RENDERING PERFORMANCE ASPECTS OF ANIMATED DATA VISUALIZATIONS

Masterarbeit

zur Erlangung des akademischen Grades

Master of Science in Engineering (M.Sc.)

Eingereicht bei:

Fachhochschule Kufstein Tirol Bildungs GmbH

Web Communication & Information Systems

Verfasser/in:

Alexander Ries

1410738550

BetreuerIn / ErstgutachterIn:

Prof. (FH) Dipl.-Informatiker Karsten Böhm

ZweitgutachterIn:

Dr. Axel Uhl

Abgabedatum:

18.07.16

EIDESSTATTLICHE ERKLÄRUNG

„Ich erkläre hiermit, dass ich die vorliegende Masterarbeit selbständig und ohne fremde Hilfsmittel verfasst und in der Bearbeitung und Abfassung keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe. Die vorliegende Masterarbeit wurde nicht anderweitig für Prüfungszwecke vorgelegt.“

Kufstein, am 18.07.2016

Alexander Ries

TABLE OF CONTENTS

INDEX OF FIGURES

INDEX OF LISTINGS

KURZFASSUNG

FH Kufstein

Web Communication & Information Systems

Kurzfassung der Masterarbeit „Rendering Performance Aspects of Animated Data Visualisations“

Name: Alexander Ries

Erstbetreuer: Prof. (FH) Dipl.-Informatiker Karsten Böhm

Zweitbetreuer: Dr. Axel Uhl

Animierte Daten Visualisierungen haben zwei Vorteile. Sie verbessern das Benutzererlebnis und komplexe Prozesse werden leichter verständlich gemacht. Der Browser ist eine Plattform für die animierte Daten Visualisierungen entwickelt werden. Eine der größten Herausforderungen bei der Entwicklung von Web Anwendungen ist die Rendering Performance. Dazu existieren zahlreiche Benchmarks und Ressourcen im Internet und in der Forschung aber es wird immer nur ein sehr spezifischer Fall einer Animierten Daten Visualisierung behandelt. In dieser Arbeit ist es das Ziel eine allgemeine Aussage über die Rendering Performance von animierten Daten Visualisierungen treffen zu können. Es wurde untersucht, dass es sechs Aspekte gibt, welche die Performance von Animierten Daten Visualisierungen beeinflussen könne. Diese sind Web Technologie Kombination, Graphische Primitive, Styling, Anzahl der Graphischen Primitive, Animierte Eigenschaften und Browser. Es ist das Ziel dieser Arbeit herauszufinden welche Aspekte die Rendering Performance beeinflussen. Dafür wurde als Forschungsmethode ein Experiment gewählt, dessen Design aus bestehender Forschung abgeleitet ist. Das Design umfasst drei Konzepte. Zuerst sollen verschieden Kombinationen aus Aspekten in Web Experimenten entwickelt werden und die Performance getestet werden. Außerdem werden die zwei Metriken Rendering Time und Animation Smoothness an den Experimenten gemessen. Die Messungen sollen ferner auf einem Ansatz beruhen, welcher die Frames des Bildschirms während dem Ausführen der Experimente aufnimmt und auswertet. Um die Experimente durchführen und auswerten zu können wurde eine Software geschrieben. Die Ergebnisse zeigen, dass nicht alle Aspekte von Animierten Daten Visualisierungen gleichen Einfluss auf die Rendering Performance haben. Bei statischen Inhalten ist die Reihenfolge der Aspekte, Anzahl der Graphischen Primitiven, Graphische Primitive, Browser, Web Technologie Kombination und Styling. Bei animierten Experimenten ist die Reihenfolge Anzahl der Graphischen Primitiven, Browser, Web Technologie Kombination, Graphische Primitive und Animierte Eigenschaften.

ABSTRACT

FH Kufstein

Web Communication & Information Systems

Abstract of the Master thesis „Rendering Performance Aspects of Animated Data Visualisations “

Name: Alexander Ries

Supervisor: Prof. (FH) Dipl.-Informatiker Karsten Böhm

Co-Supervisor: Dr. Axel Uhl

Animated data visualizations are have the two advantages, that they delight the user and make complex processes more understandable. The browser is one platform data visualizations are developed for. However rendering performance is critical to browsers as performance of web applications is a general problem. On the Internet and in research there are only benchmarks, which deal with very specific data visualizations scenarios. As a result a more general approach is chosen to investigate the problem of rendering performance in animated data visualizations. It was found that animated data visualizations have certain aspects, which can influence the rendering performance. Those aspects are web technology combination, graphic primitives, styling, number of graphic primitives, animated properties and the browser. It is the goal of this thesis to determine which aspects have an influence on the rendering performance. In order to do so the research method experiment was chosen. The design which was derived from related work consists of three ideas. The first idea is to build small web experiment which consist of combinations of aspects of animated data visualizations. Second, the rendering performance of those web experiment should be measured with two metrics which are rendering time and animation smoothness. The third idea is to determine the performance of web experiments based on screen output. In order to handle a vast amount of performance tests, an experiment system was developed which run experiments automatically and saves rendering performance in a database. Later the rendering performance metrics where analyzed. There where static experiments executed, which displayed graphic primitives without animation and animated experiments which showed animations with various combinations of aspects. The results show that all aspects of animated data visualizations influence the rendering performance. However, at static experiments the aspects, ordered by influence are number of elements, graphic primitives, browser, web technology combination and style. At animated experiments the most influential aspects are number of elements, browser, web technology combination, graphic primitives and animated properties.

1. INTRODUCTION

Animated data visualizations are used for visual data exploration. Animations have two advantages. They delight the user and are useful for explaining changes to the user in a richer way, which would not be as accessible, when static content is displayed. Animated data visualizations are used in different areas. Consumer products use animation in data visualizations in order to enhance the user experience and to make applications easier to use. (Amit 2015) In research animation is used to understand dynamic processes better. Examples are blowing of wind or flowing of blood. (Steele 2010 p. 331)

The browser is one platform, where people consume animated data visualizations. In consequence, a variety of frameworks for the development of animated data visualizations have evolved. (ChartJS 2016; Fusioncharts 2016; Highcharts 2016) Browser applications are slower than applications that run natively on modern operating systems, like Windows, macOS, Android or iOS. As a result, performance optimization is especially important at browser applications, to keep up with better performing applications on other platforms.

One part of the performance of a browser application is rendering performance. As rendering performance is a metric that tells how fast content is rendered and how smooth animations appear on the screen, it is an essential for the design and development of animated data visualizations.

2. PROBLEM STATEMENT

Rendering performance is a topic in web development, where due to its importance, steady progress is made. As a result, today there are a lot of benchmarks on the web, that cover which web technology is how fast at rendering content (TUM 2016). Also there are best practices when it comes to rendering performance optimization, such as hardware acceleration for animation (Shapiro 2014) and CSS3 Transitions for browser optimized animations. (Tabalin 2015) Beneath resources on the web, there is research, which compares the rendering performance of frameworks for data visualization. (Lee et al. 2014) Moreover there exists research on the rendering performance of different combinations of web technologies. (Kee et al. 2012) However it is important to say that existing benchmarks and research on rendering performance of animated data visualizations focus only on specific data visualization scenarios. As a result current information and findings cannot be used for a wide range of animated data visualizations. It would be useful for designers and developers to have findings on what makes the rendering performance of animated data visualizations bad or good, which can be applied to a wide range of visualization scenarios.

This thesis takes an approach to find out which aspects of animated data visualizations have the biggest influence on rendering performance in general. The focus is on 2D animated data visualizations. In order to do so, it is assumed that there are six aspects of animated data visualizations, which can potentially influence the rendering performance.

1. Web Technology Combination: The combination of web technologies used to develop animated data visualizations has an influence on the rendering performance, as every web technology uses different techniques, processes and calculations to tell the browser how to render content.
2. Graphic Primitives: In this thesis the top 30 papers from the data visualization conferences InfoVis, SciVis and VAST in 2015 and 2016, which are listed on the website of the IEEE Vis conference (IEEE 2016), were analysed by their

visualisations. The analysis of data visualizations shows that 25 different 2D data visualization where used. The analysis of graphic primitives, these data visualizations consist of, shows that all 25 different visualizations used five graphic primitives. Listed by their frequency of use, these graphic primitives are square, line, circle, polyline and polygon. As a result it is assumed that a wide range of data visualizations consists of these graphic primitives.

3. Styling: The CSS attributes set to the used graphic primitives have an impact on rendering time and animation smoothness as resources on the web show. (Lewis and Irish 2013)
4. Number of Graphic Primitives: The number of elements in animated data visualization is correlated with the rendering performance. More elements mean more steps in the process of rendering the content.
5. Animated Properties: The paint operations in the browser vary for each animated property. As a result, certain properties and combinations of animated properties are more likely to slow down rendering. (Lewis 2015a)
6. Browsers: Browsers have different JavaScript and rendering engines, as described in chapter 5.3. As a result the code of the browser has an impact on how animated data visualizations are rendered, which results in different rendering performances.

Aspects of animated data visualizations have an influence on different rendering performance metrics. Rendering Performance is determined in this thesis by the metrics rendering time and animation smoothness, which are explained in detail in chapters 6.3 and 6.4. Because animated data visualizations can have two different states, which are static and animated, there need to be two different metrics for the rendering performance. The influence on rendering time and animation smoothness of the aspects Technology Combination, Graphic Primitive, Number of Graphic Primitive and Browser will be measured. It is also investigated how the aspect of styling influences rendering time and the aspect of animated properties affects the animation smoothness of animated data visualizations.

It is not clear, which aspects of animated data visualizations have the biggest influence on rendering performance. Some might have a noticeable influence and some not. When designers and developers know which aspects have the biggest influence on rendering performance, they can make better decisions on the design and the code. A designer can actively choose graphic primitives and styles for a design, which have less negative influence on the rendering performance. Developers for instance can use the knowledge to choose the technology combination, which provides the best rendering performance for a set of graphic primitives, styles and animations, which where chosen from a designer.

Based on these observations the research question for this work is: “Which aspects influence rendering performance of web-based animated data visualizations?”

3. Research Method

The research method used throughout this work is an experiment. The experiment's design is based on concepts from related work, which are explained in chapter 5. The design of the experiment consists of three ideas.

The first idea is to measure the rendering performance of aspect combinations of animated data visualizations. Such a setup, which contains all aspects of animated data visualizations, is packaged in a web experiment. A web experiment is a small web application, which renders a given number of graphic primitives, with a list of additional styles and a list of animated properties, in a browser, developed with a combination of web technologies. For every experiment iteration one aspect is changed. By changing just one aspect, the influence of the aspect on the rendering performance can be determined. To investigate the influence of a large number of aspects of animated data visualizations a lot of experiments have to be executed with various combinations of animated data visualizations aspects.

Second it can be said that graphic primitives in animated data visualizations have two states. Those are static and animated. A graphic primitive can be just rendered statically or it can be animated. As a result it needs two different rendering performance metrics. The first rendering performance metric is rendering time, which measures the time an experiment needs to render a number of graphic primitives with additional aspects of animated data visualizations. The second rendering performance metric is animation smoothness, which determines how smooth a given experiment is animating a given list of properties on graphic primitives. Both rendering methods are explained in detail in chapter 6.3 and 6.4. As experiments either measure rendering time or animation smoothness there are static and animated experiments.

The last idea of the design consists of how performance metrics are measured and experiments are run. Learning from the Mozilla Eideticker project, described in chapter 5.1, it makes sense to measure rendering performance based on the screen output and to calculate frame based performance metrics. This has two advantages. First, it enables seamless cross browser rendering performance testing. Second, the screen output is what really matters, because it’s what users perceive. Moreover the developed measurement system should run tests fully automated, as this makes it possible to investigate more combinations of animated data visualization aspects in the timeframe of this thesis. Also interactions with experiments should be run automated, because human interactions cannot produce exactly the same input repeatedly, which would result in test inconsistencies.

In order to execute and analyse results of a large amount of experiments an experiment system was developed.

4. DEFINITIONS

4.1. Performance of Web Applications

Animated Data Visualizations are subjects to the same principles of performance as web applications. Either they are integrated into a web application or they are developed standalone as a web application. As this work aims to predict the performance of web technologies used for animated data visualizations, there will be given a definition of performance of web application first.

There are different definitions of Web Application Performance. Grigorik describes performance as the page load time. (Grigorik 2013 p. 170) Google for example came up with a model called RAIL Performance Model (RAIL). This acronym stands for Response, Animation, Idle and Load. Those phrases are describing the performance aspects that a user of a web application notices. In RAIL a web application has good performance when it runs certain tasks within different time interval. As a consequence performance of web application can be defined as the ability to perform tasks with in predefined time intervals.

The smallest time interval is 16 milliseconds (ms). That’s the time an application has to bring a frame to the screen. When this requirement is not met, for example during an animation or when scrolling, the user will track variable frame rates and periodic halting. In fact there is only a window of 10 ms for the developer’s code to run for every frame, as the browser needs 6 ms for other tasks every rendering loop.

Another interval maximum is 100 ms When the users clicks or tabs on the screen there should be a reaction at least 100 ms later, at any rate the reaction should not last longer than 300 ms, which is perceived as a very short delay by the user.

The mark for page loading and changing views is 1000 ms. Beyond that time interval the user will lose focus. The absolute maximum however is at 10000 ms for content to load. After that time range the user is very likely to abandon the task. To meet those time limits, the four parts of RAIL should be considered separately.

First is Response, which is describes how reactive an application behaves. As mentioned above the time maximum for reaction to an input is 100 ms. Every response, like tabbing a button, toggling form controls or starting an animation that lasts longer, is noticed as lag.

The second aspect of RAIL is Animation. Animation is more than changing properties of elements over time. Also the visual changes that are caused by scrolling and panning gestures underlie the same time limits for processing frames to the screen.

Another aspect is Idle. It should be the goal to maximize idle time, in order to response fast to interactions by the user. As a result deferred background tasks should be divided into blocks that last at a maximum of 50 milliseconds.

Load is the fourth aspect of RAIL. Load does not only mean loading files, and optimizing networking, it also includes the time content is rendered to the screen. (Kearney 2016)

As Paul Lewis points out in a talk at the Google I/O conference in 2016, the RAIL Performance Model is used at Google to address the issues of performance in web applications. He also highlights that each aspect of RAIL is not equally important. It is the case that Load is seen as the most important, closely followed by Animation. However Idle and Response are considered as less important. (Lewis 2016)

In contrast web performance can also simply be perceived as the speed data is exchanged between the front end and the back end of a web application. In a book from Smith Peter about professional web performance, the focus is on optimizing server load, handling communication between server and database and optimizing the front end to load content efficiently. (Smith 2013). This definition of web performance is also described in a paper that analyses the main performance parameter of web applications. (Zhu and Li 2011)

In addition web application performance can be seen as optimizing the usage of web technologies, like HTML, CSS and JavaScript as well as caching, networking and the user experience. (Stefanov 2012)

To sum up web application performance is the duration components need to do certain tasks. When comparing definitions of web application performance from today with older ones, it needs to be said that the focused changed from only addressing the aspect of networking to taking also user interaction and animation into consideration.

4.2. Rendering Performance

Rendering Performance is the duration a web application needs for each step in the pixel pipeline of a web browser. Good performance means that a web application is optimized to need a minimal duration for each step. The processing of HTML, CSS and JavaScript in the pixel pipeline affects how fast content is displayed, how smooth animations are and how fast a web application reacts visually to user interactions. The pixel pipeline is executed every 16ms. In an ideal scenario this means that the browser refreshes the screen 60 times per second. The pixel pipeline consists of 5 steps in modern web browsers.

JavaScript: The first step in the rendering loop is interpreting and executing JavaScript code. Usually JavaScript is changing visual elements or loading content.

Styles: After a piece of JavaScript code, the browser loop continues with Calculating Styles. That is the process where the browser calculates the final style of every DOM element, by combining style information from the HTML element, external CSS sheets and JavaScript.

Layout: After the Browser knows the style of every element, he continues with the Layout phase. Layouting means calculating where every element is located on the screen and what the size is. This can take a lot of time, as elements influence each other’s size.

Painting: The next step of a rendering loop is Painting. This is the process of setting every pixel for each element. It involves drawing out text, colors, images, borders and shadows. Painting is done in multiple layers.

Compositing: This is the process of actually drawing multiple layers in the correct order to the screen.

(Lewis 2015b)

4.3. Animation in Data Visualizations

Animations are a series of images displayed to the user in rapid succession. Viewers of an animation assemble the images, trying to build a coherent idea of what occurred in between them. As the human perceptual system notes the changes between the frames of an animation, it can be said, that an animation is a series of visual changes in between frames. The same phenomenon occurs in the real world. Changes of items around us are perceived by the human mind as very small visual steps. These visual changes appear natural to us and let us understand our environment more deeply and richly. As a result animated things appear very familiar to us. It is important to mention that the human mind can track a maximum of five objects, which are changing the same way. An example would be objects on the screen, which animate the position towards the same direction.

Animations are used in data visualizations for various reasons. Animations can show transitions and intermediate steps. An example would be an animated GPS track which is data changing over time. Also animating data over time has been used in research. An example is the animation of a flowing wind stream. Moreover, animation can be used to animate the transition of views. This lets the user keep focus. Besides adding animations to data visualizations to let the user understand the data better, animation is also added to get the attention of the user. Moreover, animation is added to data visualizations to invoke emotions and to delight the user. However, animation in data visualizations always needs a certain purpose. There have been various studies on the effectiveness of animation in data visualization. They show that animations do not fit for all types of data visualizations. (Steele 2010 p. 329 -331)

An example is an article from Tversky, which shows that animation in process data visualization is not useful. For the article 100 studies on animation and visualization have been reviewed. The studies let the users answer questions on content, which is shown in different formats. The formats are text, data visualization and animated data visualization. Data visualizations were outperforming animated data visualizations, however user with animated data visualizations answered questions faster than user with textual data representations. (Tversky et al. 2002)

Another study, conducted by Hundhausen, Douglas and Stasko, investigated the effectiveness of animated process visualizations for learning algorithms. The visualizations where used to animate the steps of algorithms like bubble sort and insertion sort. A group of pupils were divided into two groups. One group had access to animated data visualizations, while the other group had access to not animated data visualizations. The pupils could play around with the code of the algorithms and code changes where represented in the data visualizations. The result was that animations were very effective, when the pupils tried to learn the algorithms, by changing the code. In another group, pupils were supposed to only watch the algorithms visualization to change. Here, animated data visualizations did not achieve better learning results for the pupils, compared to static visualization changes. (Hundhausen et al. 2002)

Also a study at Microsoft Research compared animated and not animated data visualizations with user tests. They implemented two different data visualizations, which show the correlation of life expectancy and infant mortality of countries in 2D scatter plots. To explore the values over time, the animated data visualization had a slider, which enabled the user to move the time point. A time point change animated the data points to the new position. While the animated data visualisation showed the data for only one time point, the visualisation with no animation displayed the data from all time points. The points in the scatter plot had a lower level of translucency to indicate movement of the points. People were asked to explore the two types of data visualizations, and were supposed to answer questions on the data afterwards. The results showed, that non-animated data visualizations enable people to answer questions faster and more accurate. However, people told that is was more enjoyable to explore data with animated data visualizations. (Robertson et al. 2008)

To sum up, animations in data visualization have advantages and disadvantages. It can be said that animation is not suitable for data visualizations, which are supposed as a tool to explore data. However, animation is suitable for visualizing view changes, as it keeps the user focused. Moreover, animations are useful when data is only presented to the user. Animations in data visualization enhance the user's experience, because they are more enjoyable and appear more natural to humans, compared to non-animated data visualizations.

5. RELATED WORK

5.1. Mozilla Eideticker

Eideticker is an open source project at the Mozilla Corporation. It is a rendering performance benchmark tool, to test new releases for the Firefox Browser on Android. It was also developed to compare new rereleases of Firefox to competing mobile browsers Google Chrome and Opera. The project takes a new approach to measure rendering performance. Instead of measuring how fast an application is able to render or process data with browser tools, it captures the HDMI output of the device during a testing session and analyses the video frames. The reason is that internal measurements in the browser are loosely correlated to what the user perceives. Another reason is that is it hard to translate rendering performance tests across browsers.

There are three different metrics that are created with the tool. The first one is unique frames. This metric is used to test how smooth a web application renders during a time interval. One example is measuring how smooth the browser responds to a pan gesture. It can be said that the more unique frames are created by the browser during a pan gesture, the smoother the visual response to the gesture is.

Another metric is the size of screen parts, that are not fully drawn for a frame. This is called checker boarding. When the rate of changing frames is very high, it sometimes occurs, that the browser is not able to finish rendering one frame until the next frame appears on the screen. In this situation, the browser will leave the bottom part of the frame blank. The amount of area, which is not rendered for one frame, can be recognized within the video output and measured. This works by looking for areas that have a certain colour. The project uses sometimes slightly changed versions of testing websites, where they changed the body background colours in order to recognize undrawn areas. The lower the amount of undrawn areas within the frames, the better the rendering performance at checker boarding is.

The third metric the Eideticker project uses is start-up and load time. To measure how fast a web application initially renders, there can be used browser internal callbacks, like the *onload* method. However, using a method to determine start-up time based on frame analysis has two advantages. The first is that results are based on the screen output the user perceives. The second is that the initial rendering of a web page can be replayed as a video and can be analysed. This is interesting as it can be seen in which order the browser renders elements. (Mozilla 2013)

5.2. Web Technology Comparisons

There are papers, which developed the exactly same data visualization multiple times with different web technology combinations and frameworks. Then the performance of those visualizations was compared.

To determine the rendering performance of various data visualization frameworks, Lee, Yo and Kim developed the exactly same visualization four times with different frameworks. The frameworks were Google Charts, Flex, OFC and D3. In the implementations they measured the metrics, layout time, data transformation time and rendering time. The performance tests for each implementation were conducted on increasing large data sets from 100 to 100.000 points. The tests were executed in the Google Chrome browser and the metrics where measured with Chrome Developer Tools. (Lee, Jo and Kim 2014)

A different paper proposes the approach to test various combinations of web technologies on their rendering performance and interactivity for one specific data visualization. There was a 2D parallel coordinate visualization developed with the web technology combinations of SVG, 2D Canvas and SVG, WebGL and SVG, Processing.js 2D and SVG, Processing.js WebGL and SVG, and the Kinetic.js framework. The tests where conducted in Google Chrome. The performance of the visualization was determined by measuring the frames per second the visualization was able to render with an increasing data set size of 500 to 10000 points. (Kee, Salowitz and Chang 2012)

5.3. Problems with Existing Performance Tests

First or all existing performance tests in research and on the web performance tests are not applicable to predict the performance for a large number of visualizations. This is because existing performance tests only predict the performance for very specific visualizations. At both examples, explained in chapter 5.2, tests where conducted for one specific visualization, that gives no performance prediction for other types of data visualizations.

The second disadvantage is a lack of quality in the performance test results. Tests that evaluate the effect of manual triggered interactions with the visualization will show different results for every iteration, as humans are not able to perform an interaction exactly identical several times. An example is a paper from Kee, which has the goal to detect the maximum amount of mouse over events, a web technology can detect. In order to do so, the mouse is manually moved over the visualization and the amount of callbacks is detected. (Kee, Salowitz and Chang 2012) The problem is that the manual mouse over interaction will be slightly different every iteration, which leads to varying test results.

In addition, existing approaches run performance tests only on one browser. But there are five browsers that are commonly used on the Internet, which are Google Chrome, Firefox, Safari, Opera and Internet Explorer. Three browser usage statistics from May 2016 show all different usage percentages for browsers, because they use browser usage data from different pools of websites. Nevertheless, in all three statistics these five browsers are the most used browsers. (Clicky 2016), (Netmarketshare 2016), (W3Counter 2016) Also those five browsers render content with different engines. Chrome and Opera use Blink. (Bright 2013), (Protalinski 2013) Firefox ships an engine called Gecko. (Mozilla 2016) Safari uses Webkit (Apple 2012) and Internet Explorer contains Trident. (Microsoft 2016) As a consequence rendering performance varies across common used browser. Running rendering performance tests for just one browser gives a narrow view on how rendering performance appears to users on the Internet.

Last, determining the rendering performance by measuring code execution time in the source code or with browser tools delivers results, which give some indication on the rendering performance. However real screen output of browser application is what the user perceives and cares about. As a result it would be better to test the rendering performance based on screen output, because it is closer to what matters to the user compared to code execution times. Also this technique enables rendering performance testing uniformly on different browsers.

6. EXPERIMENT SYSTEM

As explained in chapter 3, the research method, which is used through out this work is an experiment. An experiment consists of a little web application, which con

6.1. Functionalities

The system is able to run experiments. To start an experiment an Experiment object is constructed. The object contains a link to the web experiment. Also browsers and the drivers for each browser are collected. Afterwards the number of iterations and the increasing amount of elements the web experiment is run with are specified. Then the experiment is executed. First links to the web experiments are retrieved from the experiment objects. Then the program iterates through different browsers. Google Chrome, Firefox and Opera. For every browser the web experiments are run with increasing large numbers of elements. When the experiment is not animated the page is just loaded. When the experiment is animated, the animation is run in the browsers. While the experiments are running in the browser, the screen output is recorded in the background. After one experiment has been run, frames of the screen output are processed and rendering performance metrics are calculated. When the experiment is static the metric rendering time is determined. For animated experiments the animation smoothness metric is calculated. Afterwards the results and details of the experiment are stored in the database.

The system also has a GUI that enables the user to explore experiment results. It contains tables with experiment parameters type, date, style, and animated properties. The GUI contains a button, which triggers a chart build from experiments, specified by selected parameters in the tables.

6.2. Architecture

The architecture of the system is shown in the UML Deployment Diagram in figure 1. The system is run on a Mac Book Pro with a 2.3 GHz Intel Core i7 processor. It has 16 GB 1600 MHz memory and a NVIDIA GeForce GT 750M graphics card. The Mac Book runs OSX 10.10.5 Yosemite. The system contains five main components, which are a java application, a collection of browsers, a database, web experiments and the FFMpeg framework. First the single internals of the components will be examined, followed by a description of the components interplay.

