Performance Evaluation of JavaScript Rendering Frameworks

Adam Lindberg

Supervisor: Anders Fröberg
Examiner: Erik Berglund

External supervisor: Gustav Rosén
Upphovsrätt

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnt som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida

http://www.ep.liu.se/

Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/her own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page:

http://www.ep.liu.se/

© Adam Lindberg
Abstract

When developing interactive web applications a number of different technologies and frameworks could be used. This thesis is set to evaluate a number of popular frameworks that are using different native web rendering techniques. More specifically, the goal of this study is to find what JavaScript visualization framework is best suited for developing a visualization module capable of handling up to 1000 continuously moving nodes with retained frame rate. In this case, retained frame rate refers to keeping the average frame rate above 20 FPS. The frameworks investigated in this study are D3.js using SVG and Canvas, and PixiJS using WebGL 2D rendering. The evaluation was conducted by first developing a visualization module containing a force-directed graph. This was done three times over, once with each rendering technique. Next, the average frame rate was measured during the first 10 seconds of loading a fixed size data set. Data sets of increasing volume were then loaded to examine how the different modules handle data sets of various sizes. The results showed that the SVG module was far behind the other two in terms of retained FPS on larger data sets. The Canvas and WebGL modules were closer in the level of performance, where WebGL outperformed the Canvas implementation in the base case. However, when a Gaussian blur filter was activated in both modules, the Canvas module prevailed. This blur filter was a requested feature for the final product, which led to the choice of using D3.js with Canvas rendering for further development.
Acknowledgments

I would like to thank Entiros Integrations for the opportunity to carry out this study as a member of their development team. It has been a challenging and fun time, but I have learned a lot from taking part in the full development process, from the daily meetings to discussing requirements and design decisions. A special thanks to my supervisor at Entiros, Gustav Rosén, who has been very engaged in my work and always pushed me forward. Lastly, I would like to thank Anders Fröberg, my supervisor at Linköping university, for some great discussions and tips on how to write a good thesis.
## Contents

**Abstract** iii

**Acknowledgments** iv

**Contents** v

**List of Figures** vii

**List of Tables** viii

1 Introduction 1
   1.1 Aim . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
   1.2 Research Question . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
   1.3 Delimitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

2 Background 3

3 Theory 4
   3.1 Web Applications . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
      3.1.1 Rich Internet Application . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
      3.1.2 Single-Page Application . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
   3.2 HTML . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
   3.3 Web Animation Technologies . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
      3.3.1 Scalable Vector Graphics . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
      3.3.2 Canvas . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
      3.3.3 WebGL . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
   3.4 Web Animation Frameworks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
      3.4.1 D3.js - Data-Driven Documents . . . . . . . . . . . . . . . . . . . . . . . . 7
      3.4.2 PixiJS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
   3.5 Force-Directed Graph . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
   3.6 Frame Rate . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

4 Method 9
   4.1 Pre-study . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
      4.1.1 Requirements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
      4.1.2 Frameworks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
   4.2 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
   4.3 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

5 Pre-study and Implementation 12
   5.1 Pre-study . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
      5.1.1 Requirements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
      5.1.2 Frameworks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2 Implementation</td>
<td>14</td>
</tr>
<tr>
<td>5.2.1 Data Set Generator</td>
<td>14</td>
</tr>
<tr>
<td>5.2.2 D3.js - SVG</td>
<td>15</td>
</tr>
<tr>
<td>5.2.3 D3.js - Canvas</td>
<td>18</td>
</tr>
<tr>
<td>5.2.4 PixiJS - WebGL</td>
<td>20</td>
</tr>
<tr>
<td>6 Results</td>
<td>22</td>
</tr>
<tr>
<td>6.1 Evaluation</td>
<td>22</td>
</tr>
<tr>
<td>7 Discussion</td>
<td>25</td>
</tr>
<tr>
<td>7.1 Results</td>
<td>25</td>
</tr>
<tr>
<td>7.2 Method</td>
<td>26</td>
</tr>
<tr>
<td>7.3 The Work in a Wider Context</td>
<td>26</td>
</tr>
<tr>
<td>8 Conclusion</td>
<td>27</td>
</tr>
<tr>
<td>8.1 Future Work</td>
<td>28</td>
</tr>
<tr>
<td>Bibliography</td>
<td>29</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Starlify object model. ................................................................. 3
5.1 A snapshot of the data set generator. ........................................ 14
5.2 Calculation of graph center x- and y-position. ............................... 15
5.3 Calculation of graph center k-value. ........................................... 16
5.4 Transition to center of graph and zoom to cover 80 percent of web browser. ......................................................... 16
5.5 SVG’s Gaussian blur filter. ............................................................ 16
5.6 A snapshot of the visualization module developed in D3.js using SVG. ................................................................. 17
5.7 Canvas custom blur. ................................................................. 18
5.8 A snapshot of the visualization module developed in D3.js using HTML5 Canvas. ......................................................... 19
5.9 Pixi gaussian blur. ................................................................. 20
5.10 Pixi centering function. ............................................................ 20
5.11 Pixi zoom function. ................................................................. 21
5.12 A snapshot of the visualization module developed in PixiJS. ............... 21
6.1 Average FPS for the first ten seconds when loading a data set of 100 nodes. ......................................................... 22
6.2 Average FPS for the first ten seconds when loading a data set of 203 nodes. ......................................................... 23
6.3 Average FPS for the first ten seconds when loading a data set of 390 nodes. ......................................................... 23
6.4 Average FPS for the first ten seconds when loading data sets of different sizes. ......................................................... 24
List of Tables

6.1 Results from the framework evaluation, without blur filter. . . . . . . . . . . . . . . 24
6.2 Results from the framework evaluation, with blur filter active. . . . . . . . . . . . . 24
1 Introduction

The World Wide Web was created in the early ‘90s as a way to share information across the Internet. Much has happened since the early days of static hypertext documents with today’s web applications rather resembling desktop applications in both look and functionality. The release of the current HTML version, HTML5, made it possible to create fully interactive applications and games directly in the browser using native web technologies only. Desktop applications are still the best choice for performance-critical applications, such as graphically demanding applications and games due to web applications limited resources. However, the gap between web applications and desktop applications is decreasing.

