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

Contents

Declaration iii

Abstract vi

1 Introduction 1
- 1.1 Problem description and motivation . . . 1
- 1.2 Goals of the project . . . 2

2 D3.js – Data-Driven Documents 3
- 2.1 Introduction to D3 . . . 3
- 2.2 Explaining the D3 API . . . 4
- 2.3 Force Graphs – Real time rendered data visualizations . . . 7

3 React – A JavaScript library for building user interfaces 12
- 3.1 Introduction to ReactJS . . . 12
- 3.2 Explaining the React API . . . 14
  - 3.2.1 JSX in general . . . 15
  - 3.2.2 Explaining React components . . . 16
- 3.3 React's component lifecycle . . . 19
- 3.4 Conclusion . . . 22

4 Data Visualization 23
- 4.1 Introduction to force graphs . . . 23
- 4.2 Prototypes . . . 23
  - 4.2.1 Pure D3 prototype . . . 23
  - 4.2.2 Pure React prototype . . . 23
  - 4.2.3 D3 and React hybrid . . . 23
- 4.3 Comparison of the different proposed Prototypes . . . 23
- 4.4 Building a stable Testing Environment . . . 23
- 4.5 Testing methodologies . . . 23
- 4.6 Testing devices . . . 23

5 Performance Testing and user perception 24
- 5.1 Introduction of Human Perception of fluid animations . . . 24
- 5.2 Introducing the test results . . . 24
- 5.3 Interpreting the test results . . . 24

5.4 Summary of the different prototypes 24

6 Open source and the React community **25**
6.1 Building an open source library for React.js developers 25
6.2 Designing a library API 25

7 Conclusion **26**

References **27**
Literature 27
Audio-visual media 27
Software 27
Online sources 28

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 not only 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 [8] 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 [9]. 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 [9, /d3-scale-chromatic]. Another example in [9, /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 [9, /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 [9, /d3-scale] or pie charts, line charts, area charts, radial bar charts, tree maps in [9, /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. [9] 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 [10] 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 [9, /d3/blob/master/API.md].

2.2 Explaining the D3 API

This section aims to discuss the most vital aspects of the D3's API to understand code samples, that are presented in later chapters. Some general knowledge 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. [2, P. 625] describes the imperative programming pattern as a static division of a program into its concurrent tasks which means, that the programmer uses statements to change the programs state.

According to the documentation in [9] the library D3 was created in 2010. Thus it can be noticed, that the libraries API originates from a time, where developers did not even think about using JavaScript in productive or even enterprise environments. Therefore large codebases written with D3 tend to be hardly scalable and difficult to maintain. Multiple instruction function calls in 2.1 show, that D3 code is indeed imperative. Also, the library makes use of a software pattern called "chaining". The pattern works because each function returns an instance of itself to enable the addition of an infinite amount of functions that can be added to the chain.

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 [9, /d3-selection] shows. An example can be seen in 2.1. Because the library is imperative, 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

[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)
```

nodes and the individual nodes of the simulation are complicated DOM structures, the code quickly gets very incomprehensive as demonstrated in 2.2. The provided example in 2.1 only appends one simple circle element for instance though in comparison to the much more complicated example in 2.2.

Not only is it possible to select DOM nodes via D3 but the library also contains the feature of selecting entering and exiting nodes as lines 9 and 21 showcase in 2.1. The enter and exit selection function calls can be used to explicitly handle nodes that enter and exit the visualiization according to the data that is bound to the DOM. The chained function calls can then handle the enter and exit selections accordingly. Line 21 in 2.1 shows an enter selection which appends a circle svg element for each new data object and also applies various attributes and a style.

When the data of a visualization changes, some nodes might be deleted, some new nodes might appear but some nodes might also stay in the visualization but change their position. D3 covers that use-cases by including the possibility to add transitions to node selections. The transition feature lets developers specify how to handle DOM elements that stay in the visualization if the data is updated. Advanced animations and transitions can be added via a simple function call. Line 8 in 2.1 for example shows a selection where all nodes are selected that are removed after the data has changed. Furthermore 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. Finally the

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')
```

node is then removed completely from the DOM resulting in a nice animation of the node exiting the visualization.

