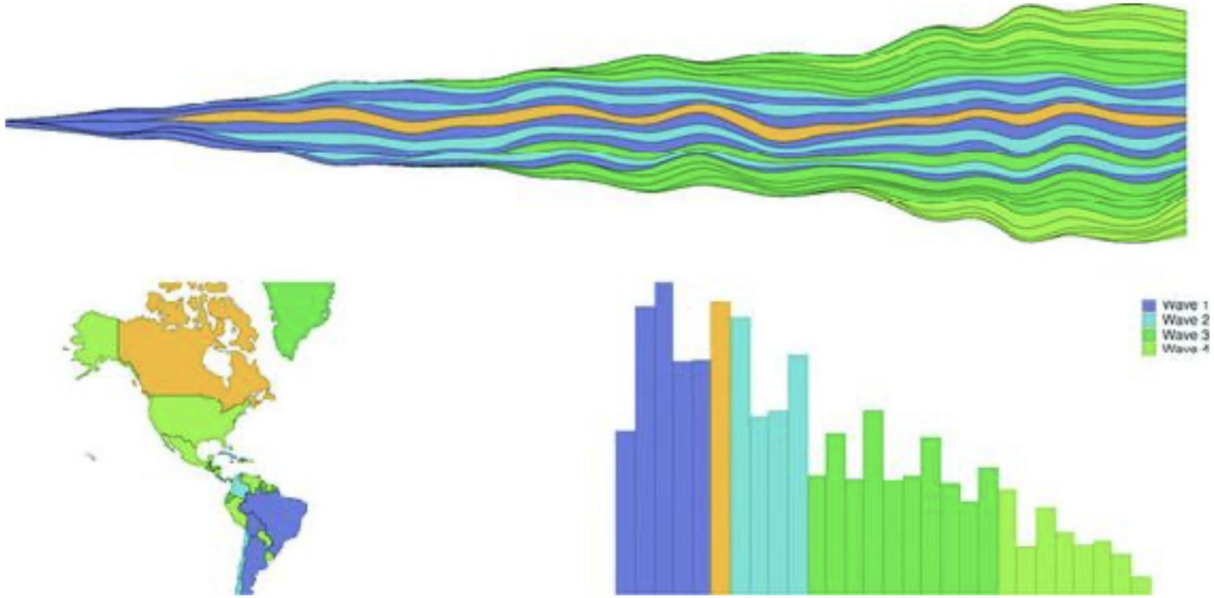
StreamGraph.js Updates

```
const stacks = stackLayout(stackData).map((d, i) => <path
    key={"stack" + i}
    d={stackArea(d)}
    onMouseEnter={() => {this.props.onHover(this.props.data[i])}}        1
    style={{fill: this.props.hoverElement === this.props.data[i]["id"] ?
    "#FCBC34" : this.props.colorScale(this.props.data[i]["id"].launchday),
    stroke: "black", strokeOpacity: 0.5 }}

/>)
```

1 Because stackData is the transformed data, we need to reference the original data to send the right object

Dashboard upgrades

Canada as our cross-highlighting example. It was earlier in the alphabet and as a result millions of Canadians were enjoying high quality European mats long before citizens of the United States could.