The advantages of web applications include platform independence and no need for installation and updates on the client-side. Platform independence comes due to the single dependency of a web browser to be able to run the application. Consequently, web applications can be run on any platform with a single code base, eliminating the need to develop a new application for each platform. Furthermore, as the application is fetched from the web server every time it is used, the need for installation and updates on the client-side is not needed. As soon as any update is needed, the code of the web server can be updated and all clients using the application will receive them the next time the website is visited. This ensures that all users are using the latest version of the application.

Because of these advantages, web applications have started gaining popularity over desktop applications. However, creating web applications handling large amounts of data and performing heavy rendering is still a challenge. Optimizations need to be considered at every point of the application to use the web browser’s limited amount of resources in an efficient way.

1.1 Aim

The aim of this thesis is to investigate JavaScript visualization frameworks that are using native web technologies for its rendering. The goal is to find the framework best suited for developing a visualization module capable of handling up to 1000 nodes that are continuously moving while responding to user interactions. This is done by first deciding on three candidate frameworks which are used to develop a prototype of the final visualization module. Thereafter, the performance of these frameworks will be evaluated based on the frame rate when loading data sets of increasing size.
1.2 Research Question

The following research question will be the basis of this thesis in order to realize the afore-
mentioned objective:

- What JavaScript visualization framework is best suited for developing a visualization
  module capable of handling up to 1000 continuously moving nodes with retained frame
  rate?

In the context of this thesis, retained frame rate means that the frame rate is kept above 20
frames per second (FPS), which Hoetzlein et.al. defines as the interactivity limit for an appli-
cation [1].

1.3 Delimitations

This study is limited to investigating three different frameworks, each using different rendering
methods. It is also limited to investigating the performance in only one metric, the frame
rate.
This thesis was carried out in collaboration with Entiros Integrations who are the leading specialized integration company in Sweden. The study was part of the development of a new product from Entiros, called Starlify\(^1\). Starlify is a tool that enables companies to organize and visualize their open application network. It provides an overview of the integration landscape, containing all applications and APIs. The visualization module to be developed is a force-directed graph containing the application network, and is the core of the application. The application network consists of three categories of nodes; systems, services, and references. Systems are the top-level nodes and can provide services. Furthermore, services can be consumed by a system through a reference. The Starlify object model is presented in Figure 2.1.

The study was conducted to aid in the decision of a framework to use when building this visualization module for Starlify. Furthermore, a minimal viable product was to be developed with the chosen framework, and was to be delivered at the end of the thesis.

\(^1\)https://starlify.entiros.se
The aim of this chapter is to form a theoretical foundation and present key concepts relevant to this thesis. To begin, the notion of web applications will be presented. This is followed by an introduction to technologies for performing animations on the Web and a few web animation frameworks. Lastly, some key concepts for this thesis will be described.

3.1 Web Applications

Since the dawn of the World Wide Web, webpages have consisted of text documents with static content and links for navigating between pages. This type of webpage still makes up the majority of the Web, but a new type of webpage is increasingly growing in number. This new type of webpage is called a web application, with looks and behavior similar to a desktop application [2].

As web technologies keep maturing and the distinction between web applications and desktop applications are getting smaller, web applications are growing ever more popular and have been gaining the upper hand over desktop applications. Web applications possess some favorable features including cross-platform compatibility, effortless system updates and lower development costs [3].

3.1.1 Rich Internet Application

A rich internet application (RIA) is an interactive web application with similar features of a desktop application. This type of application often supports features such as client-side storage and client-side computations, which reduces loading times by cutting down unnecessary communication with the server. However, one of the biggest features of RIAs is its impressive interaction and animation capabilities. RIAs are often developed as single-page applications, that with help of JavaScript dynamically change the content of the page rather than redirects to a new page. As a result, the need for full-page refreshes could be eliminated which reduces loading times further [4].
3.1.2 Single-Page Application

In the past, every time you wanted to toggle between the content of a web page, you needed to go through a link to get to a new page containing this new content. An example of this would be to go from the login page to the main page of a logged-in user. This kind of web application is called a multi-page application (MPA), as they consist of multiple different pages. The new way of developing web applications is by dynamically change the content of the web page, which results in the user being able to stay on the same page during a full session. Thus, this new type of web application is called a single-page application (SPA) [4].

In an SPA the entire web application is loaded once the web page is visited. This results in a one-time loading page, instead of loading times of each transition as in the case of MPAs. However, multimedia and data might be loaded once it is used to prevent huge initial loading times, consequently, injecting some loading of content during a session [4].

Furthermore, to create a SPA you need a way to avoid page reloads when communicating with a server or fetching data from an API. Many techniques exist for accomplishing this, with the most renowned being Ajax. Ajax isn’t a technology in itself, but a set of numerous solitary technologies that been put together in an efficacious way. This technique allows for asynchronous calls to a server for data fetching or calculations to be performed [5].

3.2 HTML

HTML, which stands for Hyper-Text Markup Language, is the base of the Web. It is a markup language that uses tags to describes how to present text contained in these tags. For example, a `<h1>` tag is a title, a `<p>` tag is a paragraph and a `<a>` tag is a link to another document on the Web. The earliest documentation of HTML is first mentioned in October 1991 by the father of the World Wide Web, Tim Berners-Lee [6][7]. It was first designed to present text and static content, e.g. images, however, as the popularity of the Web grew, so did the demand for other features. As the HTML standard at the time did not support this, many proprietary technologies were developed to extend the features of the Web, which also presented a problem. The initial idea of the World Wide Web was to provide an open and platform-independent way to share information. These new technologies require additional software to be installed by the client and force developers to buy licenses and learn proprietary technologies in addition to the HTML standard. This pushed the founding of the World Wide Web Consortium (W3C), whose purpose is to lead the progress of the Web by developing open standards for web technologies, including HTML. W3C developed multiple HTML standards during the ‘90s, up to HTML4 in 1998. Since then, the way we use the Web has changed drastically. The use of mobile devices started growing, and content changed from static text-documents to high-quality multimedia. To keep up with these new requirements W3C developed HTML5 which is the current HTML standard [8][9].

