Combining React and D3.js for data visualization purposes without introducing performance losses in the browser

Maximilian Zauner

MASTERARBEIT

eingereicht am

Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im Juli 2019

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, July 12, 2019

Maximilian Zauner

Contents

Abstract

This should be a 1-page (maximum) summary of your work in English.

Chapter 1

Introduction

This chapter introduces the reader to the general concept of the master project. Not only the initial situation and motives should be clear after reading this section of the thesis, but also what the thesis project is about and what it tries to achieve.

1.1 Problem description and motivation

With D3 being one of the most powerful data visualization libraries in the JavaScript environment, it is shocking, how bad developer experience can be when writing extensive amounts of D3 code. D3 is a reasonably old library as the development of the library started in early 2011 when the scripting language JavaScript was in a completely different state than it is now. In that era, most developers would have never even thought about utilizing JavaScript as the primary technology to use to realize big enterprise web projects.

Even tough D3 went through a few significant rewrites and got refactored multiple times throughout some major versions; the base API is still quite similar to the original API. This fact is quite noticeable when trying to write a lot of D3 production code that must be kept maintainable for multiple developers while still performing well.

Since the API might be hard to use in production environments, the idea to combine the D3 library with a widely used JavaScript library called React came to mind. React is currently used in big projects like Facebook, Airbnb, Netflix, or Spotify. The claim that developer experience improves by using D3 through React might be subjective, but even if only a few developers who are willing to use React combined with D3 over pure D3 use the thesis project in their software the project is a valuable addition to the developer community. The exciting research aspect then would be, if the combination is possible without losing render performance but still providing the convenience of programming React code.

1.2 Goals of the project

The central research aspect of the thesis focuses on the performance aspect of the combination of React and D3, not how the React version might be easier to use than pure vanilla D3. Subjective opinions can hardly be measured scientifically as the number of probands to measure developer experience on different library versions would have to be quite high to be able to get accurate heuristic results. Performance numbers, on the other hand, can easily be measured. A combination of the two libraries which would allow programmers to use some of D3's functionality by writing declarative React code without introducing any performance penalties would be a valuable software addition to the React community. Methods of how a combination of the technologies can be achieved are elaborated in this paper. Also, some already existing work is explained and analyzed.

The thesis aims to introduce the reader to the general concept of the two libraries – React and D3 – which are combined in the thesis project. The paper not only provides a general overview but also an introduction to the two libraries for an even better understanding of the thesis project. All required knowledge should be able to be acquired throughout the first chapters to then being able to understand the performance discussions in the later chapters which also compare different approaches of possible implementations regarding the combination of the libraries. It is then easy to follow the discussion by already having the necessary knowledge to understand all required aspects of the mentioned technologies.

Another significant part of the thesis is the description of the thesis project itself. The combination of the libraries React and D3 is a software project that was created out of a requirement. The goals are to create a system, that lets developers write declarative React code, avoid imperative D3 code but use D3's data visualization technologies. Also, another goal is to keep performance losses at a minimum but still use the full extent of React's features of writing web components. Of course, there are many ways to achieve the same goal; that's also why the thesis project provides 3 discussable prototypes. The thesis project chapter explains the implementation and functionality of each prototype. Ultimately, all prototypes are compared to each other.

The performance comparison of the prototypes is one of the most exciting parts of the thesis. Each prototype is tested on different devices on different browsers which generate some performance numbers that are also compared and discussed. While discussing raw performance numbers can give an insight into how the prototypes perform, they can also be essential to measure user experience. A vital part of the thesis is the explanation of user perception of animated content in the browser. The focus primarily lies on researching the threshold on which users percept an animation as not smooth anymore.

Ultimately there is the goal of introducing the reader to the open source concept that is planned for the thesis project. Initially, a specific use-case was the reason to create the library that connects D3 and React, but there are most certainly other developers that can make use of the thesis project as well. The paper provides a general overview of how the public API of the technology is designed and how the project will be published on the npm package registry to being able to include the library in any project.

Chapter 2

D3.js – Data-Driven Documents

Chapter 2 provides an overview of the popular JavaScript library D3 which is used to simplify implementations of any data visualizations in the web for developers. It provides an overview of the libraries beginnings but also goes into detail about how to implement projects with D3. The knowledge is required to understand the performance comparisons in chapter 5 and 7.