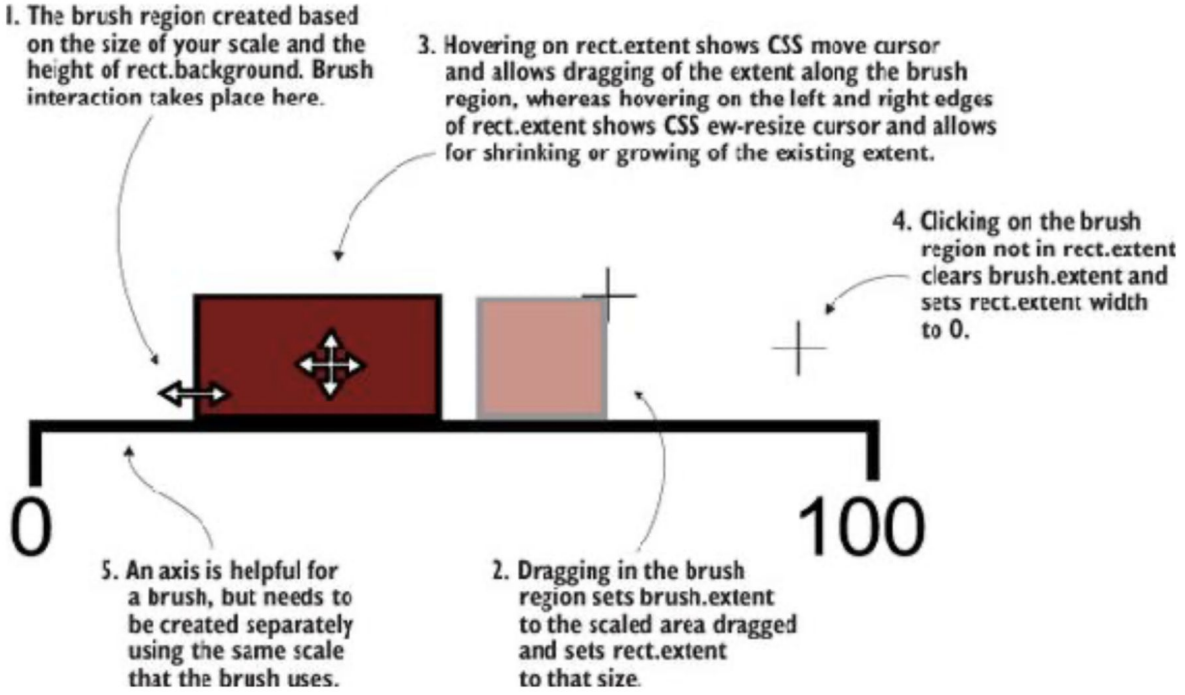
Brushing

App.js updates for a brush

```
...
import Brush from './Brush'
...

<Brush size={[this.state.screenWidth, 50]} />
```

Brushing

Components of a brush



Brushing

Brush.js component

```
import React, { Component } from 'react'
import './App.css'
import { select, event } from 'd3-selection'
import { scaleLinear } from 'd3-scale'
import { brushX } from 'd3-brush'
import { axisBottom } from 'd3-axis'

class Brush extends Component {
  constructor(props){
    super(props)
    this.createBrush = this.createBrush.bind(this)            1
  }

  componentDidMount() {
    this.createBrush()                                  1
  }

  componentDidUpdate() {
    this.createBrush()                                  1
  }
  createBrush() {
  const node = this.node
  const scale = scaleLinear().domain([0,30])
    .range([0,this.props.size[0]])                          2
```

1 Standard stuff for a React component that has D3 handle element creation and updating
2 For our axis and later for our brushed function

Brushing

```
const dayBrush = brushX()
                .extent([[0, 0], this.props.size])
                .on("brush", brushed)                    3

 const dayAxis = axisBottom()
   .scale(scale)                                  4

 select(node)
   .selectAll("g.brushaxis")
   .data([0])
   .enter()
   .append("g")
   .attr("class", "brushaxis")                        4
   .attr("transform", "translate(0,25)")

 select(node)
   .select("g.brushaxis")
   .call(dayAxis)

 select(node)
   .selectAll("g.brush")
   .data([0])
   .enter()
   .append("g")
   .attr("class", "brush")
```