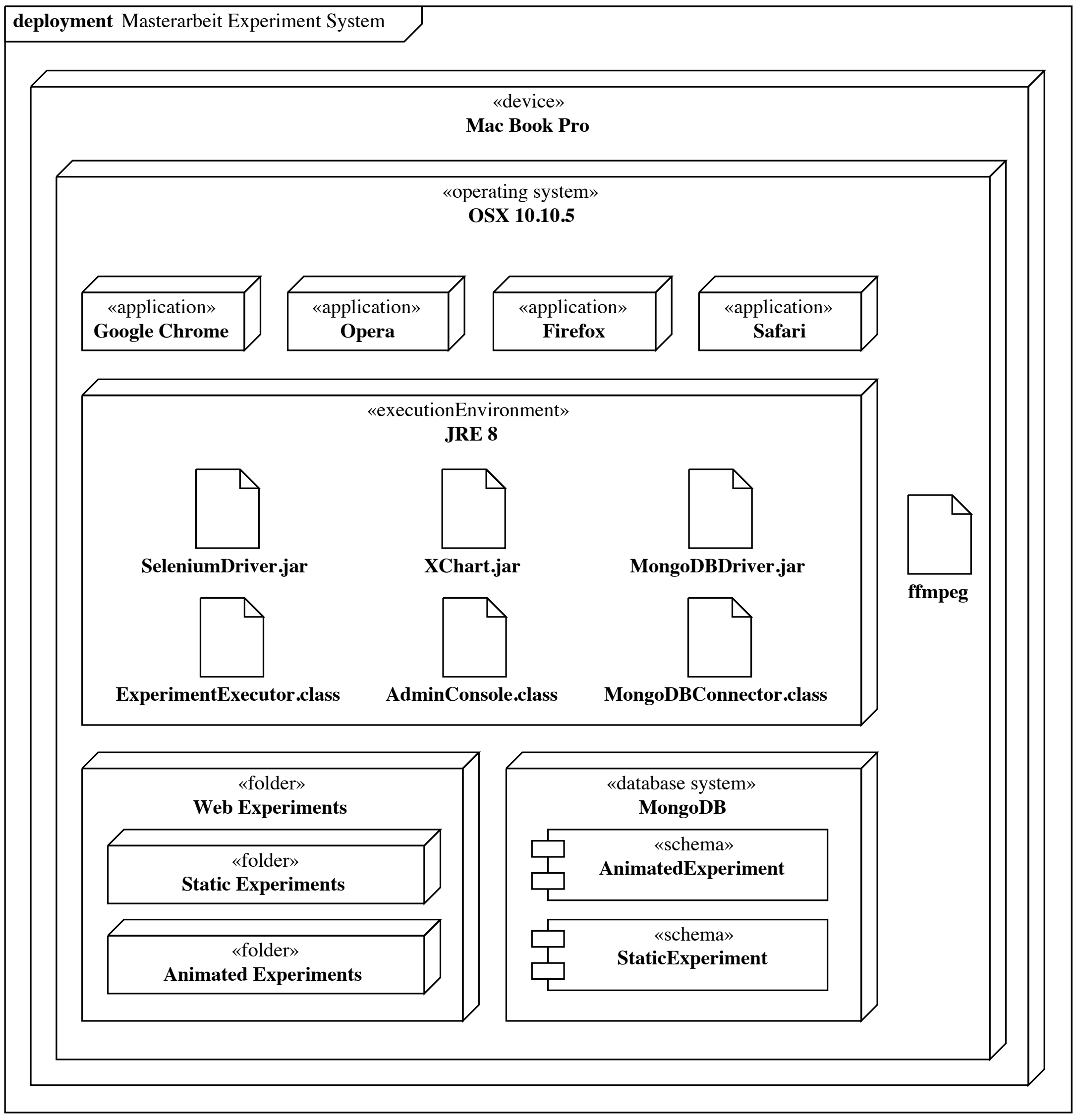

Figure 1: UML Deployment Diagram - Experiment System

6.2.1. Java Application

The java application is the core of the system. It has three main classes, which coordinate the internal operations of the application.

The *MongoDBConnector* class uses the MongoDBDriver.jar to access a local MongoDB database. It has public methods to insert *StaticExperiment* and *AnimatedExperiment* objects into the database. It has also public methods to retrieve a collection of type *StaticExperiment*, which match a *GraphicPrimitive.* To retrieve a list of type *AnimatedExperiment*, the *MongoDBConnector* class has a method that takes a *GraphicPrimitive* and a collection of type *AnimatedProperty* and returns a collection of type *AnimatedExperiment*, whose animated experiments match the criteria.

The *ExperimentExecutor* class coordinates experiments. Detailed descriptions of the process how static experiments and animated experiments are run are given in chapter 6.3 and 6.4. The class uses the Selenium framework to steer the browsers. The Selenium framework is a set of software tools with different approaches for browser automation. To use the framework, browser drivers have to be handed to the framework. Those browser drivers are located in the operating system file systems. Then the framework creates a web driver, on which certain browser operations, like open, close and loading URLs can be called. (Selenium 2016) The *ExperimentExecutor* is depending on the *MongoDBConnector* class, as it processes executed *Experiment* objects with their *ExperimentResult* objects to store in the database.

The purpose of the *AdminConsole* class is to create a GUI. The functionalities of the GUI are described in chapter 6.1. The admin console uses the XChart framework to display line charts that visualize results from experiments. XChart is a simple charting library for Java, based on the AWT framework. It provides a range of different 2D chart types and supports saving charts as images. (XChart 2016) The admin console class also retrieves data from the database with the

MongoDBConnector class. The data is displayed in the single selection tables of the GUI and in experiment result charts.

6.2.2. Browsers

Web experiments are executed in the Google Chrome, Firefox and Opera browser. The version of Google Chrome is 51.0.2704.103. Firefox runs on release 46 and Opera on version 38. The browsers were chosen by their frequency of use on the Internet according to browser statistics mentioned in chapter 5.3. However, according to the statistics the most used browsers, ordered by usage, are Google Chrome, Firefox, Internet Explorer, Safari, Opera and others. The three most used browsers should be chosen for the experiment system. Those would be Google Chrome, Firefox and Internet Explorer. As the experiment system is run on OSX, Internet Explorer is not available. As a consequence the next browser in the chain is Safari. The reason not to run the web experiments on Safari is that there is currently a bug in the latest version of Safari, which avoids it to run the selenium driver properly. (Github 2016) One possibility so solve this problem, would be to use an older version of Safari. This was not the choice in this work, because it would have meant to change the other browser versions to equivalent older releases as well in order to have comparable results. As a consequence it was decided to use the Opera browser as the third browser for the experiment system to get experiment results based on latest browser versions.

6.2.3. Web Experiments

The Web Experiments folder contains web experiments, which are small web applications. The folder contains two subfolders, Static Experiments and Animated Experiments. Web Experiments are small web applications that render graphic primitives in different combinations. Each web experiment has an URL parameter, which specifies the number of graphic primitives rendered. Web experiments are separated into static and animated. Static web experiments render a specified number of graphic primitives. Animated web experiments first render graphic primitives and run an animation on the graphic primitives, when a button is clicked. Static web experiments also differ in the graphic primitives they render and the web technology combination used to implement the web experiment. Animated web experiments differ, beneath the graphic primitives and the web technology combination in their properties they animate on the graphic primitives.

6.2.4. Database

MongoDB is the database of the system. MongoDB is a free and open source database. Also MongoDB is a NoSQL database, which means that it does not store data in a table based relational model. Instead the database uses documents with a similar format to JSON and dynamic schemas. The format in MongoDB documents is called BSON. Compared to a relational database, documents conform to table entries. Documents are stored in collection, which would be a table in a relational database. (MongoDB 2015) The system database schema contains the two collections *StaticExperiment* and *AnimatedExperiment*. Both collections have exactly the same structure, like the data type objects used in the application layer. To save an *Experiment* java object in the database, the object is converted into a JSON string. The MongoDB java driver API can handle JSON strings and stores them into a collection as a document. To convert java objects into JSON strings, the application uses the Gson library. Gson is a Java library for converting java objects into JSON strings and back. (Inderjeet Singh 2011) To retrieve data from the database, experiments are queried as JSON Strings and converted into java objects. This enables seamless communication between the java application and the database.

Figure 2 shows the data model of the application as class diagram. These data type objects are used inside the application to process data and the same structure is used to store data in the database. At the core the Experiment class exists. An experiment has a path to the web experiment it executes. Depending on the type of the web experiment the *Experiment* object is of type *AnimatedExperiment* or

StaticExperiment. A *StaticExperiment* object has fields that describe the characteristics of the static web experiment. Those are the graphic primitive and the web technology combination used. Beside the graphic primitive and the web technology combination used, an animated experiment has a field that holds an *Animation* object. The enum *WebTechnologyCombination* has a list of types of the enum *WebTechnology*. An *Animation* object has a list of various *AnimatedProperty* types that are animated at a graphic primitive.

Also *Experiment* objects have a reference on the browser they are executing the web experiment. As static and animated experiments require different recording times during testing, the *StaticExperiment* and *AnimatedExperiment* objects have a constant of the recording time. More over the an experiment more than one experiment result, because web experiments of an experiments are run multiple times in a browser with increasing large element numbers. *StaticExperimentResult* and *AnimatedExperimentResult* objects have an abstract super class that contains the number of objects of the web experiment. Also the abstract *ExperimentResult* class holds a reference on the number of different pixels to the next frame for each frame, which was recorded during the measurement session. As there are two different rendering performance metrics used for the research method, it needs two different specifications of the *ExperimentResult* class. *StaticExperimentResult* has a field that with the render time in frames for a *StaticExperiment*. *AnimatedExperimentResult* has a field of the identical frame rate per 60 frames. Both are metrics that are either used to determine the rendering performance of static and animated web experiments. The calculation processes for those metrics are described in detail in chapter 6.3 and 6.4.

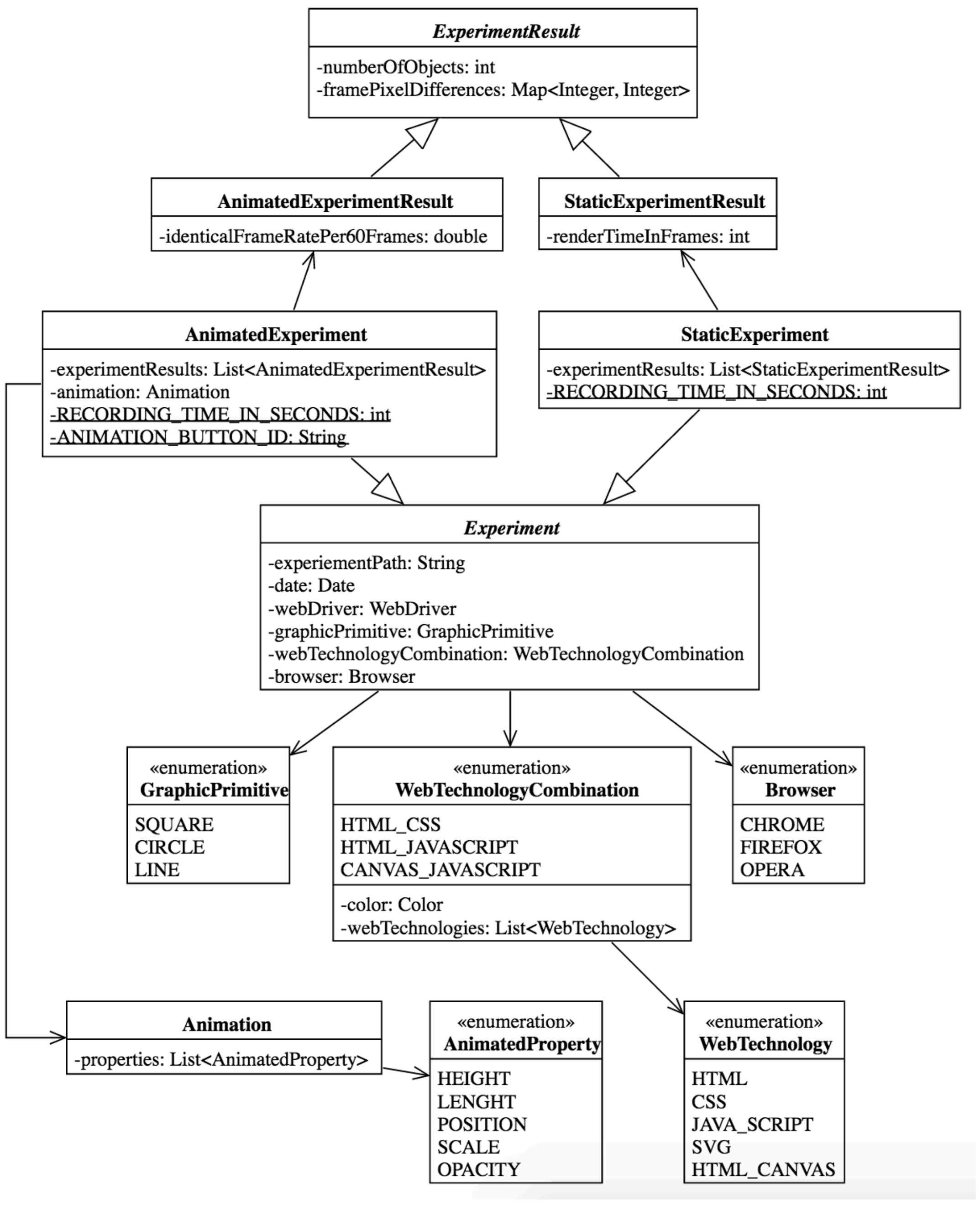
ExperimentResult
-numberOfObjects: int
-framePixelDifferences: Map<Integer, Integer>
AnimatedExperimentResult
-identicalFrameRatePer60Frames: double
StaticExperimentResult
-renderTimeInFrames: int
AnimatedExperiment
-experimentResults: List<AnimatedExperimentResult>
-animation: Animation
-RECORDING_TIME_IN_SECONDS: int
-ANIMATION_BUTTON_ID: String
StaticExperiment
-experimentResults: List<StaticExperimentResult>
-RECORDING_TIME_IN_SECONDS: int
Experiment
-experiementPath: String
-date: Date
-webDriver: WebDriver
-graphicPrimitive: GraphicPrimitive
-webTechnologyCombination: WebTechnologyCombination
-browser: Browser
«enumeration» GraphicPrimitive
SQUARE
CIRCLE
LINE
«enumeration» WebTechnologyCombination
HTML_CSS
HTML_JAVASCRIPT
CANVAS_JAVASCRIPT
-color: Color
-webTechnologies: List<WebTechnology>
«enumeration» Browser
CHROME
FIREFOX
OPERA
Animation
-properties: List<AnimatedProperty>
«enumeration» AnimatedProperty
HEIGHT
LENGHT
POSITION
SCALE
OPACITY
«enumeration» WebTechnology
HTML
CSS
JAVA_SCRIPT
SVG
HTML_CANVAS

Figure 2: UML Class Diagram - Experiment System Model

6.2.5. FFMpeg

FFMpeg is a collection of command line tools and libraries to record convert and stream audio and video of different formats. In the measurement system, the FFMpeg command line tool is used to record the screen output. The advantage of the FFMpeg tool is that is written in low-level C. This enables the tool to access directly the graphic component of the operating system. This is the component avfoundation at OSX. As a result the tool is able to record the screen at 60 frames per seconds, which is not possible with other solutions.

6.2.6. Interplay of components

The Java application is at the centre of the system. The components web browsers, web experiments, database and FFMpeg are not aware of each other. Only the java application has references to those components and steers them in order to execute experiments.

The browsers are controlled by the Java application with the Selenium framework. References to drivers for the Opera and Google Chrome browser are store in enums of the Java application. There is no reference to the Firefox driver, as the Selenium framework bundles contain and reference the driver internally. The java application steers the browser with the web drivers. For every web experiment execution a new WebDriver object is instantiated. The java application steers the browser with the WebDriver object, which includes opening and closing a browser window, making a window full screen, opening a local web experiment and clicking a button.

The web experiments are connected to the java application via hard coded folder paths, which are stored inside the enums *StaticExperimentType* and *AnimatedExperimentType* as member variable. When the Java applications starts an experiment, the path to a web experiment is retrieved from an enum and called in a browser with a WebDriver. It is important to say, that the number of elements which are rendered in a web experiment, is defined by the URL parameter *elementsNumber*, which every web experiment has. As a consequence, the number of elements is defined by the Java application, when the web experiment is executed in a browser.

The Java application communicates with the MongoDB via the MongoDB Java Driver. There are three scenarios when the Java application communicates with the database. First, Experiment objects are stored in the database, after an experiment was completed. Experiment objects are converted into JSON Strings, which are then store in the database with the MongoDB Java Driver. As mentioned in chapter 6.2.4, the Java application uses the Gson framework to convert Java objects between JSON Strings back and fourth. The second scenario is the retrieving of Experiment objects from the database, when the Admin Console is instantiated to show the collection of experiments for selection. Third, a chart with experiment results is build from the Admin Console. Here a certain Experiment is retrieved from the database, which meets the criteria set by the user in the Admin Console.

The FFMpeg framework is a collection of command line tools as described in chapter 6.2.5. Several commands are used by the Java application in the system. Commands are executed from the Java *Runtime* class. Every command starts a process whose output stream is collected and printed in an extra thread.

6.3. Measuring Rendering Time

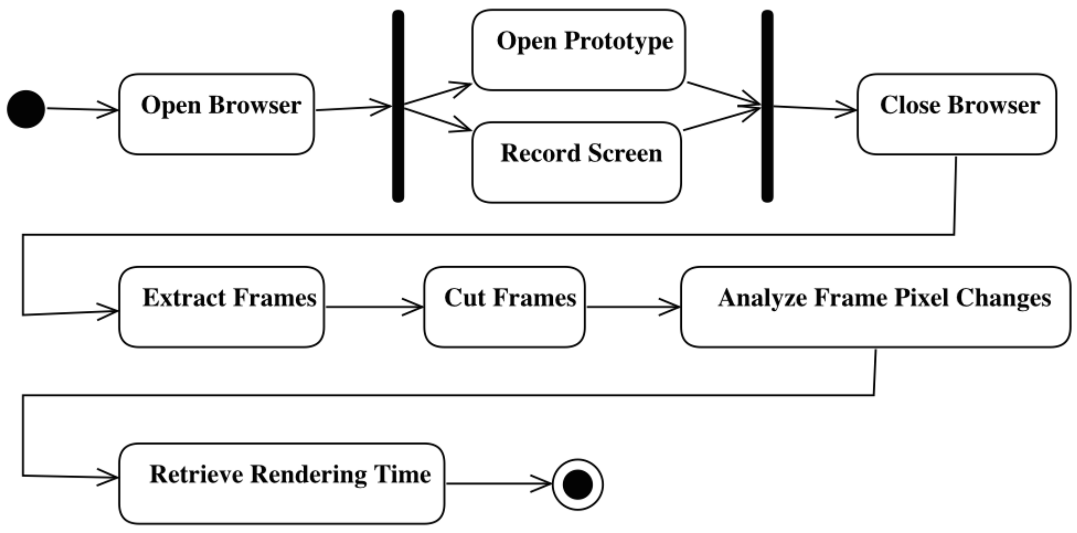

Figure 3: Measuring Rendering Time Process

To determine the rendering time of static web experiment with a given number of elements the following steps are executed in the program. These steps are also visualized in the activity diagram in figure 3.

First a browser of the current iteration is opened with one of the selenium drivers from Google Chrome, Mozilla Firefox and Opera. Also the browser window is set to full screen, to test the rendering of more objects, which are simultaneously displayed on the screen.

Second the main thread creates a new thread that is responsible for starting the screen recording process. The screen recording thread creates a new process, which records the screen for a given duration with the FFMpeg command line tool. The command line command, which records the screen output, is then executed with the java *Runtime* class. The screen recording thread waits for the screen recording process to finish by calling the *waitFor* method on the java *Process* class. That does not terminate the screen recording thread until the screen recording process has terminated. This is important because the main thread joins the screen recording thread, because it should continue only when the screen recording finished. After starting the screen recording thread, the main thread clicks the button, which triggers the adding of graphic primitives, with selenium. The button changes the background color on click, which will be recognized later at frame analysis as the start of the rendering time. Afterwards the main thread joins the screen recording thread.

In the third step the browser is closed. It is important to wait to close the browser until the screen recording process has finished, as closing a window is causing pixel changes on the screen, which would disturb the measuring results.

Afterwards the frames of the screen output video are extracted into single images. Similar to recording the screen output, this is done with the FFMpeg framework. A new thread is created and started that runs a command line process that extracts the frames into a folder. Again the main thread is only allowed to proceed running, when the frame extraction process is terminated.

In step five, a bar from the top of the frames, which shows the operating system bar and the browser bar are cropped. The reason for this is, that the clock at the operating systems bar and changes at widgets in the browser bar during the screen recording process are causing pixel changes. These pixel changes disturb the result. Afterwards an array is created, that contains the amount of pixels that have changed from one frame compared to the next frame. The program iterates though all pixel of a frame and compares every pixel with the color values of the pixel of the next frame at the same position. If the color values are different, a variable that counts the different pixels to the next frame is incremented. The amounts of different pixels to the next frame are then stored in the array for every frame, except the last frame.

Afterwards the array of different pixels to the next frame is used to calculate the rendering time in frames. The rendering time is the frame amount between the frame with first pixel changes and the frame with the last pixel changes. This is based on the idea that the page in the browser has finished to render, when pixel stop changing on the screen. The start of the rendering time is triggered by the visual feedback of the button on the page, which triggers the adding of graphic primitives.

6.4. Measuring Animation Smoothness

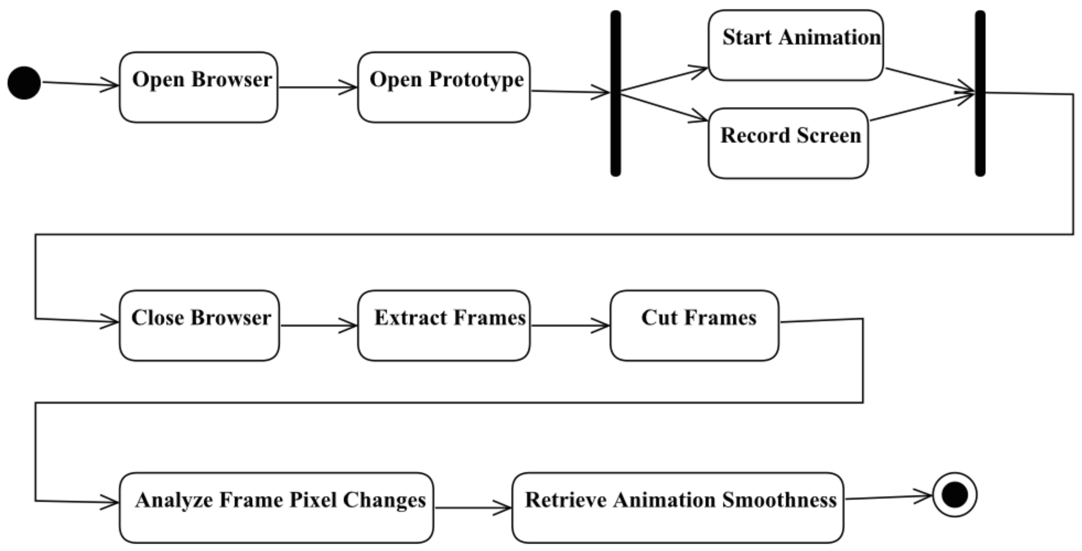

Figure 4: Measuring Animation Smoothness Process

The process of measuring animation smoothness is slightly different to the process of measuring rendering time. A visualization of the process is displayed in an activity diagram in figure 4. First the metric animation smoothness is calculated differently compared to the metric rendering time. Also the interplay of browser and screen recording process is different.

First a browser window is opened. Afterwards the prototype is loaded in the browser. The prototype contains a button that with no visual effects that starts the animation of the prototype on click.

After the prototype is loaded, a screen recording thread is created and started. When the screen recording process runs, the program triggers the animation by clicking the button with the selenium driver, which starts the animation. When the screen recording process has finished, the main thread continues to extract the frames from the screen output video, crops the frames to remove pixel disturbance and analyses the pixel changes between the frames, like in the rendering time measurement process.

Afterwards the metric animation smoothness is calculated. Animation smoothness is the amount of frames with no pixel changes per 60 frames. Modern browser render content with 60 frames per second. This means that the browser has 16 milliseconds seconds to render one frame. In fact there are 10 milliseconds available to code from the developer, because browsers need 6 milliseconds for internal operations. When the code from a website needs more than 10 milliseconds per rendering loop, the browser skips the current rendering. This results in no pixel changes on the screen for the current frame. When an animation is executed in the browser, it would be an ideal situation to render pixel changes every at every frame during the animation. However this is not the case in practice. The more complicated an animation is, the more time the browser needs to render changes for every frame. As a consequence it can be said, that the more complicated an animation is, the more rendering loops occur, where rendering is skipped, which results in frames with no pixel changes.

For calculating the metric, the first and last frames with pixel changes are determined. These frames mark the frames range, where the animation did happen. Then the duration interval of the animation is defined by the index of the last frame with pixel changes minus the index of the first frame with pixel changes. Afterwards the amount of frames with no pixel changes in the duration interval is divided by the duration interval. Then the result is multiplied with 60. The formula for the animation smoothness metric is shown in figure 5, where AF is the amount of frames with no pixel changes during the animation duration interval. IF is the index of the first frame with pixel changes and IL is the index of the last frame with pixel changes.

*Animation Smoothness = (AF/(IL – IF)) * 60*

Figure 5: Animation Smoothness Formula

An example of the animation smoothness of an animation is visualized in figure 6. The screen was recorded for one second. The index of the first frame that has pixel changes is 20. The index of the last frame with pixel changes is 45. As a result the animation duration is 25 frames. The number of frames with no pixel changes is 17. The animation smoothness is 17/25*60, which is 40,8.