3.3 Web Animation Technologies

In the early days of the Web, there were no native technologies for creating animations and RIAs. This was simply due to the fact that it was not the way the Web was intended to be used at the beginning. As the demand for such functionality arose, alternative technologies evolved, for instance, Flash and Silverlight [10]. As an answer to these proprietary technologies, W3C developed HTML5 which includes an impressive feature set covering all needs for creating desktop-like applications to be run on any device. Location-based services, `<video>` and `<audio>` tags, client-side storage and client-side rendering in the new canvas element is just a selection of the features included in the HTML5 standard [8][9]. HTML5 is the current version of HTML and was released in 2014. By 2020, most of the proprietary technologies have stopped being supported for the Web or have been declared deprecated [11][12].
3.3. Web Animation Technologies

3.3.1 Scalable Vector Graphics
Scalable Vector Graphics (SVG) is a vector image format developed by WC3. It supports both animations and interactions in 2D and is written with XML. As of the HTML5 standard, it is possible to incorporate SVG mark up inside of HTML code. Previous to HTML5, vector graphics could be included in web-pages with the help of plug-ins e.g. Flash. However, since WC3 released it as an open standard it has been freely available for everyone [13].

SVG offers plenty of features for the developer to create images and animations inside web applications. All typical graphics primitives are available, i.e. lines, polylines, polygons, rectangles, circles, and ellipses. These elements are added to the Document Object Model (DOM), and could also be grouped and organized in a tree structure. This enables for styling and transitions, which is also a feature of SVG, to be added either to a single element or a group of elements with a single instruction. Transitions included in SVG are scale, translate, rotate and skew. Additionally, durations can be added to transitions to create animations. Colors could be added to elements either as a solid fill or as gradients [13].

SVG operates in what is called retained mode, which means that a display list of objects is kept by the graphics renderer. These objects are then displayed on the screen using the properties of the objects. The properties could be the color, shape, and position [14]. Moreover, using JavaScript is possible to create new elements, change their properties and animate them dynamically by changing property values over time [10]. This is a very computational efficient technique for handling animations in high resolutions due to the use of vector graphics. However, this technique does not perform well when frequently executing transitions on a large number of elements [13].

3.3.2 Canvas
Another new feature of HTML5 is the Canvas element. The Canvas element allows for the incorporation of vector graphics, video, audio, images and animations, all in the same place. All rendering is performed on the client-side, which reduces communication with the server for a smoother and faster presentation [8][9].

HTML5 Canvas operates in what is called immediate mode, which means that the entire scene is redrawn at every frame. This is done by using the Canvas API which provides functionality to draw shapes, render text or display images on the Canvas element. This mode leaves more responsibility to the programmer, but also more freedom of how the scene should be rendered. The Canvas is drawn by using JavaScript calls to the Canvas API and build in functionality for mouse and keyboard input makes it possible to create impressive animations, interactions, and games directly in the browser [14]. It has been a vital part of the development of games as of its great performance when rendering complex scenes. However, as with SVG, the performance drops as the number of elements to draw grows [10].

3.3.3 WebGL
WebGL, short for Web Graphics Library, is yet another technology packed into HTML5. It is a JavaScript API that uses the HTML5 Canvas and its own special context to draw 2D and 3D interactive animations directly in the browser. It is based on OpenGL ES, WebGL 1.0 is based on OpenGL ES 2.0 and WebGL 2.0 is based on OpenGL ES 3.0 [15]. OpenGL ES is a graphics library used in embedded systems (ES), designed to be run on devices with less computational resources, like phones and tablets. This makes it perfectly suited for the Web, to provide stunning animations to any platform and device. Unlike SVG and Canvas, WebGL utilizes the device’s graphics processing unit (GPU), which is a processor dedicated to rendering 3D graphics. By doing this, 3D graphics can be brought to the Web, and more complex 2D scenes can be rendered with the use of less resources [16].
3.4 Web Animation Frameworks

There are countless of JavaScript animation and visualization frameworks that are utilizing the HTML5 technologies. These frameworks provides easy to use APIs to help the developer in the creation of beautiful visualizations and animations directly in the web browser without the need for any plugins. This section presents the two frameworks D3.js and PixiJS, which were also being used during the work of this thesis.

3.4.1 D3.js - Data-Driven Documents

A very popular visualization and animation framework is called D3.js, though it is often referred to as D3 which is short for Data-Driven Documents. It is a JavaScript library that utilizes features of web standards such as HTML, SVG, and CSS to provide stunning dynamic and interactive data visualizations to the Web. It is data-driven, which means that it allows you to bind data to the DOM and perform transformations to the document. For instance, an array of data could be used to generate a pie-chart, a bar-chart, a scatter plot or a simple HTML table. However, you can create more advanced visualizations like Voronoi diagrams, hierarchical edge bundling, and dynamic force-based graphs like a force-directed graph, all with full interactivity capabilities [17]. The developers of D3 describes it as a visualization “kernel” rather than a framework as it provides efficient data-based manipulations of documents rather than an abstraction layer. This results in some great advantages; it becomes easier to debug, it gets highly integrable with other frameworks and gains exceptional performance [18].

3.4.2 PixiJS

PixiJS is a JavaScript framework for creating 2D visualizations and animations using WebGL. The use of WebGL results in unmatched performance on the Web, however, not all browsers support WebGL, in which case Pixi has a fallback to HTML5 Canvas. Pixi includes full interactivity capabilities which makes it highly suitable for developing web-based games and RIAs. It provides a clean and simple to use API which gives a layer of abstraction from the complex WebGL API. Hence, the development time could be reduced for developers unfamiliar to low-level graphics libraries, without sacrificing performance and rendering speed [19][20].