The code in 2.1 also clearly shows, that every attribute and style instruction of and 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. By passing a function to property or attribute setters like the `.style(’attribute’, [style])` method, the passed handler 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

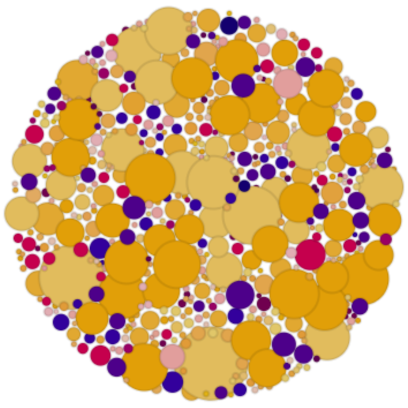


Figure 2.1: The force graph with default center force. All nodes are attracted to the same center without overlapping each other.

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.

Over the years better software patterns emerged and experience shows, 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 2.1 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. More information about declarative approaches can be found in chapter 3. D3's API still uses the imperative software pattern which forces developers to chain library function statements to control multiple elements in the DOM.

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

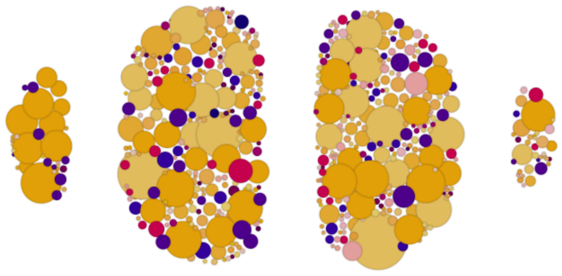


Figure 2.2: A sample force graph with more than one center force. Having more than one force centers means that different nodes are attracted to their assigned center force.

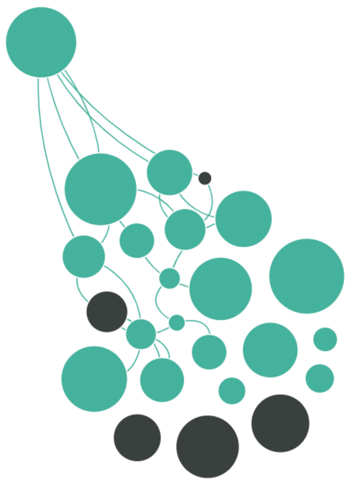


Figure 2.3: A sample force graph where the top node is dragged up to the left and the other nodes are dragged along. The force is still keeping the other nodes apart and also drawn to the center though.

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 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 affect other nodes and the whole simulation as well. 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

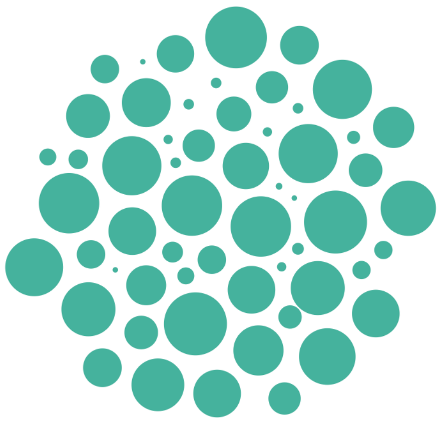


Figure 2.4: A sample force graph with one center force. The nodes are configured to reject each other with the function `r+r/2`

Program 2.3: Code snippets for D3 force simulation code

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

Program 2.4: 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)
```

in the browser as it can be read in [9, /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 of 2.3 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 [9, /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

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

”tick” of the simulation as shown in line 2 of 2.3. 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. 2.4 is a simple example that shows how to initialize a D3 force simulation.

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 simulation can be controlled via setting a custom decay function. The documentation in [9, /d3-force/blob/master/README.md] points to a few methods that can achieve said functionality. The functions on line 3, 4, and 5 of 2.3 can be used to reheat a simulation. Also the functions in line 6 and 7 of 2.3 can be used to set values directly to alter the simulation’s life span.

If a functional approach would be used the code from 2.4 would look more like in example 2.5. The difference between the two code examples in 2.4 and 2.5 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 2.4 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 there is a bug that effects multiple aspects of the application.

Program 2.5: D3 written in a fictional functional way

