Combining React and D3.js for Efficient Data Visualization 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 25, 2019

Maximilian Zauner

Contents

Preface

I want to thank all the people who not only provided me with their spare time but also with their personal devices to be able to execute the performance benchmarks on in order to obtain some results for the research aspect of the thesis. Furthermore, a special thank goes to Sigrid Huemer, as she visually enhanced some of the graphs in this master's thesis.

Abstract

D3 is one of the most powerful data visualization libraries in the *JavaScript* environment. Because the development of the library started in 2011 and although the library went through several re-writes, the API is still relatively outdated. As a consequence, projects that use *D3* become unmaintainable relatively quickly as *D3's* API was not designed to realize extensive enterprise projects.

To avoid hardly maintainable code bases, the idea to combine *D3* with a well-known and widespread *JavaScript* library called *React* came to mind. The claim that developer experience improves by writing *React* code to use *D3* features might be subjective. Thus the research question of this thesis is if the combination of *React* and *D3* can be achieved without sacrificing render performance.

Hence the master's thesis introduces the reader to three combination prototypes which were developed as a part of the thesis project. The thesis not only explains implementation details but also compares the advantages and disadvantages of the prototypes. However, the central research aspect of this thesis focuses on the performance comparison of the prototypes and shows if a combination of *React* and *D3* with performance results comparable to a native *D3* implementation can be achieved.

Kurzfassung

D3 ist eine der umfangreichsten und leistungsfähigsten Datenvisualisierungsbibliotheken im *JavaScript*-Umfeld. Da die Entwicklung der Bibliothek im Jahr 2011 begann und obwohl die Bibliothek mehrere Neuschreibungen durchlaufen hat, ist die API noch relativ veraltet. Infolgedessen werden Projekte, die *D3* verwenden, relativ schnell nicht mehr wartbar, da die API von *D3* nicht für die Realisierung umfangreicher Unternehmensprojekte konzipiert wurde.

Um kaum wartbaren Code zu vermeiden, kam die Idee in den Sinn, *D3* mit einer bekannten und weit verbreiteten *JavaScript*-Bibliothek namens *React* zu kombinieren. Die Behauptung, dass sich die Erfahrung des Entwicklers verbessert, wenn man React-Code schreibt, um *D3*-Funktionen zu verwenden, könnte jedoch subjektiv sein. Daher ist die Forschungsfrage dieser Arbeit, ob die Kombination von *React* und *D3* erreicht werden kann, ohne die Renderleistung zu beeinträchtigen.

Die Masterarbeit führt den Leser in drei Kombinationsprototypen ein, die im Rahmen des Masterarbeitsprojekts entwickelt wurden. Es werden nicht nur die Details der Implementierung erklärt, sondern auch die Vor- und Nachteile der Prototypen werden verglichen. Der zentrale Forschungsaspekt diese Arbeit konzentriert sich jedoch auf den Leistungsvergleich der Prototypen und zeigt, ob eine Kombination mit vergleichbarer Leistung wie eine native *D3*-Implementierung erreicht werden kann.

Chapter 1

Introduction

This chapter introduces the reader to the general concept of the master project. Not only should the initial situation and motives 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 its development 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 though *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 writing large amounts of *D3* production code that—while still performing well—must be kept maintainable for multiple developers.

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*, and *Spotify*. The claim that developer experience improves by using *D3* through *React* might be subjective, but even if only a few developers are willing to use *React* combined with *D3* over native *D3* in their software the thesis project is a valuable addition to the *React* developer community. The exciting research aspect would then be if the combination is possible without losing render performance but still providing the convenience of writing *React* code.

1.2 Goals of the Project

The central research aspect of the master's thesis focuses on the performance aspect of the combination of *React* and *D3*, not how the *React* version might provide a better developer experience than native *D3* version. 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 thesis. 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 master's thesis provides not only a general overview but also an introduction to the two libraries for an even better understanding of the thesis project. The first chapters explain the required knowledge to understand the performance discussion. Later chapters 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 goal is to create a software that allows developers to write declarative *React* code and to avoid imperative *D3* code while still using *D3's* data visualization capabilities. 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 three 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 important parts of the thesis. Each prototype is tested on different devices and in different browsers, which generates 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 do not percieve an animation as smooth anymore.

Last but not least, 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 thesis provides a general overview of how the public API of the technology is designed and how the project is published on a widely available package registry to allow for including the library in any project.

Chapter 2

D3.js – Data-Driven Documents

This chapter provides an overview of the popular *JavaScript* library *D3*[1] which is used to simplify implementations of data visualizations in the web for developers. It not only offers insights about 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 chapters 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 [19] explains the library as a toolkit that allows binding data to the DOM. The documentation also provides an overview of the vast amount of helpful tools that can be used to visualize data in the browser. The library includes all kinds of functionality, ranging from simple array and mathematic operations to complex simulations that are calculated in real time.

One 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 is available online in [12]. 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 *D3's* documentation[2]. Another example of the documentation[3] 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[4]. The list of useful data manipulation tools goes on, and dealing with every aspect of the library would go far beyond the

[1] https://d3js.org/
[2] https://github.com/d3/d3-scale-chromatic
[3] https://github.com/d3/d3-array
[4] https://github.com/d3/d3-random

scope of this thesis.

What makes *D3* unique is the possibility to create individual data structures for rendering sophisticated data visualizations. The library provides scatter plots[5] or pie charts, line charts, area charts, radial bar charts, tree maps[6] to name a few. *D3* also provides utility functions to add labels or user interaction to every mentioned but also not mentioned data visualization type. A significant benefit of using *D3* is that it provides simple methods to transform any *D3* visualization into being user interactable by creating floating tooltips and 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. [12] 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, every available *D3* module can be integrated into any project by using the package manager *npm*[7] via downloading it from its registry[8].

There are multiple examples on the documentation's example page in [20] 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 [12].

2.2 Explaining the D3 API

This section aims to discuss the most vital aspects of the *D3* 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 not only manipulate the visualization and its data but also bind the data to the DOM. [5, 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 [12] the library *D3* was created in 2010. Thus it can be noticed that the API of the library 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 program 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 a vital aspect of *D3's* API. Via selection *D3* can connect *JavaScript* application data to actual DOM nodes as [12] shows. An example can be seen in the code example in program 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 program 2.1 shows very well. When adding or removing

[5] https://github.com/d3/d3-scale
[6] https://github.com/d3/d3-shape
[7] https://www.npmjs.com/
[8] 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
21  .enter()
22  .append('circle')
23  .style('fill', '#45b29d')
24  .attr('r', (node) => node.size)
25  .attr('id', (node) => node.name)
```

multiple DOM nodes and the individual nodes of the simulation are complicated DOM structures, the code quickly gets very incomprehensive as demonstrated in program 2.2. The provided code example in program 2.1 showcases a simple append operation of a single circle element, for instance, in comparison to the much more complex example in program 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 program 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 program 2.1 shows an enter selection which appends a circle SVG element for each new data object and also applies various attributes and style.

When the data of a visualization changes, nodes might be deleted, new nodes might appear but nodes might also stay in the visualization but change their position. *D3* covers these 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 when the data is updated. Advanced animations and transitions can be added via a simple function call.

For example, line 9 in program 2.1 shows a selection where all nodes are selected that are removed after the data has changed. Furthermore, the color is changed and a transition effect is added, which transforms the radius attribute of the node until it

Program 2.2: Negative example of how confusing and unmaintainable *D3* code can become.

```
1 d3.select(_this).classed('active', true)
2 d3.select(_this)
3   .select('.circle')
4   .transition(500)
5   .attr('stroke', function(d) {
6     if (d.rings && d.rings.length > 0) return '#404348'
7     return d.color || COLORS[d.type.toUpperCase()] || '#27292c'
8   })
9   .attr('fill', function(d) {
10    return '#404348'
11  })
12  .style('filter', 'drop-shadow(0 3px 4.7px rgba(0,0,0,.54))')
13 d3.select(_this)
14  .selectAll('.ring')
15  .transition(500)
16  .attr('opacity', 1)
17 d3.select(_this)
18  .selectAll('.node-background')
19  .transition(500)
20  .attr('opacity', 0)
21 d3.select(_this)
22  .selectAll('.sub-circle')
23  .transition(500)
24  .attr('cx', function(d, i) {
25    let deg = ((Math.PI * 2) / 8) * i - Math.PI
26    let x = Math.sin(deg)
27    let offset = event.rings ? event.rings.length * 15 : 0
28    return x * (d.r + 5 + offset)
29  })
30  .attr('cy', (d, i) => {
31    let deg = ((Math.PI * 2) / 8) * i - Math.PI
32    let y = Math.cos(deg)
33    let offset = event.rings ? event.rings.length * 15 : 0
34    return y * (d.r + 5 + offset)
35  })
36  .attr('stroke', '#FFF')
```

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

The code in program 2.1 also clearly shows that every attribute and style instruction of added DOM nodes has to be handled via a chained function call. On line 23 in program 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 lines 24 and 25 in program 2.1. By passing a function to property or attribute setters like the `.style('attribute', [style])` method, the passed handler is called by *D3*

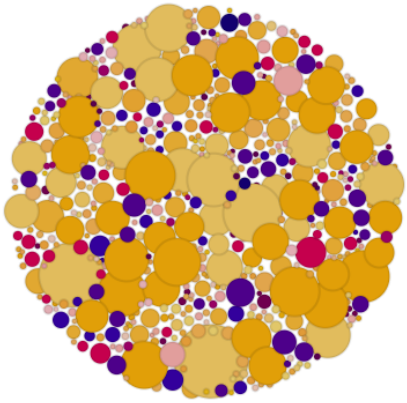


Figure 2.1: The force graph with default center force. All nodes are attracted to the same center without overlapping each other.

as a callback by providing each node's data to the callback. Therefore, when rendering 10 nodes, the attribute function on line 24 in program 2.1 which sets the radius of the node would be called 10 times, setting the radius for each specific node individually.

The code example in program 2.2 is taken from production code and shows how hard to read *D3* code can become 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 in program 2.1 is that the code cannot be reused, as it is hardcoded into a chain. Nowadays many frameworks for building front end 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 was integrated into *React* during the implementation phase of 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

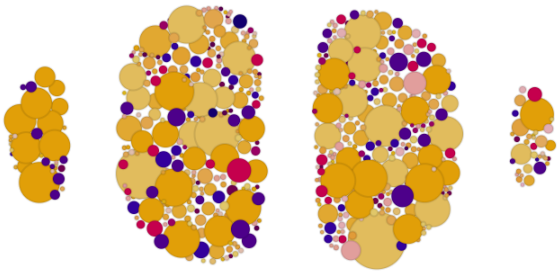


Figure 2.2: A sample force graph with more than one center force. Having multiple force centers causes each node to be attracted to their assigned center force.

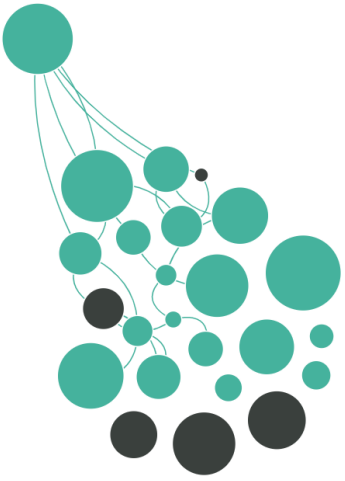


Figure 2.3: A sample force graph where the top node is dragged up to the left and the other nodes are dragged along. The force keeps the other nodes apart but also draws them to the center.

a single center force that drags all nodes towards the center but also prevents 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 dependent on each other and not only reject each other but also attract linked nodes as figures 2.2 and 2.5 show.

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 affects 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.

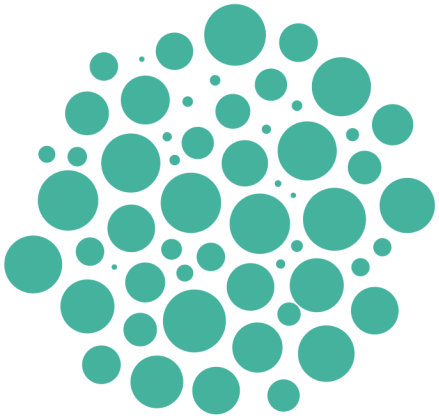


Figure 2.4: A sample force graph with one center force. The nodes are configured to reject each other with the function `r+r/2`.

Program 2.3: Code snippets for *D3* force simulation code.

```
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
8 simulation.alphaMin([min]) // sets minimum alpha value
```

Program 2.4: Sample initialization of a *D3* force graph.

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

D3 provides a somewhat simplified API to be able to quickly implement force graphs[9] in the browser. 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 program 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[10] so *D3* can correctly parse the node array.

[9] https://github.com/d3/d3-force
[10] https://github.com/d3/d3-force#simulation_nodes

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

Another very important aspect of force graphs is the so called "alpha" value system[11], which controls how long the simulation is active. The alpha value is a gradually decaying value that makes the simulation stop if a certain value is reached. Every simulation's tick handling function is called every "tick" of the simulation. Also, every time a "tick" is executed the alpha value decays via a predefinable function, it happens logarithmically per default tough. The tick handling function is not only called every tick but also for each individual node. It receives every node position in the simulation which lets developers link the data to the DOM with *D3* again. Knowing about the tick function is very important later on when the combination of *D3* and *React* is explained in more detail. Program 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. *D3's* documentation about force simulations[12] points to a few methods that can achieve said functionality. The functions on line 3, 4, and 5 of program 2.3 can be used to reheat a simulation. Also the functions in line 6, 7, and 8 of program 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 program 2.4 would look more like in example 2.5. The difference between the two code examples in programs 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 program 2.4 the simulation configuration is locked in the function chain. Chaining is a pattern which

[11] https://github.com/d3/d3-force#forces
[12] https://github.com/d3/d3-force

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)
```

can easily cause duplicated code in any codebase.

Writing functional *D3* code could also alleviate the confusing unmaintainable code in program 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.

Chapter 3

React – A JavaScript Library for Building User Interfaces

This chapter introduces the reader to the prevalent and widespread front end library called *React*. It explains an essential aspect 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 thesis elaborates how *React* uses declarative code to render data, whereas *D3* uses an imperative API to render its data.

3.1 Introduction to React

The easiest way to find information about *React* is to visit its official website[1]. There is a statement in [21] 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 [9, 05:30]. Walke wanted to create a tool that would improve the code quality of their internal tool called "Facebook ads." *Facebook* continued to develop and use *React* internally, but since the year 2013, the project is entirely open source. Starting with the initial open source release up until now not only technical engineers of *Facebook* but also the *React* open source community itself has 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 fewer restrictions than the BSD license.

According to [21], *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 [3] declarative programming is described 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. Declarative programming can be understood as a layer of abstraction that

[1] https://reactjs.org

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. Further insights into *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 on how *React* handles components and its props can be found in the upcoming 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* makes no assumptions about the technology stack that is used in a project as [21] 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.

React's documentation[2] 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 application's DOM tree in the *JavaScript* engine's heap as a representation of objects enables *React* to primarily apply updates this so-called virtual DOM. *React* compares the newly applied data with the old tree to then being able to decide if updates need to be committed to the real 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 thesis to to explore the algorithm in more depth, so it is recommended to read about *React's* reconciliation algorithm in its documentation[3].

