CS 171

Lab 7

Learning Objectives

- Know how to *link* multiple views with each other
- Understand the concept behind D3's brush component
- Understand the JS keyword *this* and the scope of execution
- Get a better understanding of *system design* and *code structure*

Prerequisites

- You have read and **programmed** along with:
    - http://www.cs171.org/2016/assets/material/cs171-lab7-reading.pdf
    - The four chapters about JS objects at http://www.w3schools.com/js/js*object*definition.asp (repetition of JS objects with more examples)
    - *[Optional]* Scope and *this* in JavaScript
- You have successfully filled in the pre-quiz for the 7th lab.

In the last weeks you have learned the fundamentals of the JS library D3 and you have gained some implementation expertise during previous labs and homeworks. You should be comfortable with the major concepts and be able to implement common charts as well as interactive and more advanced visualizations with D3.

In this lab you will work on a new problem set. It is continuous, a bit longer than the usual activities, and it will give you a better understanding of *linked views* and *system design*.

We will provide a template and many additional code snippets so that your main tasks will focus on the structure and the event handling components. However, please make sure that you understand the code in the provided templates, and take your time to read through it!

Data

Purchases for Household Supplies in the UK

In this lab you will work with a dataset that contains detailed annual statistics on family food and drink purchases in the UK.

We have already extracted the major information and created a JSON file with data between 1974 and 2013.

The JSON file is divided into two major arrays with objects:

- `years` contain the average weekly spending on food and drinks in pence (GBX) per person
- `layers` also contain the average weekly spending, but separated into 21 different *food* categories

JSON Structure:

```
{
    "years": [
        { "Year": 1974, "Expenditures": 331.437 },
        ...
        { "Year": 2013, "Expenditures": 2986.373 }
    ],
    "layers": [
        {
            "Year": 1974,
            "Milk and milk products excluding cheese": 29.228,
            "Cheese": 8.694,
            "Carcase meat": 44.479,
            "Non-carcase meat and meat products": 55.915,
            "Fish": 13.652,
            "Eggs": 11.429,
            "Fats": 14.537,
            ....
        },
        {
            "Year": 1975,
            "Milk and milk products excluding cheese": 37.239,
            "Cheese": 10.377,
            ...
        },
        ...
    ]
}
```

The data is part of the annual *Family Food Report*, it is published through UK's open government platform *gov.uk* and is updated every year. However, the dataset is not completely accurate and there is a lack of data in some areas. For example, alcoholic and soft drink figures appear for the first time in 1992, but we can assume that people have consumed these products before.