3.5 Force-Directed Graph

A graph is a great means for visualizing relationships between objects. However, how to draw a graph is not a trivial matter, though it is a widely studied area. In most cases, it is desired to have minimal edge crossing, edges of similar length and symmetry of the entire graph to get the utmost aesthetically pleasing graph. This can be achieved in various ways, one of which is by using a force-directed algorithm. A force-directed algorithm can be implemented in numerous ways, but common to all is that forces are applied to the nodes and edges of the graph. By simulating the forces acting between the bodies of the physical system, each node will find a placement in the graph where the energy of the system is minimized. Various algorithms exist that work in different ways, but generally, a repelling force is applied to the nodes of the graph while the edges are working as springs to keep connected nodes close together. Using force algorithms to draw graphs often requires little to no knowledge of graph theory which makes it accessible to most [21].
3.6 Frame Rate

Frame rate is the frequency that the screen is displaying images, and is generally specified in frames per second (FPS). The FPS of a web application is a common unit of measurement for the perceived performance, and could also be measured as time per frame. Web applications are limited to a maximum of 60 FPS, and a high frame rate is desired as the application will be perceived as smoother and more responsive \cite{22}. Hoetzlein et.al. \cite{1} measured the time per frame when they examined the graphics performance in RIAs and concluded that 20 FPS is an interactivity limit.

Claypool et.al. \cite{23} studied how frame rate and screen resolution affect the playability and perceived performance of computer games. Their results showed that the frame rate had a big impact on the playability while the perceived quality was not affected to the same extent. While these results came from studying the performance of playing computer games it could very well be applied to the usage of interactive web applications. In the same study, the authors conclude that previous research shows that a much lower frame rate, as low as 5 FPS, could be acceptable while watching videos. Moreover, they state that their findings revealed that the same was not true for games. Games and interactive applications require a significantly higher frame rate and they saw a considerable increase in performance up to approximately 30 FPS. However, the increase in performance continued all the way up to 60 FPS.
This chapter describes the method used in the thesis, including a pre-study, implementation and evaluation phase. The purpose of the pre-study phase was to decide on three different JavaScript visualization frameworks to explore and evaluate. The frameworks were reviewed by developing a basic visualization module with each of the frameworks, followed by performance testing and evaluation of the results. Additionally, the pre-study phase includes the gathering of requirements for the visualization module. The development of the visualization modules is described in the implementation phase, and lastly, the evaluation phase includes the performance testing and evaluation of the developed modules.

4.1 Pre-study

Initially, a pre-study was conducted, with two goals in mind. The first goal was to gather requirements for the visualization module. These requirements include the visuals of the UI, but also the fundamental functional requirements. The second goal of the pre-study phase was to use this set of requirements and find three suitable JavaScript visualization frameworks that could be used to fulfill the requirements established in the previous step. The frameworks were then used in the implementation phase, where each framework was used to develop a module that conformed to the requirements.

4.1.1 Requirements

The requirements for the visualization module could be divided into two categories, the visual requirements of the user interface and the functional requirements. At the start of the thesis, a design for the graph was provided. The design was delivered as a rough digital sketch of how the graph should look like. It contained the sizes and colors of the different node types, and also information about how the links should look and approximately how the nodes should be positioned for a sample data-set.

The functional requirements were set by discussions with the product owner. A list of requirements for the final version of the visualization module had already been created at the start of the thesis. This list was a starting point in the discussions of the requirements for the prototype modules. However, as the original requirements were set for a fully implemented module, they had to be reduced down to the most essential parts. The requirements for
the prototype modules were important to set to be able to verify that all basic functionality was possible to implement with every investigated framework and that each module was developed with the same goals in mind.

4.1.2 Frameworks

The next step, following the gathering of the requirements for the visualization modules, was to find a number of interesting visualization frameworks to investigate. It was decided to limit the number of frameworks to three. This number was considered high enough to get a few different options to compare between. Also, three implementations were believed to be of a reasonable extent to fit within the timeframe of the thesis. The search for frameworks was composed of reading research papers about web-based visualizations as well as scanning the web using the Google search engine.

4.2 Implementation

The aim of the implementation phase was to develop a visualization module conforming to the predetermined requirements, set in the pre-study phase. However, the implementation phase started with the development of a data set generator. This was needed to be able to verify the functionality for different sizes and structures of data.

The data-set generator was developed to create data sets of pre-defined size and with the same structure and relationships as in the final product. It was necessary to create accurate data-sets to make sure the results of the study were applicable to the final product. The data generator was created as a small React application that sent Cypher queries to a Neo4j graph database.

As the data set generator was done, the development of the visualization modules started. Three visualization modules were developed, each with a different rendering technique, which were selected in the pre-study phase. The goal was to implement each module as indistinguishable from another as possible. This was to be true for everything from appearance to functionality and feel. The core of the visualization module is a graph where the nodes are positioned with a force-directed algorithm. The algorithm simulation is animated from the beginning, with all nodes positioned in the center of the screen. As the simulation is running the nodes find their positions in the graph where the total energy of the graph is minimized. This means that the graph will have an appearance depending on the input data.

4.3 Evaluation

The purpose of the evaluation phase was to assess the performance of the modules produced in the previous phase. The performance of these modules could be measured in various ways, but the measuring unit closes connected to the perceived performance is frame rate. Frame rate is frequently used to measure the performance of web applications [22], where one example is the work of Andrews and Wright [24] who developed a web-based visualization framework. They used frame rate as a mean of assessing the performance differences between different implementations of their framework.

To be able to test these modules, data sets of various sizes were needed. This was not available and needed to be created before the evaluation could be started. Data sets of approximately 100, 200, 400, 800 and 1600 nodes were generated. These data sets were saved as JavaScript objects in JSON files on the same form that is returned from the final products API. Next, the visualization modules were given a data set at initialization. The data sets were loaded one at the time, and as the data were loaded the force-directed algorithm simulation started which were also visualized at the screen. Using the Mozilla Firefox Developer Editions performance tool, the average FPS for the first 10 seconds was measured. This was done
10 times with each data-set and each framework. Next, the average of the 10 measurements was calculated giving a final result for each module and data set. As a result, an average FPS for the first 10 seconds of the simulation was achieved for each module with varying sizes of data. All tests were run on a Macbook Pro 15” (Mid 2015) with an Intel Core i7-4870HQ CPU, Intel Iris Pro Graphics 5200 GPU and 16GB DDR3 ram.
5 Pre-study and Implementation

This chapter presents the results of the pre-study and implementation phase.

5.1 Pre-study

The pre-study consists of two parts, the first is a requirement elicitation for the visualization modules and the second part was carried out to elect the visualization frameworks to explore.