```

const forceParams = compose(
  force('charge', pipe(forceManyBody(), strength(-150))),
  force('forceX', pipe(forceX(), strength(0.1))),
  force('forceY', pipe(forceY(), strength(0.1))),
  force('center', forceCenter()),
)

const simulation = compose(
  forceParams,
  alphaTarget(1),
  on('tick', ticked),
  forceSimulation
)(data)
```

Chapter 3

React – A JavaScript library for building user interfaces

This chapter introduces the reader to the prevalent and widespread front end library called ReactJS. It explains an essential part of React – its rendering cycle – which the reader needs to understand at least on a high level to be able to follow upcoming explanations of how the thesis project was implemented. Additionally, the paper elaborates the difference of how React uses declarative code to render data, whereas D3 uses an imperative API to render its data.

3.1 Introduction to ReactJS

The easiest way to find information about React is to visit its official website [1]. There is a statement in [12] up front that says:

> React is a JavaScript library for building user interfaces.

which describes React very well. A Facebook engineer called Jordan Walke founded the library in 2011, as presented in [3, 05:30]. Walke wanted to create a tool that would improve the code quality of their internal tool called "Facebook ads". Up until then, Facebook continued to develop and use React internally, but since the year 2013, the project is entirely open source. Since the initial open-source release up until now, not only technical engineers of Facebook but also the React open source community itself have been maintaining the library. In late 2017 Facebook even changed React's BSD license to the MIT license, which is even better for the React community, as the MIT license has lesser restrictions than the BSD license.

According to [12], Facebook sees React as a declarative and component-based library. However, a question might come to mind: "What exactly does it mean for a library to be declarative and component-based?". The answer to this question might be more straightforward than initially anticipated. In [1] declarative programming is described

[1] https://reactjs.org

as a programming pattern, that expresses the logic of a program without describing its control flow. This means that the actual code only describes what has to be computed not necessarily how it should be done exactly by stating every action explicitly via a function call, for example. Declarative programming can be understood as a layer of abstraction, that makes software easier to understand for readers of the code. Declarative programming is therefore very different from the imperative programming pattern described in chapter 2. React's approach of handling the presentation layer is declarative since its API lets developers describe how the application has to look like at any given data variation, which is quite the contrary to D3's API as it can be read in 2. Further information about React's API can be found in section 3.2 though.

Enabling developers to create a highly component oriented architecture in their software is a fundamental aspect of React as well. Using a component-based library can increase productivity a tremendous amount. However, what does it mean for a library to favor component based architecture? After the initial setup of some boilerplate code, React makes it exceptionally easy to reuse existing components in the codebase to allow even faster development cycles. Once standard input components like buttons or text fields and layout components like page or header components are implemented, they can be reused throughout the whole app; thus significant progress can be achieved in a very short amount of time. Components can be manipulated by passing different properties, which might result in different presentation results of the components. More in-depth information about how React handles components and its props can be found in section 3.2.

React components can have multiple applications. There are presentational components, for example, which are pure functions that represent the current application state. Though there are also stateful Components which can hold some application state and react to state changes accordingly via rendering again, react however makes no assumptions about the technology stack that is used in a project as [12] claims. This means that users of the library can decide for themselves if they want to use the built-in state management functionality or if they want to use a third-party library for solving specific problems like global application state for example.