Source: https://www.gov.uk/government/statistical-data-sets/family-food-datasets

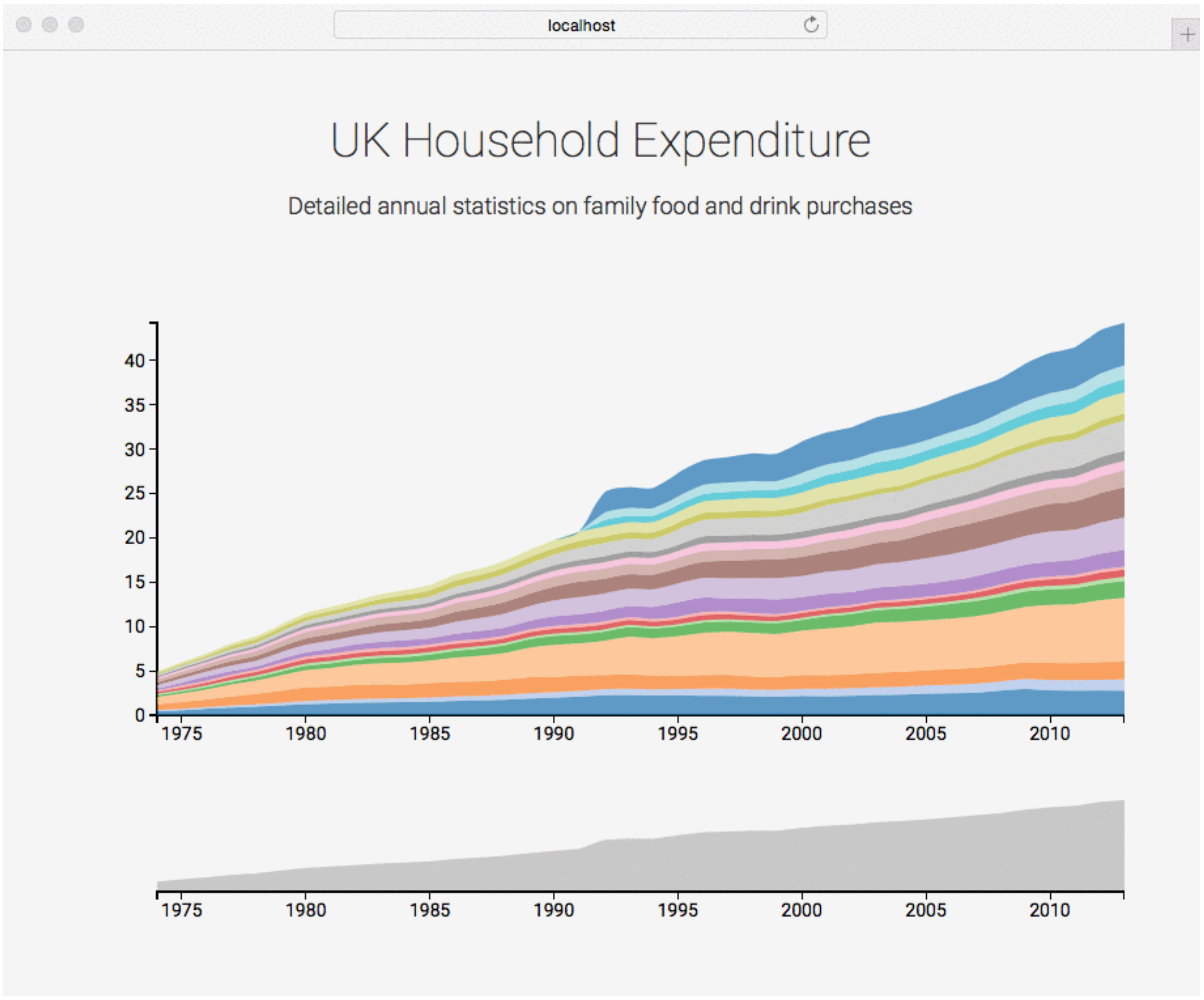
Template

This lab will bring you a step closer to implementing more complex visualizations which makes it necessary to adapt the way how we organize our application. However, the provided template is quite similar to previous labs and homeworks. It is based on Bootstrap and it contains a basic HTML structure, some CSS rules and pieces of JS code.

During this lab we will introduce a concept which will help you to structure a D3 project with multiple visualizations. For this reason, we will also describe the file structure in more detail. The main change is that each visualization is a separate JS object and the implementation of the different visualizations is moved into separate JS files.

Preview

After implementing all tasks your visualization should look like this:



Conceptual and technical background

Thinking about the structure of your project early on can save you a lot of time and will make your implementation more robust, extensible and reusable.

Files and folders

The provided template is structured as follows:

- **index.html** is the default file that appears when a user invokes your webpage. It should include a basic structure and placeholders for your visualizations.
- **/js** contains the JS files for the visualization and external libraries (can be further sub-divided)

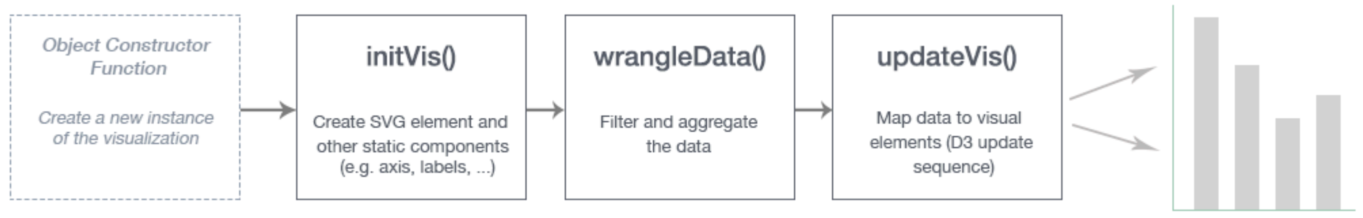
- **/data** contains the data files
- **/css** contains the stylesheet files

Divide and conquer

You should always try to split a complex problem into smaller, easier-to-tackle sub-problems. Each sub-problem will be solved independently and afterwards integrated into the final system.

Visualizations should be organized and structured into individual code components and, if possible, implemented as flexible and reusable components. Therefore, we will organize each visualization as an object.

The visualization specific implementation should be done in object functions and should follow this pipeline:


Object Constructor Function
Create a new instance of the visualization
initVis()
Create SVG element and other static components (e.g. axis, labels, ...)
wrangleData()
Filter and aggregate the data
updateVis()
Map data to visual elements (D3 update sequence)

Visualizations as objects

The following example will give you a better understanding of these steps.