2.1 Introduction to D3

D3 is a JavaScript library that helps developers create highly sophisticated data visualizations on the web via a universal tool that is platform agnostic: the browser. The official documentation of D3 in [1] explains the library as a toolkit, that allows binding data to the DOM. Also, it gives an overview of the vast amount of helpful tools that can be used to visualize data. The library includes all kinds of functionality, ranging from simple array and mathematic operations to complex simulations, that are calculated in real time.

Some of the most popular features of D3 is to render user interactable animated charts. Not only is it possible to easily create a bar chart for instance, but all other kinds of charts as well. A full list of available packages can be found in [3]. The library is prevalent amongst data scientists as it is quite easy to create complex data visualizations in the web quickly.

D3 also provides some other utility functions that can be useful in many use-cases. There is, for example, a module, that calculates chromatic colors for charts to get colors that have the maximum diversity to each other to be easily distinguishable as seen in [3, /d3-scale-chromatic]. Another example in [3, /d3-array] shows, that the library also provides some useful array manipulating functions which come in handy when having to deal with big data sets. Also generating random numbers via various distributions is no problem when using the d3-random package in [3, /d3-random]. The list of useful data manipulation tools goes on and dealing with every aspect of the library would go far beyond the scope of this paper.

What makes D3 unique though is the possibility to create individual data structures for rendering sophisticated data visualizations. The library provides scatter plots in [3, /d3-scale] or pie charts, line charts, area charts, radial bar charts, treemaps in [3, /d3-shape] to name a few. D3 also provides utility functions to add labels or user interaction to each mentioned and not mentioned data visualization type. Also, a significant advantage of using D3 is, that it also provides simple methods to transform any D3 visualization into being user interactable by creating floating tooltips or sliders, switches, or knobs, which control the visualization.

Due to the immense size of the library and its many data manipulation tools, D3 is divided into different sub-modules to prevent users of the library having to download the full library code bundle in the browser to be able to use the library. [3] shows a full list of every available tool that can be used in composition with the base package of D3. When using D3 in a big production project, all modules can be integrated into any project via using nodes package manager npm and download it from its registry[1].

There are multiple examples on the documentation's example page in [2] which show what developers can achieve by using the D3 library. The API documentation is a comprehensive documentation of the complete feature set of D3 as seen in [3, /d3/blob/master/API.md].

2.2 Explaining the D3 API

The explanation of D3's API is of utmost importance as the knowledge is crucial for understanding the comparisons of React and D3 in chapter 4. As mentioned before, the D3 API mostly consists of consecutive chained imperative function calls, that manipulate the visualization data and binds it to the DOM. This section aims to discuss the most vital aspects of the D3's API to understand code samples, that are presented in later chapters.

Selecting DOM nodes and creating a D3 selection model is, therefore, a vital aspect of D3's API. Via selection D3 can connect JavaScript application data to actual DOM nodes as [3, /d3-selection] shows. An example can be seen in 2.1. Because the library is not declarative, each node that is added or removed is handled via a chained function call as the append function in 2.1 shows very well. When adding or removing multiple dom nodes (in case the individual nodes of the simulation are complicated dom structures), the code quickly gets very incomprehensive. The provided example in 2.1 only appends one simple circle element for instance.

Besides the feature of selecting entering and exiting nodes, there is also the possibility to add transitions to node selections. With the help of D3, advanced animations and transitions can be added via a simple function call. Line 8 in 2.1 shows a selection where all nodes are selected that are removed after the data has changed. The color is changed, and a transition is added, which transforms the radius attribute of the node until it reaches the specified amount, 1e-6 in this case.

The code in 2.1 also clearly shows, that every attribute and style instruction of and

[1] https://www.npmjs.com/search?q=d3

Program 2.1: D3 selection, enter, and exit example

```
1 // earlier in the script
2
3 const svg = d3.select('.container')
4 const node = svg.selectAll('.node')
5
6 // handling data changes of the simulation
7
8 node
9   .exit()
10  .style('fill', '#b26745')
11  .transition(t)
12  .attr('r', 1e-6)
13  .remove()
14
15 node
16  .transition(t)
17  .style('fill', '#3a403d')
18  .attr('r', (node) => node.size)
19
20 node = node
21  .enter()
22  .append('circle')
23  .style('fill', '#45b29d')
24  .attr('r', (node) => node.size)
25  .attr('id', (node) => node.name)
```

added DOM nodes have to be handled via a chained function call. On line 23 in 2.1 the `fill` property is added to the `<circle>` svg element. Each additional style property would require a consecutive call of the `style('property', [style])` function.