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* once again. As mentioned in [9, 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

[2] https://reactjs.org/docs

[3] https://reactjs.org/docs/reconciliation.html

algorithm, which determines if some nodes have to be changed on the real DOM. If there was a *React* component that would always render the same `<div>` with the same data, rendering that component multiple times would not result in *React* writing multiple DOM nodes to the browser. The reconciliation algorithm would acknowledge that the virtual DOM's old tree matched the new data tree in this case, which would also result in *React* not updating the browser's DOM. Of course, if the component's content was dynamic, it would sometimes have 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, removed, or updated nodes are committed to the DOM. Even though the reconciliation algorithm prevents expensive DOM operations, the algorithm itself can also be expensive. There is an article[4] in *React's* documentation which advises developers to try to avoid reconciliation to further improve performance.

Unidirectional data-flow implicitly also implies that there is no data binding and no template language. The library only uses `createElement([element])` calls internally, which are hidden behind the so-called "JSX" *JavaScript* language extension. JSX will be explained more in 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 [13] explains exceptionally well, what immutable data structures are and how they are 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*. After reading this section, it should not only be clear how the two libraries play together but also how they are also entirely different.

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

[4] https://reactjs.org/docs/optimizing-performance.html#avoid-reconciliation

Program 3.1: Creating a *React* element with JSX.

```
1 const ReactElement = (
2   <div className="hello-world">
3     Hello <span className="emph-text">World</span>!
4   </div>
5 )
```

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 );
```

more readable code. The example in program 3.1 shows an example *React* element that is written in JSX. When looking at the transpiled output in program 3.2 it is clear how JSX helps to reduce the amount of code and how it greatly improves readability. The code in program 3.2 also shows that *React* is just a big composition of `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 two to infinite parameters as shown in *React's* documentation[5]. 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 and ongoing parameters describe the component's children. The infinite amount of children parameters makes it possible to compose multiple *React* components together, as children are nestable.

Line 2 and 3 in program 3.2 show how a *React* element is created. A node of type `div` is created and the property `{className: "hello-world"}` is passed. Each parameter after line 3 is a child of the created `<div>` node. The *React* element has three children which is demonstrated by the code example where the element is written in JSX in program 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 program 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 program 3.2. A

[5] https://reactjs.org/docs/react-api.html

nested `createElement()` call shows how components can be composed together.

Because JSX is a language extension, a transpilation step is needed to produce production code that can be interpreted by the browser. The common tool to use is called "Babel." There is a caption in [10] that says, "Use next-generation *JavaScript* today." The documentation in [10] 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 not only work in modern but also 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 program 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[6] 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 program 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.

Without going too deep into the details of how a *React* application is rendered, lines 33 and 34 in program 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 program 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.

One of the most important aspects of *React* is that props—once they are passed to a component—are static and immutable inside the receiving component as *React's* documentation[7] demonstrates. Components that only render props and do not man-

[6] https://vuejs.org/

[7] https://reactjs.org/docs/components-and-props.html#props-are-read-only

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

```
1 import React from "react";
2 import ReactDOM from "react-dom";
3
4 const HelloComponent = props => {
5   const name = props.name;
6   return <div>Hello World to {name ? name : "you"}!</div>;
7 };
8
9 const PageComponent = props => {
10  props.customFn("I get passed to the handler function!");
11  return (
12    <div>
13      <h1>{props.title}</h1>
14      <div>{props.content}</div>
15      here are my children: [{props.children}]
16    </div>
17  );
18 };
19
20 const App = () => {
21  return (
22    <PageComponent
23      customFn={console.log}
24      title="I render the title prop"
25      content="I render the content prop"
26    >
27      <HelloComponent />
28      <HelloComponent name={"Max"} />
29    </PageComponent>
30  );
31 };
32
33 const rootElement = document.getElementById("root");
34 ReactDOM.render(<App />, rootElement);
```

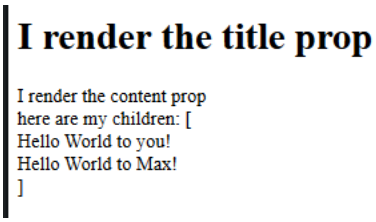


Figure 3.1: React hello world sample output.

age their own application state can be seen as pure functions which render the exact data they receive every render 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 section 3.1. Another important aspect of *React's* prop mechanism is that props cannot be changed inside the receiving component. Props that are passed

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

```
1 import React from "react";
2 import ReactDOM from "react-dom";
3
4 const Count = props => (
5   <div>
6     <div>I render the count</div>
7     <div>The count is currently {props.count}</div>
8   </div>
9 );
10
11 class StatefulComponent extends React.Component {
12   constructor(props) {
13     super(props);
14     this.state = {
15       count: 1
16     };
17   }
18
19   counterHandler = () => {
20     this.setState(state => ({ count: state.count + 1 }));
21   };
22
23   render() {
24     return (
25       <div>
26         <h1>{this.props.title}</div>
27         <Count count={this.state.count} />
28         <button onClick={this.counterHandler}>+1</button>
29       </div>
30     );
31   }
32 }
33
34 const App = () => <StatefulComponent title={"Counter Example"} />;
35
36 const rootElement = document.getElementById("root");
37 ReactDOM.render(<App />, rootElement);
```

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 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 program 3.5 shows the difference between a purely presentational component on line 4 and a stateful component that keeps track

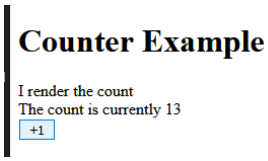


Figure 3.2: React counter component output.

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 on line 11 of the program in program 3.4. The constructor on 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 even 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 program 3.4 demonstrates well, how the current state is used in the render method on line 27. 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 on line 20 of the program 3.4. When the `this.setState` method is called, *React* is informed that there has 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 program 3.4 was enhanced in program 3.5. A few lifecycle methods—some of which can be found on lines 12, 19, 23, 27, 31, and 39 in program 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 includes a comprehensive guide[8] 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.

A good visualization of the full lifecycle of a *React* component can be found in the *GitHub* repo in [15]. The web application in [24] visualizes the separation of the

[8] https://reactjs.org/docs/react-component.html

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

```
1 import React from "react";
2 import ReactDOM from "react-dom";
3
4 const Count = props => (
5   <div>
6     <div>I render the count</div>
7     <div>The count is currently {props.count}</div>
8   </div>
9 );
10
11 class StatefulComponent extends React.Component {
12   constructor(props) {
13     super(props);
14     this.state = {
15       count: 1
16     };
17   }
18
19   componentDidMount() {
20     console.log("I did mount.");
21   }
22
23   shouldComponentUpdate(nextProps, nextState) {
24     return this.state.count !== nextState.count;
25   }
26
27   componentDidUpdate(prevProps) {
28     console.log("I did uptate, previous props are", prevProps);
29   }
30
31   componentWillUnmount() {
32     console.log("I am about to vanish...");
33   }
34
35   counterHandler = () => {
36     this.setState(state => ({ count: state.count + 1 }));
37   };
38
39   render() {
40     return (
41       <div>
42         <Count count={this.state.count} />
43         <button onClick={this.counterHandler}>+1</button>
44       </div>
45     );
46   }
47 }
48
49 const App = () => <StatefulComponent />;
50
51 const rootElement = document.getElementById("root");
52 ReactDOM.render(<App />, rootElement);
```

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 [18] from Dan Abramov, one of the core contributors to the *React* library.

The whole lifecycle consists of three 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 figure 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 in *React's* render cycle 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 as it would go out of the scope of the 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 not only avoid unnecessary commits to the DOM but also unnecessary reconciliation cycles which can greatly improve performance.

The `shouldComponentUpdate` method is the first lifecycle method that gets called in the example in program 3.5, aside from the constructor, which is only 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 program 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 in program 3.5 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. Therefore, the rest of the lifecycle methods are omitted. 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 program 3.3 are pretty self explanatory. There are a few best practices though, according to the documentation, in [21]. The `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 `componentWill`

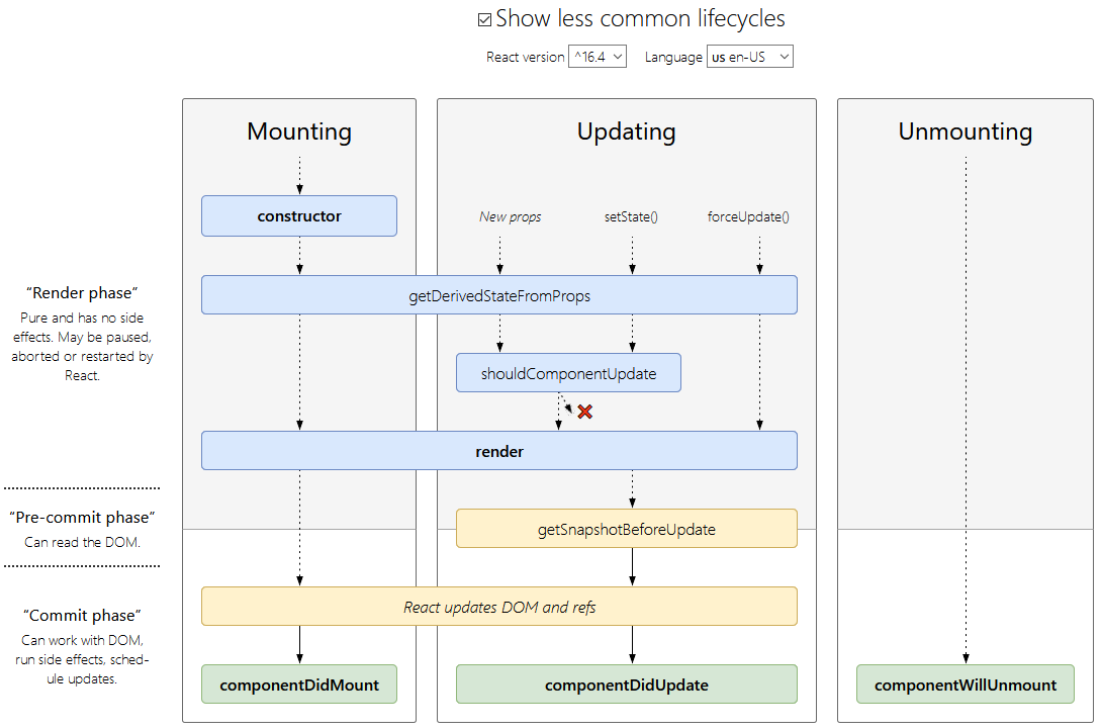


Figure 3.3: React lifecycle methods diagram taken from [24].

`Unmount` method.

3.4 Conclusion

All in all, understanding the lifecycle of *React* components is a key aspect of also understanding how the 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 versions of virtual component trees 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 consecutive virtual DOM tree versions are equal to each other. It is also important to remember that these so-called "reconciliation cycles" can completely be avoided by implementing the `shouldComponentUpdate` lifecycle method.

Chapter 4

Combining React and D3

The most important aspect of the master's thesis is the thesis project. This chapter introduces the reader to the project implementation and the resulting prototypes that were developed during the development phase of the thesis project. A complete walk-through helps the reader to understand the implementation differences of all prototypes. Finally, the chapter introduces the reader to the performance testing methodology and the different devices that were used to benchmark the prototypes.

4.1 Introduction and Motivation of the Project

When trying to find the best combination of two libraries, developing prototypes is extremely important. Combining *React* and *D3* in a few different ways in the thesis project ultimately lead to one prototype that then came out on top. The project was primarily realized to provide *React* developers with an alternative to native *D3* force implementations. The resulting software makes it possible to use *D3's* force simulations by writing *React* code. Finding a *React* implementation of *D3's* force simulation which performs better than the native *D3* implementation was not the primary goal of the project.

The obvious question "Why do I even need a combination of *React* and *D3* if I could just use native *D3* instead?" can quickly be answered. The initial idea of the project came to mind when a client requested a fully fledged *React* web application, which also included some complex data visualization aspects that have to be animated in the browser. Instead of implementing the visualization part of the application in native *D3*, the combination of *React* and *D3* enables all developers of the project to write declarative *React* code. Using a combination layer API results in only one code base that has to be maintained, instead of two. Also, as mentioned before, *D3* code bases tend to become exponentially more challenging to maintain as the project grows.

4.2 Project Setup

To be able to test multiple prototypes, one crucial aspect of the project is the force simulation builder setup. All prototypes are based on the simulation builder module which contains a few *D3* specific methods that allow developers to construct their

own individual force simulation variants. The prototypes can reuse some of the already composed methods, and some new custom methods can be implemented as well.

By implementing a force simulation builder module, it is possible to ensure that all prototype *D3* simulations are initialized and updated the same way. The most interesting code snippet of the module can be seen in program 4.1. Individual building blocks of the simulation can be piped together because the module uses a functional approach. Lines 12, 21 and 30 show how the updater functions for the three prototypes are composed together by using smaller force simulation function blocks.

The `buildForceSimulation()` method in line 42 in program 4.1 is the most important function in the module. The options parameter has to contain a type, which is used to determine what updater function is applied to the force simulation. The functional switch case statement on line 36 in program 4.1 decides based on the type which updater function to return. The builder function returns an instance of a *D3* force simulation and the associated updater function that can be used by each prototype to update the simulation every time the data changes.

Another interesting aspect of the code snippet in program 4.1 is that the functional compositions can be nested. Lines 1 and 7 show the composition of two functions that in turn can then be composed with all three force simulation variants. Even though they are piped together, the result can then be piped again into every updater function composition as shown in lines 16, 25, and 32, for example, where the composed method `applyForceHandlers` is piped into the updater functions of the different prototypes. Again, the way the updater function composition is implemented ensures the equality of the force simulation initialization of all three prototypes.

Of course, since all three prototypes work fundamentally different, some custom apply functions have to be implemented to ensure the functionality of all three prototypes. Applying the custom functions is no problem, as all updater functions are functional compositions. If some custom method has to be added, it can just be applied to the function composition of the associated prototype. Line 23 in program 4.1 shows how a custom node reference applying function is composed into the updater function.

Each prototype implements the same API interface and receives its data via props. All visualization data and every setting parameter, therefore, has to be passed to the prototype component via *React* component props. Using props also means that the data and the settings must be updateable. The parent container which renders a prototype component must be able to update the prototype's data by just passing different props. The newly applied props, as a consequence, have to be handled correctly in each prototype individually.

Program 4.1: Simple example of a *React* component and its usage.