We can create an instance of an object by using the keyword `new`. The object constructor should be used to define properties:

```
// Create an object instance
var barchart = new BarChart("bar-chart-container", data);
```

We can define properties within an object by using the keyword `this`:

```
// Object constructor function
BarChart = function(_parentElement, _data){
    this.parentElement = _parentElement;
    this.data = _data;

    // Call an object function
    this.initVis();
}
```

We will implement methods as `Object.prototype` properties:

```
BarChart.prototype.initVis = function(){
    ...
}
```

> **Object.prototype**
>
> - All JavaScript objects inherit their properties and methods from their *prototype*
> - The methods are in the execution context of the object
> - The object property `prototype` allows you to add new properties/methods to an existing prototype

The variables should be stored in the visualization object by using the `this` keyword. We would recommend to create another variable (for example `vis` or `_this`) to store the *this*-accessor. Otherwise the scope of `this` will change and it will cause undesirable side-effects:

```
BarChart.prototype.initVis = function(){
    var vis = this;

    vis.margin = { top: 20, right: 0, bottom: 60, left: 60 };

    vis.width = 800 - vis.margin.left - vis.margin.right,
    vis.height = 400 - vis.margin.top - vis.margin.bottom;

    vis.svg = d3.select("#" + vis.parentElement).append("svg")
        .attr("width", vis.width + vis.margin.left + vis.margin.right)
        ...
}
```

At the end of `initVis()` we call the function `wrangleData()` to follow our implementation pipeline:

```
BarChart.prototype.initVis = function(){
    ...

    vis.wrangleData();
}
```

It is quite usual that you have to process and prepare data (filter, aggregate, …) for a visualization whenever the data changes or after a user interaction. The function `wrangleData()` should take the raw data and modify it in a way that it can be mapped to the screen afterwards (`updateVis()`):

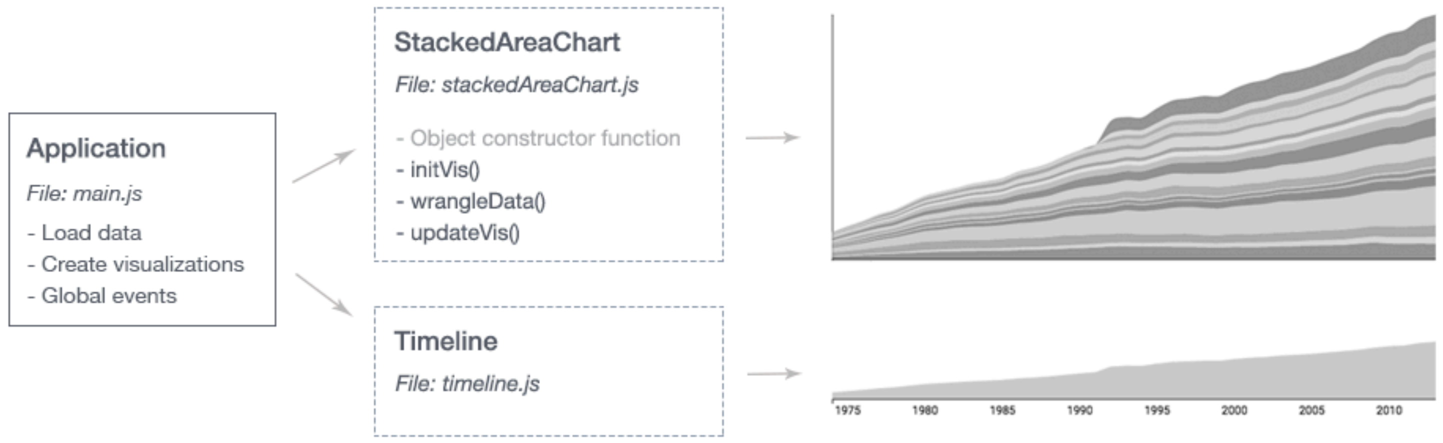
```
BarChart.prototype.wrangleData = function(){
    var vis = this;

    // Filter example
    vis.displayData = vis.data.filter(function(d){
        return d.year > 2000;
    });

    // Draw visualization
    vis.updateVis();
}
```

You should already be familiar with the last part. The function `updateVis()` contains the D3 update pattern (*enter, update, exit*). We use the variable `displayData` that we have created in `wrangleData()` before:

```
BarChart.prototype.updateVis = function(){
    var vis = this;

    // Update domains
    vis.y.domain([0, d3.max(vis.displayData, function(d) { return d.price; })]);
    ...

    // Draw the actual bar chart
    var bar = svg.selectAll(".bar")
      .data(data);

    bar.enter().append("rect")
        .attr("class", "bar")
        ...
}
```

Project overview

The previous example showed the implementation of a bar chart in an object-oriented way. To follow the main divide-and-conquer concept (i.e., splitting up a complex problem into various sub-tasks) we should also apply this to the file structure of our project.



We separate the visualization specific code into external files and create object instances in the file `main.js`, which is the entry point of our application. For example, if we want to use the same data for multiple charts we have to load the data only once in *main.js*. Thereby, our code stays clean and understandable.