5.1.1 Requirements

The requirements elicitation was performed through discussions with the product owner. A simple implementation of the visualization module was already available at the start of the thesis, however, it was limited in both functionality and design. The discussions used the previous implementation as a starting point, and desired functionality was added, and a new design was considered. The existing module was created with SVG using the D3.js framework and was displaying a force-directed graph. The new implementation was also going to present a force-directed graph, but additional graph types were desired for future implementations. A necessary feature linked to this is to be able to smoothly transition between graph types during a session. The next requirement was that the application needs to be responsive at all times, even when animations are run in the graph. This is an important requirement as the graph is dynamic and changes layout depending on what is shown in the graph. If a node is added or removed during a session, this should be reflected in the graph as well, without the application being unresponsive. Furthermore, requirements were added for features for zoom and pan of the graph, and the possibility to drag nodes around in the graph in a smooth fashion. Lastly, performance was discussed. The current implementation was handling about 200-300 nodes with an accepted FPS. This was quite a bit lower than what was required by the system. The first number that came up during the discussions was for several thousand nodes, which was later lowered to approximately 1000 nodes. It was concluded that 1000 nodes were more than enough nodes to be able to grasp what is actually showing. In the case of data sets with more nodes, it was decided to group multiple nodes together instead. However, the feature of grouping nodes was not added as a requirement for the visualization module in the first step. Furthermore, it was requested that the nodes in the graph should be blurred out rather than being solid circles. This was to get the impression of a starry sky, as
both the application and the company developing the application has a space theme. Lastly, a requirement was added that the graph should be centered in the browser window upon a load of a new data set. Additionally, the graph should cover approximately 80 percent of the browser window, independent of how large data set is loaded. This requires that both a zoom and pan is performed once the size of the graph is determined.

**Functional Requirements**

- Display a force-directed graph
- Possibility to add more graph types in future implementation
- Smooth transitions between different graph types
- Zoom and pan of graph
- Dragging of nodes
- The graph should be dynamic and based on input data
- Blurred nodes
- Centering and scaling of graph

**Non-Functional Requirements**

- The graph should be responsive to user input at all times
- Handle up to 1000 nodes with a frame rate above 20 FPS

### 5.1.2 Frameworks

The search for frameworks to investigate started with finding suitable technologies to use for creating interactive and dynamic web applications. This was mainly done by reading research papers and blog posts on the subject. Additionally, informal discussions were held with the members of the project team. It was concluded that there exist mainly three technologies for creating such applications, which are; SVG, Canvas, and WebGL. This is the same conclusion as Andrews and Wright came to when creating Fluidiagrams [24].

Based on the three rendering technologies it was decided to choose a framework for each technology. When researching rendering frameworks it was found that D3 is a widely used and popular framework, which has support for both SVG and Canvas rendering. Additionally, D3 comes with a module called force, which can be used to simulate force algorithms when creating graphs such as a force-directed graph. Furthermore, a simple implementation was already created in D3 with SVG rendering which could be used as the base for the SVG implementation. This was considered enough advantages to go with D3 as the framework of choice when creating the SVG implementation. As D3 also includes support for Canvas rendering, it was decided to use D3 for the Canvas implementation as well to reduce the overhead in learning yet another framework.

D3 does however not support WebGL rendering and could not be used for the last implementation. A few different WebGL frameworks were considered, and eventually, Pixi was selected. Pixi is an extremely popular WebGL renderer concentrated on producing fast 2D graphics. As 3D graphics were not going to be used this seemed like a good fit for the project.

**Frameworks**

- D3.js, SVG
- D3.js, HTML Canvas
- PixiJS
5.2 Implementation

As the pre-study phase was completed and three promising visualization frameworks were identified the implementation phase started. The aim of the implementation phase was to create a small prototype of a visualization module in each of the frameworks chosen in the pre-study phase. However, the implementation phase started with the development of a data set generator. Data sets were considered necessary at the beginning of the development as the visualization modules were in need of input data to be visualized.

5.2.1 Data Set Generator

At the beginning of the implementation phase, there were already some data available, but everything had been manually created. The process of creating data sets manually was tedious and hampered the testing of large data sets. For this purpose, a data set generator was needed, which would also be vital in the evaluation phase when multiple data sets of varying sizes are tested with each framework.

It was decided that the data generator should be created as a small web application with the React framework, as the main application was also developed with React. This facilitated deployment of the data generator as a module of the main application as well as allowing for a refresh of React-knowledge and getting acquainted with the data model of the project.

The data generator application was developed as a simple single page application and consists of a form with options to select where to create the new data and how large the new data set should be. Additionally from the options form there are two buttons, one for generating a new data set and one for deleting previously generated data. In the options form, you first select what domain you are interested in creating or removing data from. The domains are fetched from the main applications API where authorization is performed to provide domains for a public user or an authenticated user. As the domain has been selected, the models residing in the domain are fetched and can be selected in a new drop-down menu.

![Starlify Data Generator](image)

Figure 5.1: A snapshot of the data set generator.

The next step is to decide what the new data set should look like. To do this we first need to understand how the data model is structured. The data model has three main object types called systems, services, and references. Systems are the top-level objects and can contain the two other types, or even other systems which are called sub-systems. However, the data set generator was limited to only creating top-level systems. A service object is something a system can provide for other systems to consume. Consequently, a service node is always connected to a parent system. To represent a system consuming a service provided by another system the reference object is used. These three simple objects are used in the data model to represent a full application network.

The data set generator gives the user a possibility to tune the total number of systems of a data set before generating it, by entering the number of systems as seen in Figure 5.1. The number of services and references is selected randomly for each system. Although, the range
can be set by entering the maximum number of services and references before creating the data. The bottom value of the range is always set to zero. Default values are set for the input values of the number of systems, max services per system and max references per system to ten, three and three respectively.

5.2.2 D3.js - SVG

The next step in the implementation phase was to develop a basic visualization module in each of the frameworks chosen in the pre-study phase. The goal in this step was to get the visualization module in each framework to conform to the set of functional requirements decided in the pre-study phase, and have the same look and feel. The development of a visualization module had already been begun by Entiros by the start of the thesis, and this formed the foundation of the first implementation. This module had been developed in the JavaScript visualization framework D3.js and used SVG for its visualizations. The development of the first module started off with ensuring that all pre-determined requirements were fulfilled for the existing module, which involved adding new features and modifying some of the existing behavior. Existing behavior included the rendering of nodes, edges, and text, dragging of nodes and lastly, zoom and pan of the graph. These features required few or no modifications. In addition to the already implemented features, some new ones needed to be implemented.