The documentation in [12, /docs] claims that the library makes use of a so-called ”virtual DOM”. This means that React keeps track of its state data to prevent unnecessary writes to the actual DOM object. JavaScript performs exceptionally well when handling pure JavaScript objects in memory. Keeping the DOM tree of the application in the JavaScript engine's heap as a representation of objects enables React primarily apply updates this so-called virtual DOM. React then compares the newly applied data with the old tree to then being able to decide updates need to be committed to the DOM. Writing or committing to the DOM is the most expensive type of work in the browser, so React tries to keep DOM manipulating actions to a minimum. The React team calls the diffing algorithm ”reconciliation algorithm”. It would go out of the scope of this paper to go more in depth of the algorithm, so it is recommended to read about React's reconciliation algorithm in its documentation [12, /docs].

React is a view layer that favors unidirectional data-flow. Every time the application state changes, the whole new data object is passed to React again. As mentioned in

[3, 6:50], the speaker describes the functionality very well via explaining React as a simplified function that could look like this: `f(data) = UI`. Hence, React can be seen as the view layer that handles presentation as a function of state and data. Once the data has updated the virtual dom, the virtual dom is then passed to React's reconciliation algorithm, which then determines if any nodes have to be changed on the real DOM. If there would be a React component that always renders the same `<div>` with the same data, rendering that very component multiple times would not result in React writing multiple DOM nodes to the browser. The reconciliation algorithm sees that the virtual dom matches the real dom in this case, which results in React not updating the real DOM. Of course, if the component's content is dynamic, the component sometimes has to be re-rendered according to the data changes. If some parts of the data stay the same even after being reapplied to a component, only newly added or removed nodes are committed to the DOM. Even though the reconciliation algorithm prevents expensive DOM operations, the algorithm itself can also be expensive. The documentation in [12, /docs/optimizing-performance.html#avoid-reconciliation] advises developers to try to avoid reconciliation to improve performance.

Unidirectional data-flow implicitly means to React developers, that there is no data binding and no template language. The library only uses `React.createElement([element])` calls internally, which are hidden behind the so-called "JSX" JavaScript language extension. JSX will be explained more i depth in section 3.2. As mentioned before, React is just a pure idempotent function of its application state, which means that the same data always produces the same presentation. That fact also implies that if the application data has to be changed, a new "patched" version of the application data has to be created instead of mutating the currently available application state. The newly created data then flows into the React render cycle again. Unidirectional data-flow is also the reason why React works well with immutable data structures. This paper assumes that the reader knows about immutable data structures, but [11] explains exceptionally well, what Immutable data structures are and how they're used in JavaScript. Going more in-depth on how React works well with immutable state would go out of the scope of this thesis though. It is just essential to know that every time the data changes, React triggers a whole new render cycle of the component tree. The immutable data structures help React to work out changes in the data structure when using immutable data structures. Instead of having to implement recursive data comparison functions, nested data object tree differences can be checked via a cheap equality check.

3.2 Explaining the React API

To follow performance discussions and elaborations about the thesis project's prototypes, a general high-level understanding of the API is required. This section introduces the reader to React's public API. The section does not aim to be a tutorial on how to program React applications, but rather to be a high-level explanation of how the API works. Reading this section makes it easy to understand the differences and similarities of React and D3 and how the two libraries play together and how they're also completely different.

Program 3.1: Creating a React element with JSX