Also, a key component of D3's API is the possibility to pass attribute handling functions instead of hardcoded values. Those computed properties can be found in line 24 and 25 in 2.1 for example. By passing a function to property or attribute setters, the passed function is called by D3 as a callback by providing each node's data to the callback. When rendering 10 nodes, the attribute function on line 24 in 2.1 which sets the radius of the node would, therefore, be called 10 times as well, setting the radius for each specific node individually.

The code example 2.2 is taken from production code and shows how hard to read D3 code can get if multiple DOM changes have to be handled imperatively. Not only the addition of the nodes has to be handled via function calls, but also some general node properties like CSS styles or custom attributes.

D3 also has the concept of entering and exiting selections. The code in 2.1 shows, there are the enter and exit selection function calls that can be used to explicitly handle nodes that enter and exit according to the data that will be bound to the DOM. There is also a transition call, which lets developers specify how to handle DOM elements that stay in the visualization if the data is updated. The chained function calls then can handle those selections accordingly. Again, the provided example is a straightforward form of a

Program 2.2: Negative example of how confusing and unmaintainable D3 code can get

```
d3.select(_this).classed('active', true)
d3.select(_this)
  .select('.circle')
  .transition(500)
  .attr('stroke', function(d) {
    if (d.rings && d.rings.length > 0) return '#404348'
    return d.color || COLORS[d.type.toUpperCase()] || '#27292c'
  })
  .attr('fill', function(d) {
    return '#404348'
  })
  .style('filter', 'drop-shadow(0 3px 4.7px rgba(0,0,0,.54))')
d3.select(_this)
  .selectAll('.ring')
  .transition(500)
  .attr('opacity', 1)
d3.select(_this)
  .selectAll('.node-background')
  .transition(500)
  .attr('opacity', 0)
d3.select(_this)
  .selectAll('.sub-circle')
  .transition(500)
  .attr('cx', function(d, i) {
    let deg = ((Math.PI * 2) / 8) * i - Math.PI
    let x = Math.sin(deg)
    let offset = event.rings ? event.rings.length * 15 : 0
    return x * (d.r + 5 + offset)
  })
  .attr('cy', (d, i) => {
    let deg = ((Math.PI * 2) / 8) * i - Math.PI
    let y = Math.cos(deg)
    let offset = event.rings ? event.rings.length * 15 : 0
    return y * (d.r + 5 + offset)
  })
  .attr('stroke', '#FFF')
```

Program 2.3: Sample initialization of a D3 force graph

```
const simulation = forceSimulation(data)
  .force('charge', forceManyBody().strength(-150))
  .force('forceX', forceX().strength(0.1))
  .force('forceY', forceY().strength(0.1))
  .force('center', forceCenter())
  .alphaTarget(1)
  .on('tick', ticked)
```

Program 2.4: D3 written in a fictional functional way

```
1
2 const forceParams = compose(
3   force('charge', pipe(forceManyBody(), strength(-150))),
4   force('forceX', pipe(forceX(), strength(0.1))),
5   force('forceY', pipe(forceY(), strength(0.1))),
6   force('center', forceCenter()),
7 )
8
9 const simulation = compose(
10   forceParams,
11   alphaTarget(1),
12   on('tick', ticked),
13   forceSimulation
14 )(data)
```

simulation. If the DOM structure would be more complex, handling any selection would produce confusing and unreadable code very quickly as 2.2 clearly shows.

According to the documentation in [3] the library D3 was created in 2010. Thus it can be noticed, that the libraries API originates from a time, where developers would not even think about using JavaScript in production. Therefore any code written with D3 is not very scalable. The example 2.3 shows, that the API is written in a very imperative way. Also, the library makes use of a pattern called ”chaining”. The pattern works because each function returns an instance of itself so an infinite amount of functions can be added to the chain, making the code more flexible.

Over the years better software patterns emerged as experience had shown, that chaining is a software pattern that was new at the time but can cause code that is hard to maintain. Nowadays this library would probably be written with a functional approach, letting developers compose their simulations via functional composition as various libraries and frameworks on the internet already do. The problem with the example 2.3 is that the code cannot be reused, as it is hardcoded into the chain. Nowadays many frameworks for building frontend applications use a declarative approach for binding the view to the data model. D3 still uses an imperative software pattern which forces developers to chain multiple library functions to control multiple elements in the dom.