```
const applyForceHandlers = pipeAppliers(
  applyGeneralForce,
  applyLinkForce,
  applyCollisionForce
)

const applyEndHandlers = pipeAppliers(
  applyOnEndHandler,
  applySimulationReheating
)

const pureD3Updater = pipeAppliers(
  applyNewNodeData,
  applyPureD3Selection,
  applyTickHandler,
  applyForceHandlers,
  applyDragHandlers,
  applyEndHandlers,
)

const hybridUpdater = pipeAppliers(
  applyNewNodeData,
  applyNewRefs,
  applyTickHandler,
  applyForceHandlers,
  applyDragHandlers,
  applyEndHandlers,
)

const pureReactUpdater = pipeAppliers(
  applyNewNodeData,
  applyForceHandlers,
  applyEndHandlers
)

const getUpdaterFunction = switchCase({
  [SIMULATION_TYPE.PURE_D3]: pureD3Updater,
  [SIMULATION_TYPE.REACT_D3_HYBRID]: hybridUpdater,
  [SIMULATION_TYPE.PURE_REACT]: pureReactUpdater,
})(null)

export const buildForceSimulation = (options) => {
  const simulation = forceSimulation()
  const updateSimulation = getUpdaterFunction(options.type)
  updateSimulation({ simulation, options })
  return { simulation, updateSimulation }
}
```

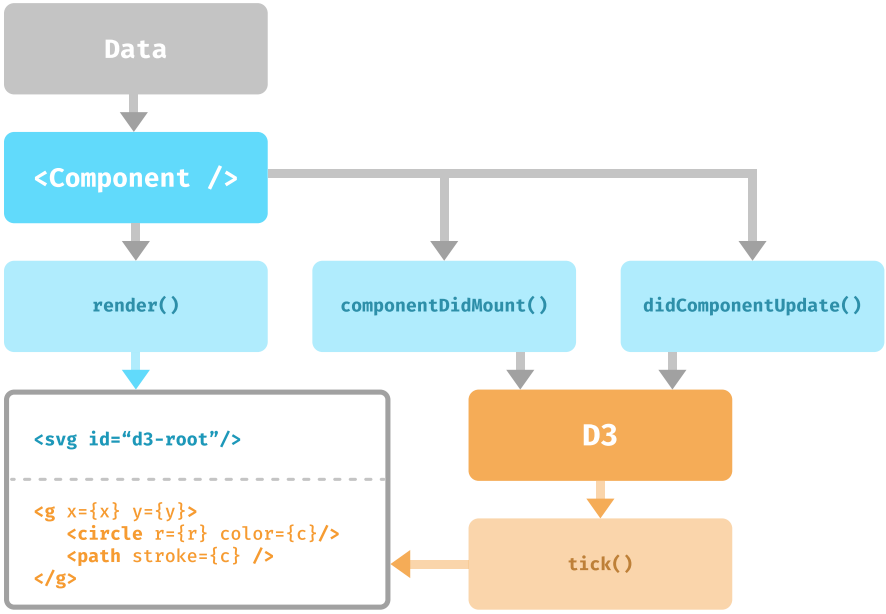


Figure 4.1: Pure *D3* force graph life cycle visualization.

4.3 Prototypes

This section introduces the reader to the prototypes of the thesis project. It also explains how the project initially originated. Furthermore, every resulting prototype of the project is listed and explained extensively. All in all, the thesis project yielded three viable prototypes that are described in this chapter.

4.3.1 Pure D3 Prototype

The first prototype of the project was developed from an early experimental implementation. The sole reason it was realized was to test if it is even possible to combine *React* and *D3* but still maintaining *React's* philosophy of unidirectional dataflow and idempotent render function components. The main goal, therefore, was to create a prototype that would always correctly update itself and thus visualize current data props on every render cycle. If the parent component updated the visualization component's data or options, the prototype would have to instantly reflect the changes as well. The main difficulty with this prototype is to combine *React's* declarative approach with *D3's* imperative way of rendering data.

The implementation heavily relies on the fact that *React's* reconciliation algorithm omits updates to the DOM if the same element is rendered consecutively. The pure *D3* graph component renders a single static base SVG, as shown in the code snippet in program 4.2. After the component mounts, *D3* hooks into the base SVG component via the provided reference to the actual base DOM node and builds its force simulation

Program 4.2: Render function of the pure *D3* prototype.

```
1 render() {
2   const { width, height } = this.props
3   return <svg ref={this.ref} width={width} height={height} />
4 }
```

on top. *D3* also appends and removes the DOM nodes according to the data that was passed to *D3*. Figure 4.1 demonstrates via color coding how *React* only renders the SVG element and *D3* renders the simulation via the tick function.

As a result, *React's* reconciliation algorithm does not handle nodes that are inside the *D3* force simulation if the data changes. The component only renders a static SVG element that is not updated. Because the SVG element is static, *React's* reconciliation algorithm does not commit anything to the DOM since every time the render function is called the SVG tag stays the same. Instead, *D3* entirely takes over the DOM manipulation and adequately handles the simulation by itself without *React* interfering in any way.

Every time the data updates, the new data is provided directly to *D3* via the life cycle method `componentDidUpdate()` as figure 4.1 shows. Of course, every data update causes *React* to render the base SVG component, but, due to the virtual DOM implementation of *React*, the SVG is never newly rendered, as it never changes. A static component is a *React* node that does not contain any dynamic content and is therefore never updated by the reconciliation phase. *React's* reconciliation algorithm prevents the browser from newly committing the SVG tag to the DOM. *D3*, therefore, works completely separate from *React*. *D3*, on the other hand, can be implemented like in any other native *D3* web project. Not only the initial force graph generation functionality has to be implemented but also the update logic that handles the updated data and applies it to the force graph simulation.

There are two component life cycle methods from *React* that are crucial to this implementation. First, `componentDidMount()` is used to initialize *D3*, select the base node, and then build the whole force simulation on top as demonstrated in program 4.1. It is important to use the life cycle method that triggers *after* the initial commit phase, as it makes sure, the component has already been rendered once for *D3* to being able to select the existing real SVG DOM node. Looking at figure 4.1 again, it is apparent that the second important life cycle method is `componentDidUpdate()` which provides the latest most up to date data directly to *D3*. That way *D3* can handle the update in the force graph.

Implementation Details

Once the component is initialized, the `componentDidMount()` life cycle method directly calls the initializer function which can be seen on line 1 in the code snippet in program 4.3. The initializing function appends the base `<g>` tag and also handles the translation of the current height and width of the component. Then the `buildForceSimulation()` function is called with the current options and parameters in order to

Program 4.3: Pure *D3* force graph initializing function.

```
1 initGraph = () => {
2   const { width, height, onSimulationStart } = this.props
3
4   onSimulationStart()
5
6   const svg = select(this.ref.current)
7   svg
8     .append('g')
9     .attr('transform', 'translate(' + width / 2 + ',' + height / 2 + ')')
10
11  const simOptions = this.extractSimOptions()
12  const { simulation, updateSimulation } = buildForceSimulation({
13    type: SIMULATION_TYPE.PURE_D3,
14    ...simOptions,
15  })
16
17  this.simulation = simulation
18  this.updateSimulation = updateSimulation
19 }
```

get the simulation and the correct updater function. Note how the simulation type is passed to the builder function as well. The resulting simulation and updater function is then saved in the current component.

What is also worth mentioning is the fact that the simulation and the updating functions are saved directly to the `this` context as seen in lines 17 and 18 of the code snippet in program 4.3. As stateful components are just plain *JavaScript* classes, they are capable of having member variables as well. It is of utmost importance not to confuse member variables with *React's* component state, as *React* is agnostic to class member variables. *React* not noticing the member variables is a desired effect in this case, as the simulation and the updater function have to be saved in the component without *React* going through a new render cycle.

Looking at the code in program 4.4, the update applying functionality can be seen very well. If given a current simulation object, the function handles all newly entering, transitioning, and exiting nodes accordingly. Even an animation is applied. Each time, the pure *D3 React* component goes through the `componentDidUpdate()` function, the `applyNodeUpdateCycle()` function is called as well. The force simulation building module can take in the updater function from the example in program 4.4 and composes it directly into the updating function as seen in line 14 in program 4.1.

Another vital function of the pure *D3* force graph is the tick handler that can be seen in program 4.5. The ticking function also gets passed to the force simulation builder function. Each iteration of *D3's* simulation tick the function updates the position of all nodes and links in the simulation.

Program 4.4: Function that applies the data update to *D3* on data changes.

```
applyNodeUpdateCycle = (simulation) => {
  simulation.linkSel.exit().remove()

  simulation.linkSel = simulation.linkSel
    .enter()
    .append('path')
    .attr('stroke', '#45b29d')
    .attr('fill', 'none')
    .merge(simulation.linkSel)

  let t = transition().duration(750)

  simulation.nodeSel
    .exit()
    .style('fill', '#b26745')
    .transition(t)
    .attr('r', 1e-6)
    .remove()

  simulation.nodeSel
    .transition(t)
    .style('fill', '#3a403d')
    .attr('r', ({ size }) => size)

  simulation.nodeSel = simulation.nodeSel
    .enter()
    .append('circle')
    .style('fill', '#45b29d')
    .attr('r', ({ size }) => size)
    .attr('id', ({ name }) => name)
    .merge(simulation.nodeSel)
}
```

Program 4.5: Tick handling function of the pure *D3* prototype.

```
ticked = () => {
  this.simulation.nodeSel.attr('cx', ({ x }) => x).attr('cy', ({ y }) => y)
  this.simulation.linkSel.attr('d', (d) =>
    this.props.linkType === LINK_TYPES.CURVED ? getCurvedLinkPath(d) :
    getStraightLinkPath(d),
  )
}
```

Advantages

One of the most apparent advantages of the pure *D3* force graph implementation is its performance, of course. Since the implementation uses a native *D3* approach to render and update the nodes and links in the simulation, the performance is also comparable to a native *D3* implementation. In chapter 5, the performance of the pure *D3* prototype

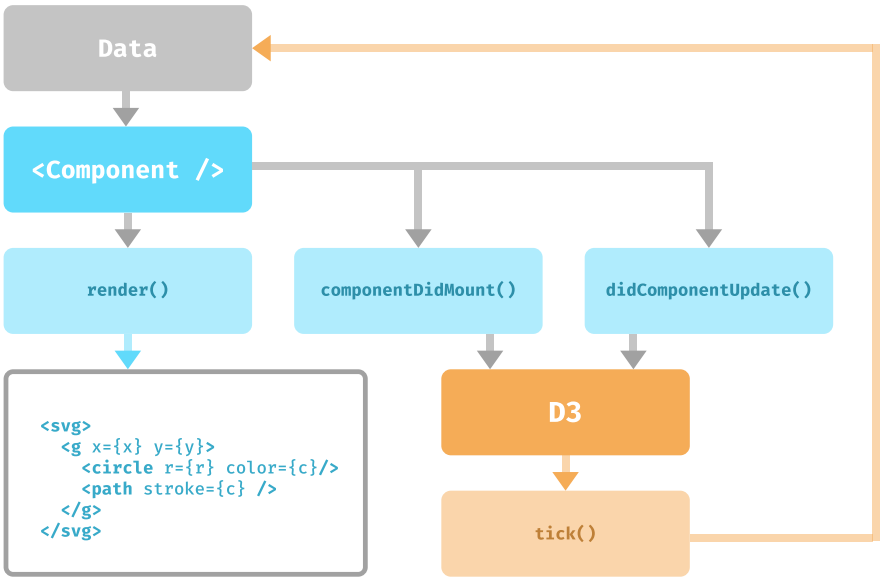


Figure 4.2: Pure *React* force graph life cycle visualization.

is compared to the other implementations.

Disadvantages

The most significant disadvantage of the pure *D3* implementation is the fact that all DOM manipulations are handled via imperatively chained function calls on the node selections of *D3* which also implies that the node rendering functionality cannot be customized by passing custom render functions for instance. The force graph's code itself has to be changed to get different node and link appearances, which leads to the previously described problem of encountering unmaintainable code over time.

4.3.2 Pure React Prototype

The pure *React* force graph implementation is far more complicated than the pure *D3* force graph, as the *D3* implementation is just a *React* wrapper for *D3*. To achieve a pure *React* implementation, *D3* needs to be deeply integrated into the rendering cycle of *React* itself. As mentioned in chapter 2, *D3* provides developers a tick function when using animated simulations. This tick function is executed as often as the browser has an animation frame available according to *D3's* documentation[1]. Explaining the request animation frame functionality of the browser would go out of the scope of this thesis, but MDN provides a good explanation on the MDN website in [22].

As figure 4.2 shows, the complete presentation layer is handled by *React* itself. The whole SVG component tree is completely rendered by *React*. The figure also demon-

[1] https://github.com/d3/d3-timer

strates how the data is handled not only in the component's constructor but also in its `shouldComponentUpdate()` life cycle method. As mentioned in the previous chapter 3, the constructor is the first life cycle method that is executed in any *React* component. The pure *React* prototype takes advantage of that fact and initializes the complete *D3* force simulation before any data is rendered. The `shouldComponentUpdate()` life cycle is used to determine if the component's props have changed and possibly apply an update to the *D3* force simulation.

The key to the integration of *D3* into *React's* render cycle is the tick function on the *D3* force simulation. Every time the tick function is executed, the applied tick handler then queries the force simulation data, fetches all link and node positions, calculates the current position and then passes the newly calculated data to the *React* component again via a `setState()` call. Figure 4.2 demonstrates how the data flows back into the component after each tick cycle of the force simulation.

Keeping all node and link positions in the *React* component state, as a result, unlocks full control to the presentation layer. Thus, *React* acting as a function of state can completely handle each element in the force simulation. The component state includes not only appearance properties like background or stroke color but also exact node and link positions. As a consequence, the data has to pass the complete rendering cycle of *React* on every tick execution, including the reconciliation phase, which can lead to poor rendering performance. Not only does it take time for *D3* to calculate a new version of the force data each tick, but also *React* then has to process the whole new data tree. If the simulation contains a high number of nodes, the whole rendering process—which figure 4.2 shows—should massively impact performance in theory.

An existing project of *Uber*[2] strongly inspired the pure *React* implementation. *Uber's* project is called vis-force and can be found on its git page in [17]. During the research phase of the thesis project, *Uber's* project was found and also thoroughly examined. The pure *React* prototype implementation is essential as it can be used to compare the render performance to the other two prototypes of the thesis project.

Implementation Details

The initialization of the pure *React* force graph pretty much looks the same as in line 12 in the pure *D3* code example in program 4.3. The pure *React* prototype passes its correct simulation type of course though. The biggest and most important difference to the pure *D3* implementation is that the initialization of the force simulation takes place inside the constructor instead of the `componentDidMount()` life cycle method. As mentioned before, the pure *React* prototype contains all node and link positions in the internal state. Consequently, the internal state must exist before calling the `render()` life cycle method the first time.

Due to the fact that the pure *React* force graph component provides the complete dataset about all nodes and links in the simulation, the render function can be 100% declarative code as seen in the code example in program 4.6. The link and node properties which contain presentational data like the size of a node or the color of a link come from the component's props, as they are passed in from the parent container. However, the link and node positions come from the components internal state. The example code

[2] https://www.uber.com

Program 4.6: Render life cycle method of the pure *React* force graph prototype.