Each node, link and text element is represented on the screen as an actual DOM-element. Hence, the DOM-tree will grow very large as the data sets get larger. This comes with the advantage that nothing needs to be redrawn if something changes. If the position of a node is updated in the data set, the DOM-element will also need to be updated with the new position and will then be automatically re-rendered in the new position. To perform updates on nodes and links, a tick function was added to the D3 force simulation. This function simply gets new positions from the simulation and updates the elements in the DOM accordingly, which in turn triggers a new render.

The first feature that was needed was a zoom and repositioning of the graph upon the selection of a new data set. It was requested that the graph should cover approximately 80 percent of the web browser, leaving at least a ten percent margin between the graph and the edge of the browser window on all sides of the graph, with the graph positioned in the center of the web browser. This was done by first getting the center position of the graph. The center of the graph was calculated by getting the maximum and minimum x and y position from the node array. These positions were used to calculate the center of the graph, by adding half the graph width to the minimum x-position and adding half the graph height to the minimum y-value, as seen in Figure 5.2.

```javascript
const x = range.x.min + ((range.x.max - range.x.min) / 2);
const y = range.y.min + ((range.y.max - range.y.min) / 2);
```

Figure 5.2: Calculation of graph center x- and y-position.

The next part of the zooming was to determine the $k$-value, the level of zoom needed to fill 80 percent of the screen. This was done by first resolving what orientation has the smallest margin, i.e. if the height or width of the web browser is the smallest in proportion to the height and width of the graph. The chosen orientation is left with a ten percent margin on each side. As the position of the nodes is calculated from the center of the node, it introduces an issue for graphs with a small number of nodes. In these cases, the node radius will be equal or more than ten percent of the whole graph size which will leave no margin between the graph and the browser after the zoom. This was resolved by adding an additional margin on each side of the graph, with size of the radius of the largest node. Consequently, we
get a ten percent margin from the edge of the nodes on the fringe, rather than the center of the same nodes. The code for calculating the k value is presented in Figure 5.3 where target equals 0.8 and width and height is the width and height of the browser.

```javascript
k = ((x.max - x.min) / width) > ((y.max - y.min) / height) ? (target / ((x.max - x.min + radius * 2) / width)) : (target / ((y.max - y.min + radius * 2) / height));
```

Figure 5.3: Calculation of graph center k-value.

A transform is then performed with the built-in D3.js zoom function with the new center position and zoom ratio provided. The transform and zoom call is shown in Figure 5.4 which causes the screen to zoom and pan to the provided position and zoom ratio in a smooth motion during 2.5 seconds.

```javascript
const transform = d3.zoomIdentity.translate(width / 2, height / 2).scale(center.k).translate(-center.x, -center.y);
graph.transition().duration(2500).call(zoom.transform, transform);
```

Figure 5.4: Transition to center of graph and zoom to cover 80 percent of web browser.

The second feature that needed to be implemented was a blur filter for the nodes. It was desired that the graph should look like a starry sky, with the nodes being blurred out rather than being drawn with solid edges and full opacity. This was done by using the built-in SVG Gaussian blur filter. The code for the blur filter is found in Figure 5.5 and the result of the finalized module is presented in Figure 5.6.

```javascript
const system_filter = svg.append("defs")
  .append("filter")
  .attr("id", "system_glow")
  .append("feGaussianBlur")
  .attr("stdDeviation", "4");
```

Figure 5.5: SVG’s Gaussian blur filter.
Figure 5.6: A snapshot of the visualization module developed in D3.js using SVG.
5.2.3 D3.js - Canvas

The second module was chosen to be developed with the D3.js framework as well, but using HTML5 Canvas for the rendering. As a result of using the D3.js framework again, most functionality could be reused. The dragging of nodes needed some modifications, however, the principle for how it was done was similar to the SVG implementation. The same was true for the zoom behavior, where the functionality from the SVG implementation could be reused after some modifications. The feature for repositioning and zooming the graph to cover 80 percent of the web browser was possible to be reused without any modifications at all. However, as HTML5 Canvas was going to be used instead of SVG for rendering, this part needed to be fully rewritten.

In contrast to SVG, the HTML5 Canvas is only using one single HTML element to create visualizations, the canvas itself. This canvas needs to be redrawn every time the data set gets updated, e.g. a new node is created, a node’s position is changed or the graph is zoomed or panned. Accordingly, the tick function of the SVG implementation, which updates each DOM-element with new data, is replaced with a draw function that draws a new image in the web browser with the nodes in the current state. This draw function will be triggered on every tick of the simulation which in turn is triggered on updates from the simulation or from external input, e.g. dragging of nodes. Additionally, the draw function will be triggered by the zoom function which in turn is triggered on any change in zoom or pan, either by user input from a mouse click or mouse wheel, or from zooming functions that could be triggered on any kind of event.

When creating the blur effect in the canvas implementation it was not possible to use a Gaussian blur filter, as the built-in Gaussian blur filter for canvas is only applicable on the whole canvas and not on single items. This was solved by creating a custom radial gradient of decreasing opacity towards the edges. A total of eight steps were used for creating the filter and the code is displayed in Figure 5.7.

```javascript
function createBlurFill(node, color) {
  const radgrad = context.createRadialGradient(node.x, node.y, 0,
                                             node.x, node.y, node.radius);
  radgrad.addColorStop(0, `rgba(${color}, 1)`);
  radgrad.addColorStop(0.3, `rgba(${color}, 0.9)`);
  radgrad.addColorStop(0.5, `rgba(${color}, 0.8)`);
  radgrad.addColorStop(0.65, `rgba(${color}, 0.65)`);
  radgrad.addColorStop(0.8, `rgba(${color}, 0.4)`);
  radgrad.addColorStop(0.9, `rgba(${color}, 0.2)`);
  radgrad.addColorStop(1, `rgba(${color}, 0.05)`);
  radgrad.addColorStop(1, `rgba(${color}, 0.0)`);
  return radgrad;
}
```

Figure 5.7: Canvas custom blur.

