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

Maximilian Zauner

MASTERARBEIT

eingereicht am

Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im Juli 2019

Declaration

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

Hagenberg, July 12, 2019

Maximilian Zauner

Contents

Abstract

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

Chapter 1

Introduction

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

1.1 Problem description and motivation

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

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

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

1.2 Goals of the project

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

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

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

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

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

Chapter 2

D3.js – Data-Driven Documents

Chapter 2 provides an overview of the popular JavaScript library D3 which is used to simplify implementations of any data visualizations in the web for developers. It not only provides an overview of the libraries beginnings but also goes into detail about how to implement projects with D3. The knowledge is required to understand the performance comparisons in chapter 5 and 7.

2.1 Introduction to D3

D3 is a JavaScript library that helps developers create highly sophisticated data visualizations on the web via a universal tool that is platform agnostic: the browser. The official documentation of D3 in [13] explains the library as a toolkit, that allows binding data to the DOM. Also, it gives an overview of the vast amount of helpful tools that can be used to visualize data. The library includes all kinds of functionality, ranging from simple array and mathematic operations to complex simulations, that are calculated in real time.

Some of the most popular features of D3 is to render user interactable animated charts. Not only is it possible to easily create a bar chart for instance, but all other kinds of charts as well. A full list of available packages can be found in [6]. 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 [6, /d3-scale-chromatic]. Another example in [6, /d3-array] shows, that the library also provides some useful array manipulating functions which come in handy when having to deal with big data sets. Also generating random numbers via various distributions is no problem when using the d3-random package in [6, /d3-random]. The list of useful data manipulation tools goes on and dealing with every aspect of the library would go far beyond the scope of this paper.

What makes D3 unique though is the possibility to create individual data structures for rendering sophisticated data visualizations. The library provides scatter plots in [6, /d3-scale] or pie charts, line charts, area charts, radial bar charts, tree maps in [6, /d3-shape] to name a few. D3 also provides utility functions to add labels or user interaction to each mentioned and not mentioned data visualization type. Also, a significant advantage of using D3 is, that it also provides simple methods to transform any D3 visualization into being user interactable by creating floating tooltips or sliders, switches, or knobs, which control the visualization.

Due to the immense size of the library and its many data manipulation tools, D3 is divided into different sub-modules to prevent users of the library having to download the full library code bundle in the browser to be able to use the library. [6] shows a full list of every available tool that can be used in composition with the base package of D3. When using D3 in a big production project, all modules can be integrated into any project via using nodes package manager npm and download it from its registry[1].

There are multiple examples on the documentation's example page in [14] 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 [6, /d3/blob/master/API.md].

2.2 Explaining the D3 API

This section aims to discuss the most vital aspects of the D3's API to understand code samples, that are presented in later chapters. Some general knowledge of D3's API is of utmost importance as the knowledge is crucial for understanding the comparisons of React and D3 in chapter 4. As mentioned before, the D3 API mostly consists of consecutive chained imperative function calls, that manipulate the visualization data and binds it to the DOM. [2, P. 625] describes the imperative programming pattern as a static division of a program into its concurrent tasks which means, that the programmer uses statements to change the programs state.

According to the documentation in [6] the library D3 was created in 2010. Thus it can be noticed, that the libraries API originates from a time, where developers did not even think about using JavaScript in productive or even enterprise environments. Therefore large codebases written with D3 tend to be hardly scalable and difficult to maintain. Multiple instruction function calls in 2.1 show, that D3 code is indeed imperative. Also, the library makes use of a software pattern called "chaining". The pattern works because each function returns an instance of itself to enable the addition of an infinite amount of functions that can be added to the chain.

Selecting DOM nodes and creating a D3 selection model is, therefore, a vital aspect of D3's API. Via selection D3 can connect JavaScript application data to actual DOM nodes as [6, /d3-selection] shows. An example can be seen in 2.1. Because the library is imperative, each node that is added or removed is handled via a chained function call as the append function in 2.1 shows very well. When adding or removing multiple DOM

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

Program 2.1: D3 selection, enter, and exit example

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

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

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

When the data of a visualization changes, some nodes might be deleted, some new nodes might appear but some nodes might also stay in the visualization but change their position. D3 covers that use-cases by including the possibility to add transitions to node selections. The transition feature lets developers specify how to handle DOM elements that stay in the visualization if the data is updated. Advanced animations and transitions can be added via a simple function call. Line 8 in 2.1 for example shows a selection where all nodes are selected that are removed after the data has changed. Furthermore the color is changed and a transition is added, which transforms the radius attribute of the node until it reaches the specified amount, 1e-6 in this case. Finally the

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

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

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

The code in 2.1 also clearly shows, that every attribute and style instruction of and added DOM nodes have to be handled via a chained function call. On line 23 in 2.1 the `fill` property is added to the `<circle>` SVG element. Each additional style property would require a consecutive call of the `style('property', [style])` function.

Also, a key component of D3's API is the possibility to pass attribute handling functions instead of hardcoded values. Those computed properties can be found in line 24 and 25 in 2.1. By passing a function to property or attribute setters like the `.style('attribute', [style])` method, the passed handler is called by D3 as a callback by providing each node's data to the callback. When rendering 10 nodes, the attribute function on line

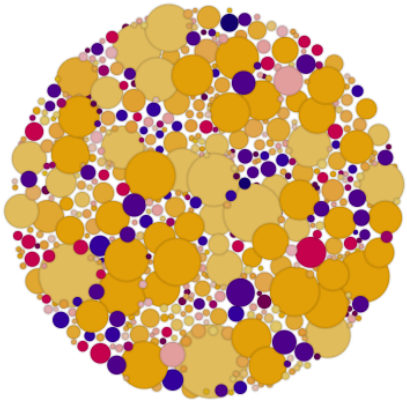


Figure 2.1: The force graph with default center force. All nodes are attracted to the same center without overlapping each other.