```
render() {
  const { height, width, nodes, links } = this.props
  const { linkPositions, nodePositions } = this.state

  return (
    <span style={containerStyle}>
      <svg height={height} width={width}>
        <g style={gStyle} transform={`translate(${width / 2},${height / 2})`}>
          <Links links={links} linkPositions={linkPositions} />
          <Nodes nodes={nodes} nodePositions={nodePositions} />
        </g>
      </svg>
    </span>
  )
}
```

in program 4.6 shows very well, how the nodes and links components have direct access to the link and node positions. Figure 4.2 also shows how the *React* components handle all the SVG properties.

Another vital aspect of the pure *React* force graph's implementation is the update handling mechanism. Due to the fact that the component is updated on every free animation frame of the browser, it is of utmost importance to only update the component if necessary. The `shouldComponentUpdate()` function in the code snippet in program 4.7 shows that not only internal state but also the props are checked, if they have updated. Only if the props provided by the parent component have changed, the component calls the force simulation updater function which it obtained via the `buildForceSimulation()` call in the constructor. The component should update though if either its props or its state has changed. If the props have changed, the simulation updater function makes sure that the force simulation calculates the data for the new nodes and links. By doing so, the next render call can already process the new node and link positions.

Unfortunately calling the `handleSimulationUpdate()` function inside *React's* life cycle method `shouldComponentUpdate()` is considered an anti-pattern according to Facebooks article[3], as side-effects should always be handled inside the `componentDidUpdate()` life cycle method. The force simulation updater function is a so-called "side effect." Facebook strongly advises against using any side effects in the `shouldComponentUpdate()` life cycle method, as it should be as fast and efficient as possible to prevent possible render cycles. In the case of the pure *React* force component calling the simulation updater function is fine though, as the side-effect is not called on every state update, but only if the component's props change. Props only change, if the parent component passes different props, which does not happen that often. The internal component state, on the other hand, updates multiple times per second and preventing the simulation update function call on every state change is crucially important for good rendering performance.

[3] https://reactjs.org/docs/react-component.html

Program 4.7: Update method of the pure *React* force graph prototype.

```
1 shouldComponentUpdate(nextProps, nextState) => {
2   const propsChanged = shallowCompare(this.props, nextProps)
3   const stateChanged = shallowCompare(this.state, nextState)
4   const shouldUpdate = propsChanged || stateChanged
5   propsChanged && this.handleSimulationUpdate(nextProps)
6   return shouldUpdate
7 }
```

Program 4.8: Simulation tick handler of the pure *React* force graph prototype.

```
1 updatePositions = () => {
2   this.setState({
3     linkPositions: getLinkPaths(this.simulation),
4     nodePositions: getNodePositions(this.simulation),
5   })
6 }
```

Another interesting aspect of the pure *React* prototype implementation is the fact that the tick handler function does not manipulate the DOM directly as the pure *D3* example does. Instead, the code example in program 4.8 demonstrates how *React's* `setState()` component function is called which updates the component's state with the new link and node positions. The functions on lines 3 and 4 take in the current force simulation as a parameter and extract either the node or link positions and return them. As mentioned before, the update function is called as often, as the simulation tick function ticks.

Every `setState()` call updates the component and triggers a new render cycle of the force component. The component update loop goes on until the alpha in the simulation has fully decayed. The `updatePositions()` call on line 1 in the code snippet in program 4.8 itself is also wrapped inside a `requestAnimationFrame()` call to ensure a consistent framerate in the browser. The component updater function should only be called if the browser is ready to render a new simulation tick.

Advantages

A considerable advantage of a pure *React* implementation is the excellent developer experience. Programmers have full control over the simulation data. All force data can be represented declaratively via *React* components. The most significant difference to *D3* is that bigger *D3* codebases often contain a significant amount of hardly maintainable code, as DOM nodes have to be added via imperative `append()` and removed via imperative `remove()` calls which exist on multiple different places in the code base. Since DOM nodes have to be conditionally added, changed, and removed, the code quickly gets confusing and possibly produces inconsistent UI states. When writing the components with *React*, the library accurately renders nodes and links that exist in the currently calculated data tree that was produced by the current ticking cycle of the *D3*

force simulation. *React* itself can not only completely handle properties like filling color or stroke properties but also the nodes' positions.

Rendering the complete force graph via *React* components yields one other significant advantage compared to the pure *D3* prototype. If desired, the force graph component can be upgraded to accept custom rendering functions that can be passed from outside. If the pure *React* force graph component offered a custom rendering function, users of the component could write their personal rendering functions with *React* code to achieve a customized version of the force graph.

Disadvantages

The most significant advantage sadly is also the biggest disadvantage of the pure *React* force graph. Because the pure *React* prototype not only stores all node and link positions in the internal component state but also updates them every fraction of a second more CPU cycles are lost by calculating not only the *D3* tick cycles but also complete *React* render cycles as a consequence. Even though sometimes some node and link positions are only changed ever so slightly when the alpha value approaches the minimum value, *React's* reconciliation algorithm determines an updated DOM node and completely recommits the whole node to the DOM, which, of course, has an impact on performance.

Another considerable disadvantage of the pure *React* force graph implementation is that some native *D3* functionalities like dragging nodes or zooming in and out of the graph have to be implemented from scratch in pure *React* which can be a tedious task. Due to the component being rendered multiple times a second, implementing well performing drag handlers is a very complicated task, for example.

4.3.3 D3 and React Hybrid Prototype

Last but not least, there is the *D3* and *React* hybrid force graph implementation. The prototype not only provides the developing experience of writing declarative *React* code but also puts the complete feature set of *D3* at the developers' disposal. The hybrid implementation makes it possible to handle the representational aspect of the simulation via *React* and let *D3* not only calculate but also manipulate the DOM node positions of the individual nodes and links in the force graph. Figure 4.3 demonstrates via color coding what parts of the DOM content are handled by *React* and *D3*.

The way the prototype achieves the previously mentioned functionality is that the *D3* force simulation is constructed *after* the hybrid *React* component is initialized and mounted. As in the other two prototypes, all force data is passed into the hybrid force graph component via props from the parent component. *React* renders the nodes and links which are represented in the props' data and after the render phase has been passed, the data is then handed to *D3* via the `componentDidUpdate()` life cycle method. *D3* selects the already committed DOM nodes and applies the internally calculated force position data. Figure 4.3 demonstrates how only post render phase life cycle methods are used to pass data to *D3*.

As described in chapter 3, during the component's render phase the reconciliation algorithm compares newly applied data of the component's state or props with the virtual DOM and then decides which DOM nodes need to be committed to or removed

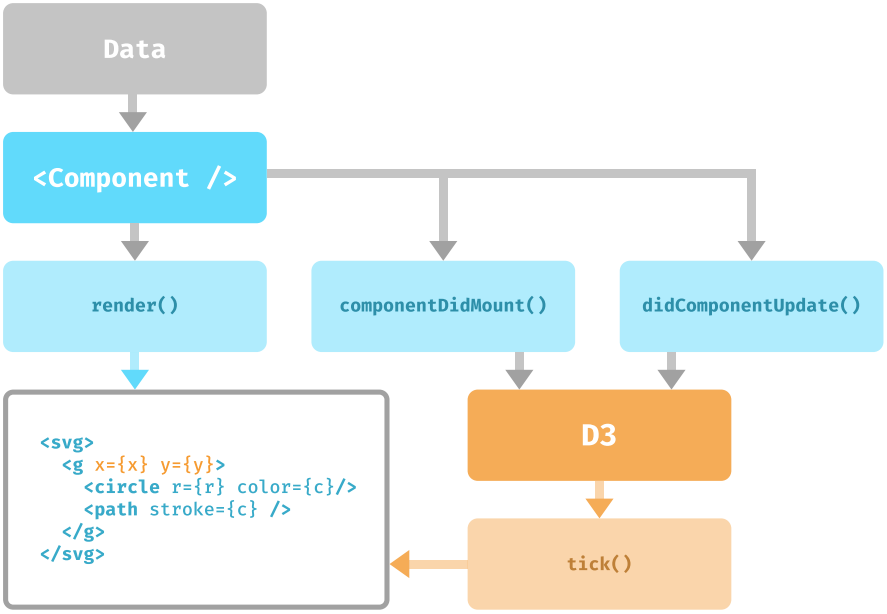


Figure 4.3: Pure *React* force graph life cycle visualization.

from the DOM. Once the simulation data updates via a prop update from the parent component, some nodes or links, therefore, might be added, changed or could just be removed, if they are non-existent anymore in the new version of the applied data.

The most important aspect of the hybrid prototype is that *D3* selects already existing DOM nodes and connects them to the current force simulation data. Since the selection process is executed in *React's* commit phase, *D3* can read the up to date version of the DOM. As described in chapter 3, reading or manipulating the DOM in the commit phase guarantees the DOM to be a representation of the component's current state and data. Therefore *D3's* selection happens *after* the render phase to always apply its selection to the most recent version of the DOM.

Implementation Details

As the name of the component already reveals, the implementation of the hybrid prototype is a combination of the previous two prototypes. The initialization of the hybrid graph component also looks very similar to the initialization on line 12 in the code snippet in program 4.3. The only difference in the hybrid implementation is that the hybrid simulation type is passed to the builder function. A very notable difference to the pure *React* component is that the initializer function is called in the `componentDidMount()` life cycle method, not in the component's constructor. As described before in the introduction of section 4.3.3, the component needs to be in the commit phase for *D3* being able to hook into already committed DOM nodes.

As described in section 4.2, the `buildForceSimulation()` function is very important

Program 4.9: Component update handler of the hybrid force graph prototype.

```
1 componentDidUpdate() {
2   this.updateSimulation(this.extractSimUpdateParams())
3 }
```

to ensure every prototype the same *D3* force simulation object. Any changes in given option parameters yield a different simulation of course, but the builder function is an idempotent function. If every prototype calls the builder function with the same force simulation option parameters, the method always returns the same simulation for every prototype. Like the other prototypes, the hybrid implementation of course also uses the force simulation builder function.

In contrast to the other prototypes, updating the hybrid graph component is quite simple. The code snippet in program 4.9 shows that the `componentDidUpdate()` life cycle method just calls the updater function that was obtained from the `buildForceSimulation()` call. Since *D3* handles the positioning of the nodes, updates in the components simulation data have to be passed to *D3* to update its force calculation. While the pure *React* prototype has to pass a complete render-cycle multiple times a second, the hybrid component only updates, if the parent container passes some new simulation data. As a consequence, the hybrid component can be rendered multiple times by the parent container but prevents itself from updating via implementing the `shouldComponentUpdate()` life cycle method.

The tick handler, on the other hand, contains a few *D3* specific instructions to display the current force simulation state. As mentioned before, the hybrid component makes *D3* responsible for positioning the DOM nodes correctly according to the current simulation data. Line 4 in program 4.10 demonstrates, how the tick handler works in the hybrid prototype. To enable users of the component to pass custom simulation handlers, the node and link handlers are extracted from the props of the component first. If they are set from the parent container, they are utilized to handle the simulation tick. Otherwise the standard tick handlers on lines 1 and 2 are used. More information on how to customize the usage of the hybrid component can be read in chapter 6 later on.

Like the pure *D3* prototype, the hybrid prototype also supports transitions. To achieve *D3's* transition functionality, a library called *react-move*—which can be found on its *GitHub* page in [14]—provides the functionality for *React* components. Because *React* only renders its current state as a function of state, it is not possible to keep track of data throughout multiple render cycles. If, for example, a node with index "42" exists in render cycle one but not in render cycle two, there is no way for *React* to know that there was a node with index "42" in the first render cycle. *React* move provides an animation mechanism to tackle the previously mentioned problem. All of the simulation data is passed to the *react-move* component inside the hybrid node component, which then keeps track of data changes via an internal system. Via animation hook functions the behaviors of entering, transitioning, and exiting nodes can be specified, which is similar to the *D3* prototype. Information about the usage and how the *react-move* library works internally can be found on the *GitHub* project in [14].

Program 4.10: Simulation tick handler of the hybrid force graph prototype.

```
applyNodeTick = (nodeSel) => nodeSel.attr('cx', (d) => d.x).attr('cy', (d) => d.y)
applyLinkTick = (linkSel) => linkSel.attr('d', this.getLinkPath)

ticked = () => {
  const { nodeTickHandler, linkTickHandler } = this.props

  nodeTickHandler
    ? nodeTickHandler(this.simulation.nodeSel)
    : this.applyNodeTick(this.simulation.nodeSel)