This methodology will become very helpful for developing larger systems and more sophisticated interaction mechanisms.

Implementation

Activity I - Load data and create an instance of the stacked area chart

For all implementation parts in this lab, you should keep in mind that visualizations are implemented as objects. So if you want to store variables that belong to the visualization, make sure that you store it in the correct context (e.g., `vis.my_variable`). The same goes for accessing object parameters. If you get an error that a variable is undefined, make sure that you access it correctly!

For simplicity reasons, in the example code below, we have not always included the object oriented way of accessing variables - you will have to adjust that code to your overall code structure.

1. **Download the template**

   http://www.cs171.org/2016/assets/scripts/lab7/template.zip

   Check the `index.html` file and get a quick overview of the template.

   - The file `js/main.js` is the entry point of our application. It contains general scripts (e.g., loading the data) and it should create instances of the visualizations
   - The file `js/stackedAreaChart.js` contains the specific implementation of the stacked area

chart (see bar chart example above)

The data loading script is already complete and we have also created the file *stackedAreaChart.js* with boilerplate code.

2. **Create an instance of `StackedAreaChart`**

   After the data is loaded completely, the function `createVis()` in *main.js* is getting called. We will use this function to create instances of the visualizations.

   → Create an instance of `StackedAreaChart` and use the constructor to specify the properties: ***parentElement*** (“stacked-area-chart”) and ***data*** (“allData.layers”).

   → Open the webpage in your browser! If everything worked, you should see the loaded data in the web console and an SVG area with placeholders for the axes in the web inspector. If you get an error message in the web console you should fix it before going further.

Stack Layout

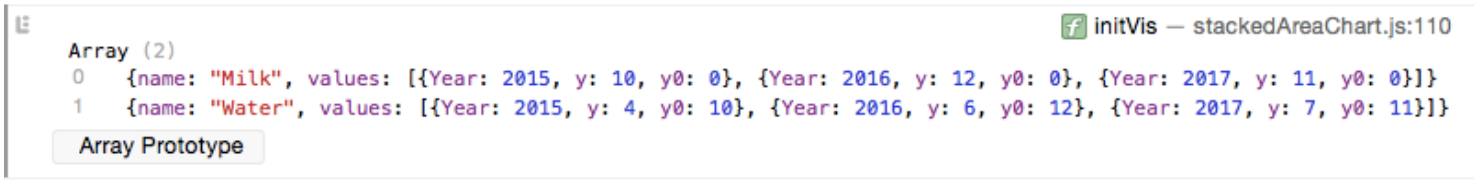
As you have seen in the preview at the beginning of this lab, you have to create a stacked area chart to visualize the different categories of UK’s household expenditures. Instead of calculating the coordinates of these layers manually you can use the *d3.layout.stack()* to generate a baseline value for each datum, so you can “stack” layers of data on top of each other.

And as a reminder from the previous lab: *The D3 layout methods have no direct visual output. They take data that you provide and re-map or otherwise transform it, thereby generating new data that is more convenient for a specific task.*

Example:

```
// Example data
var data = [
    {
        "name": "Milk",
        "values": [
            { "Year": 2015, "y": 10 },
            { "Year": 2016, "y": 12 },
            { "Year": 2017, "y": 11 }
        ]
    },
    {
        "name": "Water",
        "values": [
            { "Year": 2015, "y": 4 },
            { "Year": 2016, "y": 6 },
            { "Year": 2017, "y": 7 }
        ]
    }
];

// Initialize layout function
// with the 'values' accessor due to the multi-dimensional array
var stack = d3.layout.stack()
    .values(function(d) { return d.values; });

// Call layout function on the dataset
stack(data);

console.log(data);
```

Result:



```
                                                              initVis — stackedAreaChart.js:110
Array (2)
0  {name: "Milk", values: [{Year: 2015, y: 10, y0: 0}, {Year: 2016, y: 12, y0: 0}, {Year: 2017, y: 11, y0: 0}]}
1  {name: "Water", values: [{Year: 2015, y: 4, y0: 10}, {Year: 2016, y: 6, y0: 12}, {Year: 2017, y: 7, y0: 11}]}
Array Prototype
```

In the stacked data, each object has been given a y0 value. This is the *baseline* value. Notice that the y0 baseline value is equal to the sum of all the y values in the preceding categories. The category *"Milk"* is our first category and starts at the bottom, which means that the y0 values are zero.

Stacked Area

To stack these categories visually, we can create *stacked bar charts* or *stacked area charts*. In this lab you will create a stacked area chart. But it should be easy to adopt the workflow to a bar chart.