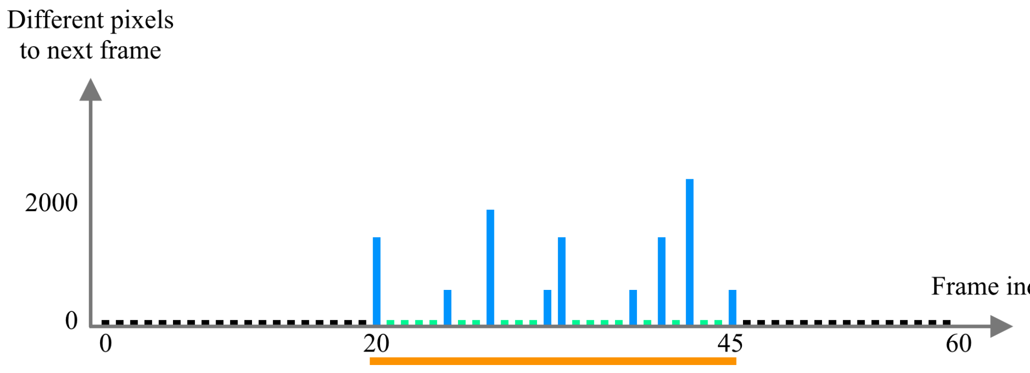

Figure 6: Animation Smothness Example

7. EXPERIMENT EXECUTIONS

To measure the influence of animated data visualization aspects on rendering performance, web experiments with different combinations of aspects will be developed. Then, for each web experiment, an experiment will be created, which runs the web experiment in a given browser multiple times with different numbers of graphic primitives. Experiments are divided in static and animated experiments, depending whether their web experiment shows displays static graphic primitives or an animation on graphic primitives. In addition, static experiments will be grouped for execution by the style of graphic primitives in their web experiment. Animated experiments are grouped by the animated properties in their web experiments. The results for every experiment will be presented. Also key code parts of the rendering of graphic primitives for each web technology combination will be explained.

During testing no static were recognized from the experiment system on the experiment objects. After running static and animated experiments multiple times, there could not be any greater variances be detected at static experiments and animated experiments running in the Firefox and Opera browser. However there were variances in performance recognized by web experiments run in the Google Chrome browser. It is not sure whether variances are caused by the performance system or rendering performance variances inside Google Chrome. The effect is discussed in detail in chapter 8.3

7.1. Static Graphic Primitives

The goal of this chapter is to investigate the rendering time of static graphic primitives, square, circle and line, and the influence on rendering time of additional CSS attributes. Three experiments will be run. First, the rendering time of graphic primitives without additional CSS attributes will be tested. Then, two experiments with additional CSS attributes opacity and shadow, added to the graphic primitives, will be run. Every experiment contains nine static web experiments. For every graphic primitive, three visually identical web experiments will be developed with the technology combinations of HTML with JavaScript, SVG with JavaScript and Canvas with JavaScript. All web experiments in an experiment are run with an increasing number of graphic primitives ranging from 1 to 10000 with 1000 steps in between. Moreover, all web experiments are run in the Google Chrome, Opera and Firefox browser. The rendering time for a web experiment iteration is measured in frames with a frame rate of 60 frames per second. The metric is described in chapter 6.3.

7.1.1. Static Web Experiments

Every static web experiment uses the same code to setup. First absolute position values are calculated for a given number of graphic primitives. These position values are later used to set the position absolute to graphic primitives in all static web experiments. Although there are web technology combinations, like HTML with JavaScript, which are able to position elements via layout in the DOM, there is also the web technology combination of Canvas with JavaScript, which only enables developers to position elements with absolute values. As it is the goal in these experiments to get results purely based on rendering time not mixed with layout time, graphic primitives are positioned absolute in all web experiments.

Also every static web experiment contains a button, which triggers the rendering process for a given number of graphic primitives. To render graphic primitives, the code iterates over all positions, which were calculated previously, and adds a graphic primitive at the provided position.

7.1.2. Rendering Time of Graphic Primitives

In this chapter the rendering time of graphic primitives square, circle and line are investigated. For that, nine web experiments have been developed. Three visually identical web experiments for every graphic primitives with the web technology combinations of HTML with JavaScript, SVG with JavaScript and Canvas with JavaScript. The results of the rendering tests are presented followed by an explanation core rendering code pieces. Figure 7 shows the results for the graphic primitive square figure 8 for circle and figure 9 for line.

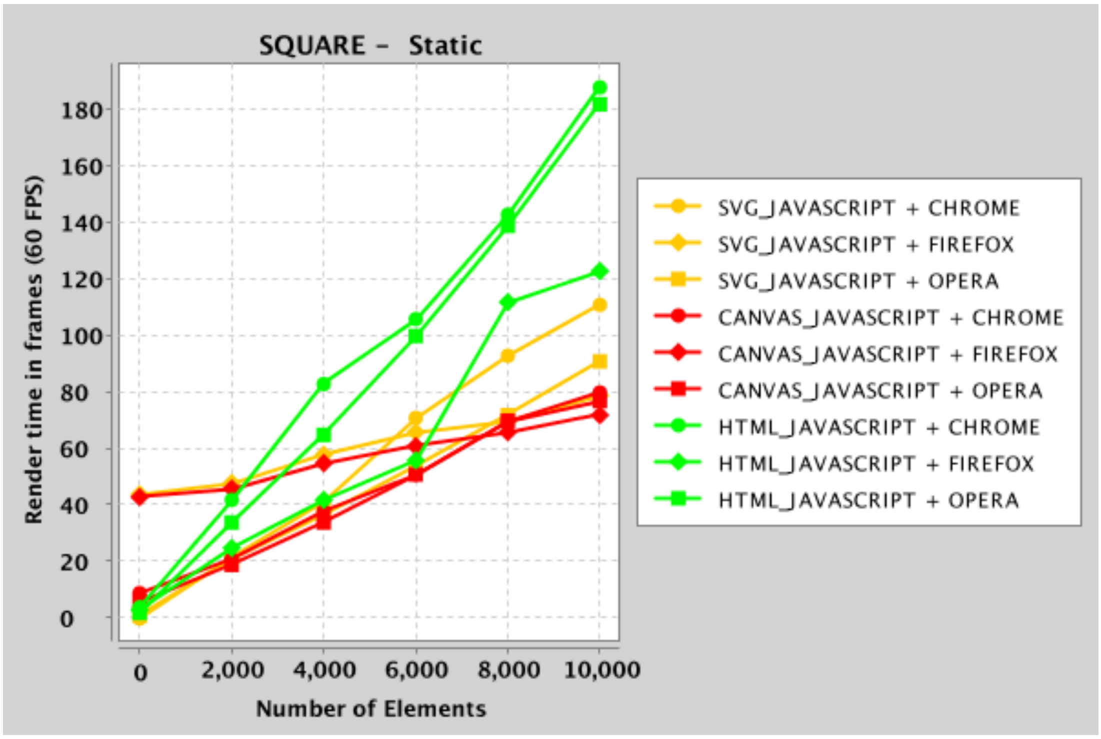

Figure 7: Render Time in Frames, Squares

At rendering one square, there is very little difference in rendering time between most browsers and web technology combinations. However, the Firefox browser needs 43 frames to render one square with SVG and Canvas, which is significantly longer, than the performance of other browsers and web technology combinations. In contrary, the Firefox browser outperforms the browser Google Chrome and Opera, at rendering 10000 squares. Here, the Firefox browser achieved the best rendering performances at all three measured web technology combinations. Also it needs to be said that for rendering large amounts of elements, the web technology combination of Canvas with JavaScript executed in the Firefox browser achieves the best rendering times. The slowest rendering time had the web technology combination HTML with JavaScript, when executed in the Google Chrome and Opera browsers at 183 frames. This is one second longer than the Firefox browser which renders 10000 elements with the web technology combination of HTML with JavaScript in 122 frames. The rendering times at web experiments for circles and lines are almost identical with the rendering times at all web experiments for squares.

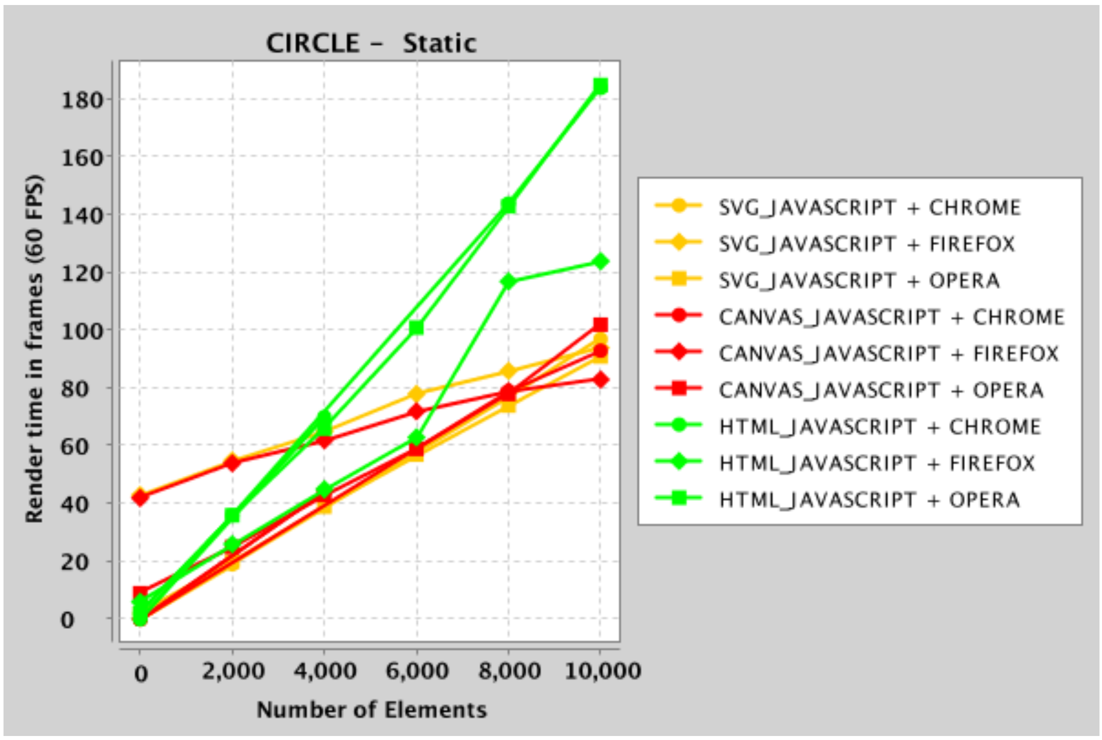

Figure 8: Render Time in Frames, Circles

The web technology combination of HTML with JavaScript renders squares by iterating over an array of positions. For every position, a div element is created and added to the DOM. The position of the element is set absolute with the CSS attributes top and left. For adding circles and lines, the process is identical. The different is that circles are div elements with a CSS border radius applied to it. Lines are very thin div elements.

The SVG renders graphic primitives by creating SVG rect, circle and line elements. The position is set via SVG attributes. All elements are added to a SVG container.

At web experiments created with the web technology combination of Canvas and JavaScript, there is first a 2D canvas element created. Then the code iterates over an array of positions and draws a rectangle to the canvas context with the *rect* method. Circles are added with the arc method. Lines are created with the *moveTo* and *lineTo* methods.

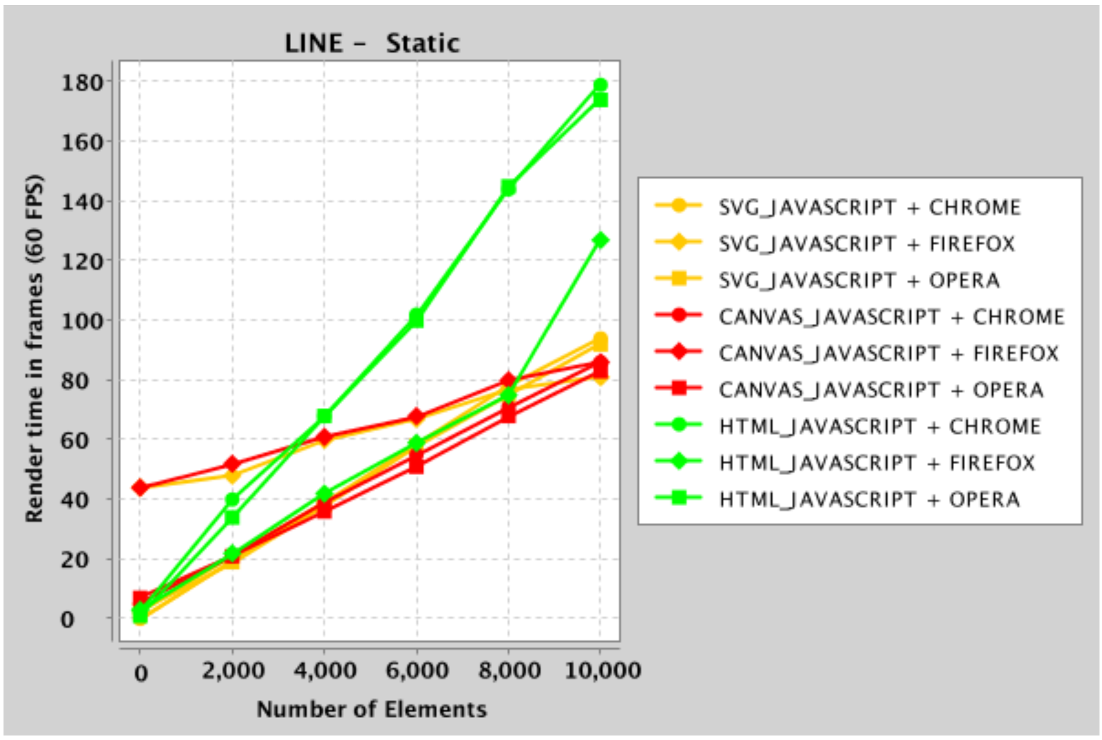

Figure 9: Render Time in Frames, Lines

7.1.3. Rendering Time of Graphic Primitives with Opacity

One styling attribute, which is very common in current data visualization solutions, is different levels of opacity. In this chapter it is investigates, how the setting of the opacity value affects the rendering time of graphic primitives, square, circle and line. There have been nine web experiments developed which display graphic primitives with a set opacity attribute of 0.5. For every graphic primitive there are three visually identical web experiments, developed with the web technology combinations of HTML with JavaScript, SVG with JavaScript and Canvas with JavaScript. Figure 10 shows the rendering times of squares with a set opacity value. Figure 11 shows the rendering times of circles and figure 12 shows lines.

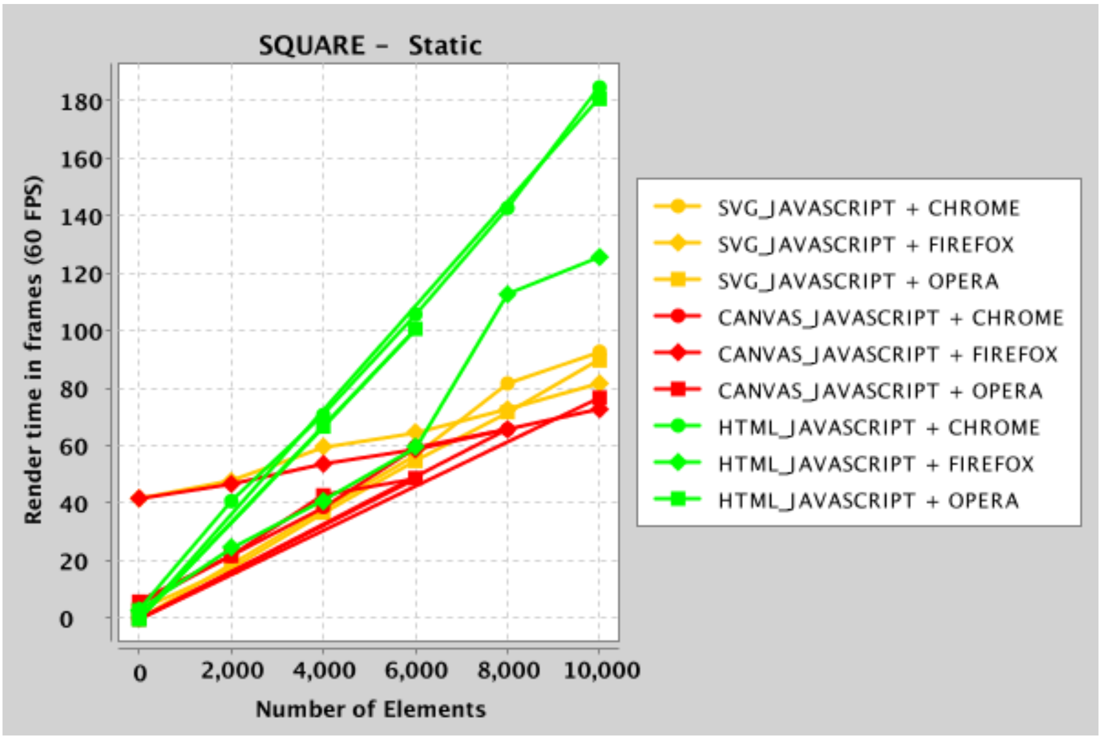

Figure 10: Render Time in Frames, Squares with Opacity

The results show that rendering time is not hugely affected by setting the opacity attribute. Even at large element numbers the rendering time does not change significantly. The difference in browser performance is about 60 frames between the fastest rendering time and the slowest performance. With increasing large datasets the rendering performance rises linear. It can be said that the rendering time does not significantly rise with the setting of the opacity at all graphic primitives and all web technology combinations.

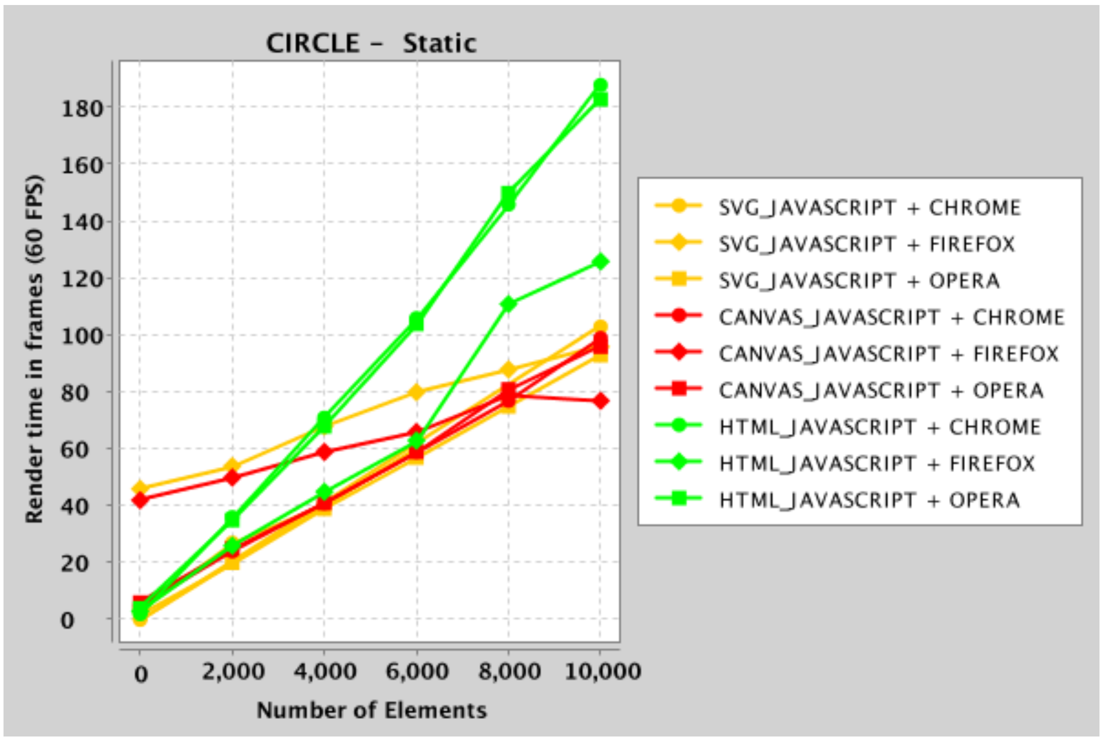

Figure 11: Render Time in Frames, Circles with Opacity

The code for displaying graphic primitives with a set opacity values is almost identical to the code of web experiments explained in chapter 6.1.2. However the difference is in setting the opacity for each graphic primitive.

For the web technology combination of HTML and JavaScript a CSS class was added which opacity attribute with value of 0.5. This class was used for all graphic primitives. At web experiments developed with SVG and JavaScript, the opacity was set by adding the fill-opacity attribute. The opacity value for graphic primitives drawn with the web technology combination of Canvas and JavaScript was set with the *fillStyle* method for squares and circles and with the *strokeStyle* method for lines.

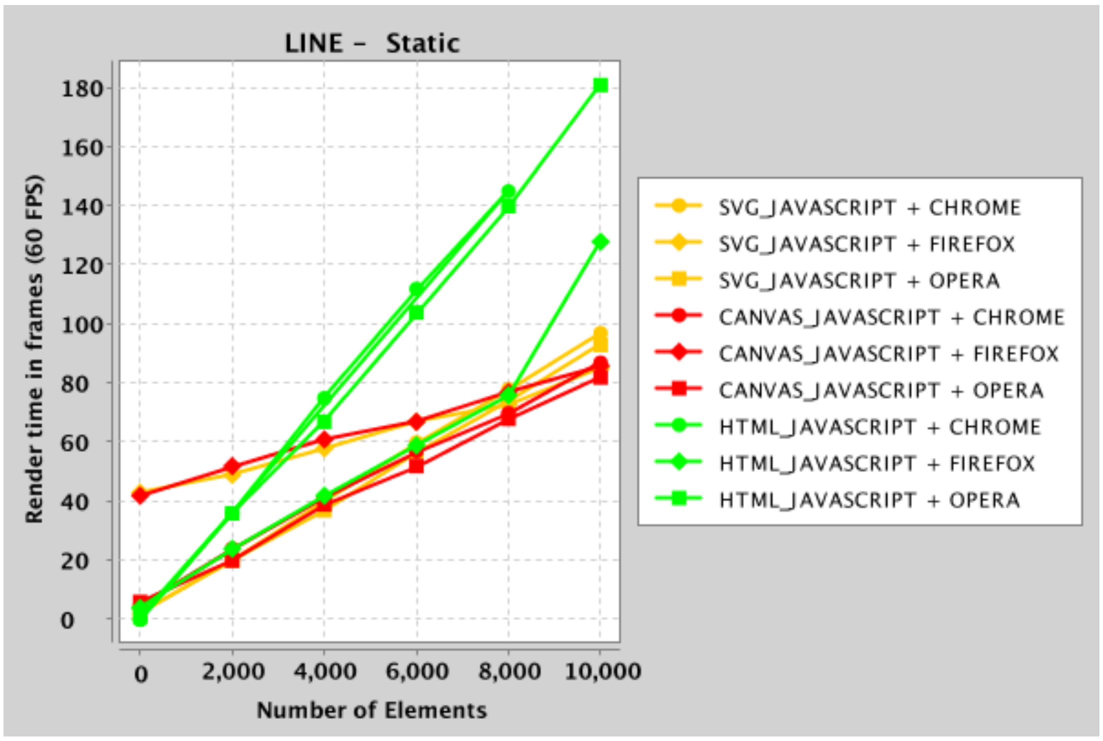

Figure 12: Render Time in Frames, Lines with Opacity

7.1.4. Rendering Time of Graphic Primitives with Shadow

This chapter measures the rendering performance of graphic primitives, which have a shadow effect applied to it. For every graphic primitive square, circle and line, there are three web experiments developed, which display a given number of graphic primitives with shadow. The web technology combinations used are HTML with JavaScript, SVG with JavaScript and Canvas with JavaScript. Figure 13 shows the rendering times of squares with shadow, figure 14 shows circles and figure 15 show lines.

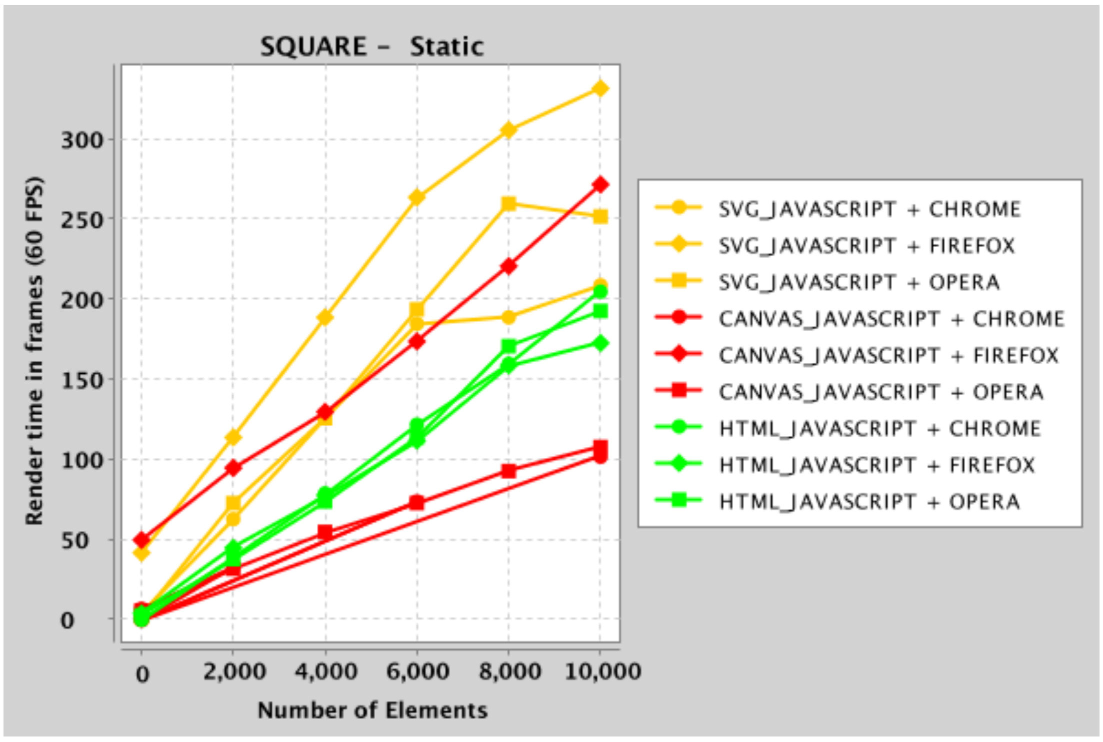

Figure 13: Render Time in Frames, Square with Shadow

In contrast to rendering time results of graphic primitives with no additional CSS attributes and graphic primitives with an opacity value, graphic primitives with a shadow effect have very different rendering times. There are almost no changes across all graphic primitives and browser fort he web technology combination of HTML with JavaScript. It renders 10000 elements with a rendering time of 200 frames. This is in average 20 frames slower than at previous experiments.