  linkTickHandler
    ? linkTickHandler(this.simulation.linkSel)
    : this.applyLinkTick(this.simulation.linkSel)
}
```

Advantages

The benefit of the hybrid implementation is that *React* can handle the node and link representation. Writing *React* code also implies that there are no disjointed code fragments of appending and removing components like in *D3* code. Using *React* components to render the force simulation nodes also implies that custom node components can be written and used in the force component, which makes the hybrid implementation highly versatile.

The rendering performance is probably close to a pure *D3* implementation, as *React* has to render the nodes for the simulation only once every data update. *D3* then takes over by altering the positions on the actual DOM nodes via its internal ticking function. Technically there is no difference between rendering the initial nodes and links of a force graph via *React* or *D3* as the DOM nodes have to be committed nevertheless. *D3's* force simulation tick cycles then directly alter the position attributes of the DOM SVG nodes. *React* is agnostic of *D3* being hooked into the DOM nodes, as the component only updates if the outside data changes. When the parent passes new simulation data, the simulation is fully updated in any case. Final numbers on performance are elaborated in chapter 5 though.

Disadvantages

A significant disadvantage of the hybrid prototype is that it might be unclear to maintainers of the library that *D3* handles the positions of the nodes and links and not *React*. Of course, documenting the positioning aspect could be a solution to the problem. Writing proper documentation is not only time but also resource consuming for maintainers. New maintainers of the library would have to completely understand how the hybrid combination of *React* and *D3* works to start being productive on the library.

4.4 Comparison of the Different Proposed Prototypes

When comparing the prototypes, an essential aspect is to understand, which technology renders what aspects of the simulation's nodes and links. Looking at the pure *D3* prototype, *React* only renders the base SVG element so *D3* can hook into the SVG base node and build up its simulation. The pure *React* prototype, on the other hand, renders the complete DOM node tree of the whole simulation, including the nodes' and links' positions. Last but not least, the hybrid prototype introduces a mixed rendering strategy, where *React* renders the whole DOM node tree, but *D3* selects the already rendered nodes and only manipulates their positions.

The performance of the pure *D3* prototype should act as a baseline for testing the other prototypes, as the whole simulation is handled by native *D3* code. The pure *React* prototype has to iterate a whole *React* render cycle on each simulation tick, which technically is performing worse than letting *D3* manipulate the DOM nodes directly. Therefore the hybrid prototype takes advantage of both worlds by leaving the expensive calculations and node position manipulations to *D3* while Rendering the nodes with declarative *React* code.

Looking at the figures 4.1, 4.2, and 4.3 again, they might appear to be quite similar. There are some subtle differences which make big differences though. Even though all prototypes mostly use two life cycle methods, it is of utmost importance to recall in which component life cycle phase the methods are called though. Figure 3.3 can help to look up all common life cycle methods again. While the pure *D3* and hybrid prototype pass the simulation to *D3* only in the commit phase, the pure *React* prototype has to pass its data to *D3* in the render phase.

4.5 Prototype Storybook

The thesis project furthermore contains one additional aspect that makes it quite easy to compare and test out all available prototypes. A community project called *Storybook* was utilized to compare and present the prototype results. The *GitHub* page in [16] explains the *Storybook* software as a development environment for UI components. The library lets developers build component browsers for their projects which showcase the project's components in so-called "stories." A story is a self-contained page which renders a component with a pre-applied configuration and state. In the case of the thesis project, the *Storybook* library was used to present all three prototypes in action and to make them interactive and browsable.

Figure 4.4 shows a visualization of the *Storybook*, which was developed for the thesis project. Users can select a prototype and a particular configuration from the sidebar and view the result in the main content section. All components are rendered inside a state container that provides data manipulation functionalities. For instance, via a simple click, the force data can be updated, links can be toggled, data or links can be shuffled and so on. All possible data manipulation functions can be seen in figure 4.4 on top of the component section.

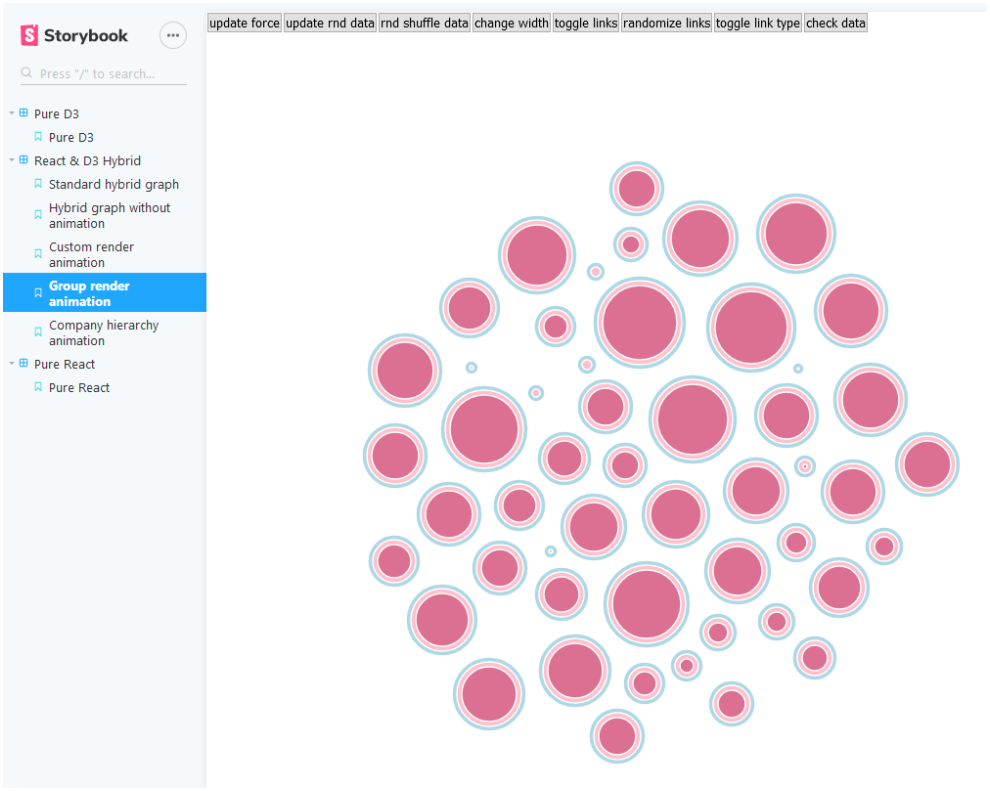


Figure 4.4: *React* Storybook currently showing the hybrid prototype.

4.6 Conclusion

All in all, each prototype has an interesting implementation on its own. The implmentation phase of the thesis project showed that every prototype has certain advantages and disadvantages. The question of which prototype is the best is a subjective decision in the end. An aspect that can be measured though is performance. Chapter 5 shows how the particular prototypes perform in comparison to each other. As letting other developers profit from the findings of this master project is desired, the hybrid prototype will be an open source project in the near future. More information about the plans of making the newly invented hybrid prototype open source can be read in chapter 6.

Chapter 5

Performance Testing and User Perception

The research question of the thesis is if *React* can be combined with *D3* without losing any performance in the browser. The following sections introduce the reader not only to the benchmark setup and testing methodologies but also present the final results of the benchmarks. There is also a discussion which elaborates all test results. Since humans cannot measure the exact amount of frames per second, a special section introduces the reader to the human perception of fluent animations and how the testing results can be interpreted even further with that knowledge.

5.1 Test Environment Setup

This section describes the testing environment that was implemented to compare the three force simulation component prototypes. There are a few challenges that had to be taken into account when realizing the testing environment. Also, the implementation details are elaborated and explained. Last but not least, the testing devices are introduced which are used to run the benchmark.

5.1.1 Challenges

One of the most challenging aspects of the thesis project is the performance measurement of the prototypes. Modern browsers have an uncountable amount of features that help to smooth out the performance to improve user perception. Getting some consistent performance numbers is much harder due to inconsistent browser optimizations. Using different browsers for benchmarks also means that different *JavaScript* runtimes are used to run the benchmarks. All engines have different execution and parsing speeds. Also, the mechanism to speed up frequently accessed script code is different, which also makes it harder to get consistent performance numbers.

When benchmarking web applications, it is very complicated to get performance values that have scientific relevance, which can be compared to get some accurate results. The next section introduces a system that was primarily implemented to measure the performance of *JavaScript* web applications. The system tries to tackle all challenges to producing detailed benchmark results.

Another big problem is the fact that browsers detect the refresh rate of monitors when utilizing the request animation frame functionality. Monitors with a refresh rate

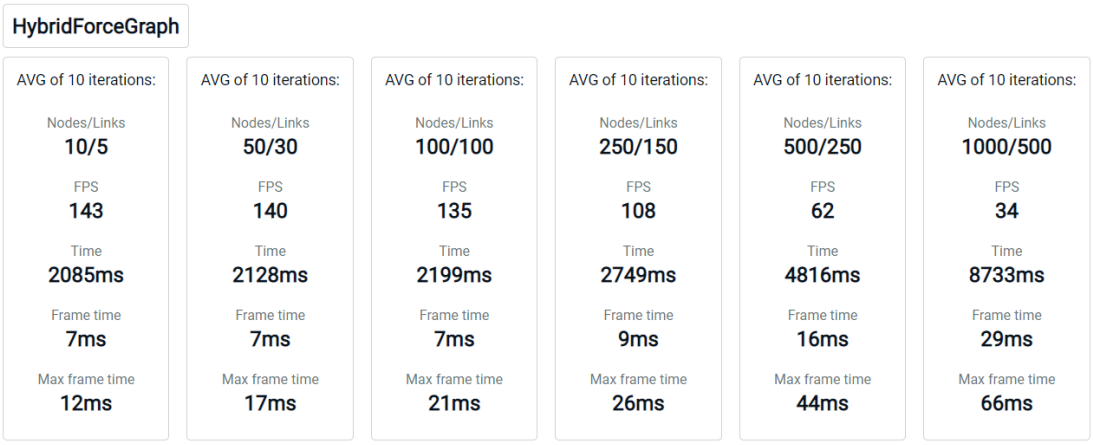


Figure 5.1: Hybrid prototype benchmark results.

of 144 Hertz allow browsers to produce up to 144 frames per second. 60-Hertz monitors, on the other hand, limit the browser's framerate to 60 FPS. The maximum amount of animation frame executions in any browser can only ever go up to the monitor's amount of Hertz the browser window is running in.

Speaking of utilizing the request animation frame functionality, it must be mentioned that each browser has a different implementation of the functionality. The *Chrome*[1] browser, for example, is smoothing out the performance by trying to execute animation frames regularly, which means that the overall performance might be lowered to achieve a smoother framerate as explained in [23]. Experience shows that *Firefox*[2], on the other hand, always fires its animation frames whenever a frame is available, which can result in higher overall but not as consistent framerates.

Last but not least, another challenge was to build a performance measuring environment that can be used without adding code to the prototypes. Theoretically, each prototype should be a complete component that can be shipped as a third party library. By adding performance measurement specific code, the components would contain functionality that is not needed when shipping production builds of the components.

5.1.2 Building a Stable Testing Environment

A testing environment which produces consistent data input across multiple iterations yields the best testing results when benchmarking all three prototypes. Testing the force simulation prototypes with the same data across multiple benchmarks, environments, and devices is of utmost importance. A valid solution is to use a pseudo-random data generator which can be restarted and reseeded each benchmark iteration.

Generating an arbitrary amount of random node and link positions is not a problem, as *JavaScript* has a built-in random generator which can be utilized to generate random data. Unfortunately, *JavaScript's* random generator cannot be seeded to achieve consistent pseudo-random results. The library *seedrandom* in [11] is the perfect technol-

[1] https://www.google.com/intl/de/chrome/
[2] https://www.mozilla.org/de/firefox/new/

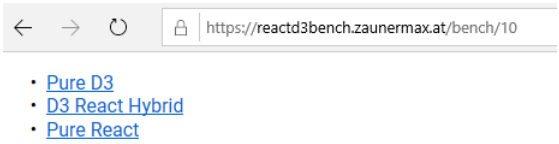


Figure 5.2: Overview of the benchmark application.

ogy to solve the problem of generating consistent data across multiple iterations of the benchmark tests.

Since browsers frequently yield different performance numbers across iterations with identical data, the amount of cycles per test data iteration has to be increased. When testing one specific iteration multiple times, the average value has much more significance than testing an iteration only once. Therefore the testing environment must have support for different iteration configurations, which can be run an arbitrary amount of times.

The testing environment is designed to be a stateful container which generates some random data and then passes it to the desired prototype. *D3* simulations provide a mechanism to add an event handler whenever the simulation stops. Therefore, it is no problem to pass a handler to the benchmark component that is executed whenever the simulation stops. The handler can be used to restart the benchmark with some newly generated data until the desired amount of benchmark iteration cycles is reached.

Pseudo-randomly generated data is obtained by a specially implemented custom helper utility module. The custom module is designed to generate a specified amount of random nodes and links. The method to generate random data takes two parameters—the number of nodes and the number of links. Generating consistent pseudo-random data is possible by internally using the previously mentioned seedrandom module.

Tackling the performance measurement problem is a much harder task, however. To be able to measure the number of frames per second, the request animation browser functionality can be used. Another custom implemented module provides some functionality that is specifically designed to measure performance intensive *JavaScript* animations. By requesting an animation frame as often as possible in a terminable infinite loop, a reference timestamp can be used to measure the amount of animation frame executions per second which is equal to the frames per second the browser produces.

Presenting the performance results must not be underestimated either. To provide benchmark results appealingly, each iteration is represented via a visual container that contains all relevant information for a specific test iteration. If there are six test iterations, six containers are rendered after the benchmark. The containers contain the number of cycles per iteration, the test configuration, the number of frames per second, the overall execution time per cycle, the average frame time, and also the highest frame time. Figure 5.1 shows how the benchmark results visually look like.

Finally, the benchmark tool has to be easy to use on all kinds of devices. The thesis project also contains a small *React* application which can be deployed on any static web hosting service that can serve single page applications. A visual representation of the basic benchmark app can be seen in figure 5.2. The *React* application is a wrapper

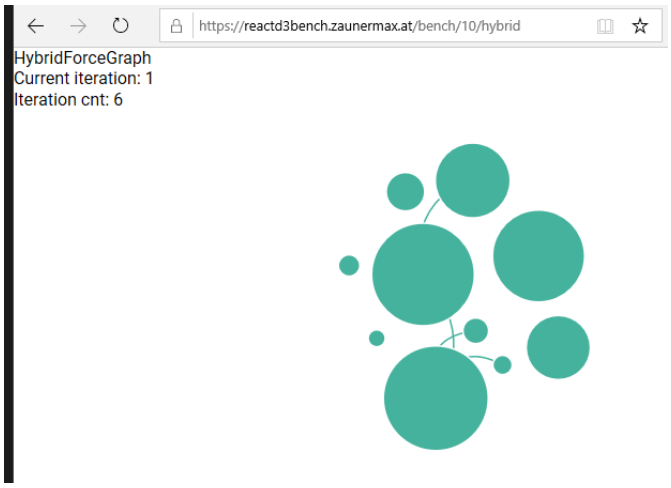


Figure 5.3: The benchmark currently iterating through hybrid prototype iterations.

Table 5.1: The table shows a list of low-end testing devices.

Device	*CPU*	*GPU*	*RAM*
OnePlus 1	Snapdragon 801	Adreno 330	3GB
OnePlus 5T	Snapdragon 835	Adreno 540	8GB
SurfaceBook	Intel i5-6300U	Intel HD 520	8GB

around the testing environment, which lets users select the desired benchmark for any of the three force simulation prototypes and then runs it in the browser. Each benchmark also has an easily distinguishable URL to be able to copy paste a specific benchmark URL into any browser. Figures 5.2 and 5.3 show how the URL contains all relevant benchmark parameters. That way the benchmark URLs can be pasted into any browser to execute the benchmark.

5.2 Testing Setup

Having a bulletproof testing setup plays a fundamental role in producing scientifically releveant test results. Therefore, the following sections of the thesis provide a thorough insight into the testing devices and on how the whole benchmark methodology is conceptionalized.

5.2.1 Testing Devices

The amount of test devices should be as high as possible while still being reasonable regarding to the effort it takes to process all resulting benchmark data. A total amount of six devices is enough to retrieve scientifically significant results because each prototype is not only tested on each device, but also on two browsers with multiple iteration

Table 5.2: The table shows a list of high-end high refresh rate testing devices.

Device	*CPU*	*GPU*	*RAM*
Tower (Max Z.)	Intel i9-7900X	2x Nvidia GTX 1080Ti	32GB
Razer Blade 15 (2018)	Intel i7-8750H	Nvidia GTX 1070 Max-Q	16GB
Tower (Patrick M.)	Intel i7-6700k	Nvidia GTX 1080	16GB

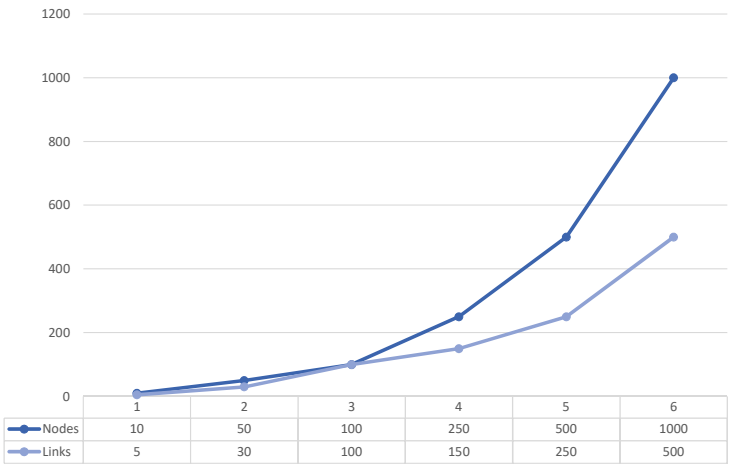


Figure 5.4: Benchmark iteration configuration.

difficulties which are executed multiple times. The range of devices is divided into two sections: high-end devices and low-end devices.

The mobile devices used for testing are a *OnePlus 1* phone, a *OnePlus 5t* phone, and a *SurfaceBook* in tablet mode. All low-end devices and their specs are listed in table 5.1. It must be noted that every selected low-end device has a monitor refresh rate of 60-Hertz.

Table 5.2 introduces all high-end devices which have a monitor refresh rate of 144-Hertz. Two of the listed devices are custom tower builds with custom specs and one device is a *Razer Blade 15* laptop with specs defined by its manufacturer. All in all the devices should provide a good overview of the performance of the force graph components.

5.2.2 Testing Methodologies

Running the benchmarks to get good quality testing results is relatively straight forward. To get the most declarative performance numbers it is best to mainly use devices with a high monitor refresh rate. Devices with a low monitor refresh rate can possibly falsify some testing results. If, for example, a browser could possibly render 100 frames per second during an iteration, a system with a 60-Hertz monitor would only be able to

measure a maximum of 60FPS as the theoretically possible 100 FPS would be capped to the monitor's 60-Hertz refresh rate.

Devices with high refresh rates can be sufficient for measuring the overall best performing prototype. One of the more interesting research aspects of the thesis though is the question, how well the prototypes perform on mobile devices with lower performance specifications than desktop PCs. Thus the testing results must be split up into different categories as a consequence. Due to the fact that not only frames per second, but also other perfromance aspects like total execution time are measured, the lower performing devices can also be compared to each other.

Each device runs through six iterations per prototype with exponentially increasing rendering difficulty. Figure 5.4 shows the benchmark configuration for the benchmark. Starting with a node count of 10 and a link count of five, the configuration ultimately goes up to 1000 nodes and 500 links. The third iteration is special, as the number of nodes and links is the same. The special configuration was added to test an extreme scenario of all nodes being connected to each other to induce some extra performance heavy force calculations.

Each iteration runs through 10 cycles which equals to a grand total of 60 cycles per prototype, browser, and device combination. Three high-end devices with 144-Hertz monitors and the three low-end devices with 60-Hertz monitors run the benchmark iterations in the *Chrome* and in the *Firefox* browser. The two browsers were selected, because they are the most significantly used, platform independent browsers worldwide according to the statistics in [25] and [26]. Taking into account that there are six devices, three prototypes, and two browsers, the total number of iterations is 36.

5.3 Benchmark Results

This section answers the research question if *React* can be combined with *D3* without introducing any performance losses in the browser. Extensive benchmark testing sessions resulted in some remarkable research results which are presented below. After presenting the test results, an introduction to the human perception of fluent animations helps the reader to follow the subsequent interpretation of the benchmark results.

5.3.1 Introducing the Test Results