24 in 2.1 which sets the radius of the node would, therefore, be called 10 times as well, setting the radius for each specific node individually.

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

Over the years better software patterns emerged and experience shows, that chaining is a software pattern that was new at the time but can cause code that is hard to maintain. Nowadays this library would probably be written with a functional approach, letting developers compose their simulations via functional composition as various libraries and frameworks on the internet already do. The problem with the example 2.1 is that the code cannot be reused, as it is hardcoded into the chain. Nowadays many frameworks for building frontend applications use a declarative approach for binding the view to the data model. More information about declarative approaches can be found in chapter 3. D3's API still uses the imperative software pattern which forces developers to chain library function statements to control multiple elements in the DOM.

2.3 Force Graphs – Real time rendered data visualizations

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

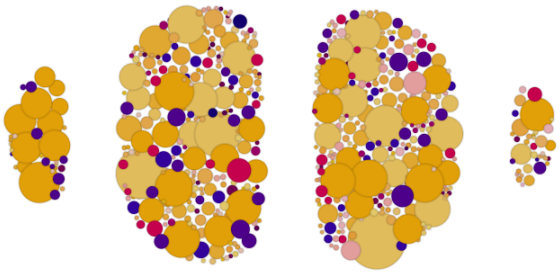


Figure 2.2: A sample force graph with more than one center force. Having more than one force centers means that different nodes are attracted to their assigned center force.

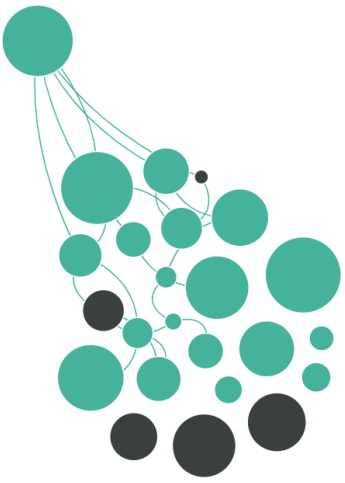


Figure 2.3: A sample force graph where the top node is dragged up to the left and the other nodes are dragged along. The force is still keeping the other nodes apart and also drawn to the center though.

each other even further than their actual size as figure 2.4 shows. It is also possible to implement so-called links, that also add some complexity to the simulation, as nodes are dependent on each other and not only reject each other but also attract linked nodes as figure 2.2 and 2.5 shows.

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

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

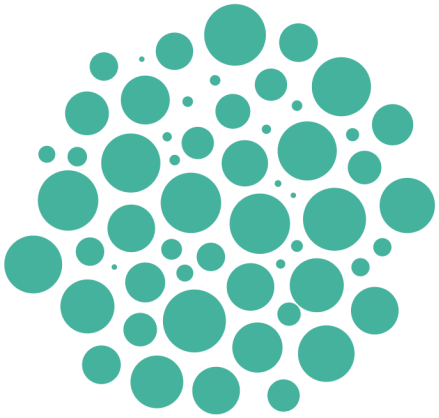


Figure 2.4: A sample force graph with one center force. The nodes are configured to reject each other with the function `r+r/2`

Program 2.3: Code snippets for D3 force simulation code

```
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
```