You have already created an area chart (see *HW4 - Za'atari Refugee Camp*) and a line chart (see *HW5 - FIFA World Cup*) and you should know how to use the D3 path generator. The built-in `d3.svg.area()` is ideally suited for the stacked area chart too. It just needs some slight modifications of the `y0()` and `y1()` parameters to consider the different baselines:

```
// Basic: area
var area = d3.svg.area()
    .interpolate("cardinal")
    .x(function(d) { return x(d.Year); })
    .y0(height)
    .y1(function(d) { return y(d.y); });

// Extended: stacked area
var area = d3.svg.area()
    .interpolate("cardinal")
    .x(function(d) { return x(d.Year); })
    .y0(function(d) { return y(d.y0); })
    .y1(function(d) { return y(d.y0 + d.y); });
```

The cardinal interpolation is optional and can be used to create smooth shapes.

Activity II - Create a stacked area chart

The file `stackedAreaChart.js` contains some functions with code snippets (append SVG, initialize scales and axes, etc). Try to get a quick overview and extend the implementation by running trough the following steps:

1. **Stack the data**

   This task should be done only once, so we would recommend an implementation in the `initVis()` function. If you want to filter the data, you can do this afterwards in the `wrangleData()` function directly on the "stacked" data.

   D3 Wiki: https://github.com/mbostock/d3/wiki/Stack-Layout

   It might seem pretty easy to "stack" the data with the built-in layout method `d3.layout.stack` (like in the example above) but for this to work, the raw data must be in a special format. You can compare the output in the web console with the *"Milk and water"*-example above and you will notice the difference.

Therefore, our first step is to transpose the loaded data and rearrange it in an array of arrays (do this directly after you init the D3 stack layout, but before you call the layout function on the dataset, all within `initVis()`):

- ***Get all categories***

  You can reuse `colorScale.domain()` to get all category names. In the file *main.js*, after the initial data preparation, we have used *d3.keys* to set the domain of the color scale. Take a look at that part in the code in *main.js* to make sure you understand what is going on, before getting the categories in `initVis()` as shown below.

  ```
  var dataCategories = colorScale.domain();
  ```

- ***Transpose the data***

  Currently the dataset consists of JSON objects for each year with 21 food categories and the corresponding date. Now, you have to transpose the data into layers. Each layer contains a name and a *values* array with the category-specific data for each year.

  ```
  // Required format
  var data = [
      {
          "name": "Milk",
          "values": [
              { "Year": 2015, "y": 10 },
              { "Year": 2016, "y": 12 }
          ]
      },
      {
          "name": "Water",
      ...
  ];
  ```

  You can rearrange the data by using the array *map()* function and iterating over all categories:

```
// Rearrange data into layers
var transposedData = dataCategories.map(function(name) {
    return {
        name: name,
        values: vis.data.map(function(d) {
            return {Year: d.Year, y: d[name]};
        })
    };
});
```

We are providing you with a lot of code here that is quite complex on first sight. Make sure that you think through each step of the data pre-processing and use debug statements until you understand what each `map()` is doing! (You could also use a bunch of for loops to rearrange the data, but using `map()` is more efficient and easier to read.)

- ***Stack the data***

  After initializing a D3 stack layout you can call it on the dataset:

  ```
  var stackedData = stack(transposedData);
  ```

2. **Initialize the D3 path generator in `initVis()`**

   After transforming the data into layers you have to initialize a path generator which maps the actual values to screen coordinates. The `updateVis()` function will use it later to draw the chart.

   Make sure to save the path generator in the variable `area` *:*

   ```
   vis.area = d3.svg.area() ...
   ```

3. **Call the `wrangleData()` function and draw the chart**

   This function will be used to filter/aggregate the dataset every time the visualization is rendered. Therefore, remove the comment and call *wrangleData()* at the end of the *initVis()* function. This will automatically call *updateVis()*. You will have to extend *wrangleData()* later.

   We have already implemented a basic version of the `updateVis()` function for you. It will update the y-axis domain, create visual elements for the stacked layers by using your path generator and finally update the axis labels.

   If you have completed the above implementation steps correctly, you should see the stacked area chart in your browser.

Otherwise, debug your code and make sure that the data preparation (stacked data) and the path generator (`vis.area`) is working as required.

Focus+Context via Brushing

In the next task you will extend the visualization with a second chart (a basic area chart; see *Preview*), to give the user the possibility to zoom and to select a specific time range.