3 Initialize the brush and associate it with our brushed function
4 Nothing new here, only creating an axis

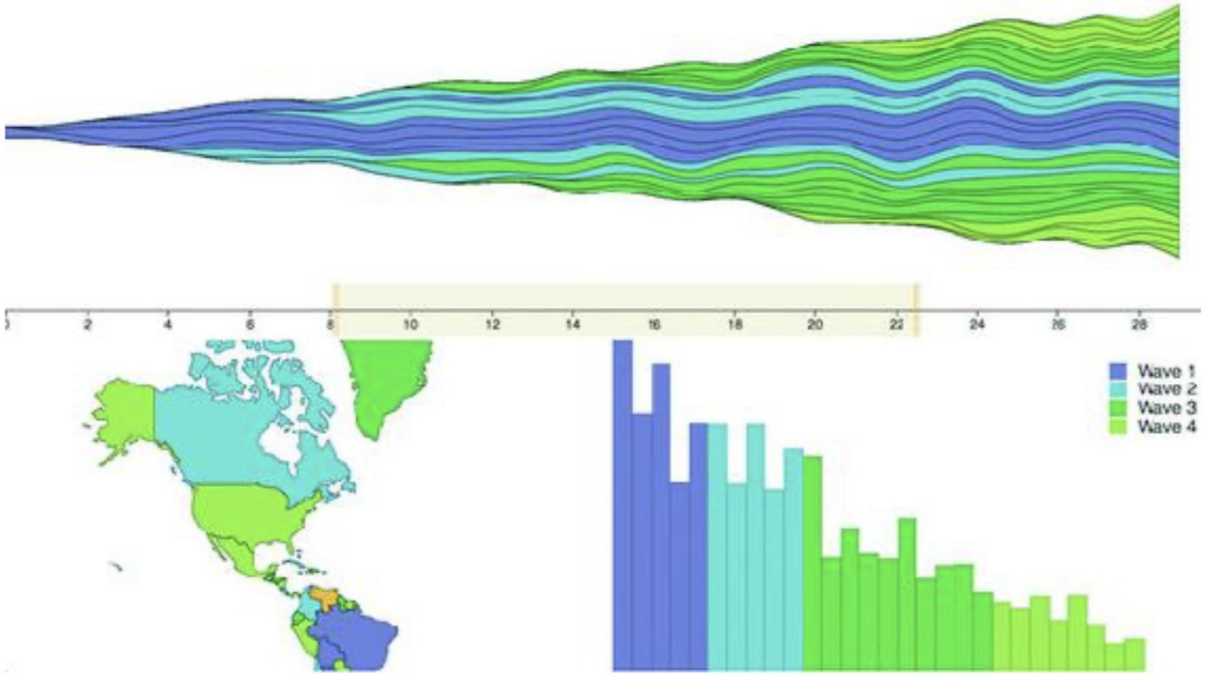
Brushing

```
  select(node)
    .select("g.brush")
    .call(dayBrush)                                    5

  function brushed() {
    console.log(event)
     // brushed code                                    6
  };

  }

  render() {
    return <svg  ref={node => this.node = node}
width={this.props.size[0]} height={50}></svg>
  }
}

export default Brush
```

5 Call the brush with our <g> to create it
6 We'll handle this below

Brushing

Here’s our brush, though without any function associated with the brush events, so it’s little more than a toy.



Brushing

The brushed function in Brush.js

```
const brushFn = this.props.changeBrush                          1
function brushed() {
    const selectedExtent = event.selection.map(d => scale.invert(d))      2
    brushFn(selectedExtent)
}
```

1 We need to bind this function to a variable since the context of brushed will be different (we could also use function.bind or function.apply but that's more cumbersome in this case)
2 Invert will return the domain for the range for a scale