Experiments developed with the web technology combination of SVG with JavaScript show very different rendering times. For rendering 10000 squares and circles, the rendering time is between 200 and 400 frames. Also the Firefox browser is the slowest browser for rendering squares and circles with shadow. In contrary, the web technology combination of SVG with JavaScript achieves the fastest rendering times of all web technology combinations at rendering lines with shadow.

Web experiments developed with the web technology combination of Canvas with JavaScript have similar rendering times across all graphic primitives when running in the Google Chrome and Opera Browser. However, when experiments run in the

Firefox browser, rendering times vary a lot. The web technology combination takes 272 frames to render 10000 squares with shadow, 498 frame to render 10000 circles and 304 frames to render 10000 lines.

Overall the fastest web technology for rendering graphic primitives is Canvas with JavaScript for squares and circles. To render lines, the fastest web technology combination is SVG with JavaScript.

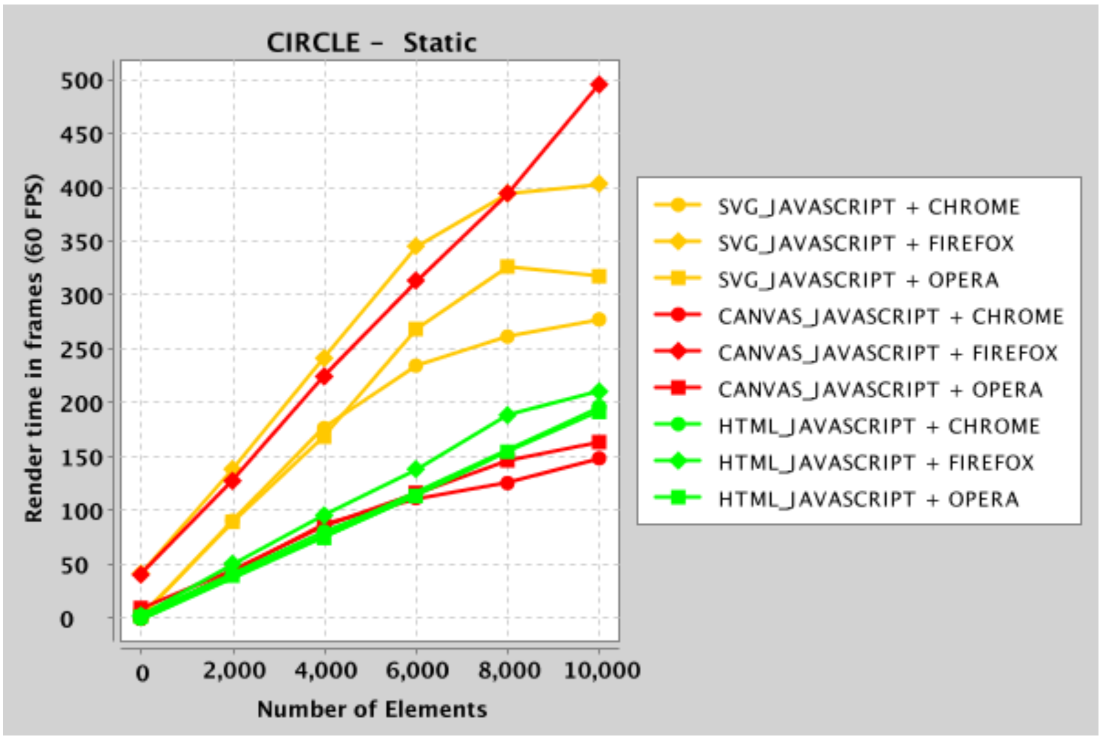

Figure 14: Render Time in Frames, Circle with Shadow

The code for displaying graphic primitives is identical with the code of the web experiments in chapter 6.1.2 which display graphic primitives without any additional styling attributes. However, in this collection of web experiments, a shadow effect is added to every graphic primitive.

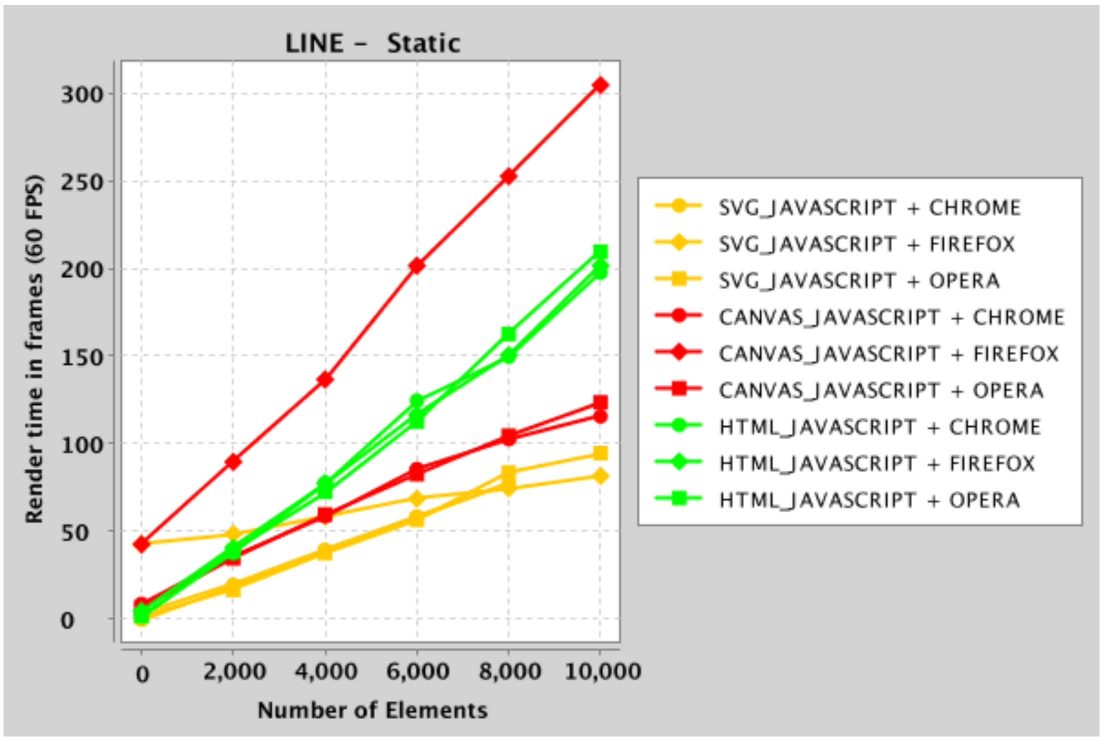

Figure 15: Render Time in Frames, Line with Shadow

The web technology combination of HTML and JavaScript sets a shadow effect to the elements by adding a class which contains the box-shadow CSS attribute. To apply a shadow effect to an SVG element, a filter has to be defined inside the svg tag. The code for the filter is shown in listing X.

```

<filter id="f1" x="0" y="0" width="15" height="15">
   <feOffset result="offOut" in="SourceAlpha" dx="20" dy="20" />
   <feGaussianBlur result="blurOut" in="offOut"
stdDeviation="10" />
   <feBlend in="SourceGraphic" in2="blurOut" mode="normal" />
</filter>

```

Listing 1: SVG Shadow Filter Definition

The svg filter tag allows developers to set more refined filters than at other web technologies. The attributes in the *filter* tag define where the filter should be positioned. The *feOffset* tag sets the colour of the filter by defining that the alpha channel of the elements pixels should be used for the filter instead of the entire RGBA pixels. This makes the filter appear black for the shadow effect. The *stdDeviation* attribute in the *feGaussianBlur* tag defines the amount of blur for the shadow. The filter is referenced with the *id* attribute in the *filter* tag. Filters are added to elements by adding the *filter* attribute, and referencing the filter as URL String followed by a hash tag. In this example, the filter was set by the key value pair of *filter="url(#f1)"*.

Web experiments which implement rendering of squares, circles and line with the web technology combination of Canvas and JavaScript apply a shadow effect with styling methods. Those methods are *shadowColour, shadowBlur, shadowOffsetX* and *shadowOffsetY.*

7.2. Animated Graphic Primitives

Animations in data visualizations are always the result of a human interaction. There is one exception, which is the scenario of live data being displayed. In that case, changes of the data appear without any human interaction. Nevertheless as interactions always trigger changes of objects in the visualization and changes of objects in very small steps are animations, the most relevant animation types can be derived from typical interactions with data visualizations. Fischer describes six interactions with data visualizations and the visual changes they trigger.

1. Changing the view. This interaction is a pan or a zoom on a map. The interaction would require the objects of the visualization to change the position or scale.
2. Change the charting surface: An example is changing the axes of a plot from linear to log scale. The objects of the visualization would need to change the position or change their size.

3. Filter the data: This removes and adds data points to the visualization following a selection criterion. Objects from the visualizations are disappearing.
4. Reorder the data: This changes the order of data points. Data is also ordered visually in data visualizations. This would change the position of objects in the visualization.
5. Change the representation: This interaction changes the style of the visualization. Examples are changing the colour of objects or changing the layout of a graph.
6. Change the data: Move data forward through a time step. Modify the data or change values displayed. An example would be the average value of a wind direction chart which changes with every new data point added. Change of data is causing objects in the visualization to change the position or scale.

Keim describes a similar list of interactions types with data visualizations which consists of *projections*, *zooming*, *filtering*, *linking and brushing*, and *distortion*. *Projections* change the mapping of data dimensions to dimension in the visualization. An example is changing the mapping of data dimensions to the axes of a three dimensional chart. *Zooming* is an interaction which is also part of *changing the view* in Fischer's definitions. Also *filtering* is mentioned by Fischer. The interaction *linking and brushing* is similar to *changing the representation*. At Keim, this is an interaction which adds colour to identical objects in two different visualizations to detect dependencies and correlations. Also Keim describes the interaction *distortion*. This interaction gives the user a detailed view of a portion of the dataset, while an overview of the rest of the data is preserved.

Keim's list is narrower as its interactions focus on data visualizations which are purely for data exploration. Therefore the list from Fisher is chosen to derive typical animations for data visualizations, as it focuses on general types of data visualizations. In chapter 2 it was pointed out that 2D data visualizations consist of only 5 graphic primitives, which are square, circle, line, polyline and polygon. In this thesis the focus is on the three most common graphic primitives, square, circle and line. As a consequence, it is assumed that the interactions by Fisher are solely applied to data visualizations which consist of squares, circle and line. As a result, the possible object changes which are triggered by interactions are more limited. The focus is on object changes, which affect the properties position, opacity, scale, height and length.

Changes of the position property are cause by two different interactions. The first is changing the view with a pan gesture which requires the graphic primitives to move on the screen. The second is *reordering* the data. Here, graphic primitives get a new position.

Interactions *changing the view* and *filtering* have the consequence that objects in data visualizations are appearing and disappearing as they are added and removed. These visual changes can also be animated by a fade-in fade-out effect which would mean an animation of the opacity property.

The scale property of objects in data visualizations changes with interactions of *change the view*, *change the charting surface*, *change the representation* and *change the data*. The interaction *change the view* can be a zoom gesture on a map object. This would cause the objects in the visualization to scale. Changing the charting surface can scale the axes of a chart which would require the objects in the chart to scale with the size of the axes. A change of the representation changes the layout of a data visualization which would also result in a scale change of its containing objects. Finally the interaction of *changing the data* can also cause the objects in the data visualizations to change the scale property. An example would be a live chart where data is added in intervals. With every added object, space for the objects gets smaller and they scale down.

The interactions *change the charting surface* and *change the data* affect the height property of objects. An example of changing the charting surface is setting the scale of axes in a chart. For example, this changes the height of bars in a bar chart. *Changing the data* can also result in a change of the height property. An example is a bar chart, which changes the height of its bars, with data changes.

The length property changes with the interactions of *changing the view* and *changing the data*. When data visualization contains lines, the length of lines would change when the user zooms into the data visualization. Also the line property can change with the interaction of *changing the data*. An example would be a line chart which redraws the data points.

Summarizing, this chapter investigate the rendering performance of animated changes of graphic primitives square, circle and line which where derived from the interactions with data visualizations defined by Fisher. The rendering performance of opacity and position animations will be measured for graphic primitives, square, circle and line. For the scale animation, web experiments will be developed with the graphic primitives square and circle. The height animation will be measured for squares and the animation of length for lines. Also it is important to say that graphic primitives can have two different animated states in animated data visualizations. The first state is animation of a single property. The second state is animation of multiple properties synchronously. This chapter also investigates the rendering performance of multi property animations. The rendering performance of animating the properties position and opacity synchronous will be measured with graphic primitives squares and circles. Also an animation with three property changes of position, opacity and scale will be measured on graphic primitives square and circle.

7.2.1. Animated Web Experiments

Every animated web experiment has the same setup code which is called when the site is loaded. To setup an animated web experiment, positions for a given number of graphic primitives are calculated. Then, graphic primitives are added to the page with absolute position values. Also every web experiment has a button which triggers the animation of graphic primitives. Depending on the web technology combination, the setup code is slightly different. Animation of elements is developed with two different web technologies. First, JavaScript is used to animate property changes of elements incrementally. To start an animation, the *requestAnimationFrame* method from the browser API is called. The *requestAnimationFrame* method requires a callback method which is executed when the browser is ready to draw the next frame. In the callback, small property changes are calculated and applied to the graphic primitives. Afterwards, requestAnimationFrame is called again with the same callback method which will change the properties of graphic primitives. By calling the *requestAnimationFrame* method repeatedly, the application can fluently animate a graphic primitive with small steps. Also web experiments use CSS transitions to animate graphic primitives. Setting a new property value of a CSS class animates the value. As a result, setting new property values to the class of graphic primitives triggers animations.

7.2.2. Animation Smoothness of Position Animated Graphic Primitives

The animation of position of graphic primitives is very common in various data visualization tools. In order to measure the animation smoothness of position animations of various graphic primitives, 15 web experiments have been developed. The graphic primitives which are used for the web experiments are square circle and line. The web technology combinations are Canvas with JavaScript, HTML with CSS, HTML with JavaScript, SVG with CSS and SVG with JavaScript. In order to measure the influence of different graphic primitives on animation smoothness, five web experiments have been developed which animate square, circles and lines. All web experiments were run on the browsers Google Chrome, Firefox and Opera. Also every web experiment was run with an increasing number of elements reaching from 1 to 2000 with steps of 200 in between. Figure 16 shows the animation smoothness values of all iterations in every browser for the graphic primitive square. Figure 17 shows the animation smoothness values for the graphic primitive circle and figure 18 for the graphic primitive line.

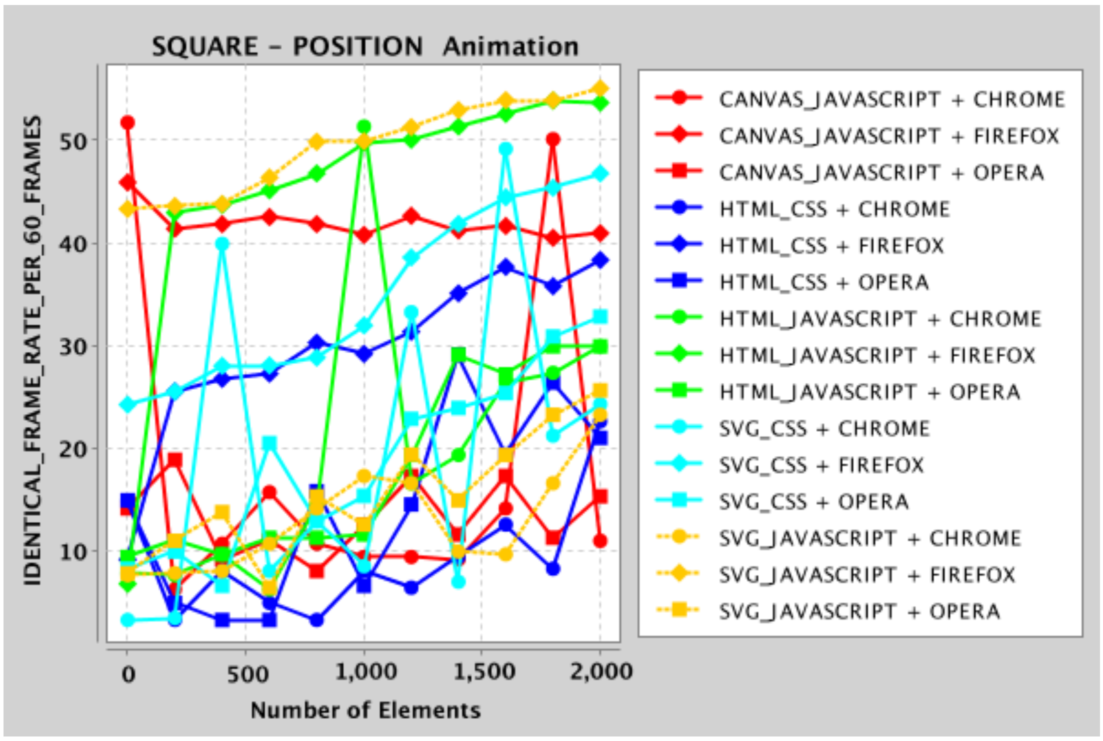

Figure 16: Position Animation Smoothness of Squares

The best animation smoothness performance at animating one element had the web technology combination of SVG and JavaScript in the Google Chrome browser, with four identical frames per 60 seconds. The worst performance had also the web technology combination of SVG and JavaScript in the Firefox browser with 46 identical frames. Animating the position of 2000 squares, the best animation smoothness was achieved by the web technology combination of Canvas and JavaScript in the Google Chrome browser. The worst performance had the web technology combination of SVG and JavaScript at animating the position of 2000 elements.

Overall, it can be said that the Firefox browser is the worst browser at rendering the position animation of squares across all technology combinations. Firefox renders the technology combinations of HTML with CSS and SVG and CSS at an average of 35 identical frames per 60 frames. The technology combinations of HTML with JavaScript, Canvas with JavaScript and SVG with JavaScript achieve the worst animation smoothness values. The animation smoothness is at an average of 48 identical frames per 60 frames.

The Google Chrome and the Opera browser achieve the best performance. The web technology combination of HTML with CSS delivers the best rendering performance overall, followed by SVG with JavaScript running in the Google Chrome Browser. The web technology combinations of Canvas with JavaScript, HTML with JavaScript and SVG with JavaScript executed in the Google Chrome and Opera browser are slightly worse with 17 identical frames per 60 frames in average.

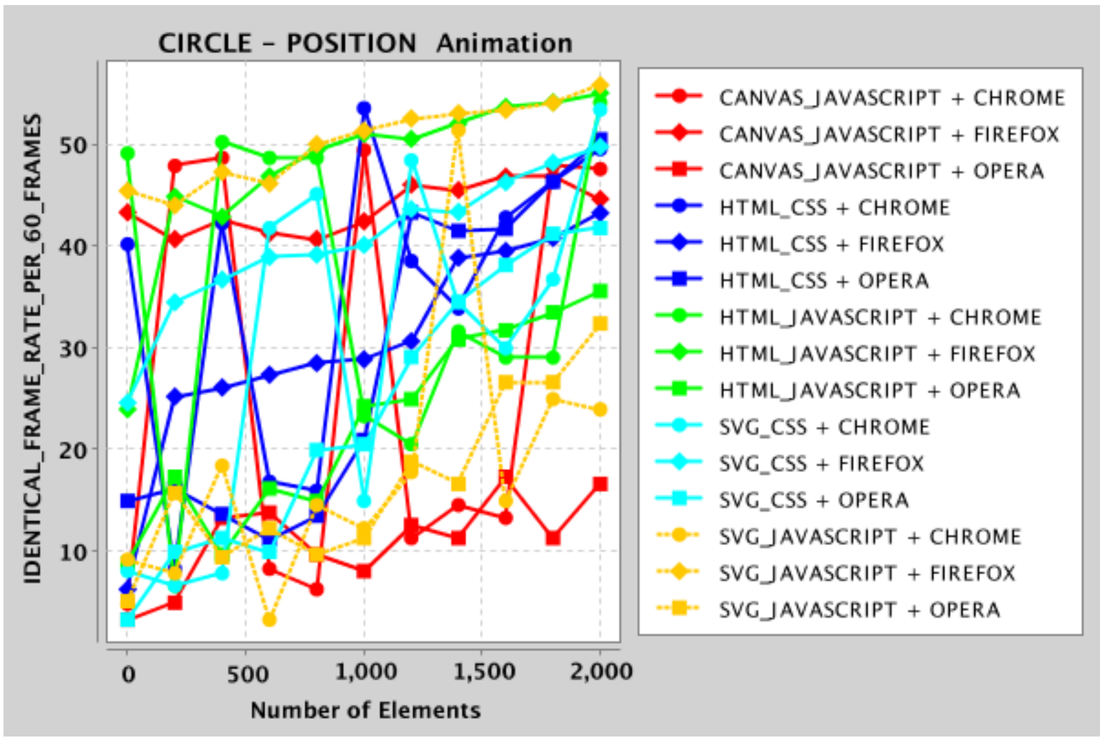

Figure 17: Position Animation Smoothness of Circles

The best animation smoothness as animating one circle was achieved by the technology combination of HTML with CSS running in the Opera browser. The worst performance had the combination of SVG with JavaScript running in the Firefox browser. Animating the position of 2000 circles, the web technology combination of Canvas and JavaScript achieved the best result with 27 identical frames per 60 frames. The web experiment was executed in Google Chrome. The web technology combination of SVG with JavaScript had the worst result in Firefox.

Overall, the best performing web technology combinations at animating the position of circles are Canvas with JavaScript, followed by SVG with JavaScript. Both web experiments were run in the Google Chrome and Opera browser. The combination of SVG and JavaScript has the worst performance compared to all other combinations. The web experiments were run in Firefox. The second worst performance was achieved by the web technology combination of HTML and JavaScript also running in the Firefox browser. Web experiments run in Google Chrome and Opera had a low animation smooth at rendering more than 1000 elements. Those web experiments were executed with web technology combinations of HTML with CSS, SVG with JavaScript and HTML with JavaScript.

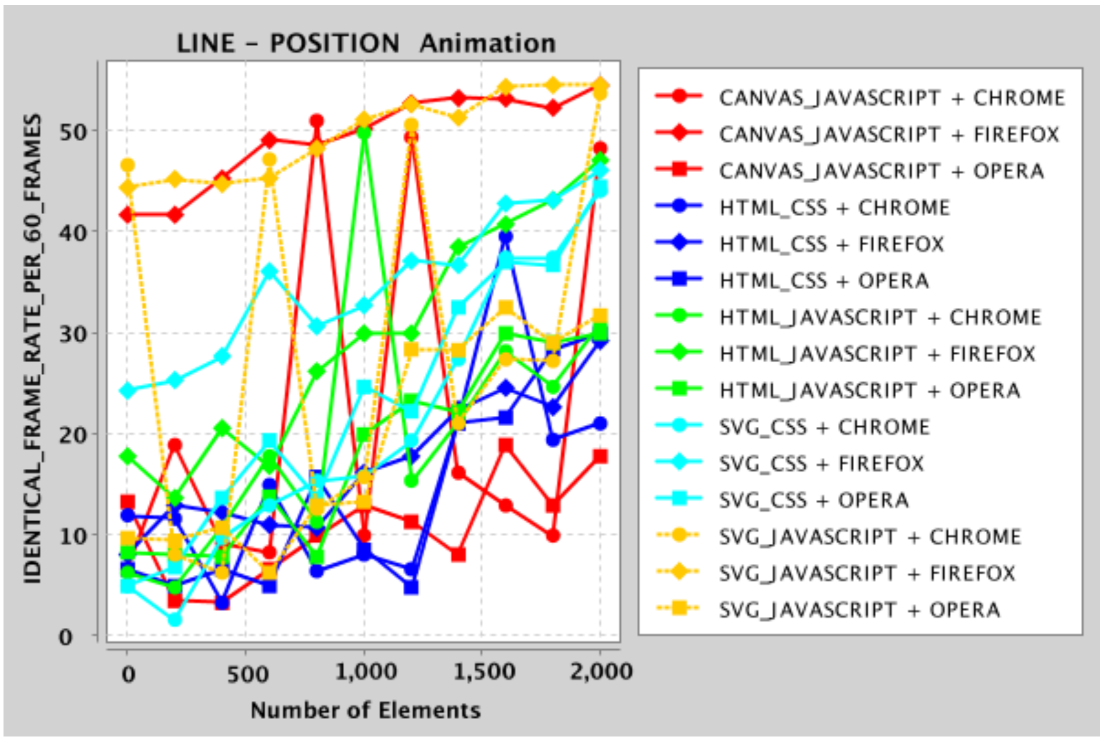

Figure 18: Position Animation Smoothness of Lines

The best animation smoothness performance at animating the position of one line has been achieved by the web technology combination of HTML with CSS in Google Chrome. The animation had an average of 3 identical frames per 60 frames. The less smooth animation was run in the Firefox browser with the technology combination of SVG with JavaScript. The best animation smoothness at animating the position of 2000 lines was achieved with the web technology combination of HTML and CSS in the Opera browser. The worst web technology combination was SVG with JavaScript running in Firefox.