First of all, the thesis project is a success, as the overall performance numbers show a clear trend that the hybrid prototype is ahead of *Uber's* pure react implementation. The total execution time of all combined benchmark iteration cycles is exactly 25326s when combining the average execution times of each test. When converting milliseconds to hours, the result is 7.04 hours. Since the benchmark environment was designed for extensive benchmark sessions, the time between running the benchmarks was minimized. The only manual task was to write down the actual benchmark results. Letting devices run the tests required no further user interaction.

Figure 5.5 shows the average FPS of the low-end devices. An overall downwards trend can immediately be seen in the FPS chart, which is expected. The more DOM nodes the browsers have to calculate, the lower the frames per second get per iteration. Each group of bars in the chart represents an average value for each prototype per

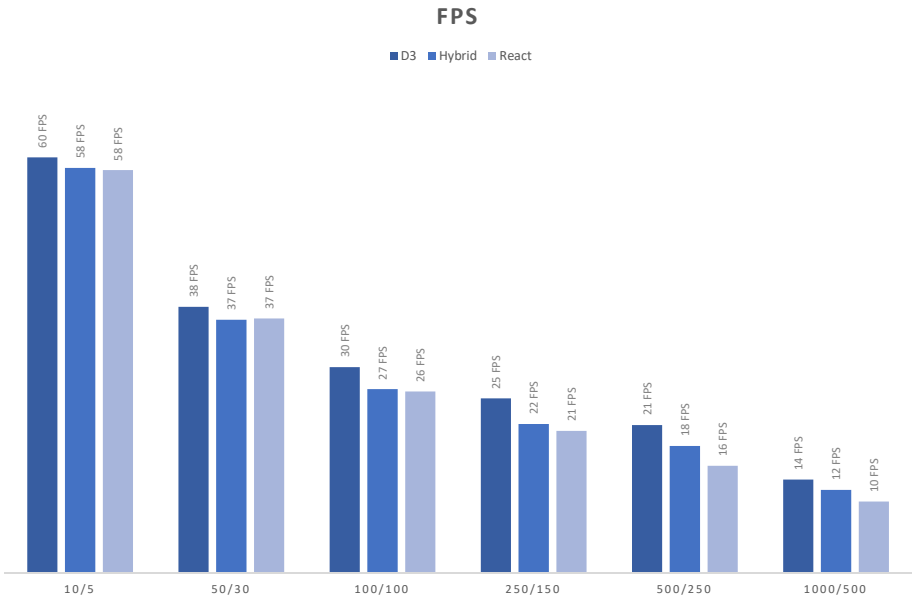


Figure 5.5: Low-end devices' average frames per second per benchmark iteration cycle (higher is better).

benchmark iteration. Note that all values are the average taken from 10 iteration cycles of the *Chrome* and the *Firefox* browser. The prototypes mostly yielded the expected performance numbers starting with the reference performance of the pure *D3* prototype, followed by the hybrid component and then finally followed by the pure *React* force graph component.

Figure 5.6 shows the average FPS values of the high-end devices' benchmark iterations. Looking at the bar chart, it is apparent that the high-end devices hit the FPS cap of 144 FPS throughout the first few iteration cycles. From the third iteration a steady decrease of FPS can be observed though, as the iteration difficulty is high enough for all devices not to hit the monitor refresh rate limit anymore.

The bar chart in figure 5.7 shows the average time it took to complete the benchmark fully. Via the browsers' performance API, exact timestamps can be measured once a benchmark cycle starts, and once it ends. By subtracting the start timestamp from the end timestamp the overall time to execute is calculated. Like the FPS chart, the time to complete (TTC) chart also shows all average values it took the different prototypes to complete the benchmark cycles.

Continuing with the high-end devices, the chart in figure 5.8 shows the average time in milliseconds it took the devices to finish one iteration cycle. The results show that not only FPS can be capped at maximum values but also the TTC can be capped at a minimum value as shown in the first iteration cycles. Since the performance measurement utility is tied to the browsers' animation frame functionality, being capped at a maximum value also means being restricted on minimum values.

Measuring the average frame time of animations can provide insights into the user's perception of fluent animation. If the value is too high, the animation may not be

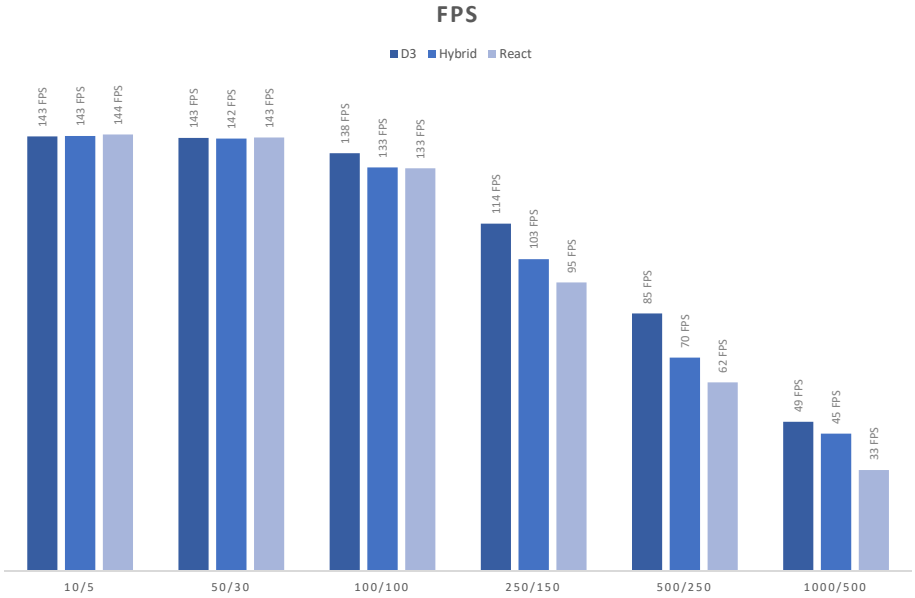


Figure 5.6: High-end devices' average frames per second per benchmark iteration cycle (higher is better).

experienced as a smooth animation. Therefore measuring the value is crucial when comparing the prototypes to each other. Figure 5.9 shows a bar chart of the low-end device benchmark. Towards the last two iterations, the benchmark configuration seems to have hit a certain threshold since the values rise exponentially. The other performance results show a similar pattern to the previous results, however.

Looking at the high-end results in figure 5.10 the same pattern as before can be observed, where the first iterations are capped to a specific minimum value. However, the rest of the results increase exponentially, which correlates to the rest of the high-end performance results. The average frame time of the last iteration stands out and spikes with an exceptionally high value.

Last but not least, a critical aspect of any animation performance measurement is the maximum time between frames measured. The value can provide critical insights to some performance issues even though the average frame time per second might look okay. The chart in figure 5.11 shows an average of the maximum frame time value to each iteration cycle. One unanticipated result was that the maximum frame rate of the hybrid component is significantly higher throughout the testing results than the pure *D3* or pure *React* components.

However, the performance graph in figure 5.12 shows the high-end devices' maximum frame time results which are more expected. Again, the average maximum frame time rises exponentially. The results contain a few irregularities, though. The maximum frame time of the reference pure *D3* prototype, for example, is sometimes higher as the value of the other two prototypes. The maximum frame time values, therefore, must be taken with a grain of salt, as any maximum value could be caused by an unexpected system or browser activity which could have decreased the overall performance of any prototype.

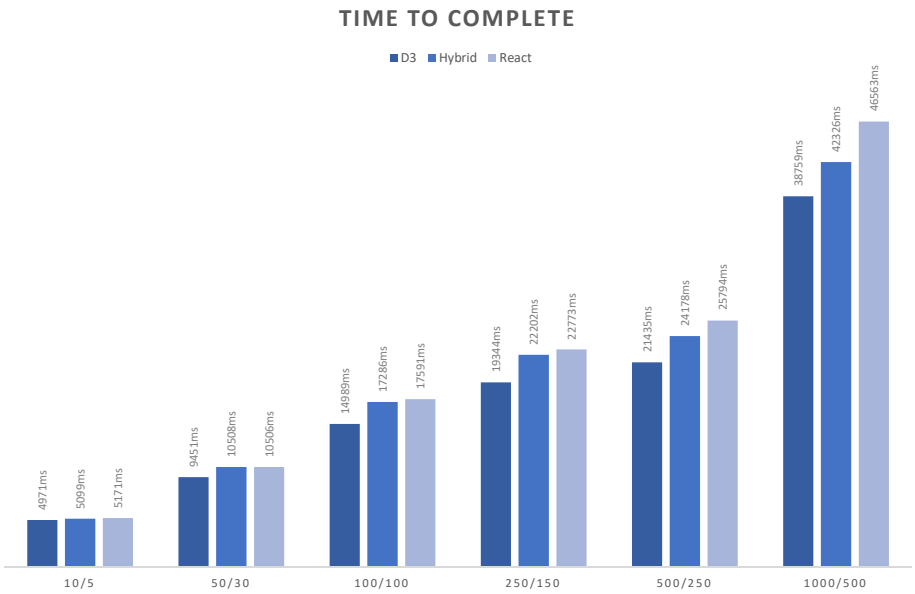


Figure 5.7: Low-end devices' average time to complete for one benchmark iteration cycle in milliseconds (lower is better).

As mentioned before, the problem is mitigated by executing one iteration multiple times, but there is still a margin of error, though.

5.3.2 Human Perception of Fluent Animations

Human vision is a very complicated topic, as there has been a large volume of studies over many years which tried to determine at which point humans perceive a series of images as fluent motion. Read & Meyer claim in [6] that humans perceive motion if the animation is displayed with at least 12 frames per second. During further research, another article in [4] was found which states that humans can detect specific images in a rapid serial visual representation (RSVP) of a series of multiple pictures. The paper in [4] found that participants could determine if the RSVP stream contained a specific picture even with a frequency of displaying each picture for just 13ms.

Another vital question is at which point an observer does not perceive an animation as fluent anymore. As mentioned before, the time per frame should be below 13ms in the best case to provide the perception of fluent animation. Although animations with at least 12 frames per second can be perceived as motion, they are not necessarily experienced as fluent motion though. As the performance measurements of the thesis project yield some final numbers, they can be used to determine if the animation of the test can be experienced as fluent motion or not.

Further research in the field of human perception of animation revealed yet another interesting result. Countless studies throughout the last decades have tried to find an answer to the question at which point humans do not experience an animation as stuttering or flickering anymore. Previous research has established that flickering or so-called "stuttering" cannot be detected if the human perception cannot distinguish between

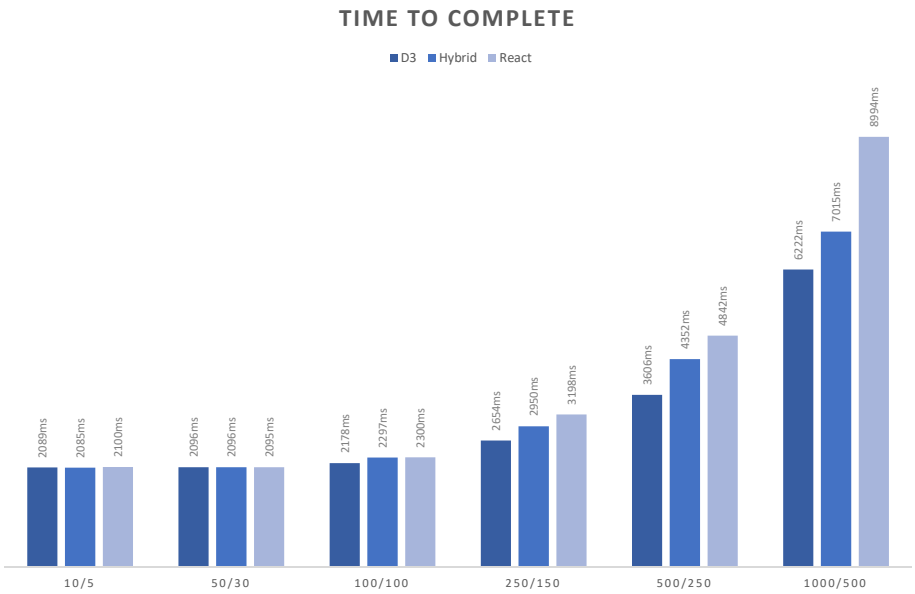


Figure 5.8: High-end devices' average time to complete for one benchmark iteration cycle in milliseconds (lower is better).

modulated light and a stable field anymore. The rate seems to be between 50 and 90 Hertz according to several resources in [1, 2, 7]. Even though the findings are mostly about the refresh rate of monitors, the same principle also applies for the displayed frames per second of an animation rendered in the browser.

5.3.3 Interpreting the Test Results

When interpreting the test results, it is essential to keep in mind that the hybrid implementation uses a custom animation library internally, as mentioned in subsection 4.3.3 which can be turned off to improve performance. The animation feature was turned on during the execution of the benchmark tests even though the test iterations do not include any individual node animation. As a result, the pure *React* component has a clear advantage over the hybrid component by not having to calculate node animations as they are not implemented on the pure *React* component. Even though the additional performance decreasing feature was kept on during all tests, the hybrid component generally yielded equal or better performance numbers in comparison to the pure *React* prototype.

Another clear advantage of the *React* prototype is that newer versions of *React* provide functionality which assigns a lower priority to DOM nodes which are outside the viewport of the browser. Lin Clark explains the functionality of the library during the *React* conference in [8], where the feature was first introduced. Due to the fact that mobile devices mostly have smaller viewports, a large proportion of the nodes in higher iteration difficulties is rendered outside of the viewport again providing an advantage to the *React* prototype.

Looking at the frames per seconds, the charts in figures 5.5 and 5.8 indicate how

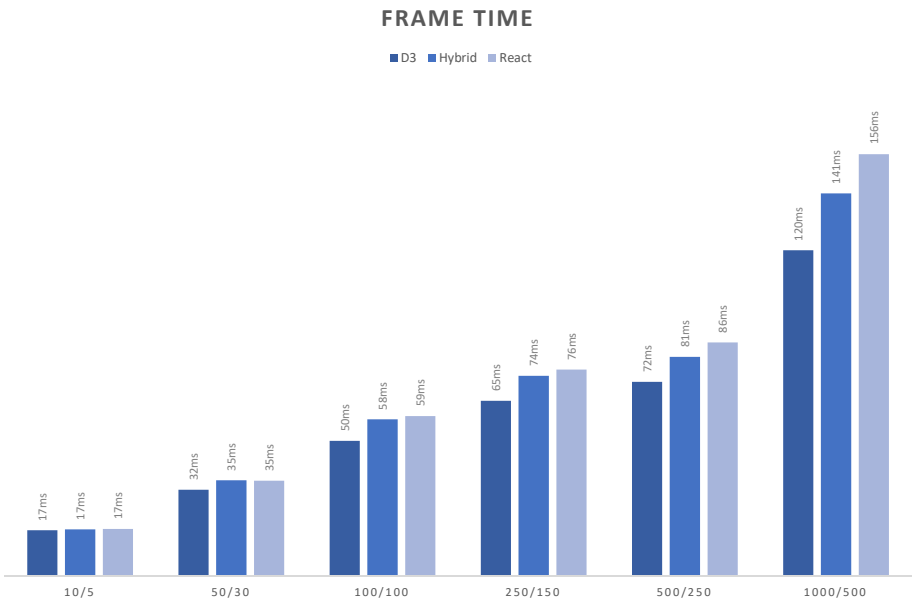


Figure 5.9: Low-end devices' average frame time per benchmark iteration cycle (lower is better).

the *D3* prototype yields the best result in every iteration, followed by the hybrid implementation with the second best results and lastly the pure *React* prototype with the slowest results. Closer inspection of the two charts shows that the performance decrease is linear, whereas the increase of render difficulty in figure 5.4 is exponential. Because the frames per second are calculated via the `requestAnimationFrame()` function, the results show that the browsers regularly try to provide animation frames even though the calculations get exponentially harder.

In comparison to the earlier presented results, the overall TTC values in figures 5.7 and 5.8 do not correlate with the frames per second which is a rather unexpected result. As mentioned before, browsers try to provide as many animation frames as possible, whereas the overall TTC is measured via two timestamps. A remarkable outcome though is the fact that the first and most lightweight iteration results show that the low-end devices' TTC is roughly 40% of the high-end devices' TTC. This supports the theory that the monitor refresh rate is directly tied to the browsers' request animation frame functionality as 40% of 144 roughly equals 60.

The high-end and low-end TTC values do not correlate either. While the high-end benchmark values in figure 5.8 show the expected exponential increase of TTC, the low-end results in figure 5.7 show a linear increase for the TTC one iteration cycle. The findings can be explained that browsers cannot complete full render and paint cycles anymore during one animation frame due to the low-end hardware. By generating a constant backlog of due animation frames, some of them might get dropped due to new animation frame requests that are more recent. Once animation frames get canceled because of more recently requested animation frames, the pattern of increasing completion time is more comparable to the pattern of the overall FPS increase in the

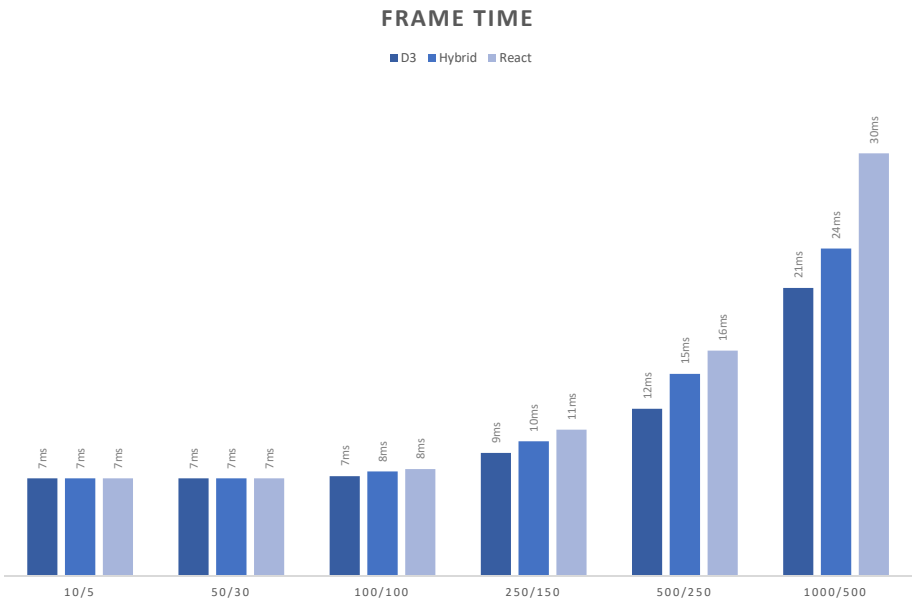


Figure 5.10: High-end devices' average frame time per benchmark iteration cycle (lower is better).

charts in figures 5.5 and 5.8. The high-end results in figure 5.8 show the expected exponential increase of TTC an iteration cycle, as browsers can process the calculations completely within the requested animation frame.

Figures 5.9 and 5.10 show the average frame times of the prototypes. The results can be seen as another way to describe frames per second. As described in section 5.3.2, the frame times play a crucial role for humans to percieve an animation as fluent without stuttering. For all low-end devices, benchmarks showed rather high average frame times starting from the second benchmark iteration. While the results stayed well above the previously mentioned 12 frames per second threshold, the lower frame time is definitely noticeable. Probably one of the most significant findings when looking at the general trend, the hybrid prototype regularly comes out ahead, when comparing the average frame times. The high-end chart in figure 5.10 shows a clear spike of the pure *React* implementation in the last iteration. A possible explanation for this outcome might be that *React* reaches the previously already mentioned threshold when performing the calculation for an animation frame. *React* performs pretty well, considering it has to process 1500 DOM nodes every 30ms completely and put them through a complete render cycle.

Last but not least, the maximum frame time results can also provide valuable insights into how the prototypes perform compared to each other. These results must be interpreted with caution though, because various unforeseeable reasons can cause maximum frame time spikes. The results for the low-end devices in figure 5.11 show, for example, that the hybrid component had the highest maximum frame time in the last two iterations. Now, if this only happens once in the animation, this might not be noticeable, but if this would be a continuous trend, the data could point to a performance

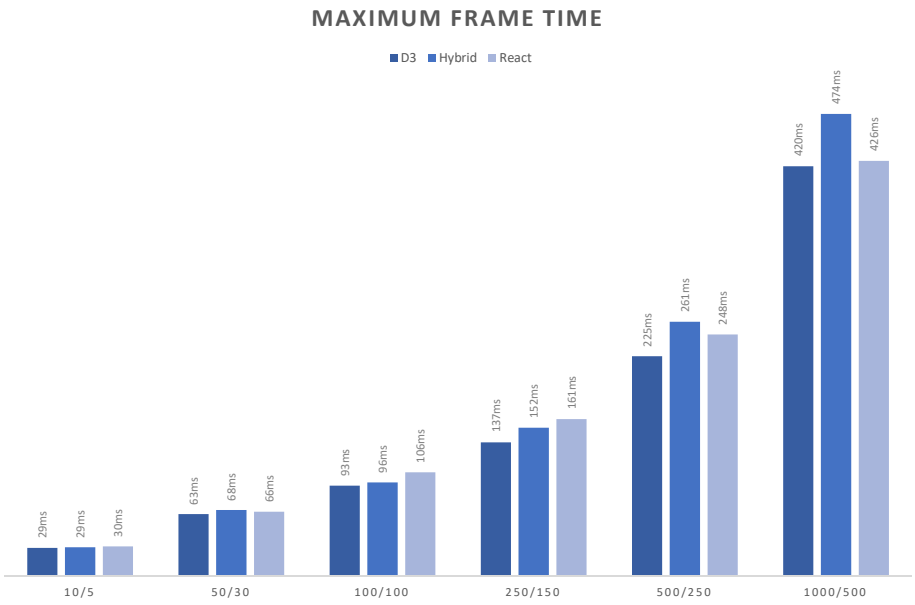


Figure 5.11: Low-end devices' average maximum frame time per benchmark iteration cycle (lower is better).

problem in the implementation of the prototype. The explanation for the performance spikes is most likely also tied to the fact that low-end devices run into the animation frame limit of not being able to calculate a full render cycle in one animation frame. The hybrid component not only has to build up the whole component tree via react but also calculate the animations with react move. If both instances are limited to animation frame constraints, the calculation could be spread out across multiple frames, making the whole animation slower in the process.

On the contrary, the high-end maximum frame time tests in figure 5.12 yield more expected results. When having to deal with many nodes and links, the maximum frame times should be equally affected as a consequence. The last iteration shows the longest time between frames for the pure *React* prototype, which is 76 milliseconds. The pure *D3* and hybrid component seem to be able to maintain a maximum frame time at about 60 milliseconds during the cycles of the last iteration.

5.4 Conclusion

All in all, the different prototypes performed quite well in general. The question "Why does the benchmark use numbers as high as 1000 nodes and 500 links?" which might come up during the examination of the testing results can quickly be answered. The test that was used to benchmark the prototypes uses a simple force simulation configuration where each base node is just a single circle element, and each link is a path element. When using more sophisticated nodes which contain multiple SVG elements, the outcome could be the same. If an example simulation would use ten SVG elements in just one node, rendering 100 of those nodes would yield a similar outcome as rendering the

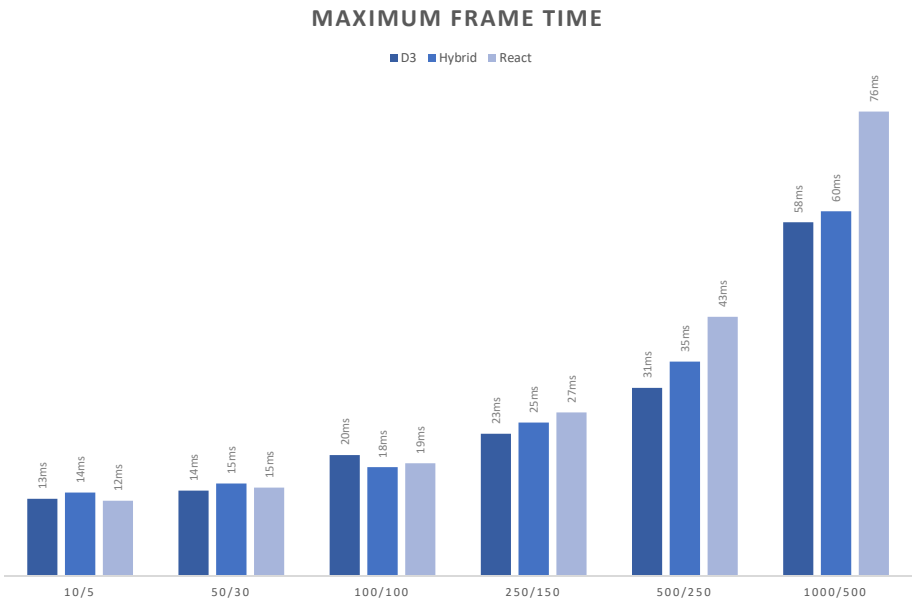


Figure 5.12: High-end devices' average maximum frame time per benchmark iteration cycle (lower is better).

6th benchmark iteration.

Overall all the prototypes performed pretty well. As mentioned before, the pure *D3* component is the clear winner and best performing prototype, but that was the expected result, as it was developed to serve as a baseline for the results of the other prototypes. When glancing over the rest of the results, the hybrid nearly always comes out ahead of the pure *React* prototype. The primary purpose of the study was if *React* can be combined with *D3* without losing performance in the browser. The definitive answer to that question has to be no, unfortunately, as there are some performance penalties when combining two full grown libraries. The test results also support the answer to the research question.

When talking about user experience though, the prototypes are pretty much all usable in production projects, as the divergence of the performance numbers mostly stays within the bounds of a smooth animation experience. The benchmark results of the third and fourth iteration are acceptable when looking at the FPS and average frame time. When using the force graphs on mobile devices, the performance is worse overall, but due to the fact that even the pure *D3* prototypes performed a lot worse on low-end devices, the combination prototypes cannot magically yield a better performance, as they technically not only have to calculate *D3* force simulation ticks but also *React* render cycles. As a consequence, the performance is worse than just letting *D3* handle the whole simulation on its own.

Chapter 6

Open Source and the React Community

Ultimately the thesis project was always planned to be an open source project at some point in the future. The current *React* community is extremely active and productive because of countless open source projects that provide useful libraries that can be used in any project free of charge. As the thesis project is quite the success in terms of the achieved goals and the performance numbers, the overall result should not only be presented in this masters' thesis. Instead, the project should be published to the *React* community who truly benefits from the software.

This chapter is about the publishing plans of the hybrid prototype and what API it should have to provide the most benefit to the users of the component. The thesis project was created out of necessity for a component that would handle a *D3* force simulation but can be written in *React* code. The first versions of the API are probably very opinionated, as there was a particular use case for the component. However, the community might alter the proposed component API in the future.

6.1 Building an Open Source Library Component

Conceptionally speaking, if there is a library that ships a single component, the *React* component has to be somehow packaged and then published to *npm*[1] to be usable by other developers. Since *npm* is one of the biggest package managers for web projects, the component can be installed in any web project once it has been published to the *npm* registry. Bundler tools like *webpack*[2], *rollup*[3], or *parcel*[4] make it easy to create static library assets that can be published to *npm*.

Furthermore, the code should be hosted on a public collaboration platform. The best option to host a public VCS repository is *GitHub*[5] of course, as public projects can be hosted free of charge. As a consequence the ultimate goal of open-sourcing the component is not only to publish the component to *npm* but also to *GitHub*. Most if not almost all of the open source third-party react component code bases are hosted on

[1] https://www.npmjs.com/
[2] https://webpack.js.org/
[3] https://rollupjs.org
[4] https://parceljs.org/
[5] https://github.com/

Program 6.1: Alpha version of the force graph component API.