The filter is created separately for each node, and it could be used with any color. This made it possible to use the filter for both system nodes and service nodes. The final result of the canvas implementation is presented in Figure 5.8.
5.2. Implementation

Figure 5.8: A snapshot of the visualization module developed in D3.js using HTML5 Canvas.
5.2. Implementation

5.2.4 PixiJS - WebGL

It was decided that the third and last module would be developed with PixiJS which is a JavaScript framework built on top of WebGL. WebGL utilizes the GPU, in contrary to the other frameworks examined in this thesis. Consequently, this would presumably improve performance when rendering large data sets, thus it was considered a path worth exploring further. As for the first two implementations, some functionality was possible to reuse for the Pixi implementation as well. The simulation itself was the same through all three implementations with D3’s force module keeping track of the nodes’ positions. Pixi was closely connected to the Canvas implementation in regard to the main ideas for how to build the module. In the SVG implementation there existed actual objects that could be dragged across the screen, whereas in the Canvas and Pixi implementation, it was needed to calculate what node was closest to the mouse and whether or not the mouse pointer actually is over the node. The rendering was the biggest difference between Canvas and Pixi. However, this was purely based on implementation details and both implementations built on the same fundamentals. Both used similar functionality for the dragging of nodes, zooming, and panning. Pixi came with built-in filters, one of which was Gaussian blur, the same as was used in the SVG implementation. The filter was simply called `BlurFilter` and the code for creating this is shown in Figure 5.9.

```javascript
const blurFilter = new PIXI.filters.BlurFilter();
blurFilter.blur = 4;
```

Figure 5.9: Pixi gaussian blur.

Another feature that needed to be reimplemented was the centering function. The zoom and pan work similar in Pixi as in SVG and Canvas, however, Pixi doesn’t have a built-in transition from one state to another. Previously when going from a state to another, that is, from a level of zoom and position in the graph to a new level of zoom and/or position in the graph, it could be done with one call, providing the transition time and new state. In Pixi, a custom transition function was needed to handle this. This was done by creating arrays of 100 interpolated values between the two states, as seen in Figure 5.10.

```javascript
function center() {
    const target = getGraphCenter();
    const current = {x: viewport.center.x, y: viewport.center.y, k: viewport.scale._x};
    const transition = {x: [], y: [], k: []};
    for (let i = 0; i < 100; i++) {
        transition.x.push(current.x + ((target.x-current.x) / 99 * i));
        transition.y.push(current.y + ((target.y-current.y) / 99 * i));
        transition.k.push(current.k + ((target.k-current.k) / 99 * i));
    }
    zoom(transition);
}
```

Figure 5.10: Pixi centering function.

These arrays were then used when calling the zoom function, which zoomed into the graph one step at the time, with the step size 1/100. The zooming starts with 10 ms between each zoom for the first third of the values. It then continues zooming one step at the time with 7 ms between each zoom. The last third of the values is zoomed with a 5 ms interval.
This creates the effect of a slow zoom which accelerates until the target is reached. The code for the zooming is presented in Figure 5.11.

```javascript
function zoom(a, i = 0, timeout = 10) {
    setTimeout(() => {
        viewport.setZoom(a.k[i], false);
        viewport.moveCenter(a.x[i], a.y[i]);
        i++;
        if (i < a.k.length/3)
            zoom(a, i);
        else if (i < a.k.length / 3 * 2)
            zoom(a, i, 7);
        else if (i < a.k.length)
            zoom(a, i, 5);
    }, timeout);
}
```

Figure 5.11: Pixi zoom function.

With all these features implemented, the Pixi module was almost indistinguishable to the other modules in both looks and behavior. The final result of the Pixi implementation is displayed in Figure 5.12.

Figure 5.12: A snapshot of the visualization module developed in PixiJS.
6 Results

This chapter presents the results of the evaluation phase.

6.1 Evaluation

The evaluation of the visualization modules was carried out by measuring the FPS of the application for the first ten seconds after loading a data set of a fixed size. This was done ten times for each data set, and with five different data sets of increasing size. Furthermore, the tests were run with the blur filter active and disabled. A more comprehensive description of the framework evaluation is found in section 4.3. The smallest data set consisted of 100 nodes, and the result for loading this data set with each framework, with and without blur filter, is presented in Figure 6.1.

Figure 6.1: Average FPS for the first ten seconds when loading a data set of 100 nodes.
6.1. Evaluation

Each data set for the evaluation is approximately double the size of the previous data set. Accordingly, the next data set was of size 203 nodes. The results from loading this data set with each framework, with and without blur filter, is presented in Figure 6.2.

Figure 6.2: Average FPS for the first ten seconds when loading a data set of 203 nodes.

Next, is the data set of 390 nodes. This data set was also tested with and without the blur filter, and the result is presented in Figure 6.3.

Figure 6.3: Average FPS for the first ten seconds when loading a data set of 390 nodes.

The data sets following after, with size 777 and 1617 were only tested without the blur filter as the performance loss with the filter active was already too high. A complete view of the test results is presented in Figure 6.4, where all data sets and modules are shown side by side. In this figure, all results are without the blur filter.
6.1. Evaluation

Figure 6.4: Average FPS for the first ten seconds when loading data sets of different sizes.

Lastly, the test results without the blur filter active are found in Table 6.1 and the test results with blur filter active are found in Table 6.2.

Table 6.1: Results from the framework evaluation, without blur filter.

<table>
<thead>
<tr>
<th>Number of nodes</th>
<th>Average FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SVG</td>
</tr>
<tr>
<td>100</td>
<td>51.92</td>
</tr>
<tr>
<td>203</td>
<td>38.66</td>
</tr>
<tr>
<td>390</td>
<td>19.87</td>
</tr>
<tr>
<td>777</td>
<td>9.40</td>
</tr>
<tr>
<td>1617</td>
<td>4.95</td>
</tr>
</tbody>
</table>

Table 6.2: Results from the framework evaluation, with blur filter active.

<table>
<thead>
<tr>
<th>Number of nodes</th>
<th>Average FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SVG</td>
</tr>
<tr>
<td>100</td>
<td>37.52</td>
</tr>
<tr>
<td>203</td>
<td>16.98</td>
</tr>
<tr>
<td>390</td>
<td>9.29</td>
</tr>
</tbody>
</table>
In this section, the results will be discussed and analyzed. This is followed by some thoughts on the method used and what could have been done differently.