Overall, it can be said that the best web technology combination is HTML with CSS across all browser Google Chrome, Firefox and Opera. Also web experiments running with HTML with JavaScript and Canvas with JavaScript had from rendering one to 2000 lines fewer than 30 identical frames per 60 frames. Those web experiments where run in Google Chrome and Opera. However, the same web technology combinations of HTML with JavaScript has an animation smoothness of over 30 with more than 1000 line elements when running in Firefox. The worst rendering performances where achieved by the web technology combinations of SVG with JavaScript and Canvas with JavaScript. Both combinations were run in the Firefox browser and have an average identical frame rate per 60 frames of 50.

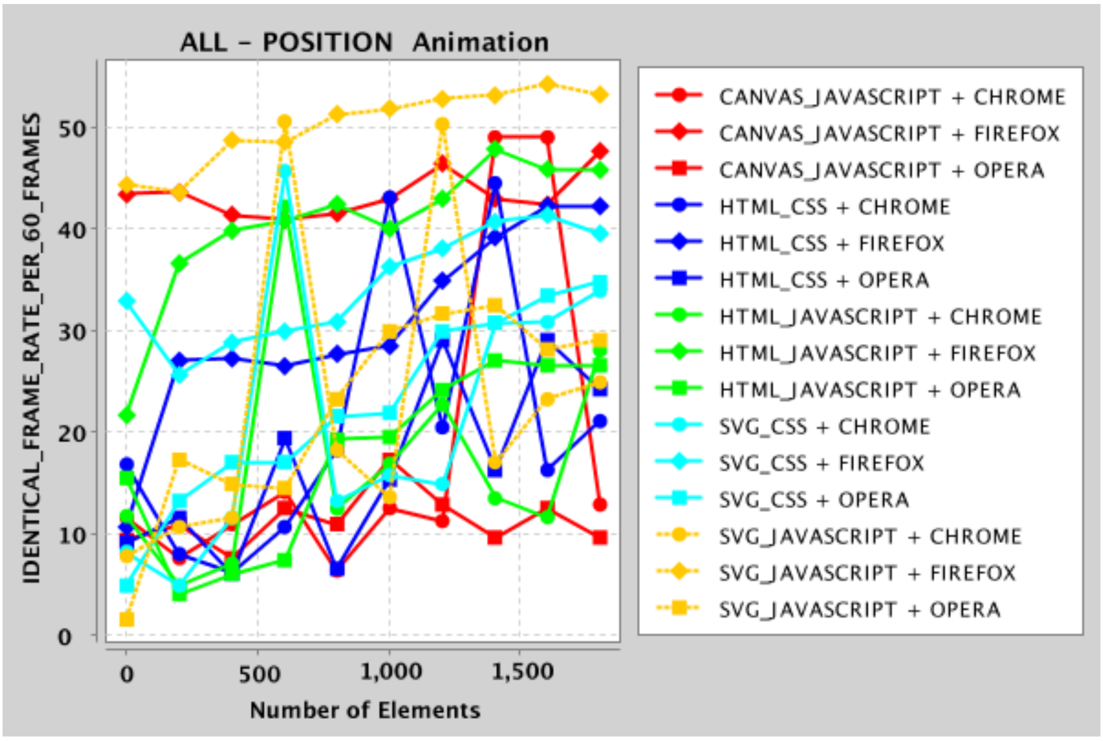

Figure 19: Position Animation Smoothness of Square, Circles and Lines

Also web experiments have been developed which animate the position of squares, circles and lines. Overall, it can be said that the web technology combination of Canvas with JavaScript achieves the best rendering performance with almost not increase of identical pixels for all numbers of elements. However, the same web experiment the second worse performance compared to other experiment iterations when run in the Firefox browser. The worst animation smoothness had the web experiment developed with the web technology combination of SVG with JavaScript running in the Firefox browser. Decent performances show the web technology combinations of HTML with CSS and SVG with CSS.

As the code of the implementation is one factor of rendering performance in animated data visualization, this chapter will explain how elements where animated in each web technology combination. The animation duration in all web experiments was 600 milliseconds. The timing function of the animation was linear. Every web experiment has a button without styling effects which triggers the animation. All elements are positioned absolute on the screen, before they animate.

At the web technology combination of SVG and JavaScript, elements are stored in an array for later use during the animation. When the animation starts, the *requestAnimationFrame* method from the browser API is called. On the callback the new position for all elements is calculated and applied to all elements by setting the CSS transform attribute to translate with the new position. After setting the transform attribute, the *requestAnimationFrame* method is called again. The iterations end when the elements have reached the new position.

The same technique is applied to the web technology combinations of HTML with JavaScript and Canvas with JavaScript. The difference from HTML with JavaScript to SVG with JavaScript is that div elements are created which have CSS attributes that make them appear as different graphic primitives like square, circle and line. In SVG are certain tags for each graphic primitive which are rect, circle and line. Web experiments with the technology combination of HTML with JavaScript change the position of elements every frame by setting the transform attribute. The web experiments of Canvas with JavaScript also use the *requestAnimationFrame* method to ask the browser repeatedly for the next available frame. The canvas context is completely cleared and the elements are drawn again at the new position every frame.

The web experiments with the web technology combination of HTML with CSS and SVG with CSS use CSS transitions to animate the element. In every web experiment, a CSS class with the transition attribute is set with the values which transform attribute changes should be animated by CSS with the duration of 600 milliseconds. Also it is defined that the CSS transition should have a linear timing function. In contrast to web experiments which animate the elements by calling the *requestAnimationFrame* method repeatedly the animation is handled by the browser. To start the animation, the final position of the elements is set via the transform CSS attribute.

7.2.3. Animation Smoothness of Opacity Animated Graphic Primitives

Animating the opacity is a tool to enhance the user experience of applications. As it is uncommon for objects in the real world to change opacity abrupt, animating the opacity of elements in applications makes them more engaging. To investigate the influencing factors on the smoothness of opacity animations in the context of animated data visualizations, a collection of 20 web experiments has been developed. There have been four groups of web experiments developed. Every group contains one web experiment developed with one of the web technology combinations of HTML with CSS, HTML with JavaScript, SVG with CSS, SVG with JavaScript and Canvas with JavaScript. Three groups animate the opacity of one graphic primitive, which are square, circle and line. In order to measure the influence on animation smoothness of different graphic primitives, five web experiments have been developed which animate square, circles and lines all in one page. The opacity is animated from 1 to 0 in every web experiment. The duration of the animations is 600 milliseconds with a linear timing function. The web experiments have been executed with an increasing number of elements from 1 to 2000 elements. Also web experiments have been executed in the browsers Google Chrome, Firefox and Safari.

Figure 20 shows the animation smoothness values for all opacity animations for the graphic primitive square. Figure 21 shows the animation smoothness for Circles and figure 22 for lines.

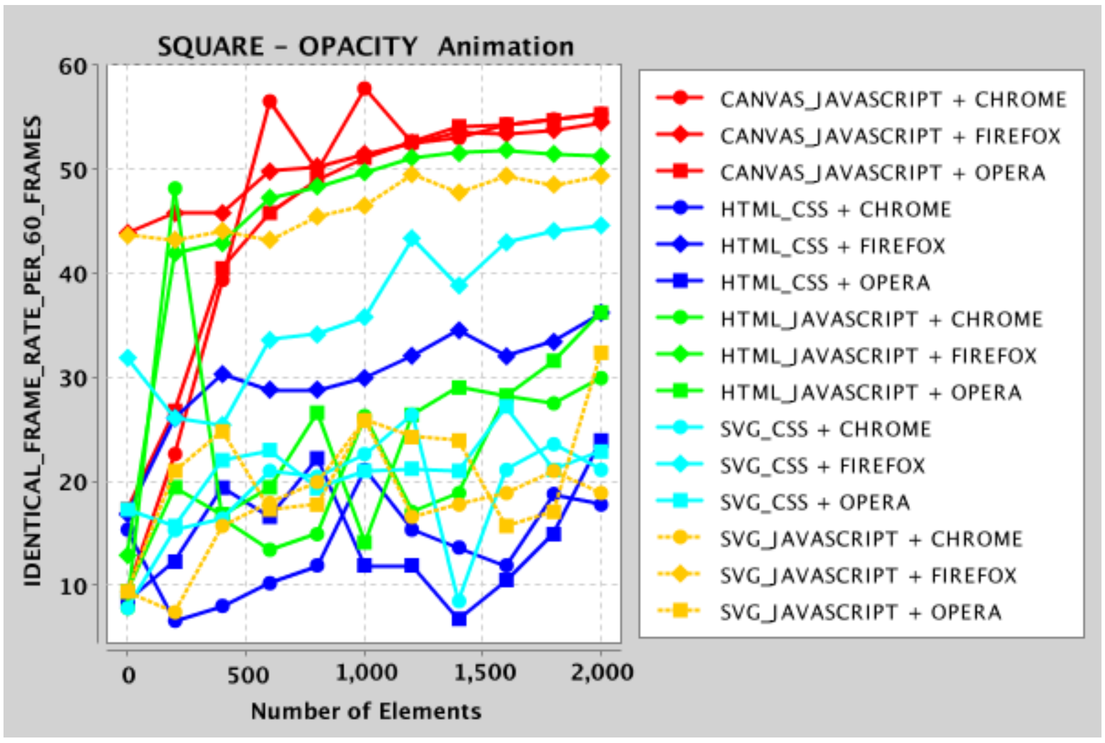

Figure 20: Opacity Animation Smoothness of Squares

The best smoothest an animation for animating the opacity of one square to zero was achieved by the web technology combination of SVG with CSS in the Google Chrome Browser. The less smooth opacity animation had the web technology combination of Canvas and JavaScript executed in the Firefox browser. The smoothest animation at animating 2000 squares was achieved by the web technology combination of HTML with CSS, run in the Google Chrome browse, with 18 identical frames per 60 frames. The worst animation at the animation of 2000 squares had the web technology combination of Canvas with JavaScript. The web experiment was run in the Opera browser.

The web technology combination of HTML and CSS has been the best performing overall. Also animating the opacity of squares with the web technology combination of SVG and JavaScript is also under 30 identical frames per 60 frames. More over the web technology combinations of HTML with JavaScript and SVG with CSS executed in the Google Chrome and Firefox browser had less than 30 frames per 60 identical frames. The worst overall animation smoothness had the web technology combination of Canvas with JavaScript executed in the Firefox browser. Also the same web experiments had very a high identical frame rate in the Google Chrome and Opera Browser. More over the web technology combinations of HTML with JavaScript and SVG with JavaScript are having high identical frame rates when executed in the Firefox browser.

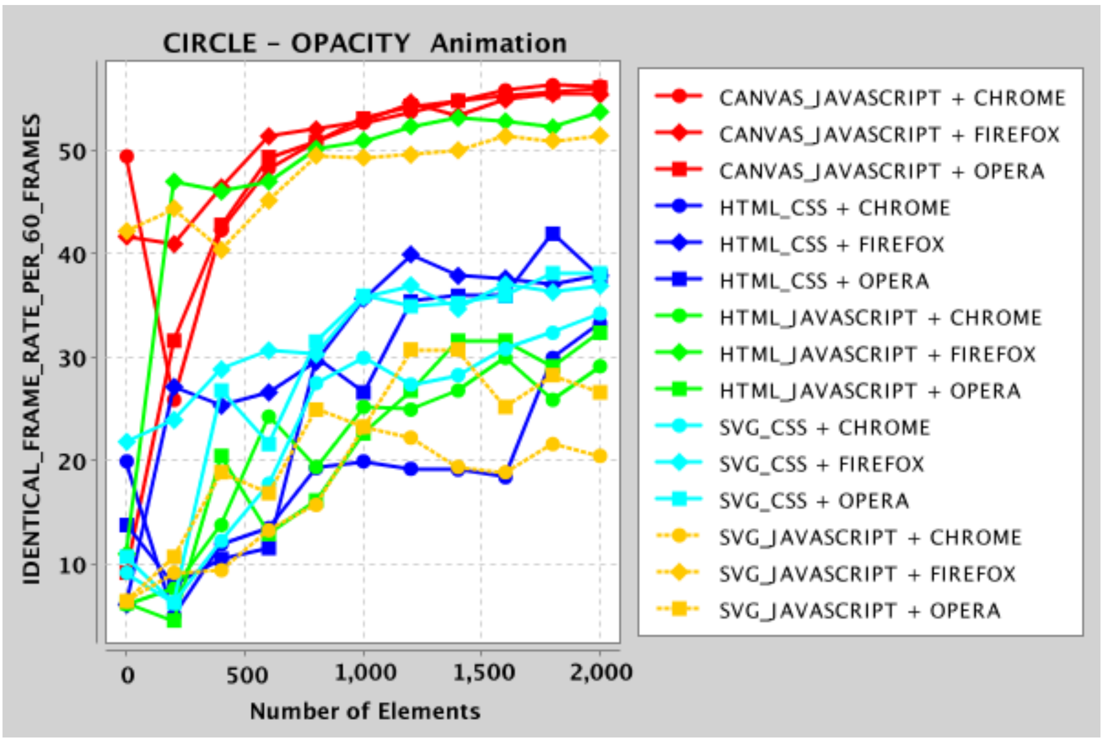

Figure 21: Opacity Animation Smoothness of Circles

At animating the opacity of one circle the web technology combination of HTML and JavaScript had the smoothest animation with 6 identical frames per second. The web experiment was executed in the Google Chrome browser. The worst performance at one circle had the web technology combination of Canvas with JavaScript run in the Firefox browser. The best animation smoothness at animating

2000 circles had also web technology combination of HTML with JavaScript, executed in the Google Chrome browser. Identical to rendering one element, the web technology combination of Canvas with JavaScript had the less smooth animation with 53 identical frames per second.

Overall the web technology combinations of HTML with JavaScript and SVG with JavaScript, executed in the browsers Google Chrome and Firefox achieved the smoothest animations in average. The web technology combinations of HTML and CSS and SVG and CSS, executed in the browsers Google Chrome, Firefox and Opera achieve animation smoothness values with less than 30 identical frames per second. The worst performance overall had the web technology combinations of Canvas with JavaScript, SVG with JavaScript and HTML with JavaScript, when executed in the Firefox browser.

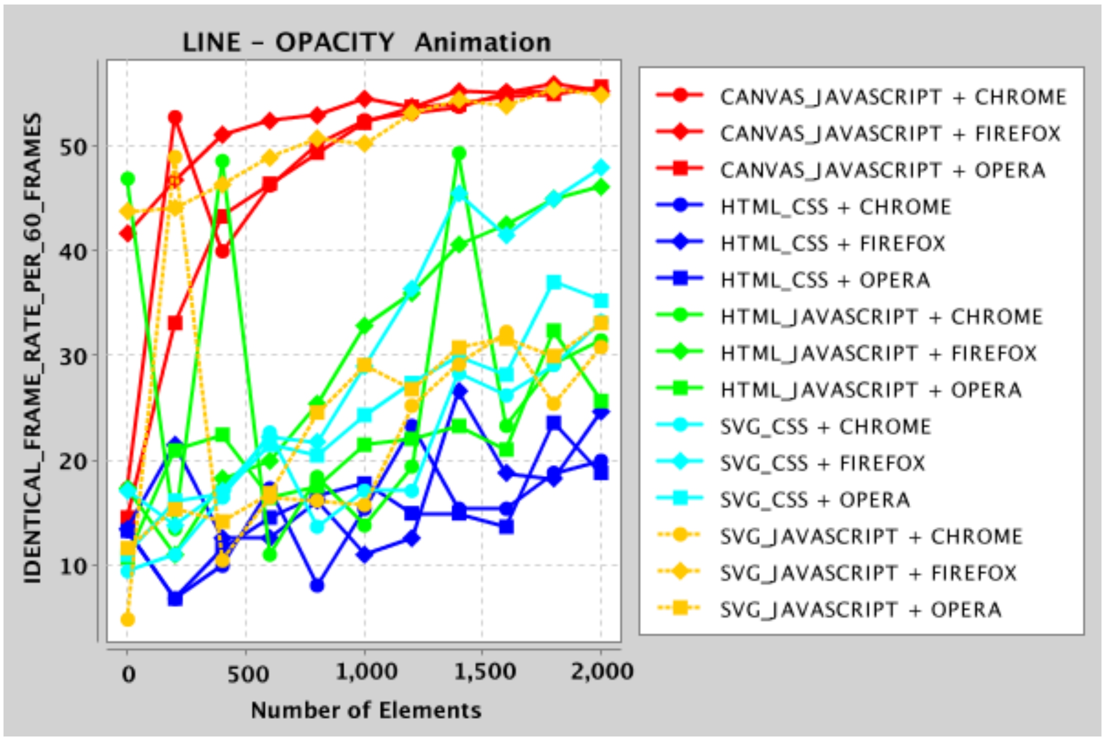

Figure 22: Opacity Animation Smoothness of Lines

The best animation smoothness had the opacity animation for one line, developed with the web technology combination of SVG with JavaScript, executed in the

Google Chrome browser. The worst at animating one element had the web technology combination of HTML with JavaScript, executed also in the Google Chrome browser. At animating the opacity at 2000 lines the web technology combination of HTML and CSS achieved the best performance with 17 identical frames per second. The worst performance had the web experiment, developed with the web technology combination of Canvas with JavaScript. The web experiment was executed in the Opera browser.

Overall the web technology combination of HTML with CSS achieved the best animation smoothness values across all tested browsers. Web experiments developed with the web technology combinations of HTML with JavaScript, SVG with CSS and SVG with JavaScript and run in the browsers Google Chrome and Opera had animation smoothness values with less than 30 identical frames per second across the entire range of element numbers. The worst overall performance had the web technology combination of Canvas with JavaScript across all tested browsers. Also the web experiment developed with SVG and JavaScript had bad animation smoothness with 51 identical frames per second in average.

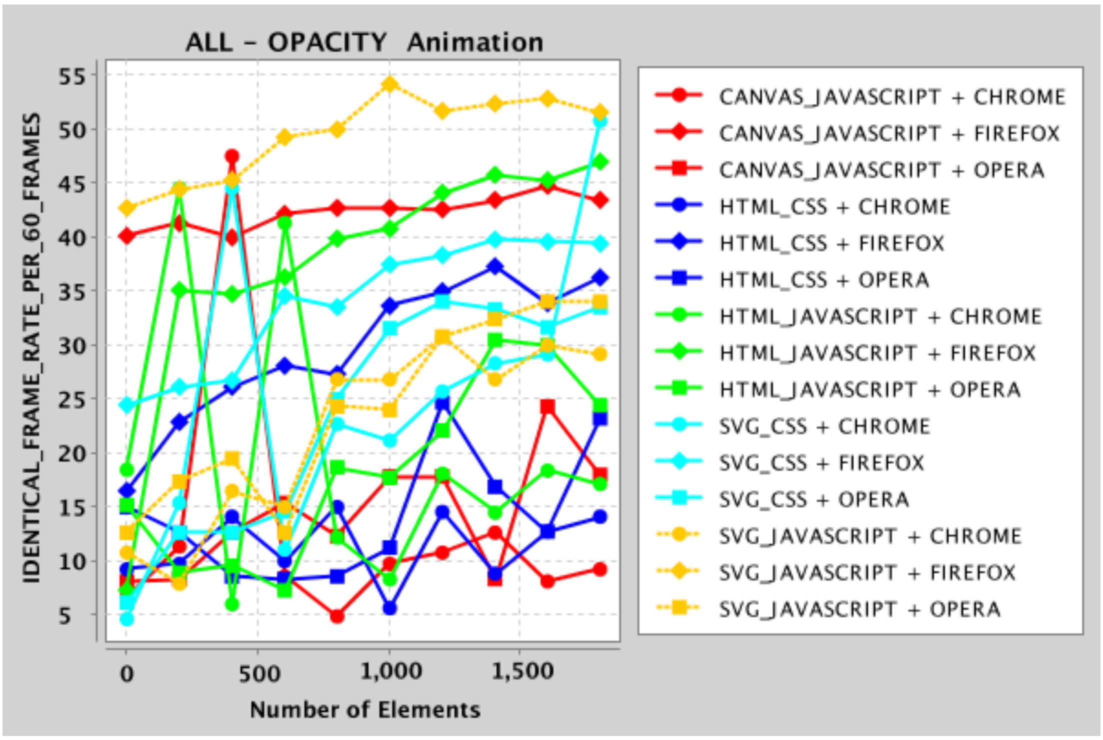

Figure 23: Opacity Animation Smoothness of Squares, Circles and Lines

The web technology combination of Canvas with JavaScript achieves the best animation smoothness values for animating the opacity of squares, circles and lines in one page, when executed in the Google Chrome browser. The worst performances where had web experiments developed with SVG with JavaScript, Canvas with JavaScript and HTML with JavaScript, when run in the Firefox browser. Compared to web experiments, that animate only one graphic primitive, it can be said, that the web technology combinations of HTML with CSS and SVG with CSS worse overall animation smoothness values, especially at small numbers of graphic primitives.

The code of a web application is one factor with influences the rendering performance. As a result the code for each web experiment will be explained.

All web experiments animate the opacity of the elements from 1 to 0 in 600 milliseconds. The timing function is linear.

All web experiments, which use JavaScript for the animation, use the *requestAnimationFrame* method from the browser API. With this method a callback method is registered, which gets called when the browser is ready to draw the next frame. By calling the *requestAnimationFrame* method repeatedly, the application can fluently animate an element with small steps. At every *requestAnimationFrame* callback, small opacity step changes, as part of the whole animation, are calculated and applied to the element.

The web experiments developed with the web technology combination of HTML and JavaScript save all elements in an array, when they are added initially to the DOM. At every *requestAnimationFrame* callback, all elements are iterated and a small opacity value change is set for every element. All graphic primitives are div elements in all web experiments. At the web experiment that animates circles, the border-radius attribute is set. The graphic primitive line is a div element, which has a very small width attribute value.

The web experiments using the web technology combination of SVG and JavaScript are using the exact same code animating the elements, like web experiments with the web technology combination of HTML and JavaScript. The difference is, that elements are added to a svg element. Also every element has its own specific tag, which are rect, circle and line.

At the web technology combination the difference is that elements are initially drawn to a Canvas 2D context. At every frame iteration the context is cleared and all elements are drawn again with changed opacity values. The opacity is set at squares and circles with the *fillStyle* method. The method expects a color string. In this case the color was a rgba color value, where the alpha value changed every iteration. At lines the *strokeStyle* method was used.

At the web experiments developed with the web technology combinations of HTML with CSS and SVG with CSS, the animation was realized with CSS transitions. When the animation was triggered, all elements are iterated and the new opacity value added as CSS attribute to the class, which animates the opacity value.

7.2.5. Animation Smoothness of Scale Animated Graphic Primitives

Scale property changes are triggered by the interactions Change the view, Change the charting surface, Change the representation and Change the data, as described in chapter 7.2. The animation smoothness is measured on graphic primitives square and circle. Five web experiments have been developed for every graphic primitive with the web technology combinations of Canvas with JavaScript, HTML with JavaScript, HTML with CSS, SVG with JavaScript and SVG with CSS. Every web experiment is executed in the browser Google Chrome, Firefox and Opera. Also web experiments are executed with increasing number of elements from 1 to 2000 with 200 steps in between. The duration of the scale animation is 600 milliseconds. The timing function is linear. Graphic primitives are scaled from the size of 10 to 10 pixels with factor four. The center point of the graphic primitives is preserved. The animation smoothness values of scale animations of squares, circles and lines are visualized in figures 24, 25 and 26.

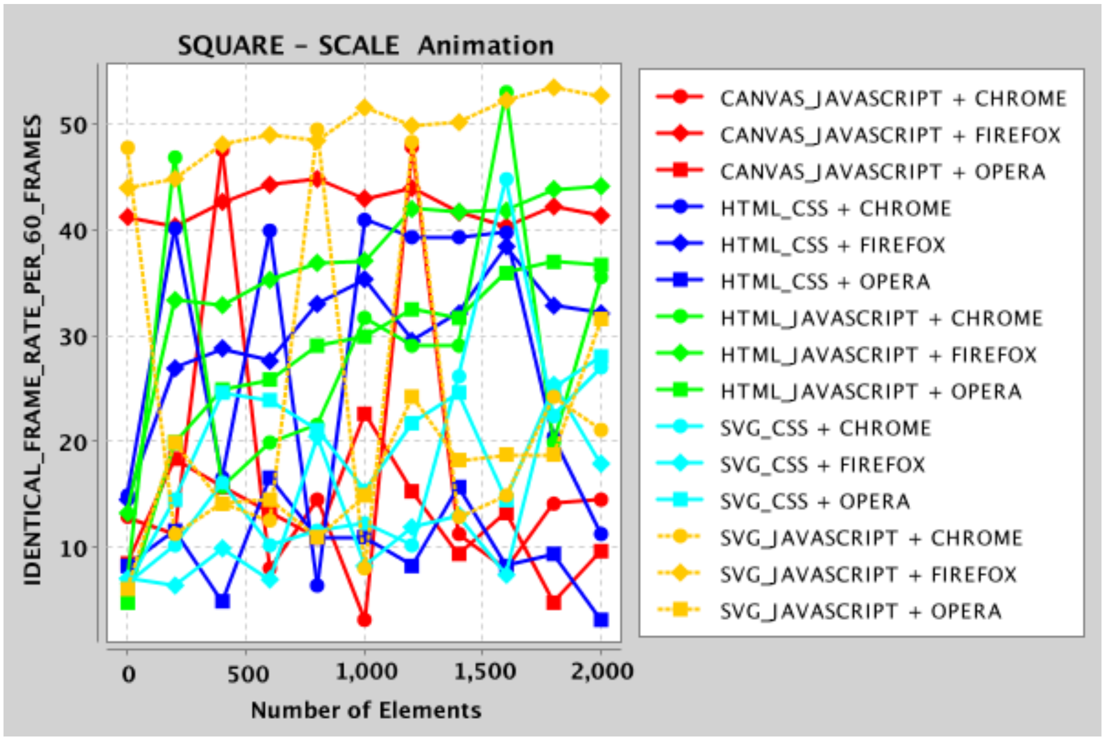

Figure 24: Scale Animation Smoothness of Squares

The smoothest scale animation of one square achieved the web technology combination of HTML with JavaScript executed in the Opera browser. The worst performance had the Technology combination of SVG with JavaScript run in the Google Chrome browser. At animating the scale of 2000 elements the web technology combination of HTML with CSS had the best result. The less smooth animation had the technology combination of SVG with JavaScript.