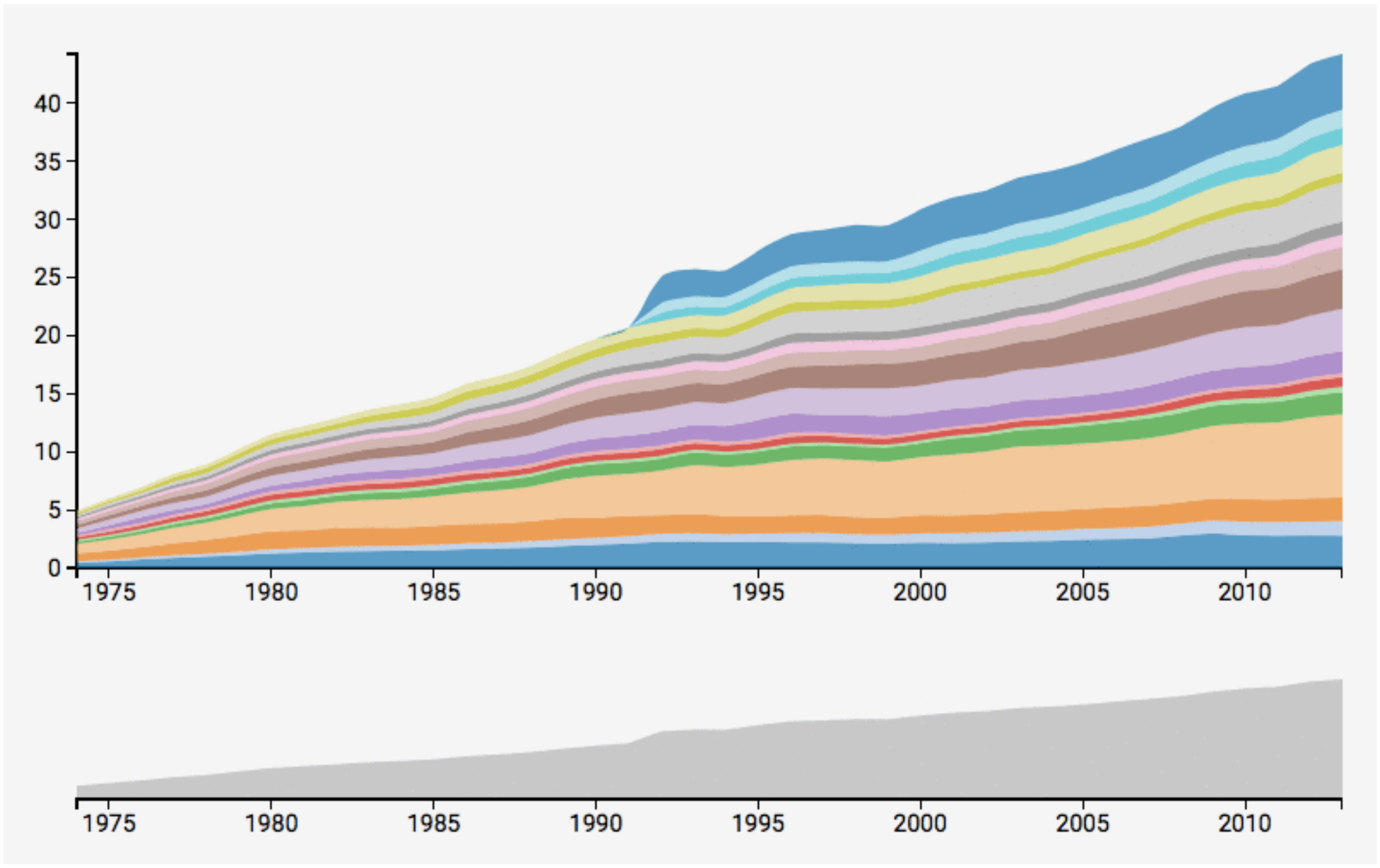
Focus+Context

- The stacked area chart should show selected regions in greater detail (*focus*)
- The basic area chart preserves a global view at reduced detail without layers (*context*)

This technique allows us to show the user detailed information linked with an overview (context) simultaneously.

Brushing & Linking

The idea of *brushing* is to allow the user to select a subset of data interactively. In combination with *linking* - changes are automatically reflected in *linked* visualizations - you can get the desired focus+context visualization.



Brushing with D3

D3 Wiki: https://github.com/mbostock/d3/wiki/SVG-Controls

The D3 brush is comparable to D3's axis component. You have to initialize the brush and assign a scale, which (most of the time) is the same scale function that is already used for the x-axis.