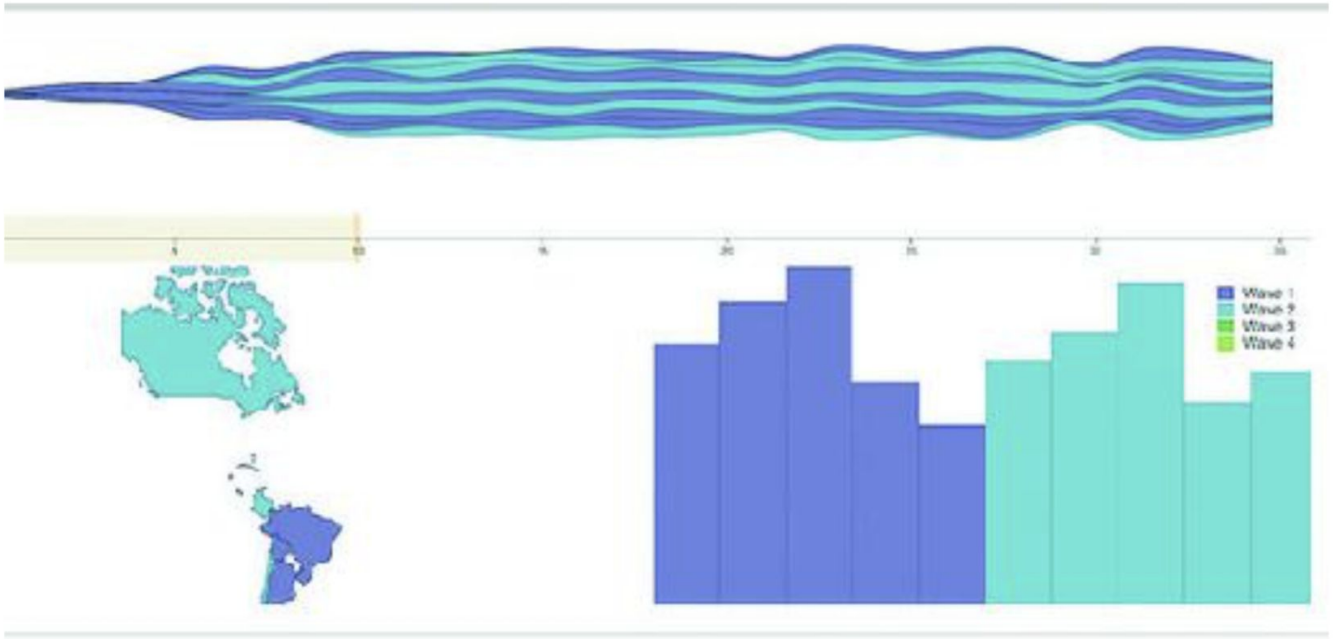
Brushing

App.js changes to listen for a brush

```
...
this.onBrush = this.onBrush.bind(this)
this.state = { screenWidth: 1000, screenHeight: 500, hover: "none",
    brushExtent: [0,40] }

...
onBrush(d) {
    this.setState({ brushExtent: d })
}
render() {
    const filteredAppdata = appdata.filter((d,i) =>
        d.launchday >= this.state.brushExtent[0] &&
        d.launchday <= this.state.brushExtent[1])
...
<StreamGraph hoverElement={this.state.hover} onHover={this.onHover}
     colorScale={colorScale} data={filterdAppdata}
     size={[this.state.screenWidth, this.state.screenHeight / 2]} />
<Brush changeBrush={this.onBrush} size={[this.state.screenWidth, 50]} />
<WorldMap hoverElement={this.state.hover} onHover={this.onHover}
     colorScale={colorScale} data={filterdAppdata}
     size={[this.state.screenWidth / 2, this.state.screenHeight / 2]} />
<BarChart hoverElement={this.state.hover} onHover={this.onHover}
     colorScale={colorScale} data={filterdAppdata}
     size={[this.state.screenWidth / 2, this.state.screenHeight / 2]} />
...
```

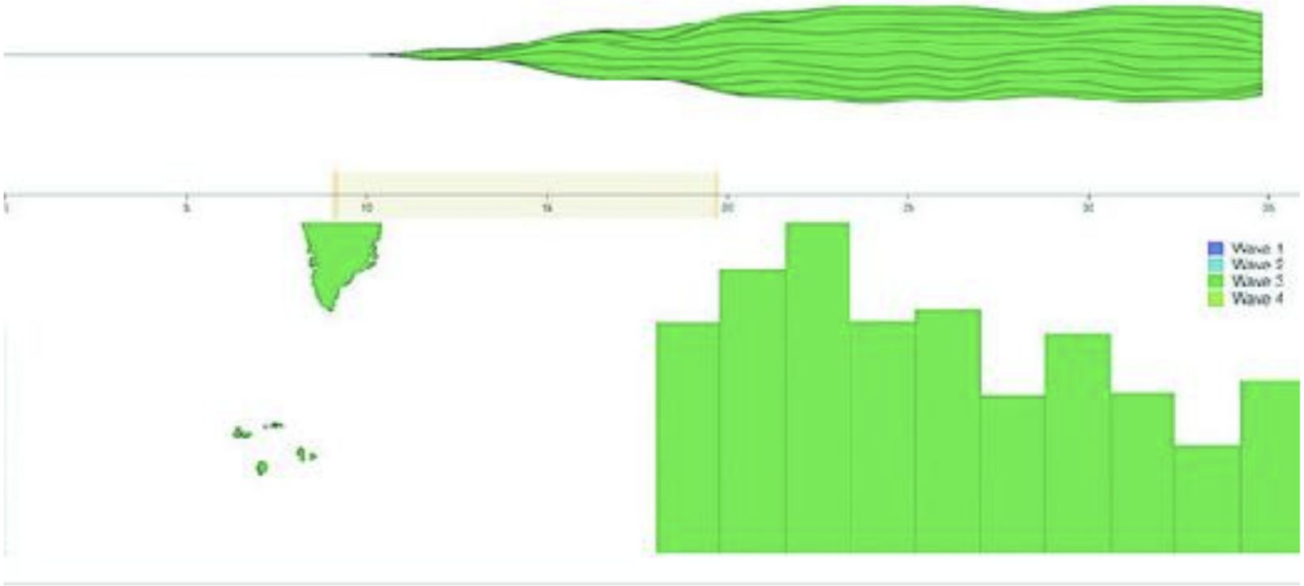
Brushing

The results of our brushed() function showing only wave 1 and 2, then wave 3, and finally wave 4 countries.