```
1 <HybridForceGraph
2   height={height}
3   width={width}
4   nodes={nodes}
5   links={links}
6   forceOptions={simulationOptions}
7   nodeTickHandler={nodeTickHandler}
8   renderNode={customNodeRenderer}
9   renderLink={customLinkRenderer}
10  animation={animationConfig}
11 />
```

GitHub.

The usage of the react component should be as easy and straigt forward as possible to enable developers an easy starting point to using the component. The API should be designed in a way that programmers can incrementally opt into more complicated features. By simply using the component with the standard required props, a standard *D3* force simulation should be rendered. After going through the documentation and the tutorial, developers should also be able to gracefully opt into the more complicated features of the component like custom node and link rendering.

As mentioned in section 4.3.3, a big disadvantage of the hybrid component is the fact that developers must understand how the hybrid implementation works in order to being able to start contributing to the open source component. The library should therefore have a very elaborated and well written contribution documentation to make it easier to contribute to the component.

6.2 Technical Details of the Component API

A draft version of the API was already implemented throught the implementation phase of the master thesis project. The storybook already utilizes the API to show customized versions of the hybrid force graph. The program 6.1 shows, how the component API looks like. There are a few mandatory props like the height and width of the SVG element. The nodes and links could be omitted, but it wouldn't make any sense, as the force graph has to get its data from somewhere. The two data properties are designed to take falsey values in case the data comes from a web API and is not available during the first parent component render cycle. Every prop starting on line 6 and onwards is optional and can be fully customized by the user of the component.

In chapter 4 the figure 4.4 shows a component story of a customized force graph component. The snippet in program 6.2 shows the source code for the custom component. Instead of using the default node renderer, a custom node rendering function is used as it can be seen on lines 4 and 19. The custom node renderer renders a base `<g>` SVG element per node, which contains three circles with different radius settings.

Due to the fact that the base element is not a circle element anymore, the ticking function has to be customized as well. Line 1 in program 6.2 shows the implementation

Program 6.2: Alpha version of the force graph component API.