```
const ReactElement = (
  <div className="hello-world">
    Hello <span className="emph-text">World</span>!
  </div>
)
```

Program 3.2: Creating a React element without JSX

```
 1 const ReactElement = React.createElement(
 2   "div",
 3   { className: "hello-world" },
 4   "Hello ",
 5   React.createElement(
 6     "span",
 7     { className: "emph-text" },
 8     "World"
 9   ),
10   "!"
11 );
```

3.2.1 JSX in general

Probably one of the most important aspects of React's API is the JavaScript language extension called ”JSX” which simplifies the use of React greatly and produces much more readable code. The example in 3.1 shows an example React component that is written in JSX. When looking at the transpiled output in 3.2 it is clear how JSX helps to reduce the amount of code and how it greatly improves readability. The code in 3.2 also shows that React is just a big composition of `React.createElement([element])` calls under the hood. When writing JSX code, in reality, it is writing declarative code that is just a functional composition of React components.

Notice, how the createElement function takes up to 3 parameters as documented in [12, /docs/react-api.html]. The first parameter is the element type (the type can also be a custom component that was created by a user or a downloaded third-party component). The second parameter is used to pass the element's current properties, and the third parameter describes the component's children. That third parameter makes it possible to compose multiple React components together, as children are nestable.

Line 2 and 3 in 3.2 show, how a React element is created. A node of type `"div"` is created and the property `{className: "helo-world"}` is passed. Each parameter after Line 3 is a child of the created `<div>` node. The React element has 3 children which is demonstrated by the code example where the element is written in JSX in 3.1. First, there is the string ”Hello ”, then there is a `<span>` which also has children, and finally there is the exclamation mark string at the end. When going back to the transpiled code example in 3.2, lines 4 to 10 exactly show what kind of children are passed to React's element creating function. Notice, that the class property has to be ”className” in JSX instead of ”class” because JSX is *not* HTML, but extended JavaScript. Something also worth looking at is line 5 in 3.2. A nested `createElement()` call shows, how components can be composed together.

Because JSX is a language extension, a transpiler step is needed to produce production code. The common tool to use is called ”Babel”. There is a caption in [7] that says:

> Use next-generation JavaScript today.

The documentation in [7, /docs/en] explains, how modern JavaScript features can be used in any JavaScript project. The code which includes those modern features is normalized and transpiled by Babel to also work in older browsers. The tool accomplishes this by transforming the JavaScript code via its core implementation but also via some third-party plugins. A babel plugin has been created to transform JSX components into the syntax that can be seen in the code in 3.2. Just as a side note, although JSX became popular in conjunction with React, there are also other web technologies that make use of JSX like Vue.js [2] for example.

3.2.2 Explaining React components

React’s components can be split up in two categories: stateful and stateless components. The following paragraphs explain the difference between the two types of components and how they can be used in a React application.

Functional stateless components

As mentioned before, React is an extremely component oriented web technology. The example code in 3.3 includes a purely presentational component called ”HelloComponent” on line 4, a page layout component called ”PageComponent” on line 9, and the base App component called ”App” on line 20. React enables developers to create reusable and configurable components by providing the possibility to pass an arbitrary number of props to components. Generally speaking, props are used to not only control presentational details like color or layout variations but also to configure some initial state for example or to pass some application state data to a textbox component for example.

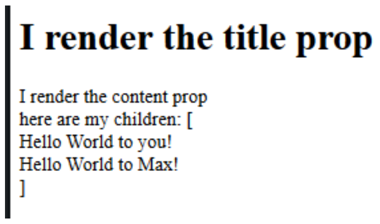
Without going too deep into the details of how a React application is rendered, lines 33 and 34 in 3.3 show, how the app component is rendered into a specific entry point in the static `index.html` page. The app component on line 20 renders the page layout component, passes a few props which should demonstrate what types of props are possible and then renders the `HelloComponent` twice inside the layout component. One time the `HelloComponent` receives the prop `name` and one time the property is omitted. The output of the hello world example can be seen in figure 3.1.

The example in 3.3 visualizes, how components can be reused throughout the application with different configurations and in different arrangements. The page layout component could be declared in an individual file to be reused in every page of the app. Via props React provides a reliable mechanism to control static state of the components that receive the props.

[2]https://vuejs.org/

Program 3.3: Simple example of a React component and its usage

```
import React from "react";
import ReactDOM from "react-dom";

const HelloComponent = props => {
  const name = props.name;
  return <div>Hello World to {name ? name : "you"}!</div>;
};

const PageComponent = props => {
  props.customFn("I get passed to the handler function!");
  return (
    <div>
      <h1>{props.title}</h1>
      <div>{props.content}</div>
      here are my children: [{props.children}]
    </div>
  );
};

const App = () => {
  return (
    <PageComponent
      customFn={console.log}
      title="I render the title prop"
      content="I render the content prop"
    >
      <HelloComponent />
      <HelloComponent name={"Max"} />
    </PageComponent>
  );
};

const rootElement = document.getElementById("root");
ReactDOM.render(<App />, rootElement);
```



```
I render the title prop

I render the content prop
here are my children: [
Hello World to you!
Hello World to Max!
]
```

Figure 3.1: React hello world sample output

One of the most important aspects of React is that props – once they're passed to a component – are static and immutable inside the receiving component as React's documentation in [12, /docs/components-and-props.html#props-are-read-only] demonstrates. Components that only render props and don't manage their own application state can be seen as pure functions which render the exact data they receive every ren-

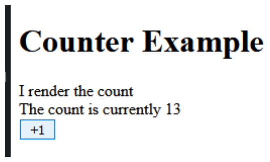
Program 3.4: Simple example of a React component and its usage