Overall the technology combinations of Canvas with JavaScript and SVG with JavaScript had animations smoothness values fewer than 30 identical frames per second. Also the web technology combination of SVG with CSS, run across all browsers, had average animation smoothness fewer than 30.

The worst rendering performances overall had the web technology combination of SVG with JavaScript followed by Canvas with JavaScript, both executed in the Firefox browser. Also the Firefox browser achieved a worse rendering performance at animating the scale of squares with the web technology combination of HTML with JavaScript at an average animation smoothness of 38.

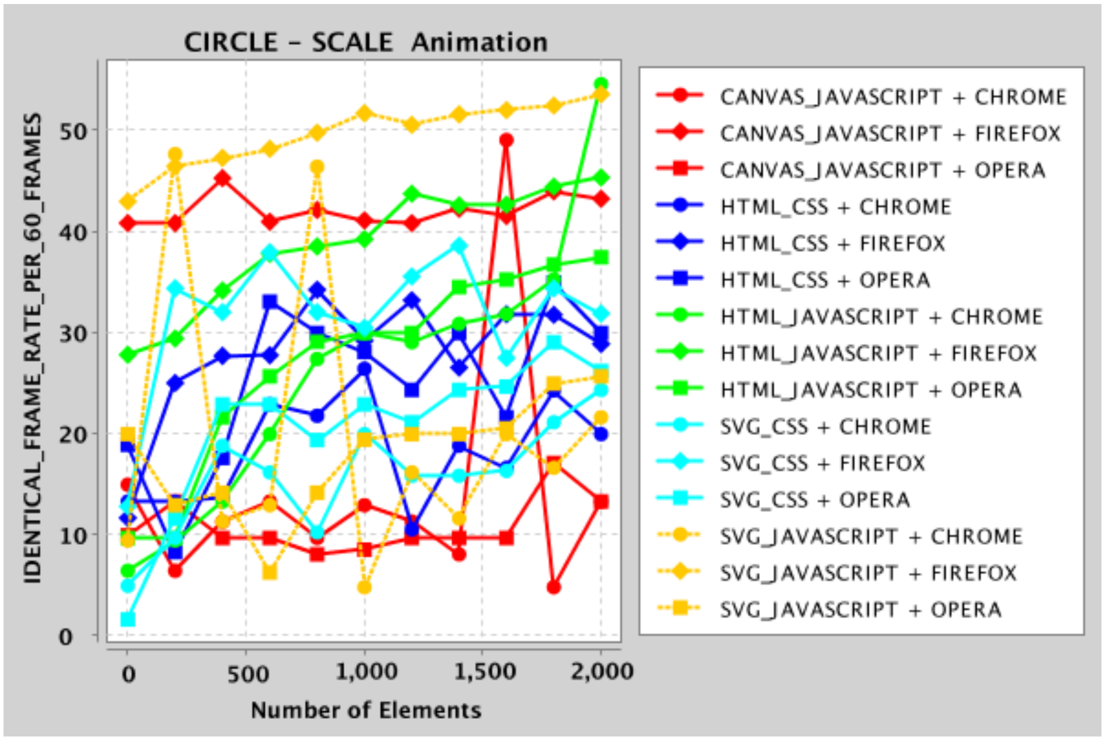

Figure 25: Scale Animation Smoothness of Circles

At animating the scale of one circle the web technology combination of SVG with CSS achieved the best rendering performance, when executed in the Opera browser. The web technology combination of SVG with JavaScript, executed in the Firefox browser, had the worst animation smoothness value at animating on circle. The best rendering performance at animating the scale of 2000 circles had the web technology combination of Canvas with JavaScript, executed in the Opera browser. However the worst performance had the technology combination of HTML with JavaScript run in the Google Chrome browser.

Overall it can be said, that the web technology combinations of Canvas with JavaScript and SVG with JavaScript have the best animation smoothness values with an average identical frames per seconds of fewer than 20. Those web experiments were run in Google Chrome and Opera. Also the web technology combination SVG with CSS has good rendering performance values, with an average animation smoothness of less than 25, when executed in Google Chrome and Opera.

Similar to animating square, the worst overall rendering performance had the technology combinations of Canvas with JavaScript and SVG with JavaScript, and HTML with JavaScript when executed in the Firefox browser.

The web experiment developed with the web technology combinations of Canvas with JavaScript, HTML with JavaScript and SVG with JavaScript use the *requestAnimationFrame* method like web experiments described in chapters 7.2.2 and 7.2.3. Every requested frame a small step of the animation is calculated an applied to the elements.

At the web technology combination of Canvas with JavaScript, the entire canvas context is cleared at every frame an graphic primitives are drawn completely new with their new attributes. Frame by frame the scale of graphic primitives is changed by multiplying the initial width and height values with a scaling factor. Also the initial center point of the graphic primitives is kept, by adapting the x and y coordinates.

Web experiments with the technology combinations of HTML with JavaScript and SVG with JavaScript set the CSS transform attribute at every frame. It is important to say, that at SVG, the transform-origin CSS attribute needs to be set to “50 % 50%” to keep the initial center points of elements. The containing svg element is also affected by CSS transforms.

HTML with CSS and SVG with CSS use CSS transitions to animate the elements. Every element has a class with a set transition for all properties with the duration of 600 milliseconds and a linear timing function. To start the animation the transform attribute is added to the class with a scale value. Web experiments, which use SVG have to set the transform-origin attribute to “50% 50%” to keep the initial center point.

7.2.6. Animation Smoothness of Height Animated Squares

To investigate the rendering performance of height animations on squares five web experiments have been developed, with the web technology combinations of Canvas with JavaScript, HTML with CSS, HTML with JavaScript, SVG with CSS and SVG with JavaScript. Every web experiment animates the height of a square for 50 pixels in 600 milliseconds. The timing function of the animation is linear. Also all web experiments have been run in the browser Google Chrome, Firefox and Opera with increasing number of elements from 1 to 2000.

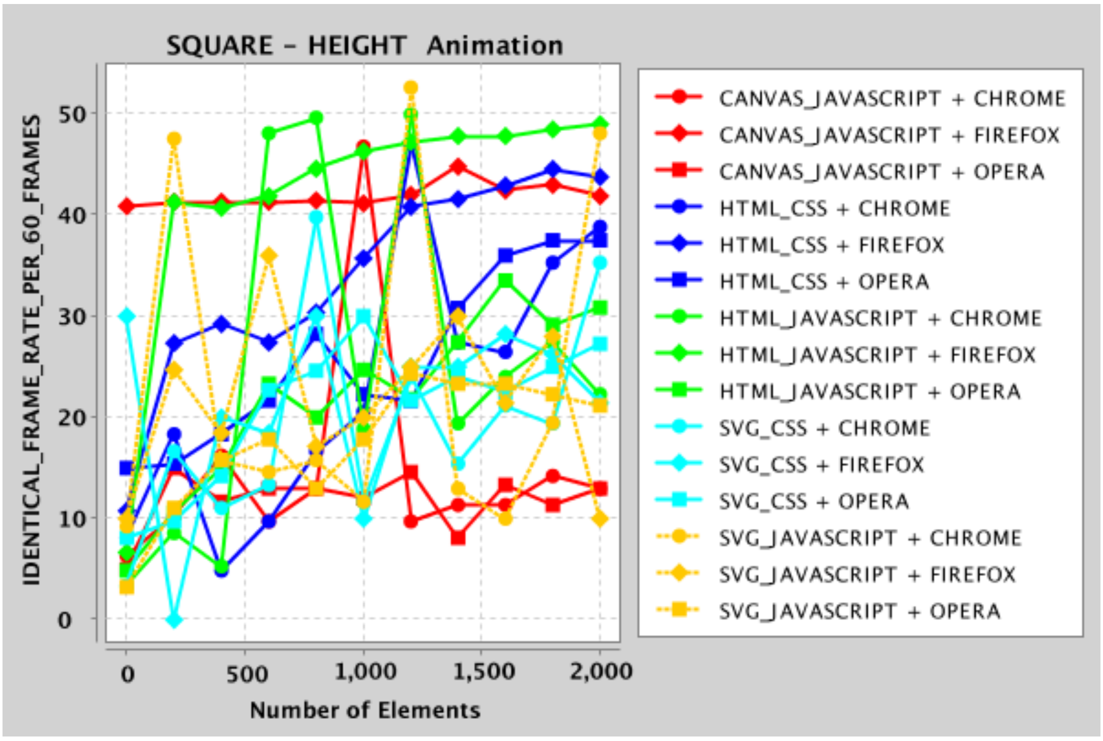

Figure 26: Height Animation Smoothness of Squares

The best rendering performance for animating the height of one square was achieved by the web technology combination of SVG with JavaScript executed in the Opera browser. The worst performance had the web technology combination of Canvas with JavaScript executed in the Firefox browser. The best rendering performance at animating the height of 2000 squares had the web technology combination of SVG with JavaScript at nine identical frames per second. The worst performance at animating 2000 squares had the web technology combination of HTML with JavaScript executed in the Firefox browser.

Overall it can be said that the web technology combinations of Canvas with JavaScript, SVG with JavaScript, SVG with CSS, HTML with JavaScript and HTML with CSS achieved good rendering performances with less than 30 identical frames per second in average. However the web technology combinations of Canvas with JavaScript, HTML with JavaScript and HTML with CSS had less smooth animations, when executed in the Firefox browser.

Like in previous web experiments, the web technology combinations, which use JavaScript to trigger and steer the animation use the requestAnimationFrame method to constantly request for new frames to by rendered by the browser. At every frame, the browser is ready to render, they incrementally change the height of the square by a small portion. The web technology combination of Canvas with JavaScript clears the entire canvas context every frame and redraws the square with a changed height value. The web technology combinations of HTML with JavaScript and SVG with JavaScript set the CSS height attribute at every square at every frame. The difference is, that the combination with HTML uses *div* elements to draw square, while the combination with SVG draws SVG *rect* elements on a svg container. The web technology combinations of HTML with CSS and SVG with CSS use the same elements respectively. However they animate the height of the elements with CSS transitions. In order to start the animation they change the height attribute of the elements class to the final height value.

7.2.8. Animation Smoothness of Length Animated Lines

In order to tests the influence of length animations on the animations smoothness five web experiments have been developed with the web technology combinations of HTML with CSS, HTML with JavaScript, SVG with CSS, SVG with JavaScript and Canvas with JavaScript. The results of the experiments are shown in figure 27.

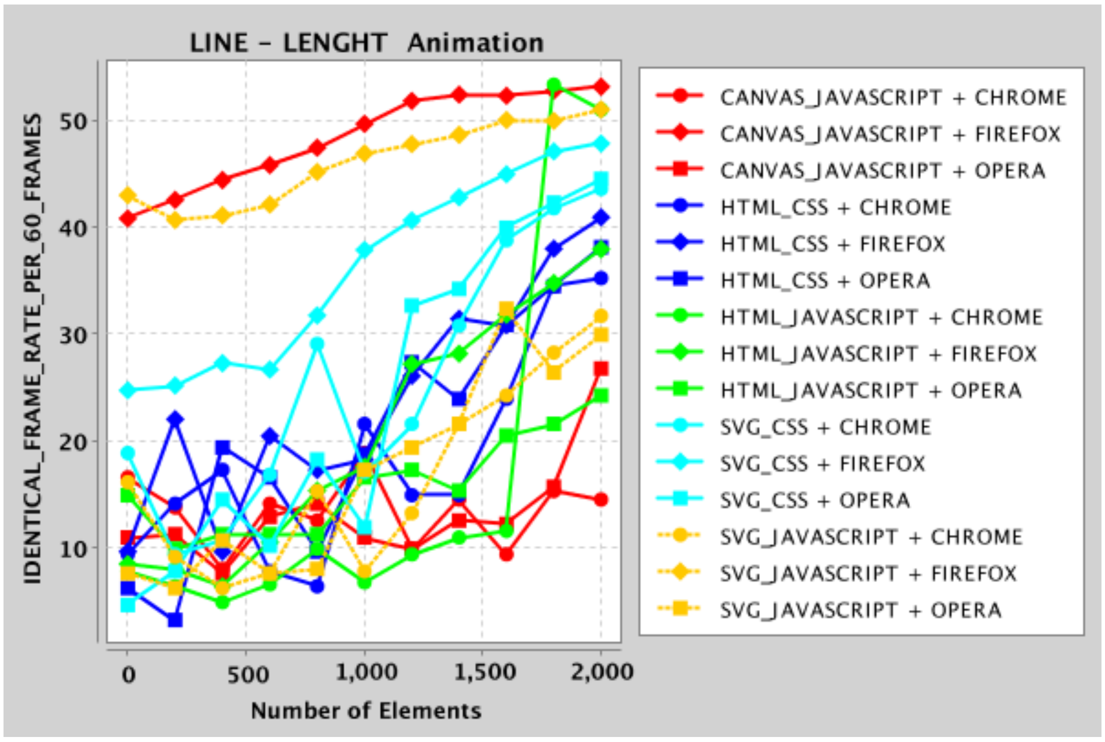

Figure 27: Lenght Animation Smoothness of Lines

Overall the web technology combination of Canvas with JavaScript achieves the best animation smoothness values, when executed in the browsers Google Chrome and Opera. Animation smoothness rises at a maximum of 16 frames between the animation of one line and 2000 lines. Also good animation smoothness was achieved by web experiments developed with HTML with JavaScript, though out all browsers. The worst performance had the web technology combinations of Canvas with JavaScript and SVG with JavaScript, when executed in the Firefox browser.

The web technology combination of Canvas with JavaScript animates the length of the line, by changing incrementally the endpoint of the line with the *lineTo* method. It is important to say, that at every iteration of the *requestAnimationFrame* callback, the canvas context is cleared and graphic primitives are redrawn completely again.

Experiments developed with the web technology combination of HTML with CSS, animate the length of the line by setting the height property of a very slim div element to the final value. Also at HTML with JavaScript the length is animated by setting the height property, however, steps of the animation are calculated and applied incrementally in every callback of the *requestAnimationFrame* method. In SVG lines are defined by start and end coordinates. However these coordinates cannot be animated with CSS. In the web experiment, which uses the combination of SVG with CSS the length of lines are animated by setting the *dashOffset* attribute to the zero. This is a trick in SVG to animate lines, which is often applied on the web. It works by setting the *dashArray* attribute to the length of the line. The *dashArray* attribute is used to set the space between dashes at dashed lines. However, when the *dashArray* attribute is the length of the line, it will disappear. Changing the offset of the dashes with the *dashOffset* attribute animates the length of the line, because dashOffset can be animated with CSS transitions. At the web experiment with the technology combination of SVG with JavaScript, the same trick is applied, but changes are set to the line incrementally within the *requestAnimationFrame* callbacks.

7.2.9. Animation Smoothness of Position and Opacity Animated Graphic Primitives

The second state of graphic primitives in animated data visualizations is animated. It can be divided into single property animations and multiple property animations. The focus on this collection of web experiments is on multiple property animations. The experiments animate the position and the opacity of squares and circles. In total there are ten web experiments developed, which display exactly the same position and opacity animation. The animation is 600 milliseconds long with a linear timing function. The position of the graphic primitives is animated 50 pixels down, while the opacity value is animated from 0 to 1. Also every animation is developed with the technology combinations of Canvas with JavaScript, HTML with CSS, HTML with JavaScript, SVG with CSS and SVG with JavaScript, for each graphic primitive. Figure 28 shows the results from experiments on the smoothness of position and opacity animations on squares. Figure 29 shows the smoothness values for the exact same animation on circles.

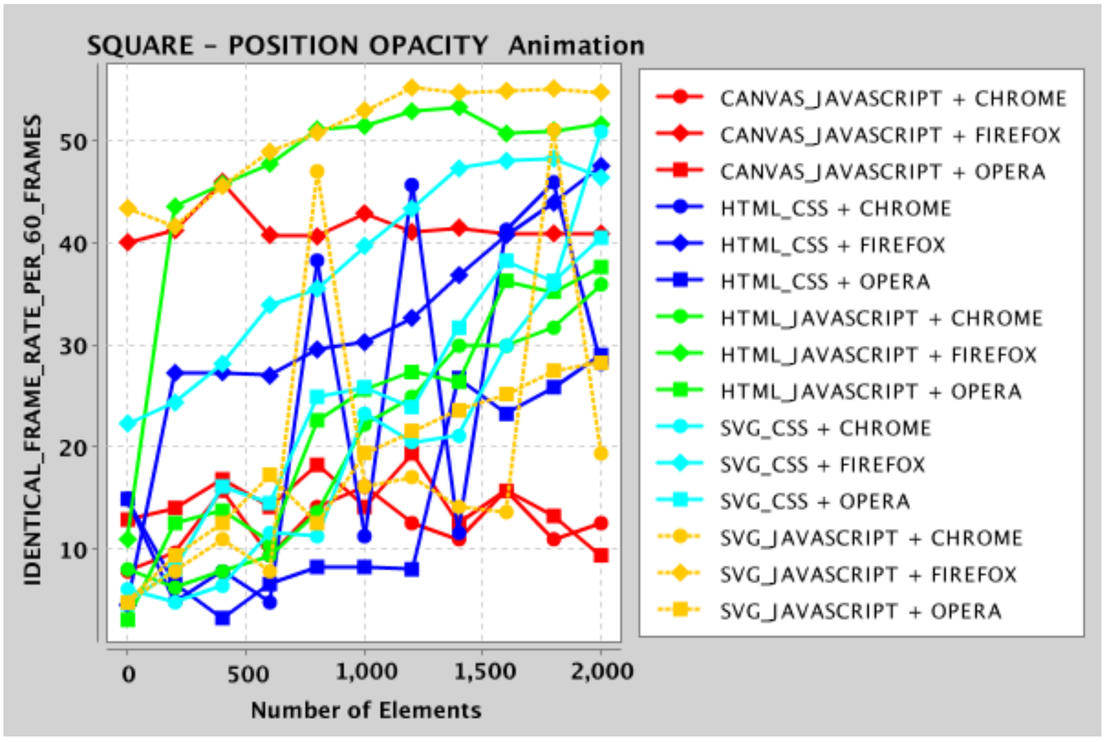

Figure 28: Position Opacity Animation Smoothness of Squares

The smoothest position and opacity animation of one square was achieved by the web technology combination of HTML with JavaScript executed in the Opera browser. At animating one element and 2000 elements, the worst performance had the technology combination of SVG with JavaScript, run in the Firefox browser. The best rendering performance at animating 2000 squares had the web technology combination of Canvas with JavaScript, executed in the Opera browser. Overall it can be said, that the technology combination of Canvas with JavaScript achieved the best rendering performance with an average animation smoothness of fewer than 20 identical frames per second. However this applies only to web experiments, which are run in the Google Chrome and Opera browser. Web experiments, run the combination of Canvas with JavaScript, executed in Firefox, have worse animation smoothness of 43 identical frames per second in average. The worst overall animation smoothness values had web experiments with the web technology combinations of SVG and JavaScript and HTML and JavaScript, when they were run in the Firefox browser.

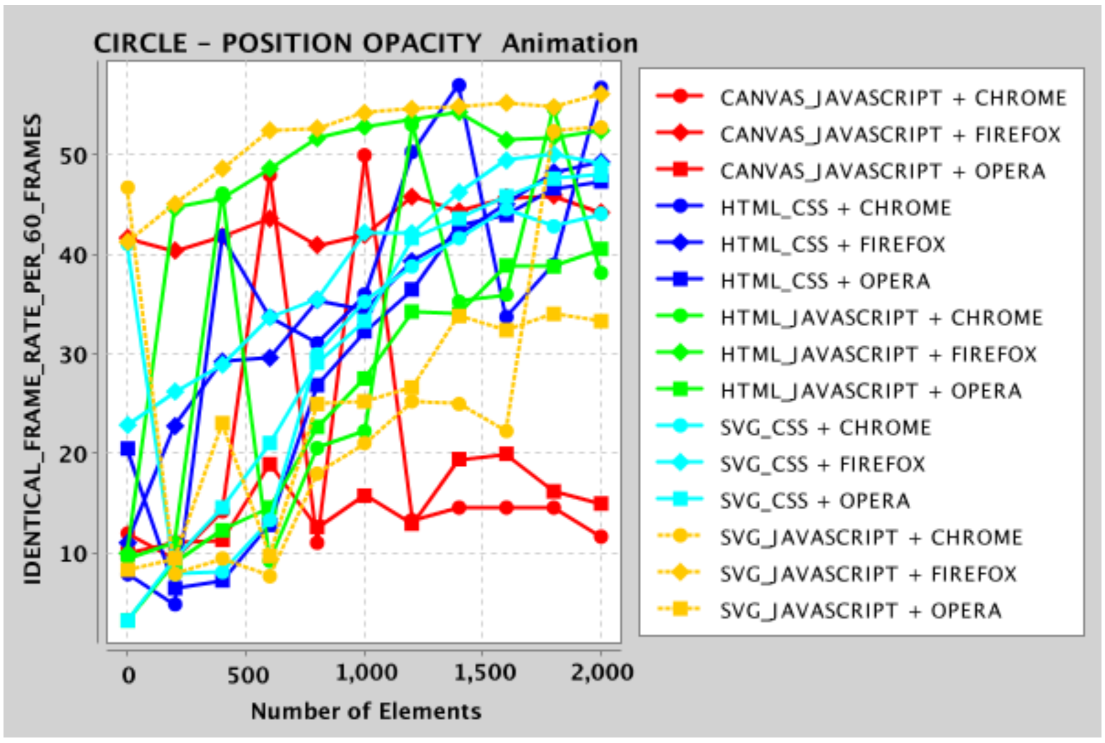

Figure 29: Position Opacity Animation Smoothness of Circles

The best rendering performance at animating the position and opacity of one circle had the web technology combination of SVG with CSS, when executed in the Opera browser. The worst performance at animating one circle had the web technology combination of SVG with JavaScript, executed in the Google Chrome browser. The smoothest animation of position and opacity of 2000 elements was achieved by the web technology combination of Canvas with JavaScript executed in the Google

Chrome browser. The web experiment had an animation smoothness of 11 identical frames per second. The web technology combination of HTML with CSS had the worst animation smoothness performance, when run in the Google Chrome browser.

The overall best performance for animating the position and opacity of squares was achieved by the web technology combination of Canvas with JavaScript, when executed in the browsers Google Chrome and Opera. Also identical to the animation of squares, the worst performance had the web technology combinations of HTML with JavaScript and SVG with JavaScript.

The code of position opacity animations is almost identical to the code of position animations of squares and circle, described in chapter 7.2.2. The difference is, that the opacity value is changed with the position. Web experiments, which use JavaScript for the animation, calculate a small opacity change for every frame rendering cycle, requested with the *requestAnimationFrame* method. The web technology combination of Canvas with JavaScript clears the canvas context every frame and redraws the graphic primitives with slightly changed position and opacity. The web technologies of HTML with JavaScript and SVG with JavaScript animate the opacity by setting the CSS attribute opacity. The web technology combinations of HTML with JavaScript and SVG with JavaScript use CSS transitions for animation. To start the animation they set the CSS opacity attribute on the class of the graphic primitives.

7.2.10. Animation Smoothness of Position Opacity Scale Animated Graphic Primitives

This chapter focuses on multi property animations of graphic primitives square and circle. Ten web experiments with visually identical animations of position opacity and scale were implemented. The duration of the animation is 600 milliseconds with a linear timing function. The animation for every graphic primitive was developed with the technology combinations of HTML with JavaScript, SVG with JavaScript, Canvas with JavaScript, HTML with CSS and SVG with CSS. The measured animation smoothness values of simultaneous animations of position, opacity and scale on squares are visualized in figure 30. Animation smoothness values for the exact same animation with circles are displayed in figure 31.

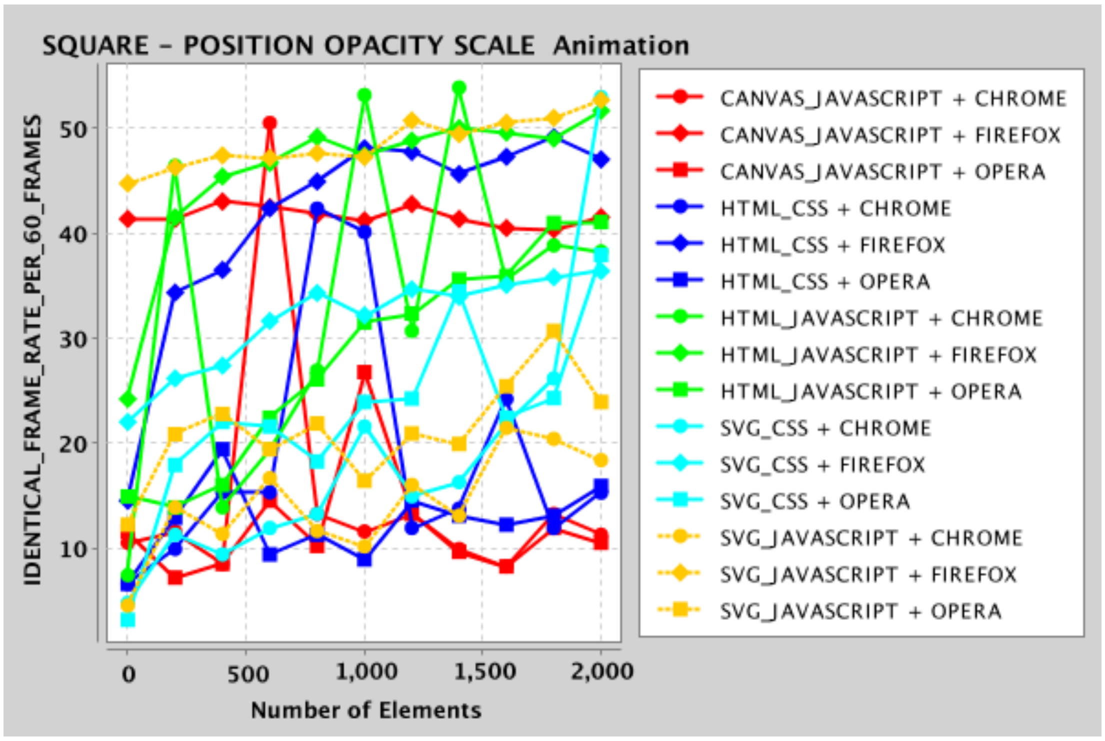

Figure 30: Position Opacity Scale Animation Smoothness of Squares

Overall it can be said, that the smoothest position, opacity, scale animations of one square are achieved by the web technology combinations of Canvas with JavaScript and HTML with CSS when executed in the Google Chrome browser. Also the web technology combinations of SVG with JavaScript and SVG with CSS have good animation smoothness values with fewer than 30 identical frames per second in average. The worst overall animation smoothness values had the technology combinations of SVG with JavaScript, HTML with JavaScript and HTML with CSS, executed in the Firefox browser. The Firefox browser proved to be worse than other browsers at animating with the web technology combination of Canvas with JavaScript.