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)
```

in the browser as it can be read in [6, /d3-force/blob/master/README.md]. The way force simulations work is that developers first have to define or build the simulation. There is a factory method as seen in line 1 of 2.3 which takes the nodes of the graph as an argument and builds a default simulation. The nodes have to be provided in a particular scheme so D3 can correctly parse the node array.

Another very important aspect of force graphs is the so called ”alpha” value system as documented in [6, /d3-force/blob/master/README.md], which controls how long the simulation lives. The alpha valueis a gradually decaying value that makes the simulation stop if a certain value is reached. Every simulationhas a function that is called every

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

”tick” of the simulation as shown in line 2 of 2.3. Everytick the alpha value decays via a predefinable function, it happens logarithmically per default. The tickingfunction takes a handling function will be passed every node position in the simulation which then letsdevelopers link the data to the DOM with D3 again. The tick function will be very important lateron when thecombination of D3 and React is explained in more detail. 2.4 is a simple example that shows how to initialize a D3 force simulation.

If there is a user interaction, the simulation sometimes has to be restarted or reheated. Programmers can set alpha values and targets to reheat or restart the simulation in case a node is dragged by the user which would possibly require many other nodes in the simulation to react to that user input. That way also the speed of the simulation can be controlled via setting a custom decay function. The documentation in [6, /d3-force/blob/master/README.md] points to a few methods that can achieve said functionality. The functions on line 3, 4, and 5 of 2.3 can be used to reheat a simulation. Also the functions in line 6 and 7 of 2.3 can be used to set values directly to alter the simulation’s life span.

If a functional approach would be used the code from 2.4 would look more like in example 2.5. The difference between the two code examples in 2.4 and 2.5 might appear to be very subtle, but in reality, it is very significant. By looking closer, it is clear, that the variable `forceParams` is a functional composition of methods, that can be reused multiple times in the application. In the first example in 2.4 the simulation configuration is locked in the function chain. Chaining is a pattern which can easily cause duplicated code in any codebase.

Writing functional D3 code could also alleviate the confusing unmaintainable code in 2.2 as developers can easily compose repeating DOM manipulation sequences and reuse them throughout the codebase without having to touch code on multiple files in case there is a bug that effects multiple aspects of the application.

Program 2.5: D3 written in a fictional functional way

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

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

Chapter 3

React – A JavaScript library for building user interfaces

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

3.1 Introduction to ReactJS

The easiest way to find information about React is to visit its official website [1]. There is a statement in [15] up front that says:

> React is a JavaScript library for building user interfaces.

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

According to [15], Facebook sees React as a declarative and component-based library. However, a question might come to mind: "What exactly does it mean for a library to be declarative and component-based?". The answer to this question might be more straightforward than initially anticipated. In [1] declarative programming is described

[1] https://reactjs.org

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

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

React components can have multiple applications. There are presentational components, for example, which are pure functions that represent the current application state. Though there are also stateful Components which can hold some application state and react to state changes accordingly via rendering again, react however makes no assumptions about the technology stack that is used in a project as [15] claims. This means that users of the library can decide for themselves if they want to use the built-in state management functionality or if they want to use a third-party library for solving specific problems like global application state for example.

The documentation in [15, /docs] claims that the library makes use of a so-called "virtual DOM". This means that React keeps track of its state data to prevent unnecessary writes to the actual DOM object. JavaScript performs exceptionally well when handling pure JavaScript objects in memory. Keeping the DOM tree of the application in the JavaScript engine's heap as a representation of objects enables React primarily apply updates this so-called virtual DOM. React then compares the newly applied data with the old tree to then being able to decide updates need to be committed to the DOM. Writing or committing to the DOM is the most expensive type of work in the browser, so React tries to keep DOM manipulating actions to a minimum. The React team calls the diffing algorithm "reconciliation algorithm". It would go out of the scope of this paper to go more in depth of the algorithm, so it is recommended to read about React's reconciliation algorithm in its documentation [15, /docs].

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

[3, 6:50], the speaker describes the functionality very well via explaining React as a simplified function that could look like this: `f(data) = UI`. Hence, React can be seen as the view layer that handles presentation as a function of state and data. Once the data has updated the virtual dom, the virtual dom is then passed to React's reconciliation algorithm, which then determines if any nodes have to be changed on the real DOM. If there would be a React component that always renders the same `<div>` with the same data, rendering that very component multiple times would not result in React writing multiple DOM nodes to the browser. The reconciliation algorithm sees that the virtual dom matches the real dom in this case, which results in React not updating the real DOM. Of course, if the component's content is dynamic, the component sometimes has to be re-rendered according to the data changes. If some parts of the data stay the same even after being reapplied to a component, only newly added or removed nodes are committed to the DOM. Even though the reconciliation algorithm prevents expensive DOM operations, the algorithm itself can also be expensive. The documentation in [15, /docs/optimizing-performance.html#avoid-reconciliation] advises developers to try to avoid reconciliation to improve performance.

Unidirectional data-flow implicitly means to React developers, that there is no data binding and no template language. The library only uses `React.createElement([element])` calls internally, which are hidden behind the so-called "JSX" JavaScript language extension. JSX will be explained more i depth in section 3.2. As mentioned before, React is just a pure idempotent function of its application state, which means that the same data always produces the same presentation. That fact also implies that if the application data has to be changed, a new "patched" version of the application data has to be created instead of mutating the currently available application state. The newly created data then flows into the React render cycle again. Unidirectional data-flow is also the reason why React works well with immutable data structures. This paper assumes that the reader knows about immutable data structures, but [7] explains exceptionally well, what Immutable data structures are and how they're used in JavaScript. Going more in-depth on how React works well with immutable state would go out of the scope of this thesis though. It is just essential to know that every time the data changes, React triggers a whole new render cycle of the component tree. The immutable data structures help React to work out changes in the data structure when using immutable data structures. Instead of having to implement recursive data comparison functions, nested data object tree differences can be checked via a cheap equality check.

3.2 Explaining the React API

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

Program 3.1: Creating a React element with JSX

```
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 );
```

3.2.1 JSX in general

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

Notice, how the createElement function takes up to 3 parameters as documented in [15, /docs/react-api.html]. The first parameter is the element type (the type can also be a custom component that was created by a user or a downloaded third-party component). The second parameter is used to pass the element's current properties, and the third parameter describes the component's children. That third parameter makes it possible to compose multiple React components together, as children are nestable.

Line 2 and 3 in 3.2 show, how a React element is created. A node of type `"div"` is created and the property `{className: "helo-world"}` is passed. Each parameter after Line 3 is a child of the created `<div>` node. The React element has 3 children which is demonstrated by the code example where the element is written in JSX in 3.1. First, there is the string "Hello ", then there is a `<span>` which also has children, and finally there is the exclamation mark string at the end. When going back to the transpiled code example in 3.2, lines 4 to 10 exactly show what kind of children are passed to React's element creating function. Notice, that the class property has to be "className" in JSX

instead of "class" because JSX is *not* HTML, but extended JavaScript. Something also worth looking at is line 5 in 3.2. A nested `createElement()` call shows, how components can be composed together.

Because JSX is a language extension, a transpiler step is needed to produce production code. The common tool to use is called "Babel". There is a caption in [4] that says:

> Use next-generation JavaScript today.

The documentation in [4, /docs/en] explains, how modern JavaScript features can be used in any JavaScript project. The code which includes those modern features is normalized and transpiled by Babel to also work in older browsers. The tool accomplishes this by transforming the JavaScript code via its core implementation but also via some third-party plugins. A babel plugin has been created to transform JSX components into the syntax that can be seen in the code in 3.2. Just as a side note, although JSX became popular in conjunction with React, there are also other web technologies that make use of JSX like Vue.js [2] for example.

3.2.2 Explaining React components

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

Functional stateless components

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

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

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

[2] https://vuejs.org/

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