If a functional approach would be used the code from 2.3 would look more like in example 2.4. The difference between the two code examples in 2.3 and 2.4 might appear to be very subtle, but in reality, it is very significant. By looking closer, it is clear, that the variable `forceParams` is a functional composition of methods, that can be reused multiple times in the application. In the first example in 2.3 the simulation configuration is locked in the function chain. Chaining is a pattern which can easily cause duplicated code in any codebase.

Writing functional D3 code could also alleviate the confusing unmaintainable code in 2.2 as developers can easily compose repeating DOM manipulation sequences and reuse them throughout the codebase without having to touch code on multiple files in case

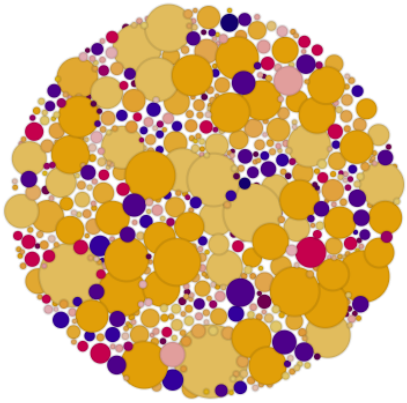


Figure 2.1: The force graph with default center force. All nodes are attracted to the same center without overlapping each other.

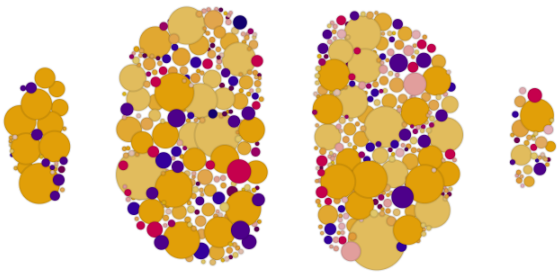


Figure 2.2: A sample force graph with more than one center force. Having more than one force centers means that different nodes are attracted to their assigned center force.

there is a bug that effects multiple aspects of the application.

2.3 Force Graphs – Real time rendered data visualizations

Due to the immense size of D3, the focus of this thesis and its project lies on a rather ”small” but quite important part of the library – the force graph simulation. It is the graph type that is integrated into React as showcased by the thesis project. The visualizations consist of objects that interact with each other in a two-dimensional space. By interacting and moving objects all other objects in the animation are also affected. Figures 2.1 and 2.2 show an example of D3’s force simulation. In figure 2.1 there is a single center force that keeps all nodes in the center but also keeps individual nodes from overlapping each other. Force graphs can also be configured to make nodes reject each other even further than their actual size as figure 2.4 shows. It is also possible to implement so-called links, that also add some complexity to the simulation, as nodes are

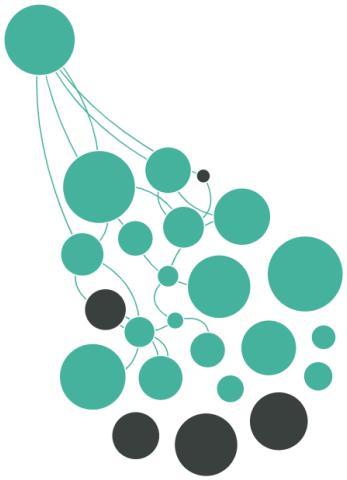


Figure 2.3: A sample force graph where the top node is dragged up to the left and the other nodes are dragged along. The force is still keeping the other nodes apart and also drawn to the center though.

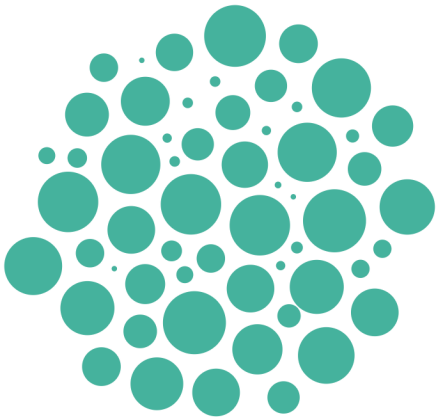


Figure 2.4: A sample force graph with one center force. The nodes are configured to reject each other with the function `r+r/2`

dependent on each other and not only reject each other but also attract linked nodes as figure 2.2 and 2.5 shows.

As previously mentioned, all force simulations are calculated, animated, and rendered in the browser which also includes user interaction. The user can for example drag nodes around which of course then affects other nodes and the whole simulation. Figure 2.3 shows well, how dragging one node affects the whole force graph, as all connected nodes follow the dragged node while still rejecting each other and while being attracted to the center force.