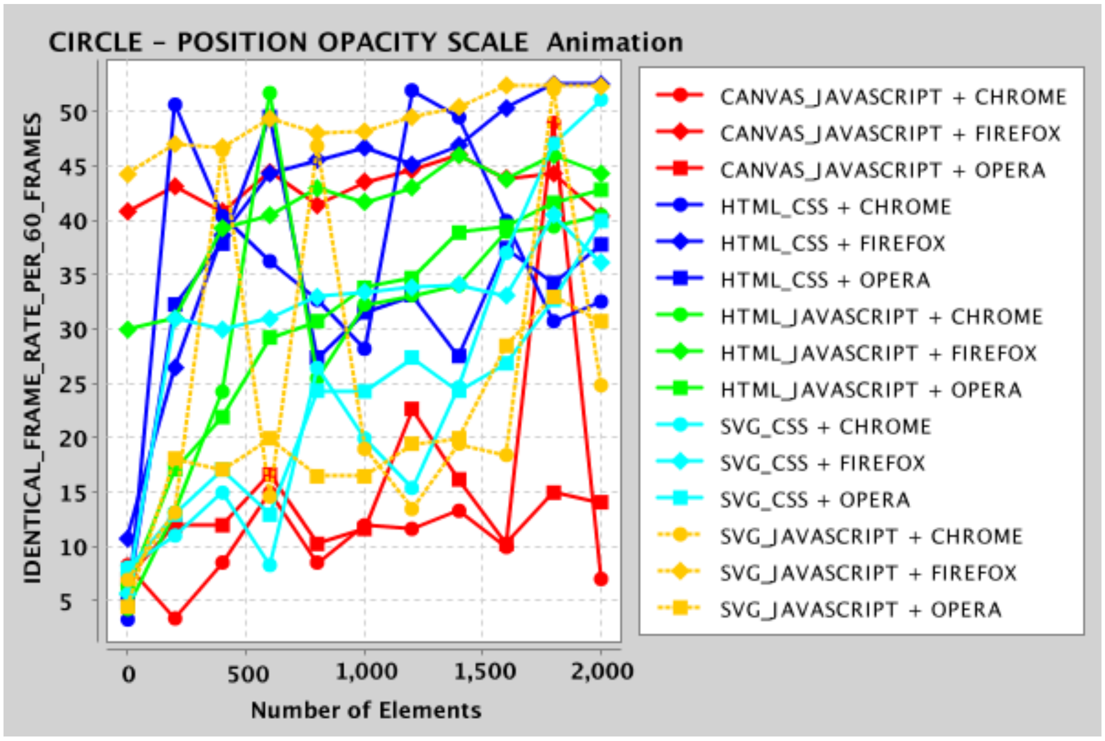

Figure 31: Position Opacity Scale Animation Smoothness of Circles

The smoothest animations overall was achieved by the web technology combination of Canvas with JavaScript executed in browser Google Chrome and Opera. The next best overall animation smoothness had the web technology combination of SVG with JavaScript, followed by SVG with CSS, also run in browsers Google Chrome and Opera. Identical to animating squares the worst animation smoothness had the web technology combination of SVG with JavaScript, executed in the Firefox browser. In contrast to animations of squares, the web technology combination of HTML with CSS shows bad animation smoothness across all tested browsers.

8. Results

In order to find whether an aspect of animated data visualizations is influencing the rendering performance, experiments results will be aggregated and analysed. For every aspect of animated data visualizations, it will be pointed out whether there is an influence on rendering performance. Also it will be explained how big the influence is.

8.1. Web Technology Combinations

The web technology combination has an influence on the rendering performance of static graphic primitives. For every static experiment, the rendering times have been grouped by the web technology combination and the additional styles of graphic primitives. Then the average values have been calculated on the groups of rendering times. Figure 32 shows the average rendering times for web technology combinations.

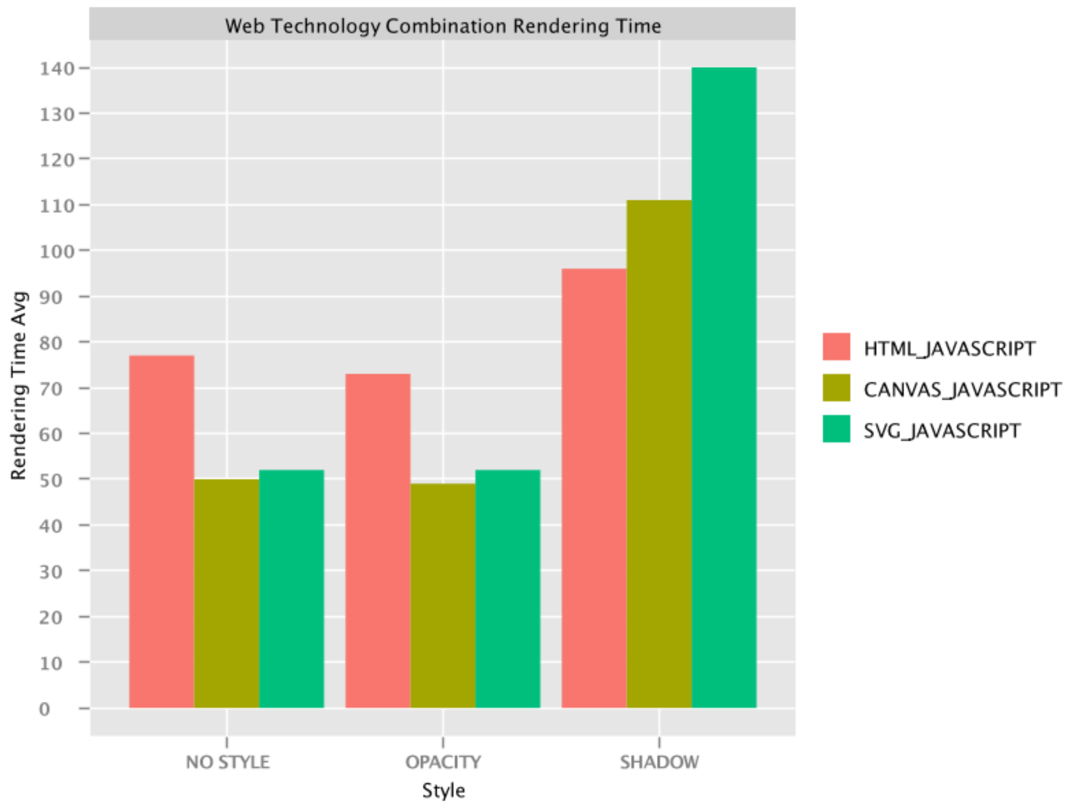

Figure 32: Rendering Times by Web Technology Combination

The results show, that there are significant influences of the web technology combination of the rendering time. At web experiments displaying static graphic primitives with no additional style and shadow, web experiments developed with the web technology combination of HTML with JavaScript took in average 27 frames longer to load, than web experiment developed with SVG with JavaScript or Canvas with JavaScript. Also the web technology combination of HTML with JavaScript need significantly more time to rendering graphic primitives with opacity, compared to other web technology combinations. The fastest web technology combination at rendering graphic primitives with no additional style and opacity is the web technology combination of Canvas with JavaScript. Very close in rendering time is the web technology combination of SVG with JavaScript. In contrast, at rendering graphic primitives with shadow the web technology combination of HTML with JavaScript is the fastest, followed by Canvas with JavaScript and SVG with JavaScript.

To find the influence of the web technology combination on rendering performance of animated graphic primitives, the averages of animation smoothness values have been calculated grouped by the web technology combination and the animated properties. Results are displayed in figure 33. In order to keep the charts with animated properties on the x-axis readable, shortcuts are used for different animated properties throughout the analysis of results. “P“ is position, “O” is opacity, “S” is scale, “H” is height, “L” is length, “PO” is position opacity and “POS” is position opacity scale animation.

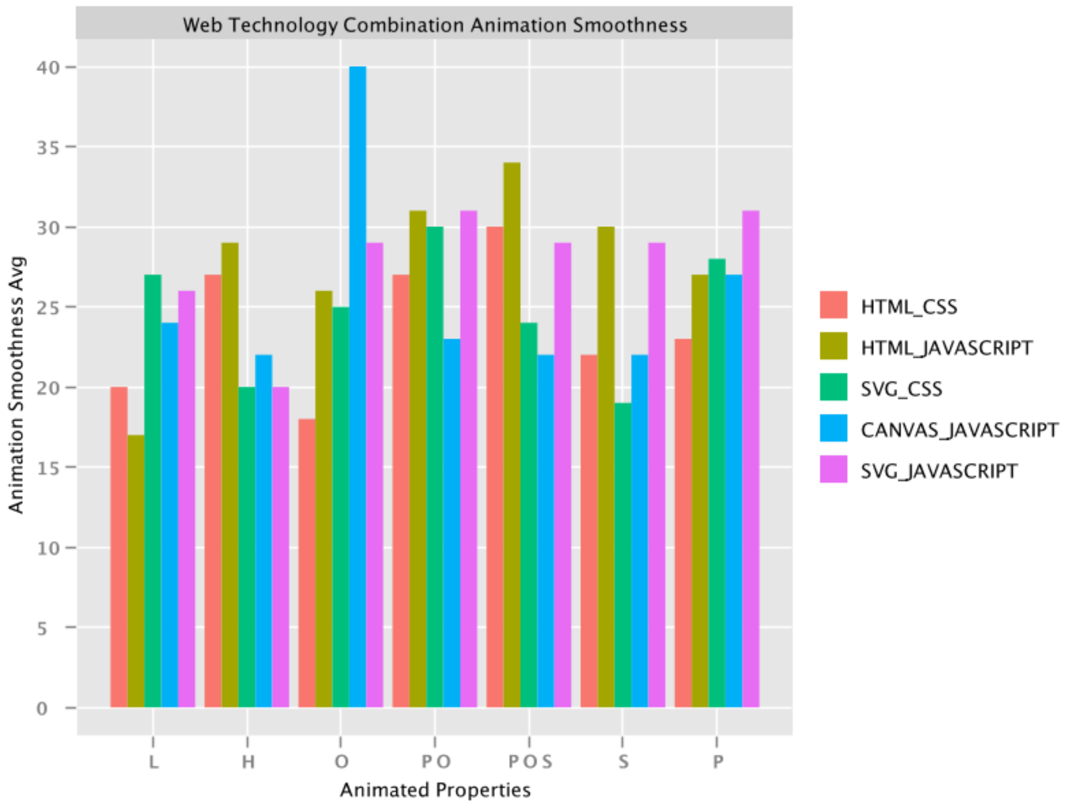

Figure 33: Animation Smoothness by Web Technology Combination

The results show that the influence of web technology combinations on the rendering performance is depending on the animated properties. Different web technology combinations show different strengths and weaknesses, when it comes to animating properties of graphic primitives. The web technology combination of HTML with CSS is best at animating the opacity and the position of graphic primitives. However the web technology combination achieves worse animation smoothness compared to other technology combinations at animating multiple properties simultaneously like position and opacity and position opacity and scale. HTML with JavaScript is the best web technology combination for animating the length of lines. However at height, scale, position opacity and position opacity scale animations, it has the worst animation smoothness overall compared to other web technologies. Web experiments developed with the web technology combination of SVG with CSS have decent animation smoothness values across all animated properties. But the web technology combination achieves the best performance at animating scale. The web technology combination of Canvas with JavaScript is the best technology combination for animating multiple properties simultaneously. However, the animation smoothness values are very bad at animating opacity. Overall, the web technology combination of SVG with JavaScript has the worst animation smoothness value compared to other technologies at all animated property combinations. The performance is decent at animating the height of graphic primitives.

8.2. Graphic Primitives

In order to determine the influence of graphic primitives on the rendering performance static and animated web experiments, results have been analysed separately. At static experiments, all rendering times have been grouped by the styling of graphic primitives and the graphic primitive. The average values have been calculated for each group. Figure 34 shows the rendering times for graphic primitives, among visually identical web experiments.

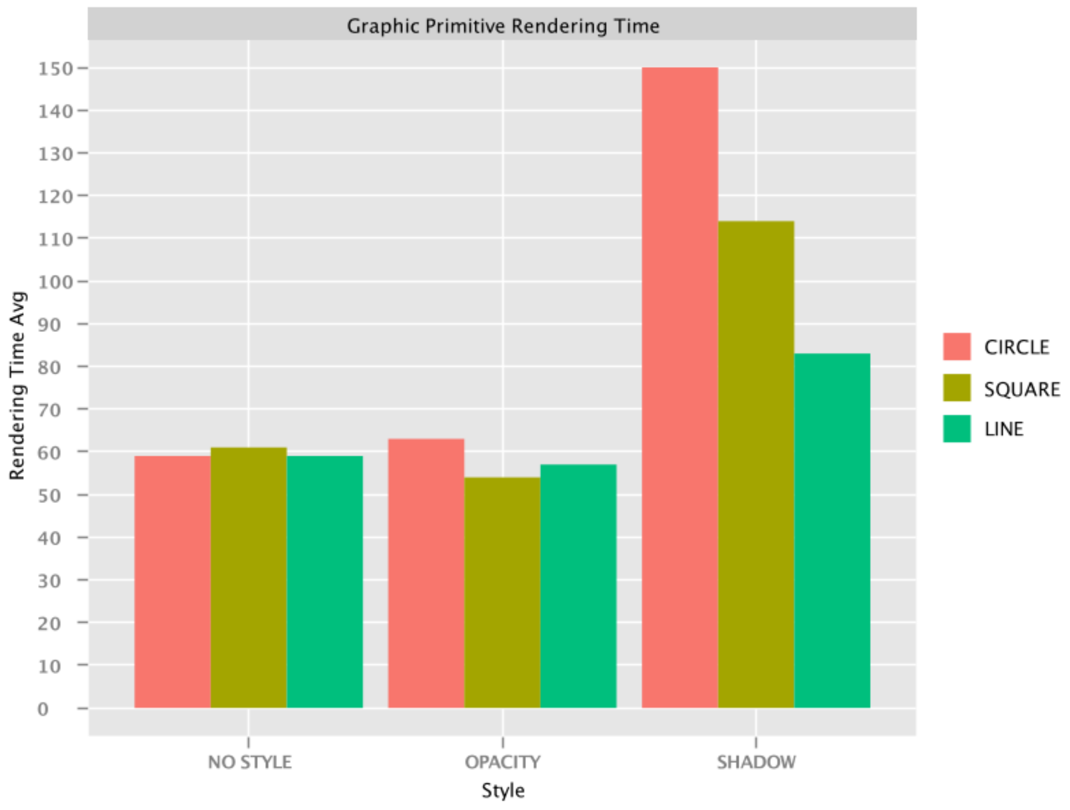

Figure 34: Rendering Times by Graphic Primitives

The chart shows that graphic primitives have almost no influence on the rendering time at web experiments which render graphic primitives with no additional style or an opacity value. However, graphic primitives with a shadow show different rendering times. Circles with shadow have the longest overall rendering time followed by squares and lines.

Also the influence of graphic primitives on the rendering performance of animated web experiments is investigated. Animation smoothness values are grouped by animated properties and graphic primitives. Then the average of all animation smoothness values is build per group. The results are shown in figure 35.

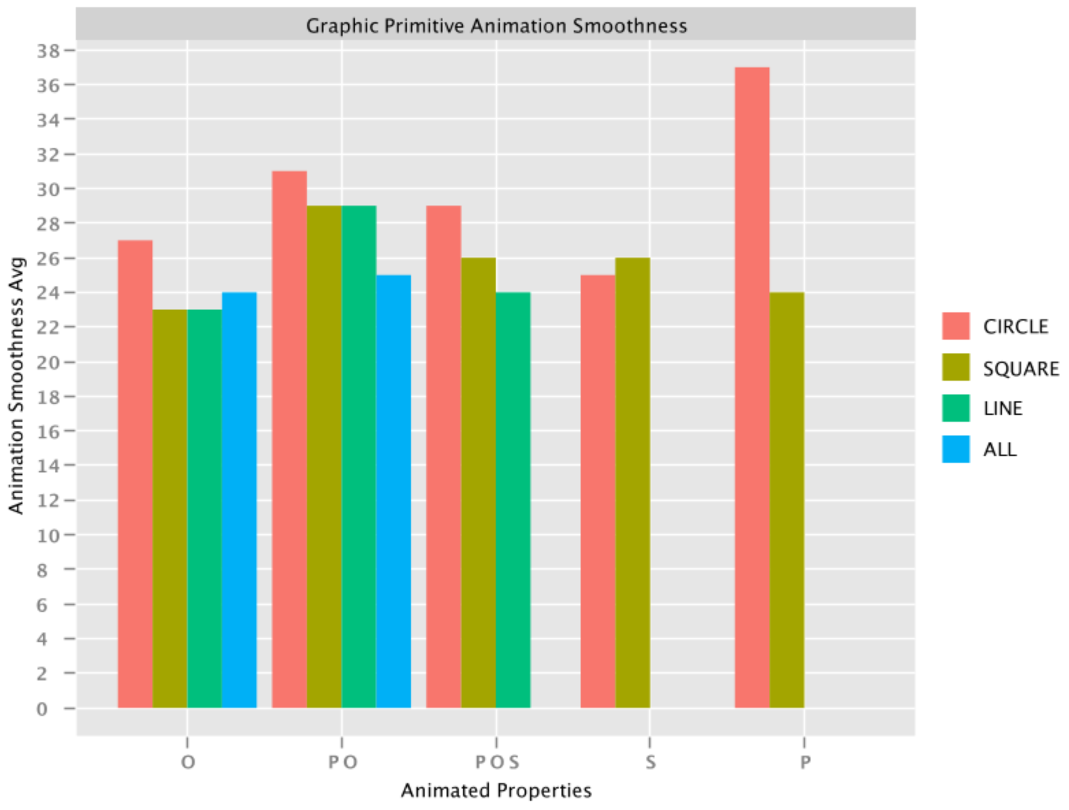

Figure 35: Animation Smoothness by Graphic Primitives

The chart shows, that graphic primitives have an influence on the animation smoothness. Web experiments which animate circles have the worst animation smoothness overall. The average animation smoothness is the worst at multi property position opacity scale and position opacity and opacity animations. At animating the position of graphic primitives, circles cause significantly higher identical frames rates than squares. In contrast, the opacity and scale animations of circles are the smoothest compared to other graphic primitives. Squares have almost no influence on the smoothness of position, opacity and scale animations. Also at the animation of other properties, the usage of squares has little influence on the rendering performance. Similar to squares, lines prove to have little influence on the rendering performance of animated web experiments. Also the animating the position and opacity of combinations of squares, circles and lines does not have a big influence on the rendering performance of animated web experiments.

8.3. Browsers

Identical to the analysis of previous dealt aspects, the influence of browsers on the rendering performance is separated into static and animated. In order to determine the influence of browsers on the rendering time of static experiments, average values of rendering times of all static experiments have been calculated. Rendering times have been grouped by additional style of graphic primitives and browser. Figure 36 shows the rendering times sorted by browser.

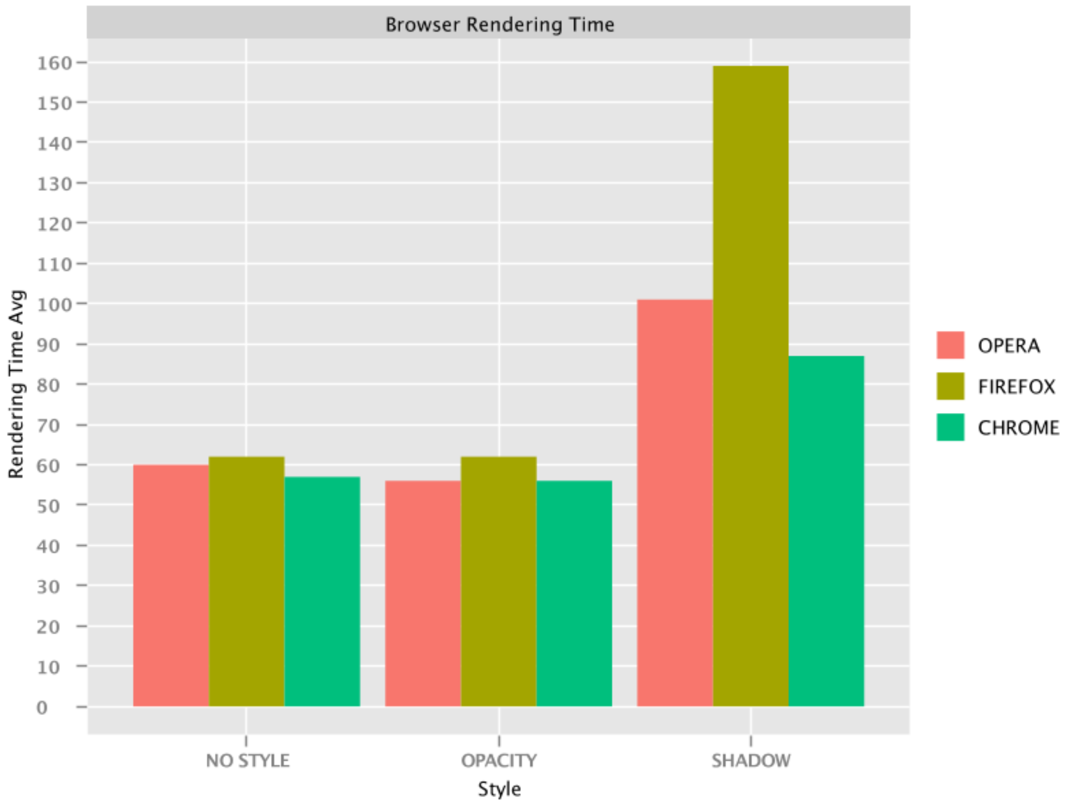

Figure 36: Rendering Times by Browser

The chart shows, that the Google Chrome, Firefox and Opera browser have very little influence on the rendering performance at rendering graphic primitives with not additional style and an opacity value. However, at rendering graphic primitives with shadow the Firefox browser has significantly longer rendering times, compared to the Google Chrome and Opera browser.

To determine the influence of browsers on the animation smoothness of animated web experiments, all animation smoothness values have been grouped by the animated properties and the browsers, the web experiments were run on. Then the averages of animation smoothness groups have been calculated. Figure 37 visualizes the average animation smoothness grouped by animated properties and browsers.

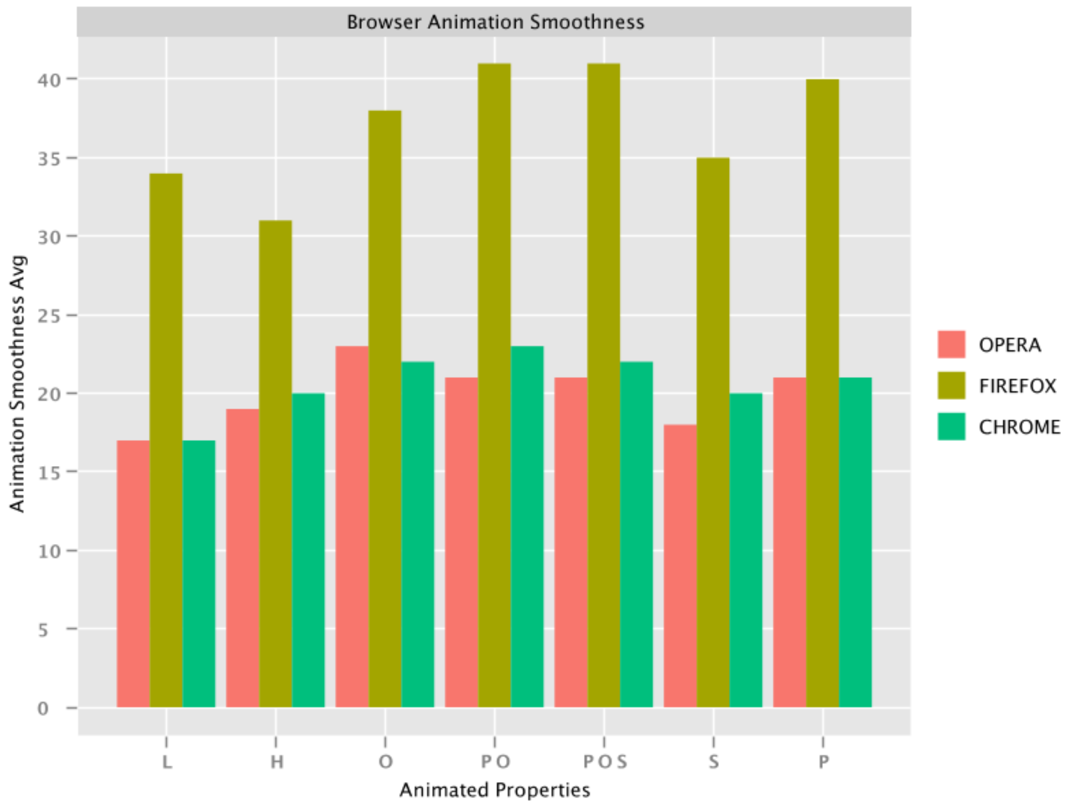

Figure 37: Animation Smoothness Sum by Browser

The chart shows, that browsers influence the smoothness of animations on graphic primitives in a clear pattern. Web experiments run in browsers Google Chrome and Opera have very similar animation smoothness values. Having that said, web experiments run in the Firefox browser have much worse animation smoothness values compared to the other browsers.