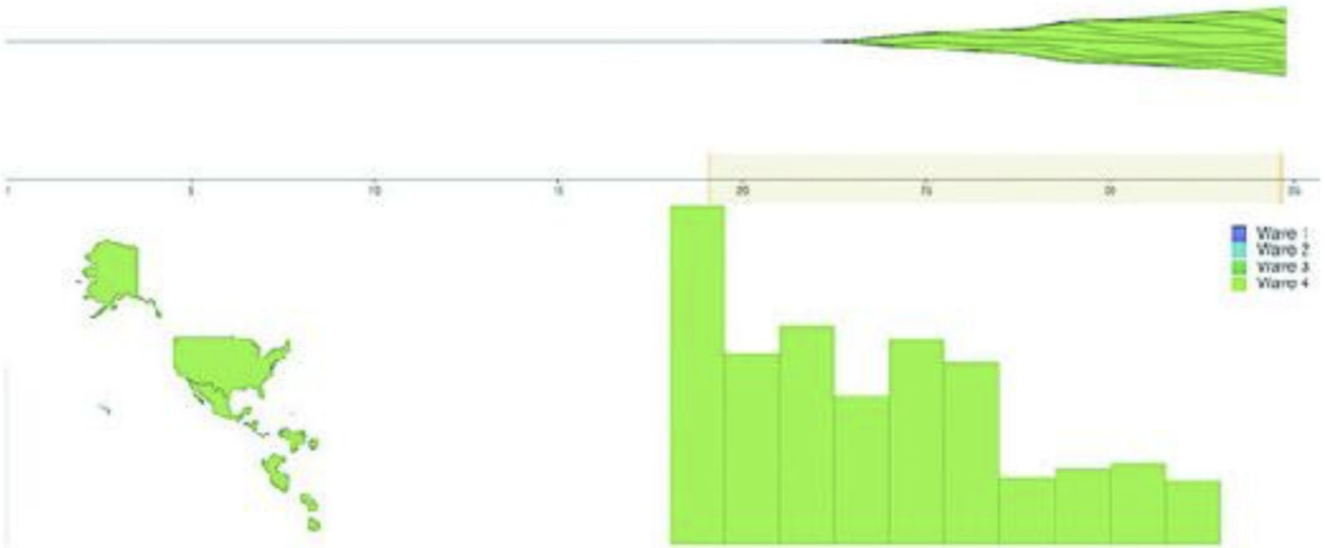
Brushing

The results of our brushed() function showing only wave 1 and 2, then wave 3, and finally wave 4 countries.



Brushing

The results of our brushed() function showing only wave 1 and 2, then wave 3, and finally wave 4 countries.



Show me the numbers

Pure Render StatLine.js

```
import React from 'react'
import { mean, sum } from 'd3-array'

export default (props) => {                                    1
  const allLength = props.allData.length                          2
  const filteredLength = props.filteredData.length
  let allSales = mean(props.allData.map(d => sum(d.data)))
  allSales = Math.floor(allSales * 100)/100                        3
  let filteredSales = mean(props.filteredData.map(d => sum(d.data)))
  filteredSales = Math.floor(filteredSales * 100)/100
  return <div>                                                  4
  <h1><span>Stats: </span>
  <span>{filteredLength}/{allLength} countries selected. </span>
  <span>Average sales: </span>
  <span>{filteredSales} ({allSales})</span>
  </h1>
</div>
}
```

1 We're exporting a function that takes props
2 Notice we're using props, not this.props, because there is no this, and props is being passed to the function when it's a pure render function
3 This is a simple way to round to two decimal places
4 Pure render components return a DOM elements

Show me the numbers

Our final dashboard, showing a statline at the top indicating the number of countries we've selected out of the total number of countries in the data as well as the average sales of the selected countries compared to the average sales overall. Here it's resized to be smaller and because we don't resize the map, we only see North America.