```
import React from "react";
import ReactDOM from "react-dom";

const Count = props => (
  <div>
    <div>I render the count</div>
    <div>The count is currently {props.count}</div>
  </div>
);

class StatefulComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 1
    };
  }

  counterHandler = () => {
    this.setState(state => ({ count: state.count + 1 }));
  };

  render() {
    return (
      <div>
        <h1>{this.props.title}</div>
        <Count count={this.state.count} />
        <button onClick={this.counterHandler}>+1</button>
      </div>
    );
  }
}

const App = () => <StatefulComponent title={"Counter Example"} />;

const rootElement = document.getElementById("root");
ReactDOM.render(<App />, rootElement);
```

der cycle. Props could thus be understood as parameters of a pure function, leading us back to the previously explained context of the function `f(data) = UI` in 3.1. Another important aspect of React's prop mechanism is that props cannot be changed inside the receiving component. Props that are passed into a component are immutable and trying to mutate them results in React not noticing any changes in the data and therefore not activating a new render cycle.

Stateful components

The attentive reader now probably has the following questions: "How do I introduce mutable application state, if data coming from props is immutable?" or "How does React notice, if I introduce changes to the application state?". At this point, it is important



Counter Example

I render the count
The count is currently 13
+1

Figure 3.2: React counter component output

to remember, that React works best when keeping the unidirectional data-flow model in mind. The library provides a built-in mechanism for handling mutable application state out of the box. The code example in 3.5 shows the difference between a purely presentational component on line 4 and a stateful component that keeps track of its application state on line 11.

First of, creating stateful components is quite effortless, as the component class simply derives from `React.Component` as shown in line 11 of the program in 3.4. The constructor in line 12 calls its super constructor – `React.Component` in that case – and then initializes its state to `{count: 1}` right away even before the component has mounted for the first time. The state object is available throughout the whole class component and can be used to render UI components, that depend on that current state.

The example program in 3.4 demonstrates well, how the current state is used in the render method in line 23. Once the component calls its render method, the current state is accessed and rendered. Note, that the state cannot be altered and is immutable and read-only as well as the component's props. To introduce state changes, React provides a class member function which is called ”setState”, which takes a callback to update the internal component state as shown in line 20 of 3.4. When the `this.setState` method is called, React is informed that there had been a state update and initiates a new render cycle which, as a consequence, triggers the whole lifecycle of the component again. The next section is all about React's lifecycle methods and how developers can utilize them.

3.3 React's component lifecycle

React components consist of a set of lifecycle methods that are called every render cycle of React, which is different from stateless functional components, as they are just pure functions. The previous code example in 3.4 was enhanced in 3.5. A few lifecycle methods – some of which can be found on lines 19, 23, 27, 31, and 39 in 3.5 – make it possible to exactly control how components react to certain application state updates. Their names make it pretty clear what aspect of the lifecycle they handle.

React's documentation in [12, /docs/react-component.html] includes a comprehensive guide on component lifecycle methods. By overriding the lifecycle methods inside their components, developers can add their specific logic to each lifecycle in every render cycle of the components. Overriding lifecycle methods is optional; it is, therefore, possible to not implement any lifecycle method at all. Notice, how class components always have to override the `render()` function to being able to even render content. The render function is called every time the component goes through a new render cycle.

Program 3.5: Simple example of a React component and its usage

```
import React from "react";
import ReactDOM from "react-dom";

const Count = props => (
  <div>
    <div>I render the count</div>
    <div>The count is currently {props.count}</div>
  </div>
);

class StatefulComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 1
    };
  }

  componentDidMount() {
    console.log("I did mount.");
  }

  shouldComponentUpdate(nextProps, nextState) {
    return this.state.count !== nextState.count;
  }

  componentDidUpdate(prevProps) {
    console.log("I did uptate, previous props are", prevProps);
  }

  componentWillUnmount() {
    console.log("I am about to vanish...");
  }

  counterHandler = () => {
    this.setState(state => ({ count: state.count + 1 }));
  };

  render() {
    return (
      <div>
        <Count count={this.state.count} />
        <button onClick={this.counterHandler}>+1</button>
      </div>
    );
  }
}

const App = () => <StatefulComponent />;