```
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);
```

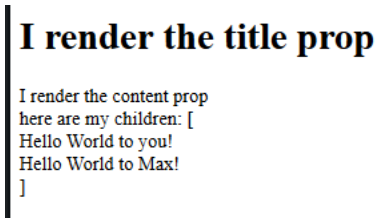


Figure 3.1: React hello world sample output

One of the most important aspects of React is that props – once they're passed to a component – are static and immutable inside the receiving component as React's documentation in [15, /docs/components-and-props.html#props-are-read-only] demonstrates. Components that only render props and don't manage their own application state can be seen as pure functions which render the exact data they receive every ren-

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

```
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);
```

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

Stateful components

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

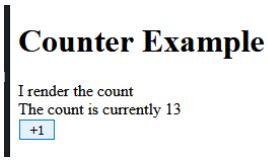


Figure 3.2: React counter component output

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

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

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

3.3 React's component lifecycle

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

React's documentation in [15, /docs/react-component.html] includes a comprehensive guide on component lifecycle methods. By overriding the lifecycle methods inside their components, developers can add their specific logic to each lifecycle in every render cycle of the components. Overriding lifecycle methods is optional; it is, therefore, possible to not implement any lifecycle method at all. Notice, how class components always have to override the `render()` function to being able to even render content. The render function is called every time the component goes through a new render cycle.

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

```
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);
```

A good visualization of the full lifecycle of a React component can be found in the GitHub repo in [9]. The web application in [17] visualizes the separation of the different phases a component goes through when iterating through its lifecycle methods. Figure 3.3 shows a screenshot of the application for the sake of being able to demonstrate the diagram in the thesis. The diagram is based off a tweet in [12] from Dan Abramov, one of the core contributors to the React library.

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

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

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

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

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

The other lifecycle methods in the code example in 3.3 are pretty self explanatory. There are a few best practices though, according to the documentation in [15]. The

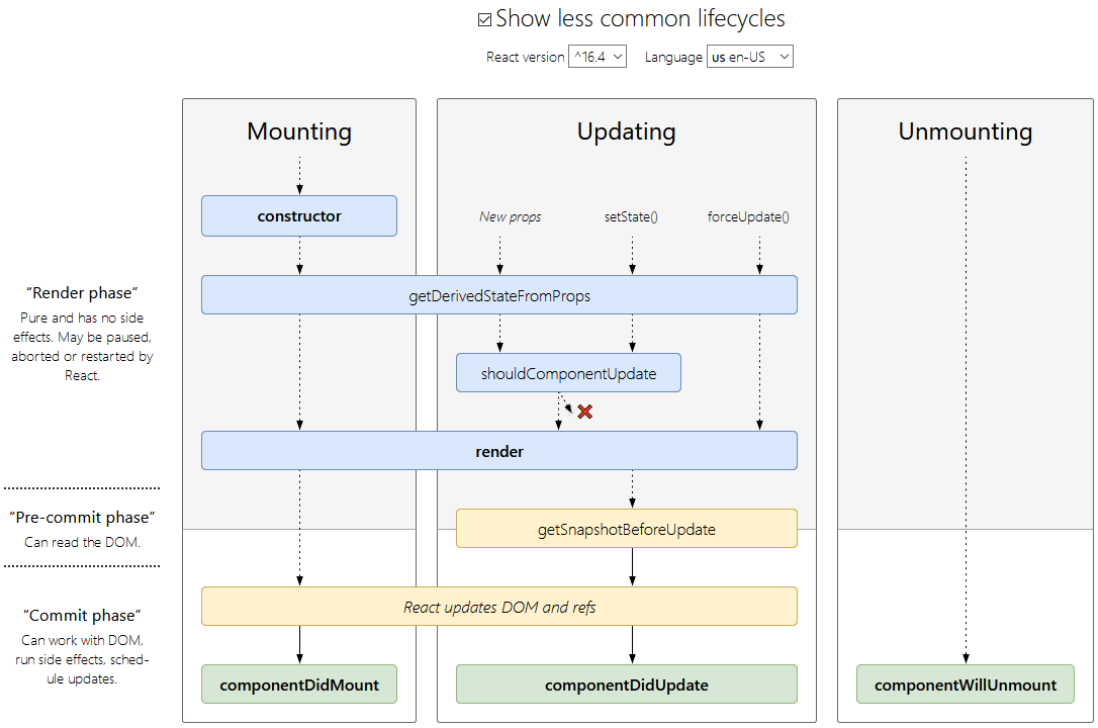


Figure 3.3: React lifecycle methods diagram taken from [17]

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

3.4 Conclusion

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

Chapter 4

Data Visualization with 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 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 3 viable prototypes that are described in this chapter.

4.1.1 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 pure D3 force implementations, which 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 vanilla D3 implementation was not the primary goal of the project.

The obvious question "Why do I even need a combination of D3 and React if I could just use pure 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 are animated in the browser. Instead of having to implement the visualization part of the application in pure D3, the combination of React and D3 enables all developers of the project to write declarative D3 code. Using a combination layer API would then result in only one code base that

has to be maintained, instead of two. Also, as mentioned before, D3 tends to become exponentially more challenging to maintain as the project grows.

4.1.2 Project setup

To be able to test multiple prototypes, one crucial aspect of the project is the force simulation builder setup. All prototypes are built on top of the simulation builder module. The module contains a few D3 specific methods that allow developers to build their own individual force simulation prototypes. All prototypes can reuse some of the already existing 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 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 3 prototypes are composed together by using smaller force simulation function blocks.

The most important function in the module of course is the `buildForceSimulation()` method in line 42 in 4.1. 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 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 if the data changes.

Another interesting aspect of the code snippet in 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 `applyForceHandlers` composed method 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 3 prototypes.

Of course, since all 3 prototypes work fundamentally different, some custom apply functions have to be implemented to ensure the functionality of all 3 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 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 each 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 }
}
```