Beneath differences in overall rendering performance, browsers showed differences in the variance of their performance. In order to analyse this all animated web experiment values have been drawn on a chart. The web experiments have been coloured by the browser they have been run. Blue is Google Chrome, orange is for Firefox and red is opera. The results are displayed in figure 38.

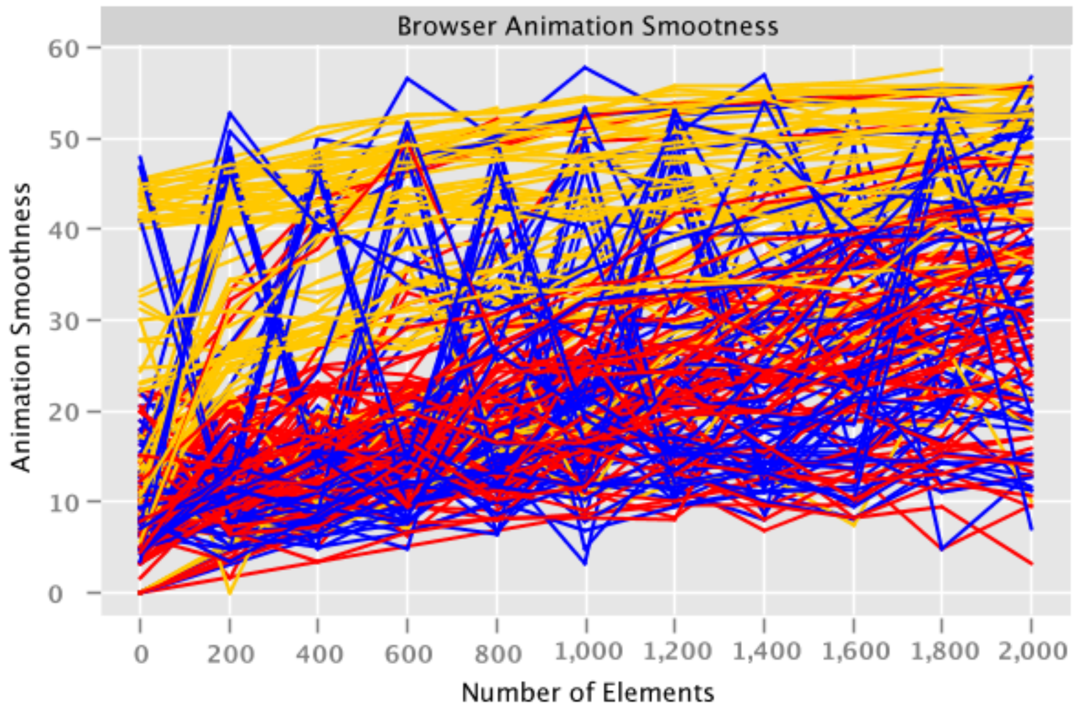

Figure 38: Animation Smoothness by Browser

The chart shows that the Google Chrome browser rendering performance is varying more than on the Firefox and Opera browser. Also it needs to be said that peaks and lows of the variances occur at every number of elements, the web experiment have been run with. As a result, not pattern can be recognized.

8.4. Number of Graphic Primitives

The number of graphic primitives is only loosely correlated with the rendering performance. The goal of this chapter is to investigate how the rendering performance changes with the increasing number of graphic primitives at static and animated web experiments.

In order to analyse the influence of the number of graphic primitives on rendering time, static web experiments have been grouped by the style of their graphic primitives. Then the average has been calculated for rendering times of web experiment iterations with the same number of graphic primitives. Figure 39 shows the rendering time averages grouped by the graphic primitive styles.

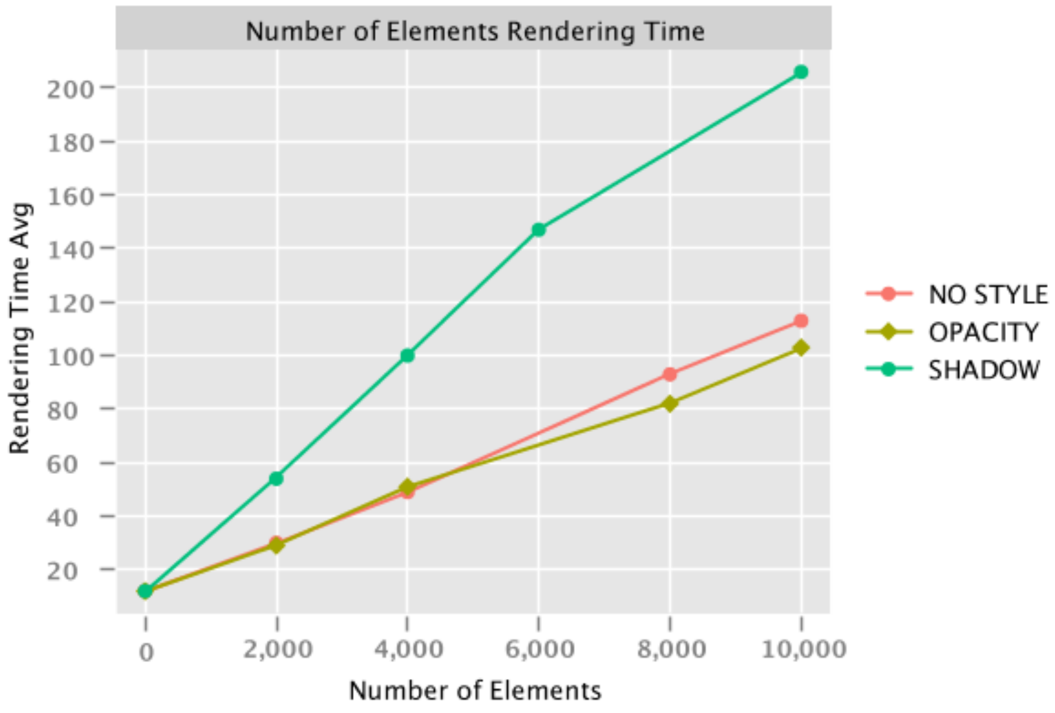

Figure 39: Rendering Times by Number of Elements

Web experiments, which render graphic primitives with no style and an opacity value, have linear rising rendering times. At rendering of graphic primitives with a shadow, the average rendering time shows a kink at 6000 elements and continues to rise slower till 10000 graphic primitives.

To investigate how the animation smoothness changes with increasing numbers of elements, web experiments were grouped by their animated properties. Then average animation smoothness values were calculated from groups of animated web experiment iterations, which had the same number of graphic primitives. The results are visualized in figure 40.

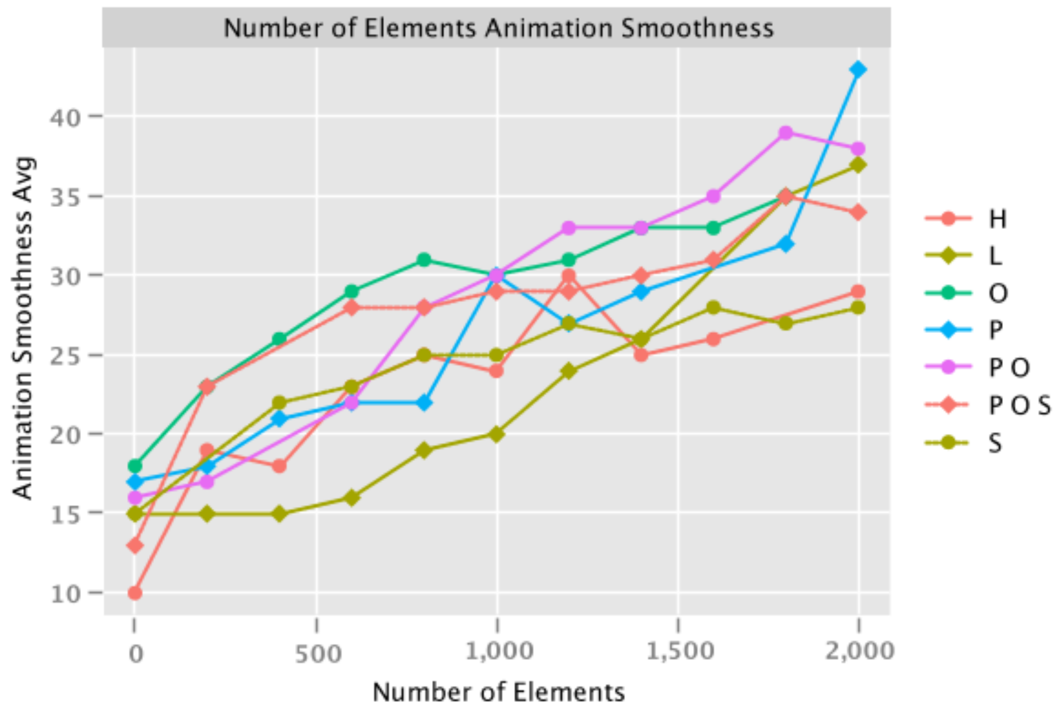

Figure 40: Animation Smoothness by Number of Elements

The average values show that the animation smoothness is not changing linear with increasing number of graphic primitives. At web experiments which animate the opacity, scale and the property combination of position, opacity and scale, the number of identical frames raises faster between one and 1000 elements, compared to other animations. The smoothness of length and position opacity animations rises slower between one and 1000 elements compared to other animations. At web experiments with position or height animations, the animation smoothness is fluctuating strongest in comparison to other animations.

8.5. Styling

In this thesis, the influence of additional styles on the rendering time of graphic primitives has been measured. In order to determine the influence on rendering times, static web experiments have been grouped by their additional style of graphic primitives which are no style, opacity and shadow. Then the average rendering time is calculated on each group. Results are visualized in figure 41.

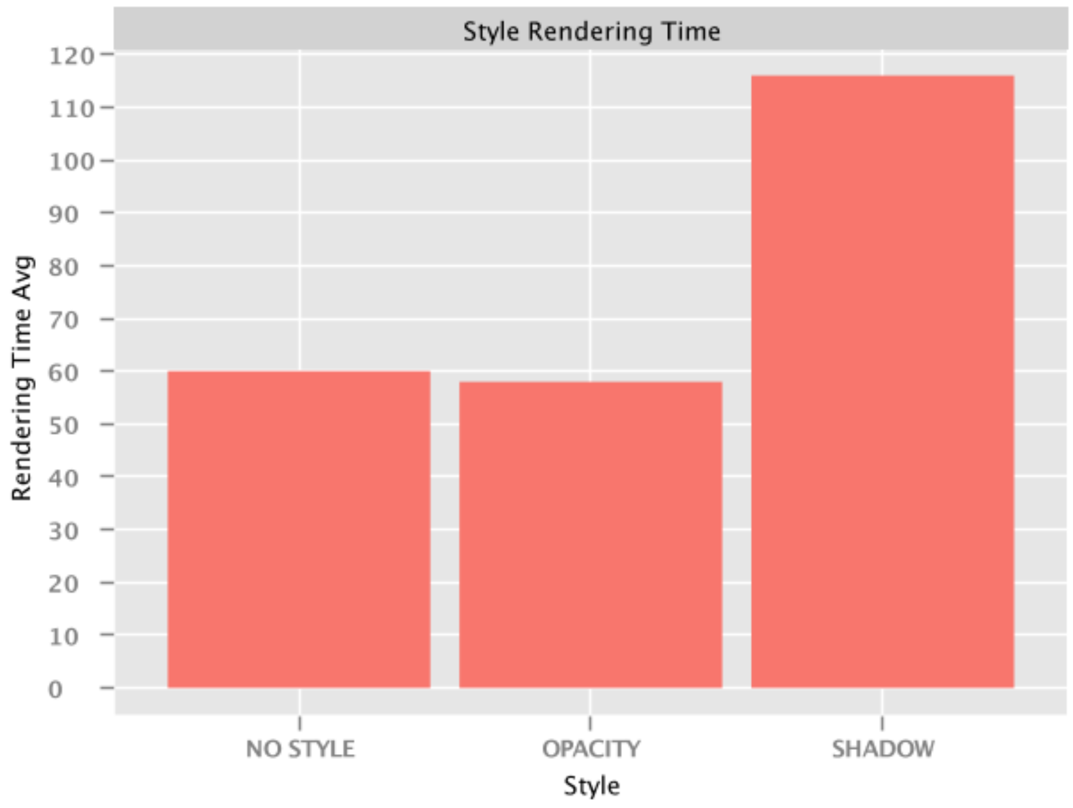

Figure 41: Rendering Time by Style

The results show that both graphic primitives with no additional style and a set opacity value have an average rendering time of 60 frames. Shadow has a big influence on rendering time as it causes graphic primitive to render double the time compared to other styles.

8.6. Animated Properties

Animated properties are an aspect of animated data visualizations which are only part of animated web experiments. In order to investigate the influence on the rendering performance, animation smoothness values of web experiment iterations are grouped by the animated properties of the web experiment. Then, an average animation smoothness value is calculated. Figure 42 shows the animation smoothness averages grouped by animated properties.

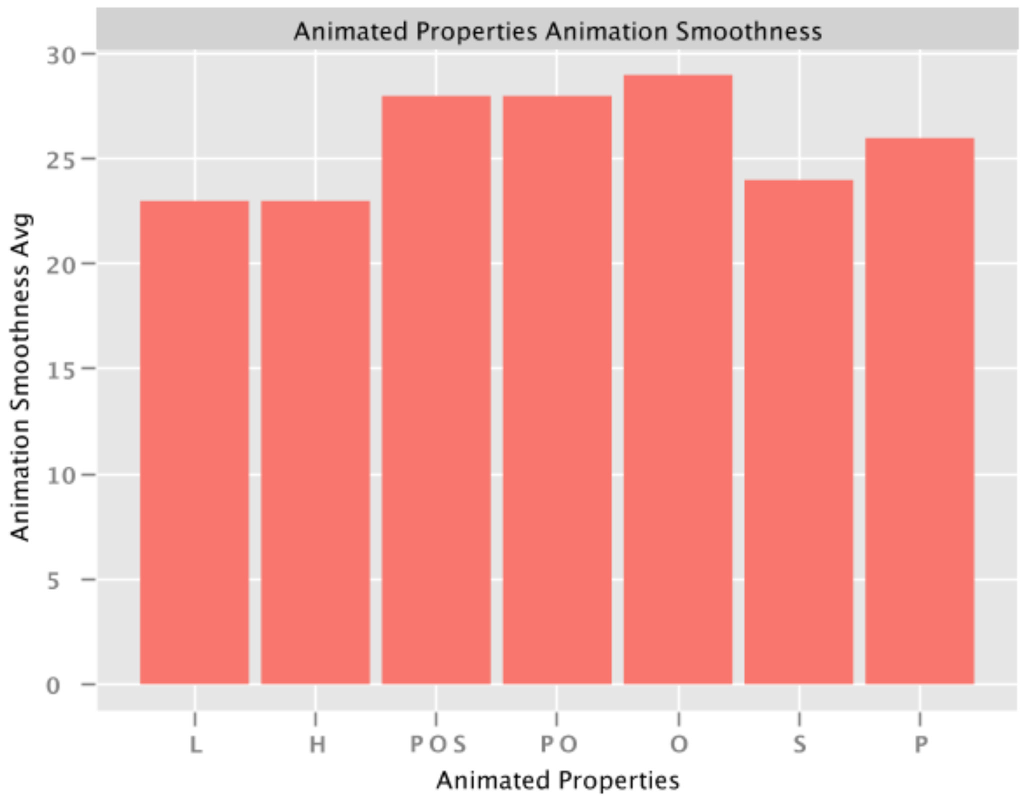

Figure 42: Animation Smoothness Average by Animated Properties

The chart shows that the animations of line and height properties have the smallest influence on rendering performance of animated data visualisations. The next bigger animation smoothness average has the scale animation, followed by position animations. The average animation smoothness value of simultaneous animation of position and opacity is with 27.6 almost identical with the average animation smoothness of position opacity scale animation. The worst overall animation smoothness has the animation of opacity with an average of 28.3. It is important to say, that increasing the number of simultaneously animated properties does not strongly correlate with decreasing animation smoothness. In the tests, the animation of position resulted in slightly smoother animations, than the position opacity and position opacity scale animations. Even only animating the opacity resulted in the highest animation smoothness value compared to position, position opacity and position opacity scale animations.

9. Conclusion

All in all it can be said that all aspects of animated data visualisations have an influence on the rendering performance. However, the degree of influence varies among aspects.

At static experiments the number of graphic primitives has the biggest influence. Graphic primitives with no style and opacity had an average rendering time of 10 for one element and an average rendering time of 100 at 10000 elements. Average rendering times for graphic primitives where ranging from 10 frames to 200 frames. The second biggest influence on rendering time has the aspect of styling. The average rendering time of graphic primitives with style shadow was with 117 frames almost twice as long as the rendering time of graphic primitives with no style or an opacity value. Here the average rendering time was about 60 frames. The third biggest influence on rendering time had the graphic primitive. While at rendering graphic primitives with no additional style and opacity almost no influence on rendering time was recognized, the average rendering time varies between 80 and 150 at graphic primitives with shadow. Similar to the aspect of graphic primitives, the browser has almost no influence on rendering time of graphic primitives with not shadow and opacity. How every at graphic primitives with shadow, the average rendering time varies about 50 frames. The next aspect is the web technology combination, which had an influence on the rendering performance at all static experiments. The influence was a little stronger at web experiments, which displayed graphic primitives with shadow.

At static experiments the most influence on the rendering performance has the aspect of number of graphic primitives animated. At a range of 1 to 2000 graphic primitives, animation smoothness was varying about 22 identical frames between the animation of one and 2000 graphic primitives. The second biggest influence on animation smoothness had the browser. At all animated experiment Google Chrome and Opera had the same performance in average. However the Firefox browser caused web experiment to run animations with 17 identical frames more in average, compared to Google Chrome and Opera. The third biggest influence on rendering performance has the web technology combination. At all combinations of animated properties, the web technology combination has been causing a difference in animation smoothness between 5 and 10 frames in average. Graphic primitives have the fourth biggest influence on rendering performance of animated data visualisations. At opacity, position opacity and position opacity scale animations, the animation smoothness values are varying about 6 to 8 frames, depending on the graphic primitive. At scale animations there seems to be no variance in animation smoothness. However graphic primitives have a big influence on position animations. The last aspect is the combination of animated properties. The animated properties are having the least influence on rendering performance. Even with considering multi property animations, the animation smoothness values were varying about 5 frames between different combinations of animated properties.

10. Future Work

Rendering performance experiments were only run on OSX. It would be interesting to see how aspects of animated data visualisations influence rendering performance on Windows and Linux. The experiment system can be run on other operating systems because all components are working cross platform. Having said that, the experiment system requires platform specific configuration steps for the components FFMpeg and Selenium. The FFMpeg framework needs to be installed for the desired platform. Moreover screen recording, frame extraction and frame cutting is realized in the experiment system by running the ffmpeg executable with the Java Runtime class. The path to the executable has to be set as constant in the *CommandExecutor* class of the experiment system. Screen recording, extract frames and cutting frames are realized by adding different parameters to the execution of the ffmpeg executable. The parameters required for screen recording are different between operating systems. (FFMpeg 2016) As a result parameters need to be adapted for the desired operating system in the *ScreenRecordingThread* class. In addition, the selenium framework requires the paths to drivers for browsers Google Chrome, Firefox and Opera. The paths to the drivers are referenced in the *Browser* enum.

11. References

AMIT, D. 2015. Functional Animation In UX Design. [Online] 26.05.16 https://www.smashingmagazine.com/2015/05/functional-ux-design-animations/

APPLE. 2012. Introduction to Safari CSS Reference. [Online] 2016. 07.04.2016 https://developer.apple.com/library/safari/documentation/AppleApplications/Reference/SafariCSSRef/Introduction.html

BRIGHT, P. 2013. Google going its own way, forking WebKit rendering engine. [Online] 2016. 07.04.2016 http://arstechnica.com/information-technology/2013/04/google-going-its-own-way-forking-webkit-rendering-engine/

CHARTJS. 2016. Animation Configuration. [Online] 22.03.16 http://www.chartjs.org/docs/ - chart-configuration-animation-configuration

CLICKY. 2016. Web browsers (Global marketshare). [Online] 2016. 06.06.16 https://clicky.com/marketshare/global/web-browsers/

FFMPEG. 2016. Capture Desktop. [Online] 2016. 22.05.16 https://trac.ffmpeg.org/wiki/Capture/Desktop

FUSIONCHARTS. 2016. Product. [Online] 22.03.16 http://www.fusioncharts.com/tour/

GITHUB. 2016. Safari 9 Selenium Bug. [Online] 06.06.16. https://github.com/seleniumhq/selenium/issues/1185

GRIGORIK, I. *High Performance Browser Networking: What every web developer should know about networking and web performance*. Edtion ed.: O'Reilly Media, 2013. ISBN 9781449344726.

HIGHCHARTS. 2016. Demos. [Online] 22.03.16 http://www.highcharts.com/demo

HUNDHAUSEN, C. D., S. A. DOUGLAS AND J. T. STASKO A Meta-Study of Algorithm Visualization Effectiveness. Journal of Visual Languages and Computing, 2002, 13(3), 259-290.

IEEE. 2016. List of Accepted Papers for VIS Conference 2015. [Online] 2016. 03.04.16 http://ieeevis.org/year/2015/info/overview-amp-topics/papers

INDERJEET SINGH, J. L., JESSE WILSON. 2011. Gson User Guide. [Online] 2016. 22.03.16 https://sites.google.com/site/gson/gson-user-guide

KEARNEY, M. 2016. The RAIL Performance Model. [Online] 2016. 05.05.16 https://developers.google.com/web/tools/chrome-devtools/profile/evaluate-performance/rail

KEE, D. E., L. SALOWITZ AND R. CHANG Comparing Interactive Web-Based Visualization Rendering Techniques 2012.

LEE, S., J. Y. JO AND Y. KIM. Performance testing of web-based data visualization. In *2014 IEEE International Conference on Systems, Man, and Cybernetics (SMC).* 2014, p. 1648-1653.

LEWIS, P. 2015a. Animations. [Online] 06.03.16 https://developers.google.com/web/fundamentals/design-and-ui/animations/

LEWIS, P. 2015b. Rendering Performance. [Online] 2016. 04.05.16 https://developers.google.com/web/fundamentals/performance/rendering/?hl=en

LEWIS, P. 2016. High Performance Web User Interfaces - Google I/O 2016. [Online] 31.06.16 https://www.youtube.com/watch?v=thNyy5eYfbc

LEWIS, P. AND P. IRISH. 2013. High Performance Animations. [Online] 04.04.16 http://www.html5rocks.com/en/tutorials/speed/high-performance-animations/

MICROSOFT. 2016. Internet Explorer Architecture. [Online] 2016. 07.04.2016 https://msdn.microsoft.com/en-us/en-us/library/aa741312(v=vs.85).aspx

MONGODB. 2015. Introduction to MongoDB. [Online] 2016. 23.05.16 https://docs.mongodb.com/manual/introduction/

MOZILLA. 2013. Project Eideticker. [Online] 2016. 05.05.16 https://wiki.mozilla.org/Project_Eideticker

MOZILLA. 2016. Gecko. [Online] 2016. 07.04.16 https://developer.mozilla.org/en-US/docs/Mozilla/Gecko

NETMARKETSHARE. 2016. Desktop Browser Version Market Share. [Online] 2016. 06.06.16 https://www.netmarketshare.com/browser-market-share.aspx?qprid=2&qpcustomd=0

PROTALINSKI, E. 2013. Opera confirms it will follow Google and ditch WebKit for Blink, as part of its commitment to Chromium. [Online] 2016. 07.04.16 http://thenextweb.com/insider/2013/04/04/opera-confirms-it-will-follow-google-and-ditch-webkit-for-blink-as-part-of-its-commitment-to-chromium/ - gref

ROBERTSON, G., R. FERNANDEZ, D. FISHER, B. LEE, et al. Effectiveness of Animation in Trend Visualization. IEEE Transactions on Visualization and Computer Graphics, 2008, 14(6), 1325-1332.

SELENIUM. 2016. Test Automation for Web Applications. [Online] 07.05.16 http://www.seleniumhq.org/docs/01_introducing_selenium.jsp

SHAPIRO, J. 2014. CSS vs. JS Animation: Which is faster. [Online] 22.03.16 https://davidwalsh.name/css-js-animation

SMITH, P. G. *Professional website performance : optimizing the front-end and back-end*. Edtion ed. Hoboken, N.J.: Wiley, 2013. ISBN 978-1-118-48752-5.

STEELE, J. Beautiful visualization : [looking at data through the eyes of experts]. In *Theory in practice.* Beijing ; Köln [u.a.]: O'Reilly, 2010, p. XVI, 397 S.

STEFANOV, S. Web performance daybook : Volume 2. In. Sebastopol, Calif: O'Reilly, 2012, p. Online-Ressource (1 online resource (1 v.)).

TABALIN, A. 2015. An introducation to Hardware Acceleration with CSS. [Online] 22.03.16. https://www.sitepoint.com/introduction-to-hardware-acceleration-css-animations/

TUM. 2016. Web Animation Benchmark. [Online] 23.06.16 http://home.cs.tum.edu/~stoeck/webanimationperf/

TVERSKY, B., J. B. MORRISON AND M. BETRANCOURT Animation: can it facilitate? International Journal of Human - Computer Studies, 2002, 57(4), 247-262.

W3COUNTER. 2016. Web Browser Market Share. [Online] 2016. 06.06.16 https://www.w3counter.com/globalstats.php

XCHART. 2016. A Simple Charting Library for Java. [Online] 25.05.16 http://knowm.org/open-source/xchart/

ZHU, L. Z. AND F. LI. The collection and analysis of performance parameters in web applications. In *Electronics and Optoelectronics (ICEOE), 2011 International Conference on.* 2011, vol. 3, p. V3-150-V153-154.