The property `on` must be used to set an event listener, whereby you can choose between three different events:

- `brushstart` - on mousedown
- `brush` - on mousemove, if the brush extent has changed
- `brushend` - on mouseup

```
// Initialize time scale (x-axis)
var xContext = d3.time.scale()
    .range([0, width])
    .domain(d3.extent(displayData, function(d) { return d.Year; }));

// Initialize brush component
var brush = d3.svg.brush()
    .x(xContext)
    .on("brush", brushed);
```

We can react to the *event* and update the linked visualizations in a separate function. Particularly noteworthy is the property `brush.extent()` which returns the current brush extent or in other words, it returns the user's selection.

```
function brushed() {
    // Set new domain if brush (user selection) is not empty
    xFocus.domain(
        brush.empty() ? xContext.domain() : brush.extent()
    );

    // Update focus chart (detailed information)
    ...
}
```

We have initialized the brush component and implemented an event listener. In the last step it is necessary to append the brush to the *context chart*. This step is also comparable to appending a D3 axis. An additional rectangle, within the SVG group element, should indicate the current selection for the user:

```
// Append brush component
svg.append("g")
      .attr("class", "x brush")
      .call(brush)
    .selectAll("rect")
      .attr("y", -6)
      .attr("height", height + 7);
```

Activity III - Implement Focus+Context

In the next activity you will have to implement a second chart with an interactive brushing component. Most of the parts are already complete but you have to connect the components to each other.

1. **Create an instace of `Timeline`**

   The result of this lab should be a stacked area chart (focus) linked with a basic area chart (context). To avoid confusion, we will call the basic area chart from now on "timeline" because it shows the full time period.

   You can find the implementation of the timeline in `js/timeline.js`. The self-contained component follows our methodology and can be easily integrated into the system.

   → Open `timeline.js` and get an overview of the implementation. It is much shorter than the stacked area chart because the area path and the axis is static and we don't need any data wrangling tasks.

   → Create an instance of `Timeline` and use the constructor to specify the properties (see *constructor function* in `timeline.js`). The timeline should not include the detailed layers of our data, but it should indicate the general trend. Therefore, the data `allData.years` is perfectly suited.

   ↻ Reload the webpage in your browser. If you have created the instance correctly you should see the timeline below your stacked area chart. Otherwise make sure that you have included the right parameters (container ID, data).

2. **Implement the brushing component**

   You have to extend the timeline and implement the described focus+context approach for the visualization. The prior brushing example should help you to include the necessary pieces of code.

   → Initialize the brushing component in the `initVis()` function

   → Create a global function `brushed` in *main.js*

   - The implementation in this file acts as a controller and can access both charts. Alternatively, and more important for larger systems, you can include a more sophisticated event handling mechanism. But for this example it is enough to create a global function and link the two visualizations directly.

   - Update the x domain of the stacked area chart every time the *brushed* event gets triggered. And don't forget to update the chart by calling:

     ```
     areachart.wrangleData();
     ```

   → Append the brush component to the *timeline* SVG

   ↻ Reload the webpage in your browser. You should be able to choose a specific time period in the timeline (context) and the stacked area chart (focus) will automatically zoom in and will show the details for the selected years.

But now there is probably an error in the stacked area chart. When you zoom in, the layers will expand and go beyond the y-axis until they reach the end of the SVG container. You should solve the problem by using a rectangle as an overlay and clipping the path:

Paste this code snippet after creating the SVG element:

```
vis.svg.append("defs").append("clipPath")
    .attr("id", "clip")
  .append("rect")
    .attr("width", vis.width)
    .attr("height", vis.height);
```

Activity IV - Implement basic tooltips

Currently it is not possible to identify layers and to get detailed information for a specific year. All of this would require an advanced tooltip solution.

You will implement a lightweight version of these tooltips which should help the user to identify just the category names.

1. **Append a tooltip placeholder**

   Open `stackedAreaChart.js` and append an empty SVG text element. This element should be created only once. It should be used as a placeholder for the category names and should get updated every time the user hovers over a layer.