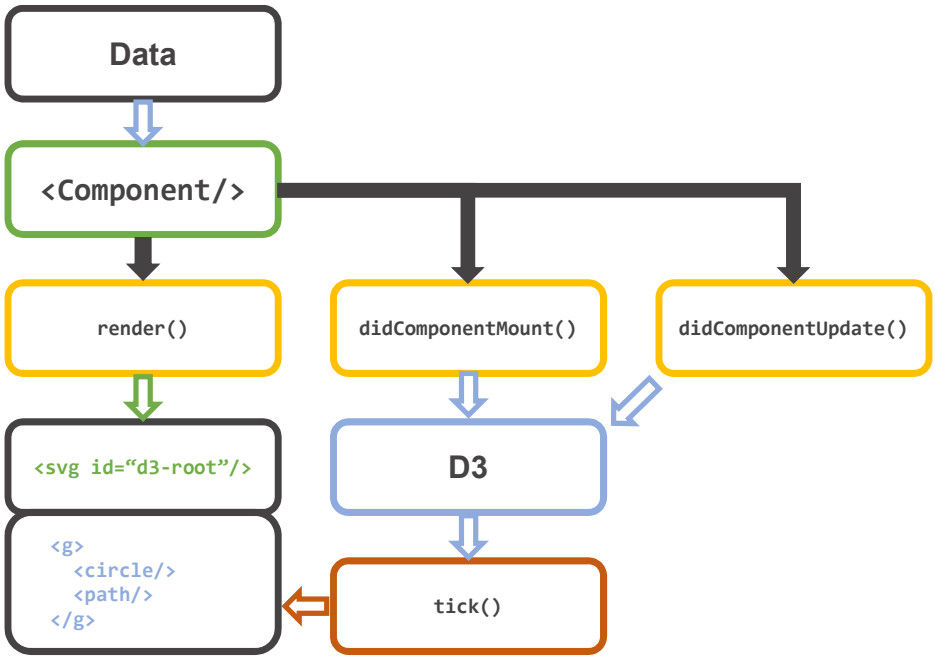


Figure 4.1: Pure D3 force graph lifecycle visualization

4.1.3 Pure D3 prototype

The first prototype of the project was developed from an 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 each render cycle. If the parent component updated the visualization component's data or options, the prototype would have to reflect the changes as well instantly. 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 only renders a static base SVG, as shown in the code snippet in 4.2. After the component mounts, D3 hooks into the base SVG component via the provided ID and builds its force simulation 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.

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 }
```

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 doesn't 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 lifecycle method `componentDidUpdate()` as figure 4.1 shows. Of course, each 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 is static. 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 on any other web project as well. Developers not only have to implement the initial force graph generation functionality but also the update logic that handles the updated data and applies it to the force graph simulation.

There are 2 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 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 then handle the update in the force graph.

Implementation details

Once the component is initialized, the `componentDidMount()` lifecycle method directly calls the initializer function which can be seen on line 1 in the code snippet in 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 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.

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 }
```

What is also worth mentioning is the fact, that the simulation and the updating function are saved directly to the `this` context as seen in lines 17 and 18 of the code snippet in 4.3. As stateful components are just plain JavaScript classes, they're 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 wanted 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 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 has 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 4.4 and composes it directly into the updating function as seen in line 14 in 4.1.