```
const customTickHandler = (nodeSel) =>
  nodeSel.attr('transform', ({ x, y }) => `translate(${x},${y})`)

const customNodeRenderer = ({ id, size }) => (
  <g id={id} key={id} className={'node'}>
    <circle r={size} fill={'lightblue'} />
    <circle r={size - 5} fill={'pink'} />
    <circle r={size - 10} fill={'palevioletred'} />
  </g>
)

const CustomForceGraph = ({ height, width, nodes, links }) => (
  <HybridForceGraph
    height={height}
    width={width}
    nodes={nodes}
    links={links}
    nodeTickHandler={customTickHandler}
    renderNode={customNodeRenderer}
    animation={null}
  />
)
```

of the custom tick handler. Instead of passing the position via `cx` and `cy` coordinates, the function applies a transform property to the the node selection which translates the base node to the `x` and `y` coordinates. The translation is necessary, as the group SVG element cannot directly accept `x` and `y` coordinates.

The code example in program 6.2 demonstrates really well, how the graceful opt-in strategy of the API works. The example does not use every possible prop that can be added to the force graph like demonstrated in the code example in program 6.1. The implementation of the custom graph omits the `forceOptions` prop for example. The hybrid force graph component falls back to the standard force simulation configuration as a result. The custom link render function prop is also omitted as the default link implementation is sufficient for the custom force graph.

6.3 Final Thoughts

In the end only time will tell, if the component is useful for the community once it is published and if there are some open source developers willing to contribute to the project. Due to fact that the force graph component is already utilized in at least one production project the library component is already a success.

Chapter 7

Conclusion

This chapter is the conclusion of all the findings of this master's thesis. Not only the prototypes are recapitulated, but also the test results are summarized. The most apparent finding emerging from this thesis is the fact that there is a possible combination of *React* and *D3* which only sacrifices a small proportion of performance to enable *React* developers to use *D3* functionalities, but write *React* code.

7.1 Prototypes

Three prototypes were developed to implement a combination of *React* and *D3*. Every prototype has its advantages and disadvantages. They all pursue the same goal though, which is to provide a *React* component that completely encapsulates the complete *D3* force simulation functionality. The force graph component has to be usable like any other *React* component, which means that the component has to be a declarative representation of state. To get a force simulation component that is a function of state, all prototypes have to *React* to state changes accordingly.

7.1.1 Pure D3 Prototype

The first prototype is the pure *D3* variant as introduced in section 4.3.1. The component is a *React* wrapper that renders a base element which *D3* then hooks into. *D3* is completely in charge of the whole simulation. Not only does *D3* append and remove the nodes from the DOM but also *D3* controls all node positions of the simulation. The prototype relies on *React's* reconciliation algorithm, which never detects any change in the staticly rendered base element and always keeps the reference to the real SVG element intact.

The implementation is very similar to a *D3* only implementation, as *React* is only used to render the static base SVG element. Much imperative *D3* code is used to append and remove nodes. Furthermore, the code contains a lot of *D3's* imperative selection functions, which help to keep track of entering and exiting nodes to apply transitions to them. The complete presentation layer is handled by *D3*, which includes not only style attributes but also the positions of the nodes and links.

A significant advantage of the pure *D3* prototype is that its performance is close to if not the same as a native *D3* implementation. The most notable disadvantage is

that almost the entire code of the component is written in pure *D3* code, which could lead to a poor developer experience if the component advances in the future. Also, no custom render functions can be used, as a custom mechanism would have to be added to enable user customized *D3* code in the component.

7.1.2 Pure React Prototype

Section 4.3.2 introduces the pure *React* prototype. *React* is completely in charge of the presentational layer. As a consequence, not only the nodes' and links' positions but also their styling and node types are completely handled by *React*. When the component mounts, *D3's* force simulation is initialized with the data. The prototype then pulls the data via the tick function and sets it in its internal state. Due to the nature of *React*, the component goes through a new render cycle to render the newly fetched node and link position. The complete cycle happens multiple times a second.

As mentioned before, the component's constructor is used to initialize *D3* even before *React* renders anything. The tick handling function pulls all node and link positions from the simulation every tick and sets it via *React's* set state function every fraction of a second as often as *D3's* tick function is called. The component then completely rerenders with the newly set node and link positions. Because the component's state updates every fraction of a second, component prop updates have to be distinguished from state updates. The `shouldComponentUpdate()` function is used to update the *D3* force simulation only if the props have changed.

One of the most significant advantages of the prototype is that maintainers of the component can use declarative *React* code to handle the complete presentation layer of the simulation. In addition to the styling and node type, also the position is entirely handled by *React*. The fact that *React* is used to handle the whole simulation tree is also the most considerable disadvantage because it implies that *React* has to go through a complete rendering cycle every time the simulation's tick handler is called. Another advantage is that props can customize the prototype's render functions. However, it is quite complicated to achieve the full *D3* functionality like dragging and enter and exit animations, as they have to be implemented from scratch.

7.1.3 React and D3 Hybrid Prototype

Last but not least, there is the *React* and *D3* hybrid prototype, which is presented in section 4.3.3. The hybrid component works by initially rendering all the simulation's nodes via *React* and then letting *D3* select the already existing nodes to handle the nodes' positioning. *React* handles the styling and node shape and *D3* handles the node and link positions. The hybrid component uses all the lifecycle methods in *React's* commit phase to being able to read the DOM nodes that have already been committed to the DOM.

The implementation of the hybrid component is very straight forward. The *React* lifecycle method `componentDidMount()` is used to initialize the *D3* simulation and the `componentDidUpdate()` lifecycle method is used to update the *D3* force simulation if the simulation data props have changed. The render method takes care of the complete simulation presentation except for the positions, which are handled in the tick updater function. The tick handler function is one of the few instances in the hybrid component

where *D3* code is used.

Being able to make the render functions pluggable is also a significant advantage of the hybrid component. Also, *D3* functionality like dragging nodes, zooming, and adding transitions is easy to implement with the hybrid prototype. The only disadvantage might be that maintainers of the prototype would have to completely understand how the rendering process works to be able to contribute to the component.

7.2 Performance Results

This thesis aims to find a well-performing combination of *React* and *D3*. The research question "Can *React* be combined with *D3* without losing performance in the browser" unfortunately has to be answered with "*React* can be combined with *D3*, but there are some slight render performance losses". The results of the performance benchmarks clearly show how the newly invented hybrid component is always ahead of *Uber's* take on combining the two libraries.

However, when looking at the performance numbers, research shows that every prototype can be used on high-end devices without notable performance issues. The low-end devices suffer from low calculation speeds and low monitor refresh rates in general. Even though the hybrid prototype is ahead of the pure *React* component when measuring the raw performance, it is clear though that the performance differences are hardly measurable without knowing the exact numbers.

7.3 Open Source

Because the hybrid prototype is easily extensible with not only *D3* but also *React* functionality, the decision was made to open source the component. The *React* community lives off of free to use components that can be integrated into any project free of charge. The hybrid component was developed out of necessity for a better solution of combining *React* and *D3*, and maybe other developers find it useful as well.

7.4 Final Thoughts

Even though the performance numbers could have been better, the project is considered a success nevertheless due to the fact that the hybrid prototype's performance is not only on par or even superior to *Uber's* pure *React* variant. The performance is worse than a native *D3* implementation, but that is to be expected when two big libraries are combined. The hybrid's component API also turned out to be very useful as the storybook examples demonstrate. Being able to not only use the power of *D3* but also to write almost pure declarative *React* code to implement a force simulation is definitely considered a success.

Appendix A

CD-ROM Contents

Format: CD-ROM, Single Layer, ISO9660-Format

A.1 PDF files

```
/
├── latex-source ..... latex source files
└── thesis_EN.pdf ..... master's thesis (main document)
```

A.2 Benchmark Test Results

```
/
└── benchmark-results.xlsx ..... excel file containing all results and calcluations
```

A.3 Thesis Project – Combining React and D3

```
/
└── combining-react-and-d3
    ├── .storybook ..... storybook config files
    ├── public ..... publicly served files
    ├── src ..... application source
    │   ├── assets ..... static assets
    │   ├── components ..... prototypes and generic app components
    │   ├── lib ..... utility functions
    │   ├── pages ..... application pages
    │   ├── stories ..... storybook stories
    │   ├── app.js ..... root application file
    │   ├── index.js ..... index file
    │   ├── routes.js ..... application route configuration
    │   └── serviceWorker.js ..... service worker configuration
    └── * ..... config files
```

A.4 Online Resources Snapshots

```
/
└── online-references
    ├── footnotes ..... snapshots of online references from footnotes
    └── ** ..... folder with resource number containing the associated snapshots
```

References

Literature

[1] J. E. Farrell, Brian L. Benson, and Carl R. Haynie. "Predicting flicker thresholds for video display terminals". In: *Proceedings of the System Insight Display*. Vol. 28. 1987, pp. 449–453 (cit. on p. 49).

[2] David M. Hoffman, Vasiliy I. Karasev, and Martin S. Banks. "Temporal presentation protocols in stereoscopic displays: Flicker visibility, perceived motion, and perceived depth". *Journal of the Society for Information Display* 19.3 (2011), pp. 271–297 (cit. on p. 49).

[3] JW. Lloyd. "Practical advantages of declarative programming". In: Conference Proceedings/Title of Journal: Joint Conference on Declarative Programming. 1994, pp. 3–17 (cit. on p. 12).

[4] Mary C. Potter et al. "Detecting meaning in RSVP at 13 ms per picture". *Attention, Perception, & Psychophysics* 76.2 (Feb. 2014), pp. 270–279 (cit. on p. 48).

[5] Terrence W. Pratt and Marvin V. Zelkowitz. *Programming Languages - Design and Implementation*. 4th ed. London: Prentice Hall, 2001 (cit. on p. 4).

[6] Paul Read and Mark-Paul Meyer. *Restoration of Motion Picture Film* -. 1st ed. Amsterdam: Elsevier, 2000 (cit. on p. 48).

[7] X. Wu and G. Zhai. "Temporal Psychovisual Modulation: A New Paradigm of Information Display [Exploratory DSP]". *IEEE Signal Processing Magazine* 30.1 (Jan. 2013), pp. 136–141 (cit. on p. 49).

Audio-visual media

[8] Facebook Developers. *Lin Clark - A Cartoon Intro to Fiber - React Conf 2017*. Youtube. Lin Clark talking about React and its fiber reconciliation algorithm. Mar. 2017. URL: https://www.youtube.com/watch?v=ZCuYPiUIONs (cit. on p. 49).

[9] TXJS. *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, 13).

Software

[10] Babel. *Babel Babel is a JavaScript compiler*. Version 7.4.0. URL: https://babeljs.io/ (visited on 05/02/2019) (cit. on p. 16).

[11] David Bau. *seeded random number generator for Javascript*. Version 3.0.1. URL: https://github.com/davidbau/seedrandom (visited on 05/30/2019) (cit. on p. 41).

[12] Mike Bostock. *D3 Data Driven Documents*. Version 5.9.2. 2011. URL: https://github.com/d3 (visited on 01/10/2019) (cit. on pp. 3, 4).

[13] 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).

[14] Steve Hall. *React Move Beautiful, data-driven animations for React*. Version 5.2.1. URL: https://github.com/react-tools/react-move (visited on 05/29/2019) (cit. on p. 36).

[15] Wojciech Maj. *React Lifecycle Methods diagram*. Version 1.0.0. URL: https://github.com/wojtekmaj/react-lifecycle-methods-diagram (visited on 05/10/2019) (cit. on p. 19).

[16] storybookjs. *UI component dev & test: React, Vue, Angular, React Native, Ember, Web Components & more!* Version 5.1.0. URL: https://github.com/storybookjs/storybook (visited on 06/03/2019) (cit. on p. 38).

[17] Uber. *UberVisForce d3-force graphs as React Components*. Version 0.3.1. URL: https://github.com/uber/react-vis-force (visited on 01/10/2019) (cit. on p. 31).

Online sources

[18] 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).

[19] Mike Bostock. *D3 Data Driven Documents*. Version 5.9.2. 2011. URL: https://d3js.org (visited on 01/10/2019) (cit. on p. 3).

[20] Mike Bostock. *Mike Bostock's Blocks*. URL: https://bl.ocks.org/mbostock (visited on 01/10/2019) (cit. on p. 4).

[21] Facebook. *React A JavaScript library for building user interfaces*. 2014. URL: https://reactjs.org (visited on 01/10/2019) (cit. on pp. 12, 13, 21).

[22] Mozilla Foundation. *Mozilla Request Animation Frame*. URL: https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame (visited on 01/25/2019) (cit. on p. 30).

[23] Google. *Chrome Request Animation Frame*. URL: https://developers.google.com/web/updates/2012/05/requestAnimationFrame-API-now-with-sub-millisecond-precision (visited on 01/25/2019) (cit. on p. 41).

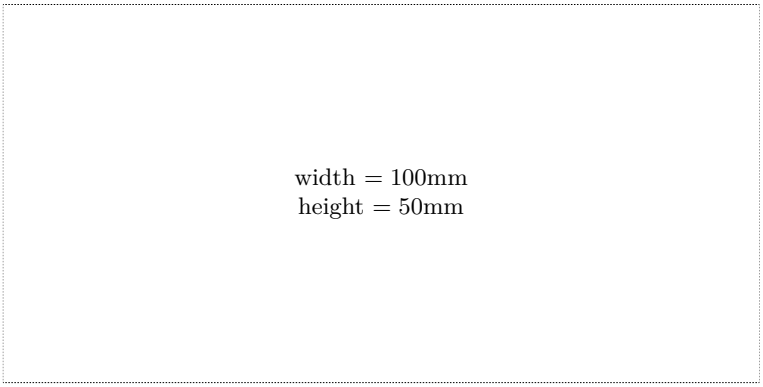
[24] Wojciech Maj. *React lifecycle methods diagram*. URL: http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/ (visited on 05/10/2019) (cit. on pp. 19, 22).

[25] StatCounter. *statcounter Browser Market Share Worldwide*. URL: http://gs.statcounter.com/ (visited on 06/07/2019) (cit. on p. 45).

[26] W3C. *W3C Browser & Platform Market Share*. URL: https://www.w3counter.com/globalstats.php (visited on 06/07/2019) (cit. on p. 45).

Check Final Print Size

— Check final print size! —



— Remove this page after printing! —