const rootElement = document.getElementById("root");
ReactDOM.render(<App />, rootElement);
```

A good visualization of the full lifecycle of a React component can be found in the GitHub repo in [4]. The web application in [5] visualizes the separation of the different phases a component goes through when iterating through its lifecycle methods. Figure 3.3 shows a screenshot of the application for the sake of being able to demonstrate the diagram in the thesis. The diagram is based off a tweet in [6] from Dan Abramov, one of the core contributors to the React library.

The whole lifecycle consists of 3 phases: The ”Render phase”, the ”Pre-commit phase” and the ”Commit phase”. Also, the component can be in three different states: The mounting state, the updating state, and finally, the unmounting state. The visualization in 3.3 shows the horizontal states and vertical phases of a React component. Every state of the component has to iterate all the phases each cycle vertically. Notice, how the update-render cycle does not call the constructor. What is also interesting is the fact, that the unmounting state only calls the `componentWillUnmount` method.

The most important lifecycle methods are `componentDidMount`, `componentDidUpdate` and `componentWillUnmount`. The `componentShouldUpdate` method is also important for improving performance, as explained later in this section. There are also more uncommon lifecycle methods for special occasions like the `getDerivedStateFromProps` method or the `getSnapshotBeforeUpdate` method. Those methods are rarely used and are not elaborated to avoid going out of the scope of this thesis. It is important to know, that the render phase is pure and can safely be aborted completely, resulting in a canceled lifecycle. Early cancelations of complete lifecycles can immensely improve performance.

The program in 3.5 is a good demonstration of how lifecycle methods can be used. The lifecycle method on line 23, for example, controls, if the component should even go through it’s rendering cycle or not. As mentioned before, Facebook advises avoiding reconciliation where ever possible. The `shouldComponentUpdate` lifecycle method already aborts the render cycle in the so-called ”render phase” which allows developers to avoid unnecessary reconciliation cycles and therefore also commitments to the DOM.

The `shouldComponentUpdate` method is the first lifecycle function that gets called in the example in 3.5 aside from the constructor, which only gets called once in the mounting state of the component. The `shouldComponentUpdate` method gets passed all future props *and* state which can then be compared to previous data. The example in 3.5 shows, that the count from the current and the next state is compared and if they differ, `true` is returned, which means to React that a new render cycle is necessary.

If for example the `StatefulComponent` on line 11 would have a parent that renders it 100 times, the stateful component is now smart enough to only run through its lifecycle methods only once, as `shouldComponentUpdate` method tells the component that nothing has changed and therefore omitting the rest of the lifecycle methods. Not calling the render function and its `createElement()` methods under the hood also has the implication, that no reconciliation has to be performed for the stateful component, resulting in improved rendering performance of the app.