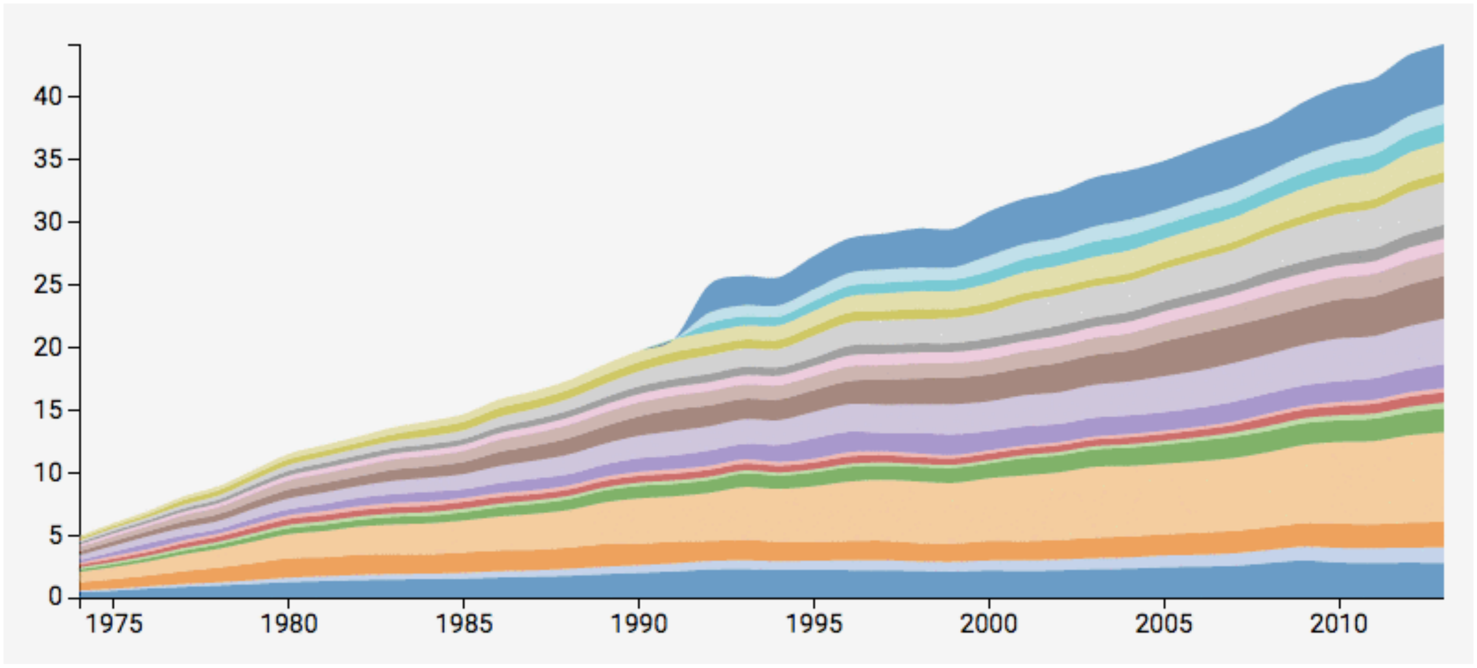
   Feel free to choose proper coordinates. In the preview the label is positioned in the top left corner of the stacked area chart.

2. **Update the tooltip**

   Create a mouse listener for the *SVG path* (layer of stacked area chart, in `updateVis()`) and update the `text` property of the tooltip-element every time the user hovers over it.

Bonus Activity (optional) - Show details for a specific layer

Due to the stacked layout it is sometimes difficult to see the exact evolution of a specific layer. An extension which allows the user to click on a category in order to see the details would solve this problem.



You should implement:

- A variable `filter` that stores the current category (if nothing is selected, the *filter* variable should be empty)
- A click listener for the layers (in addition to the previous mouse hover listener) that triggers the update of the stacked area chart. If the user clicks on the same category again the visualization should jump back to its default state.

  Example for a click listener with toggle feature:

  ```
  .on("click", function(d) {
      vis.filter = (vis.filter == d.name) ? "" : d.name;
      vis.wrangleData();
  })
  ```

- A second path generator *d3.svg.area()* in `initVis()` for displaying a single layer. You can go to *timeline.js* to see the path generator for a basic SVG area. (Basically this whole feature you are implementing is just a switch between the stacked area chart and a basic area chart.)

- An extension of the function `wrangleData()`. If the user selects a specific layer the *display data* should be filtered:

  ```
  .filter(function(d) { return d.name == vis.filter } );
  ```

  Make sure you add an IF statment and update `vis.displayData` accordingly.

- An extension of the “domain update” for the ***y-scale*** (in `updateVis()` . If you want to display a typical area chart without layers you have a fixed baseline at zero and don’t need to consider the *y0* parameter..

```
if(vis.filter) {
    vis.y.domain([0, d3.max(vis.displayData[0].values,
            function(d) { return d.y; })]);
} else {
    ...
}
```

- An IF statement in the *update* section to switch between the simple path generator (area) and the advanced path generator (stacked area) depending on wether the user has selected a category (in `updateVis()`):

```
.attr("d", function(d) {
    if(vis.filter)
        return ...
    else
        return vis.area(d.values);
})
```

Great job, you now have a focus-and-context visualization that even allows you to get details on individually selected categories!

Submission of 1-minute paper

Congratulations, you have now completed the activities of Lab 7! Please submit your 1-minute paper now!

See you next week!

Submission of lab (activity I - IV)

Please upload the code of your completed lab (the final focus-and-context visualization you created in activities I-IV) on Vocareum!

Resources

- p. 206-209 in *D3 - Interactive Data Visualization for the Web* by Scott Murray

- http://javascriptplayground.com/blog/2012/04/javascript-variable-scope-this/
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction*to*Object-Oriented_JavaScript
- http://www.w3schools.com/js/default.asp