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

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

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),
  )
}
```

to a native D3 implementation. In chapter 5, the performance is compared to other implementations as well.

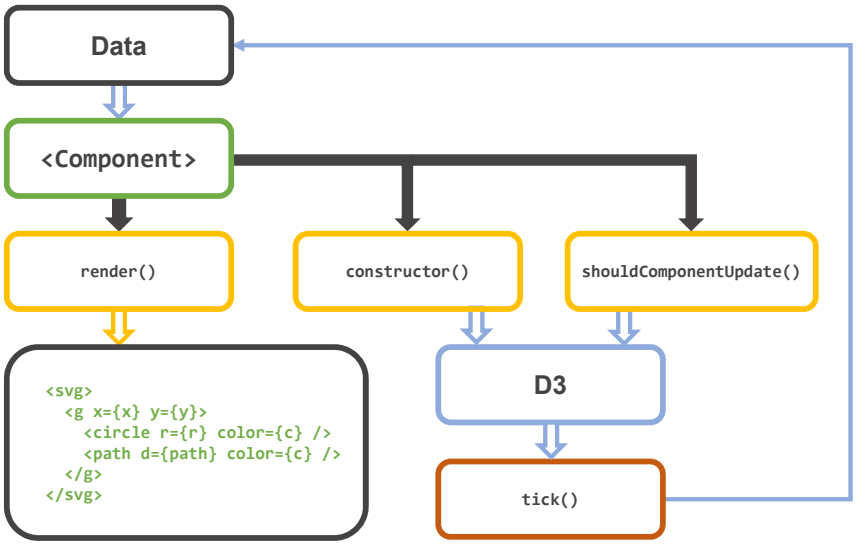


Figure 4.2: Pure React force graph lifecycle visualization

Disadvantages

The most significant disadvantage with 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 cannot be customized via 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.

4.1.4 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 though. 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 in [6, /d3-timer/blob/master/README.md]. 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 [16].

As figure 4.2 shows, the complete presentation layer is handled by React itself. The whole SVG component tree is therefore completely rendered by React. The figure also demonstrates how the data is handled not only in the component's constructor but also in its

`shouldComponentUpdate()` lifecycle method. As mentioned in the previous chapter 3, the constructor is the first lifecycle 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()` lifecycle 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 pass 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 again 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, can massively impact performance as described later in chapter 5.

An existing project of Uber strongly inspired the pure React implementation. Uber's project is called vis-force and can be found on its git page in [11]. 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 4.3. The pure React prototype passes the correct according 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()` lifecycle 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()` lifecycle 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 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're passed in from the parent container. However, the link and node positions come from the components internal state. The example code in 4.6

Program 4.6: Render lifecycle 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>
  )
}
```

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 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 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 lifecycle method `shouldComponentUpdate()` is considered an anti-pattern according to React's documentation in [15, /docs/react-component.html], as side-effects should always be handled inside the `componentDidUpdate()` lifecycle method. The force simulation updater function is a so-called "side effect". Facebook strongly advises against using any side effects in the `shouldComponentUpdate()` lifecycle 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.

Another interesting aspect of the pure React prototype implementation is the fact,

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 }
```

that the tick handler function does not manipulate the dom directly as the pure D3 example does. Instead, the code example in 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 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. 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 completely handle not only properties like filling color or stroke properties but also the position.

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 would offer a custom rendering function, users of the component could write their rendering functions with React code to achieve a customized version of the force graph. Chapter 6 is all about how the thesis project can be open sourced.

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 link positions in the internal component state but also updates it every fraction of a second as a consequence more CPU cycles are lost by calculating not only the D3 tick cycles but also complete React render cycles. 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 recommit 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 D3 functionalities like dragging nodes or zooming in and out of the graph have to be implemented from scratch in pure React as well which can be a tedious task. Due to the component being rendered multiple times a second, adding drag handlers that also perform well is a very tedious task, for example.

4.1.5 D3 and React hybrid

Last but not least, there is the D3 and React hybrid force graph component implementation. The prototype not only provides the developing experience of React 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 lifecycle 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 from the

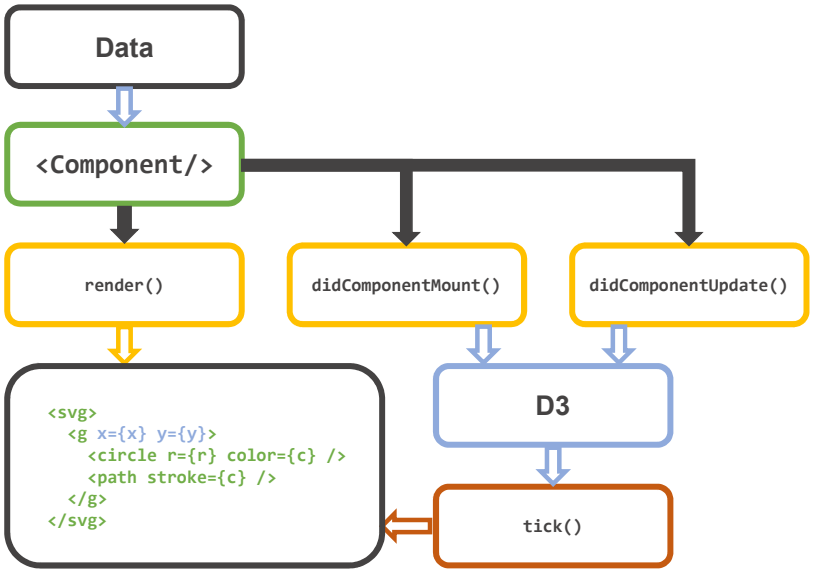


Figure 4.3: Pure React force graph lifecycle visualization

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. Again, figure 3.3 visualizes all of React's lifecycle phases.

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 4.3. The only difference also 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()` lifecycle method, not in the component's constructor. As described before in the introduction of subsection 4.1.5, the component needs to be in the commit phase for D3

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

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

being able to hook into already committed DOM nodes.

As described in the subsection 4.1.2, the `buildForceSimulation()` function is very important 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 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()` lifecycle 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 the code example in 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're 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 [8] – 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 1 but not in render cycle 2, 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

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)
}
```

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 [8].

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 quite sure 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.

4.1.6 Comparison of the different proposed Prototypes

When comparing the prototypes, an essential aspect is to understand, which technology renders what part of the DOM. 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 different 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 its 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 standard 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 2 lifecycle methods, it is of utmost importance to recall in which component lifecycle phase the methods are called though. Figure 3.3 can help to look up all common lifecycle 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.

4.1.7 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 [10] 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 project 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.