### 7.1 Results

The results from the evaluations show a clear difference between the three frameworks. D3 with SVG is easy to use once you have learned the framework. However, the performance suffers drastically as the number of elements increases. Already at 400 nodes, the SVG module dropped down below 20 FPS, as Hoetzlein et.al. mentioned as the interactivity limit [1]. If the blur filter is taken into account, this limit is reached already at 200 nodes. This concluded quite clearly that the SVG implementation was not performing to the standards that were required.

Canvas performed quite well up to 800 nodes and had no loss in performance with the blur filter active. 1600 nodes were too much for the Canvas module to handle, however, the Pixi module was not able to keep the frame rate above 20 FPS either. The Pixi module did outperform the Canvas implementation when the blur filter was deactivated and solid shapes were used. Although, the differences in performance were not that high, and if considering the blur filter the Canvas implementation was the winner in performance.

One thing to consider is if the blur filter was possible to implement in another way than with the built-in Gaussian blur filter. A custom filter was created for the Canvas implementation, and the performance of the Pixi module could have been improved by creating a custom filter there as well.

Moreover, all tests were run on a laptop without a dedicated graphics card, which means that WebGL could not be utilized to its fullest potential. This might be the answer to why the Pixi implementation did not perform better than it did. With a computer with a dedicated graphics card, the Pixi module would likely get a considerable performance boost while the other frameworks would not be able to utilize the hardware and get the same benefits. However, as the final product is a web application that is going to be used by customers who are likely to be without powerful computers, the tests run in the evaluation phase presumably reflect the results that would have been obtained by real users of the system.
7.2 Method

The goal of this study was to find out what web rendering framework is best suited for handling up to 1000 elements that continuously move around the screen and responds to user interactions. First, three rendering methods were selected, which also are the three options available for native web rendering. It was decided that each method was interesting to investigate further, and due to time limitations only one framework built on top of each method was chosen. Next, three highly used and popular frameworks well suited for the task were chosen. This is the first decision that has an effect on the end result. Other frameworks could have been chosen, or multiple frameworks with either Canvas or WebGL rendering, as those were the plausible candidates for the final product. However, it was interesting to investigate the difference from the base case with SVG, as it was used when the previous module was developed. D3 is a sophisticated and popular framework and no real competitor was found for the Canvas implementation. For the WebGL module, there were multiple options instead of Pixi, but because of time constraints, Pixi was chosen since it is a lightweight library focused on 2D graphics.

The development steps are described thoroughly, with special features described in detail and framework-specific details left out. Consequently, it should be feasible to replicate the study with similar results. Furthermore, the evaluation is described in detail making it possible to reproduce with a comparable outcome.

Moreover, the choice of measuring the frame rate only might also have an effect on the outcome of the study. Other metrics such as CPU usage or memory usage could also be interesting factors to take into account. As in the other cases of the method, this was an issue connected to time constraints. To measure and review multiple metrics would have taken too much time to fit into the timeframe of the thesis. More weight was instead put on examining multiple frameworks, and the evaluation was based on the most crucial metric that is closest coupled to perceived performance, which is the frame rate.

The information gathered for the thesis was focused on articles published at conferences with a high number of previous citations. A few books were also used when applicable, which were often the best source of information in the case for the rendering methods and frameworks. As a last resort, a few websites were used, mostly because companies developing web technologies often use their own website as a platform for sharing information with the world.

7.3 The Work in a Wider Context

As the market is shifting from desktop applications to web applications, it is important to know the limitations of the web and the tools for creating web applications when migrating from desktop or developing new applications. This study sheds some light on the options for creating interactive web animations with native web technologies.
Conclusion

This thesis was set to evaluate a number of JavaScript visualization frameworks with animation and interactivity capabilities. More specifically it was set to answer the following question:

- What JavaScript visualization framework is best suited for developing a visualization module capable of handling up to 1000 continuously moving nodes with retained frame rate?

In this case, retained frame rate refers to keeping the average frame rate above 20 FPS, as this is considered being the interactivity limit of an application. This was done by first finding what rendering methods are available for web applications. The three native web rendering methods were found to be SVG, Canvas, and WebGL. Based on these rendering methods three frameworks built on these rendering methods were chosen. For SVG and Canvas, the framework D3.js was chosen, and for WebGL the framework of choice was PixiJS. Thereafter, a visualization module was created with each of the frameworks, all conforming to the same set of requirements. Lastly, these frameworks were tested and evaluated based on the average FPS when loading data sets of increasing size.

The results showed that the SVG module was far behind the other two in terms of retained FPS on larger data sets. The Canvas and WebGL modules were closer in the level of performance, where WebGL outperformed the Canvas implementation in the base case. However, when a Gaussian blur filter was activated in both modules, the Canvas module prevailed. This blur filter was used to make the nodes in the graph appear as stars on the night sky, which was a requested feature.

To answer the research question set for the thesis, the module developed with the PixiJS framework and WebGL rendering was the winner when it came down to performance with up to 1000 nodes, and beyond. However, the module developed with the D3.js framework and Canvas rendering was the only module that conformed to all requirements set for the visualization module, and consequently became the framework of choice. This came down to two key factors. The first one was that the PixiJS module did not achieve high enough FPS with the blur filter active, while the D3.js with Canvas rendering module did. Secondly, the D3.js framework was assessed to be easier to work with and add more features with. Nevertheless, the client found the result to be of great value as a WebGL implementation.
might be of interest in the future if the demand for higher performance would arise. The WebGL implementation was estimated to have more potential than was shown in this study for two reasons. First, it's plausible that the Gaussian blur filter could be implemented in a more efficient way. Secondly, the tests in this study were all performed on a laptop without a graphics card, which WebGL in contrast to the other rendering methods could utilize.

8.1 Future Work

For future work, it would be interesting to investigate the differences between Canvas and WebGL when running on a computer with a dedicated graphics card. For this specific case, it would also be interesting to see if the Gaussian blur filter would be possible to implement in a more resource-efficient way in the WebGL module.
Bibliography