D3 provides a somewhat simplified API to be able to quickly implement force graphs

Program 2.5: Code snippets for D3 force simulation code

```
1 simulation.forceSimulation([nodes]) // factory method for a standard force simulation
2 simulation.tick([iterations]) // called on every tick the simulation goes through
3 simulation.start() // starts a stopped simulation
4 simulation.stop() // stops a started simulation
5 simulation.restart() // restarts a simuliation, resets alpha
6 simulation.alpha([alpha]) // directly sets alpha value
7 simulation.alphaTarget([alphaTarget]) // sets alpha target value
```

Figure 2.5: A sample force graph where some nodes are linked together while still rejecting each other;

in the browser as it can be read in [3, /d3-force/blob/master/README.md]. The way force simulations work is that developers first have to define or build the simulation. There is a factory method as seen in line 1 which takes the nodes of the graph as an argument and builds a default simulation. The nodes have to be provided in a particular scheme so D3 can correctly parse the node array.

Another very important aspect of force graphs is the so called ”alpha” value system as documented in [3, /d3-force/blob/master/README.md], which controls how long the simulation lives. The alpha valueis a gradually decaying value that makes the simulation stop if a certain value is reached. Every simulationhas a function that is called every ”tick” of the simulation as shown in line 2. Everytick the alpha value decays via a predefinable function, it happens logarithmically per default. The tickingfunction takes a handling function will be passed every node position in the simulation which then letsdevelopers link the data to the dom with D3 again. The tick function will be very important lateron when thecombination of D3 and React is explained in more detail.

If there is a user interaction, the simulation sometimes has to be restarted or reheated. Programmers can set alpha values and targets to reheat or restart the simulation in case a node is dragged by the user which would possibly require many other nodes in the simulation to react to that user input. That way also the speed of the simula-

tion can be controlled via setting a custom decay function. The documentation in [3, /d3-force/blob/master/README.md] points to a few methods that can achieve said functionality. There is for example the functions which can be found in line 3, line 4, and line 5 that can be used to reheat a simulation. Also the functions in line 6 and line 7 values can be set directly to alter the simulation's life span.

Chapter 3

React – A JavaScript library for building user interfaces

3.1 Introduction to ReactJS

3.2 Explaining the React API

3.3 Why immutability in combination with React works great

Chapter 4

Data Visualization

4.1 Introduction to force graphs

4.2 Prototypes

4.2.1 Pure D3 prototype

4.2.2 Pure React prototype

4.2.3 D3 and React hybrid

4.3 Comparison of the different proposed Prototypes

4.4 Building a stable Testing Environment

4.5 Testing methodologies

4.6 Testing devices

Chapter 5

Performance Testing and user perception

5.1 Introduction of Human Perception of fluid animations

5.2 Introducing the test results

5.3 Interpreting the test results

5.4 Summary of the different prototypes

Chapter 6

Open source and the React community

6.1 Building an open source library for React.js developers

6.2 Designing a library API

Chapter 7

Conclusion

References

Online sources

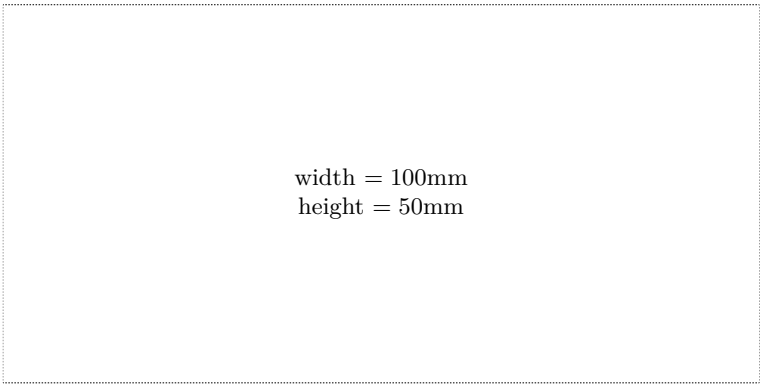
[1] Mike Bostock. *D3 Data Driven Documents*. 2011. URL: https://d3js.org (visited on 01/10/2019) (cit. on p. 3).

[3] Mike Bostock. *D3 Data Driven Documents*. 2011. URL: https://github.com/d3 (visited on 01/10/2019) (cit. on pp. 3, 4, 7, 10, 11).

[2] Mike Bostock. *D3 Data Driven Documents*. 2011. URL: https://bl.ocks.org/mbostock (visited on 01/10/2019) (cit. on p. 4).

Check Final Print Size

— Check final print size! —



— Remove this page after printing! —