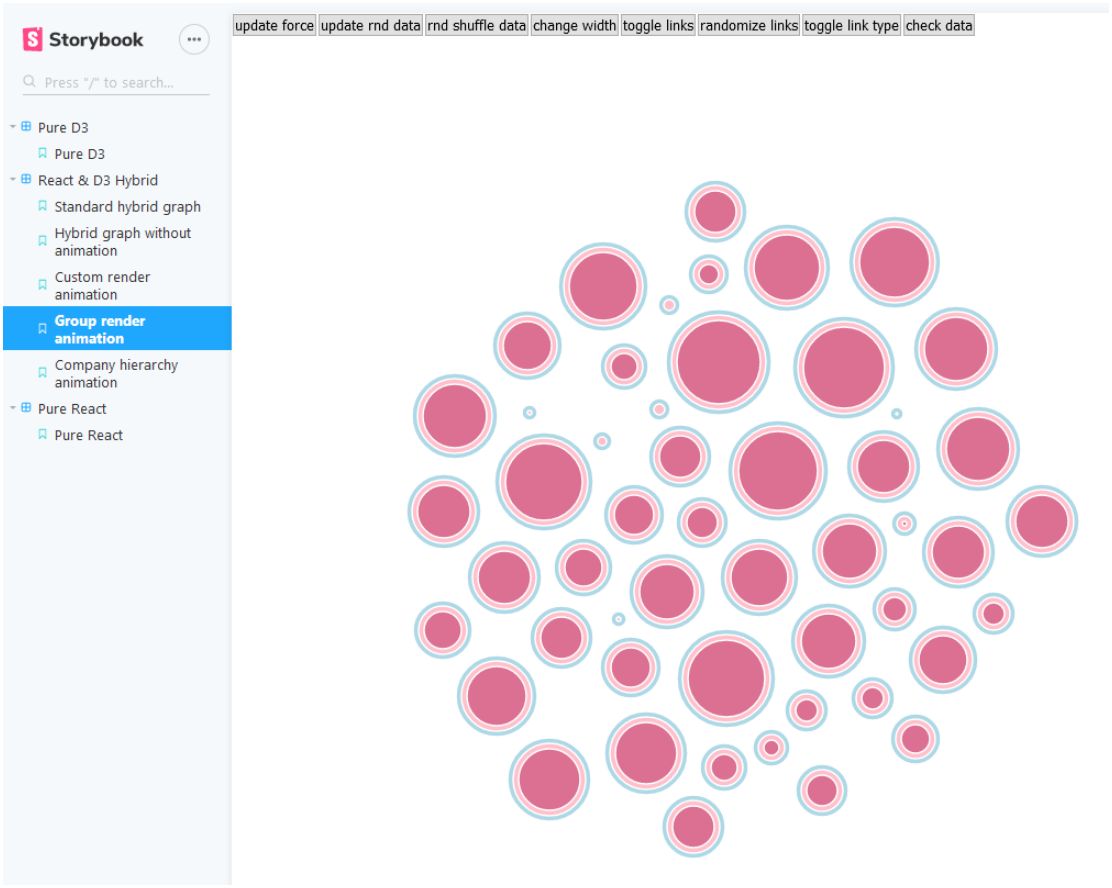


Figure 4.4: React storybook currently showing the hybrid prototype

4.1.8 Conclusion

All in all, each prototype has an interesting implementation on its own. Also, 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. More information about the plans of making the hybrid prototype open source can be read in chapter 6.

4.2 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.

4.2.1 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 research 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 produce 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 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 browser, for example, is smoothing out the performance by trying to executing animation frames regularly, which means that the overall performance might be lowered to achieve a smoother framerate. Experience shows that Firefox, 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.

4.2.2 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

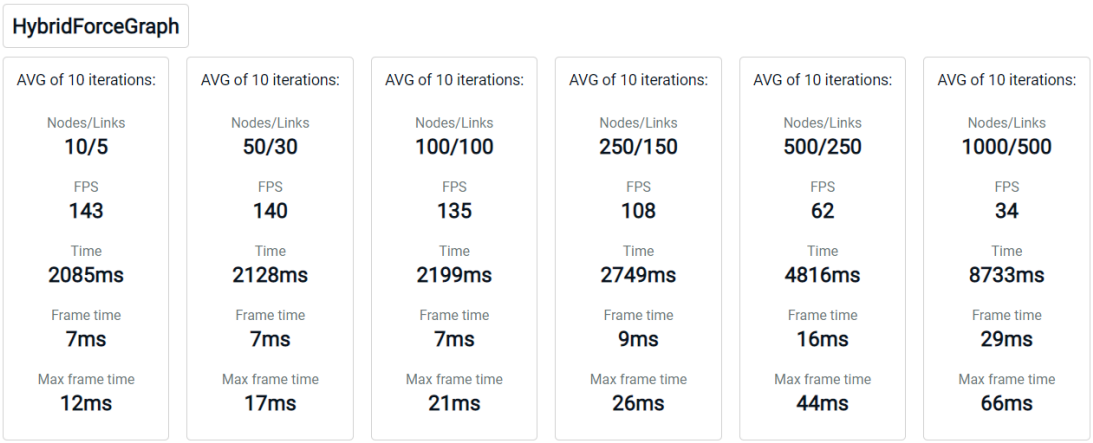


Figure 4.5: Hybrid prototype benchmark results

Figure 4.6: Overview of the benchmark application

consistent pseudo-random results. The library *seedrandom* in [5] is the perfect technology 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 cycle per test data iterations sets 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. It is, therefore, 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-random 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.