The other lifecycle methods in the code example in 3.3 are pretty self explanatory. There are a few best practices though, according to the documentation in [12]. The

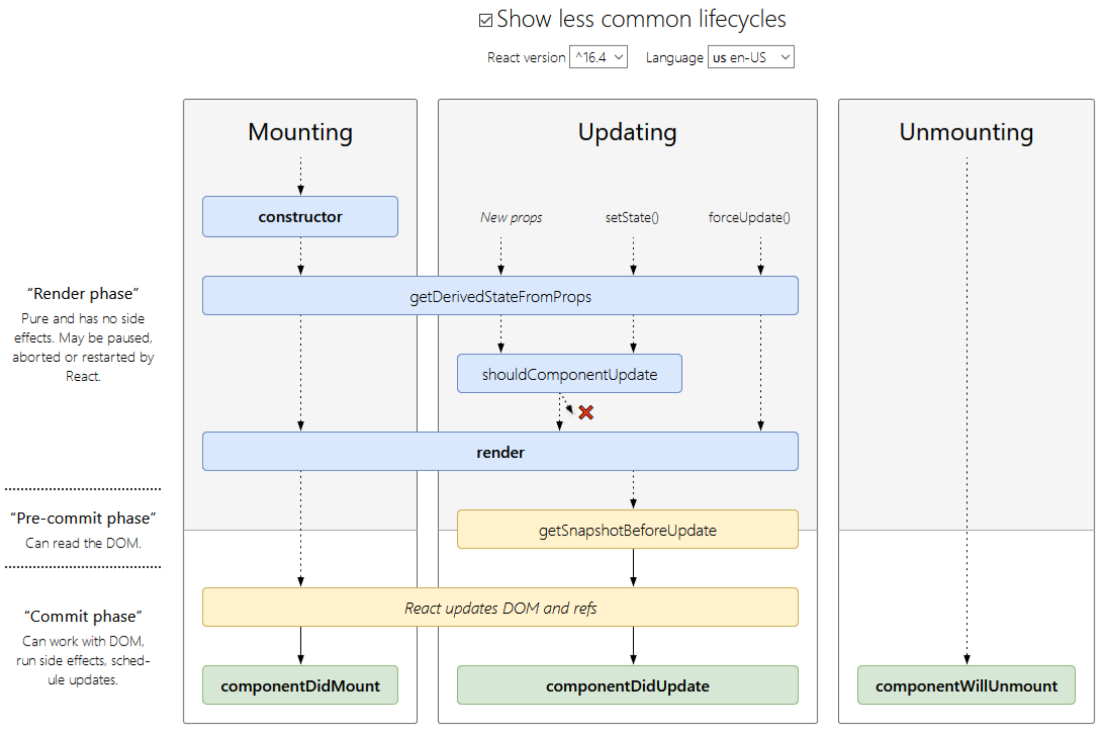


Figure 3.3: React lifecycle methods diagram

`componentDidMount` method is the one to handle side effects when fetching data from an API, for example. Another use-case would be registering event handlers in the `componentDidUpdate` method and unregistering them in the `componentWillUnmount` method.

3.4 Conclusion

All in all, understanding the lifecycle of React components is a key aspect of also understanding how thesis project was implemented. Lifecycle methods play a crucial role when combining the rendering cycle of D3 with React's rendering cycle. It is essential to know that React has a virtual DOM, which it uses to compare virtual component trees to the real DOM to decide if any updates have to be committed to the DOM. Rendering a component with the same state multiple times results in React not committing anything to the DOM as the virtual DOM then equals the real DOM. It is also important to remember, that these so-called "reconciliation cycles" can completely be avoided by implementing the `shouldComponentUpdate` lifecycle method.

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

Literature

[1] John W Lloyd. "Practical Advtanages of Declarative Programming." In: *GULP-PRODE (1)*. 1994, pp. 18–30 (cit. on p. 12).

[2] Marvin V. Zelkowitz Terrence W. Pratt. *Programming Languages: Design and Implementation (4th Edition)*. 4th. Prentice Hall, 2000 (cit. on p. 4).

Audio-visual media

[3] Pete Hunt. *Pete Hunt | TXJS 2015*. Youtube. Pete Hunt talking about React and when it was founded. June 2015. URL: https://www.youtube.com/watch?v=A0Kj49z6WdM (cit. on pp. 12, 14).

Software

[7] Babel. *Babel Babel is a JavaScript compiler*. Version 7. URL: https://babeljs.io/ (visited on 05/02/2019) (cit. on p. 16).

[9] Mike Bostock. *D3 Data Driven Documents*. Version 5. 2011. URL: https://github.com/d3 (visited on 01/10/2019) (cit. on pp. 3, 4, 9, 10).

[11] Lee Byron. *Immutable Immutable collections for JavaScript*. Version 3.8.2. URL: https://github.com/immutable-js/immutable-js (visited on 05/02/2019) (cit. on p. 14).

[4] Wojciech Maj. *React Lifecycle Methods diagram*. Version 1. URL: https://github.com/wojtekmaj/react-lifecycle-methods-diagram (visited on 05/10/2019) (cit. on p. 21).

[5] Wojciech Maj. *React lifecycle methods diagram*. Version 1. URL: http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/ (visited on 05/10/2019) (cit. on p. 21).

Online sources

[6] Dan Abramov. *Twitter Post: I just made this diagram of modern React lifecycle methods. Hope you'll find it helpful!* URL: https://twitter.com/dan_abramov/status/981712092611989509 (visited on 05/10/2019) (cit. on p. 21).

[8] Mike Bostock. *D3 Data Driven Documents.* Version 5. 2011. URL: https://d3js.org (visited on 01/10/2019) (cit. on p. 3).

[10] Mike Bostock. *D3 Data Driven Documents.* Version 5. 2011. URL: https://bl.ocks.org/mbostock (visited on 01/10/2019) (cit. on p. 4).

[12] Facebook. *React A JavaScript library for building user interfaces.* 2014. URL: https://reactjs.org (visited on 01/10/2019) (cit. on pp. 12–15, 17, 19, 21).

Check Final Print Size

— Check final print size! —

width = 100mm
height = 50mm

— Remove this page after printing! —