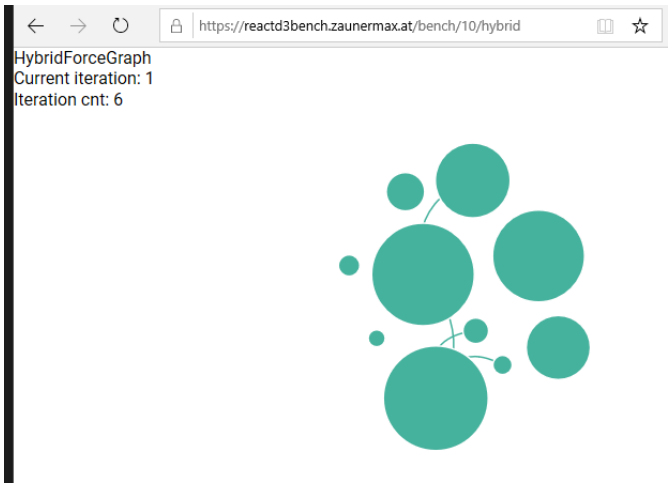


Figure 4.7: The benchmark currently iterating through hybrid prototype iterations

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 seconds which is equal to the frames per second the browser produces.

Presenting the performance results mustn't 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 6 test iterations, 6 containers are rendered after the benchmark. The containers contain the number of cycle 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 4.5 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 4.6. The React application is a wrapper 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 4.6 and 4.7 show, how the URL contains all relevant benchmark parameters. That way the benchmark URLs can be pasted into any browser to execute the benchmark.

Chapter 5

Performance Testing and user perception

5.1 Testing setup

5.1.1 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 10 devices is enough to retrieve scientifically significant results. The range of devices is divided into 2 sections: high-end devices and low-end devices.

The mobile devices used for testing are a OnePlus 1 phone, a OnePlus 5t phone, a Samsung Galaxy S8 phone, a SurfaceBook in tablet mode and a Samsung Galaxy Tab. 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.

The table 5.2 shows all high-end devices which have a monitor refresh rate of 144-hertz. It must be noted, that there are several custom tower builds with custom specs and one 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.

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
Samsung G[1] S8	Exynos 8895	Mali-G71 MP20	4GB
Samsung GT[2] A10.1	Exynos 7870	Mali-T830 MP2	3GB
SurfaceBook	Intel i5-6300U	Intel HD 520	8GB

[1] Galaxy [2] Galaxy Tab

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 (Max J.)	Intel i7-7700k	Nvidia GTX 1070	16GB
Tower (Patrick M.)	Intel i7-6700k	Nvidia GTX 1080	3GB
Tower (Julian J.)	Intel i7-7700k	Nvidia GTX 1070	16GB

5.1.2 Testing methodologies

Running the benchmarks to get valid and scientific 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 on 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 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. All test results can also be normalized by showing a percentage of how much the performance changed across all iterations.

Each device runs through 6 iterations with exponentially increasing render difficulty. Each iteration runs through 10 cycles which equals a grand total of 60 cycles per browser and device combination. Five high-end devices with 144-hertz monitors and the five 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're the most significantly used brwosers worldwide according to various statistics.

5.2 Introduction of Human Perception of fluid animations

5.3 Introducing the test results

5.4 Interpreting the test results

5.5 Summary of the different prototypes

Chapter 6

Open source and the React community

6.1 Building an open source library for React.js developers

6.2 Designing a library API

Chapter 7

Conclusion

all in all big success, as i found an implementation that combines d3 and react and is super performant. Also open sourcing the thing is really cool as other developers can benefit from using the force component by coding their nodes in react as well.

References

Literature

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

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

Audio-visual media

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

Software

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

[5] 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).

[6] Mike Bostock. *D3 Data Driven Documents*. Version 5. 2011. URL: https://github.com/d3 (visited on 01/10/2019) (cit. on pp. 3, 4, 9, 10, 30).

[7] 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).

[8] 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 pp. 36, 37).

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

[10] 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).

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

Online sources

[12] 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).

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

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

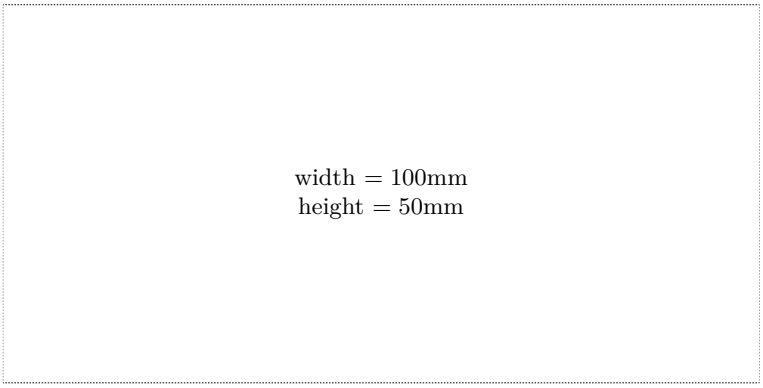
[15] Facebook. *React A JavaScript library for building user interfaces*. 2014. URL: https://reactjs.org (visited on 01/10/2019) (cit. on pp. 12–15, 17, 19, 21, 32).

[16] 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).

[17] Wojciech Maj. *React lifecycle methods diagram*. Version 1. URL: http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/ (visited on 05/10/2019) (cit. on pp. 21, 22).

Check Final Print Size

— Check final print size! —



— Remove this page after printing! —