Czech University of Life Sciences Prague

Faculty of Economics and Management

Department of Information Technologies (FEM)

Bachelor Thesis

The Impact of Selected CSS Properties and Tools on Web Page Loading and Performance

Ivan Konnikov

CZECH UNIVERSITY OF LIFE SCIENCES PRAGUE

Faculty of Economics and Management

BACHELOR THESIS ASSIGNMENT

Ivan Konnikov

Informatics

Thesis title

The Impact of Selected CSS Properties and Tools on Web Page Loading and Performance

---

Objectives of thesis

The main objective of the thesis is to analyze the influence and identify the limits of selected CSS properties or tools on web applications loading time and performance.

Partial objectives:
- Select relevant CSS properties or tools for further analysis.
- Create an experimental website (or modify an already existing one).
- Design and perform experimental measurements of the website load and performance.

Methodology

The methodology of the theoretical part is based on the study and analysis of professional information sources.

Within the practical stage, already available development tools will be utilised such as, for example, those already built into individual browsers. These tools will be used for the evaluation of how specific CSS elements reduce and affect the loading of a web page and the overall load on the end user's device. One cascading style sheet element at a time will be measured. These measurements will be repeated each time in order to eliminate potential side effects such as unstable internet connections or current CPU load on other processes.

Based on the findings from the theoretical part and the subsequent evaluation of the results from the practical part, conclusions of the thesis will be formulated.

The proposed extent of the thesis

40 – 50 pages

Keywords

CSS, styling, analysis, animations, font, transform, web page, load, grid, display, time

---

Recommended information sources

Duckett, Jon. 2011. ”HTML & CSS: Design and Build Websites”. s.l. : Wiley, 2011. 9781118008188.
Frain, Ben. 2017. ”Enduring CSS”. s.l. : Packt, 2017. 9781787282803.
Frein, Ben. 2012. ”Responsive Web Design with HTML5 and CSS”. s.l. : Packt, 2012. 1839219793.

Expected date of thesis defence

2022/23 SS – FEM

The Bachelor Thesis Supervisor

Ing. Jan Masner, Ph.D.

Supervising department

Department of Information Technologies

Electronic approval: 14. 7. 2022

doc. Ing. Jiří Vaněk, Ph.D.

Head of department

Electronic approval: 27. 10. 2022

doc. Ing. Tomáš Šubrt, Ph.D.

Dean

Prague on 09. 03. 2023

Declaration

I declare that I have worked on my bachelor thesis titled "The Impact of Selected CSS Properties and Tools on Web Page Loading and Performance" by myself and I have used only the sources mentioned at the end of the thesis. As the author of the bachelor thesis, I declare that the thesis does not break any copyrights.

In Prague on 15/03/2023 _____

Acknowledgement

I would like to thank Ing. Jan Masner, Ph. D for his continued support and advice throughout the production of this thesis.

The Impact of Selected CSS Properties and Tools on Web Page Loading and Performance

Abstract

Cascading Style Sheets (or herein "CSS") has contributed significantly towards shaping widespread modern web development standards. Arguably as a bridge between the disciplines of technology and art, it is a crucial component which every competent project is expected to include at least an extent of. Understanding its popularity and importance, the World Wide Web Consortium (W3C) makes continuous updates to the language and expansions to its ever-growing feature set. However, under certain conditions, with examples including an insufficiently fast internet connection or machines featuring less powerful hardware, these additional features hold a tendency to pose excess strain on the limited resources available to display content as originally intended.

This academic article aims towards identifying such limitations of specified CSS properties on web application loading time and overall performance, as well as potentially identifying less resource-intensive alternatives to ultimately achieving the same outcomes if applicable.

Keywords: CSS, styling, analysis, animations, font, transform, web page, load, grid, display, time

Vliv vybraných vlastností a nástrojů CSS na načítání a výkon webové stránky

Abstrakt

Kaskádové styly (nebo zde „CSS“) významně přispěly k utváření rozšířených moderních standardů pro vývoj webových aplikací. Pravděpodobně jako most mezi disciplínami a technologií a uměním je klíčovou složkou, od které se očekává, že každý kompetentní projekt bude zahrnovat alespoň určitý rozsah. World Wide Web Consortium (W3C) chápe jeho popularitu a důležitost a proto neustále aktualizuje jazyk a rozšiřuje jeho stále rostoucí sadu funkcí. Za určitých podmínek, jako je například nedostatečně rychlé připojení k internetu nebo stroje s méně výkonným hardwarem, však tyto dodatečné funkce mají tendenci představovat nadměrné zatížení omezených zdrojů dostupných pro zobrazení obsahu, jak bylo původně zamýšleno.

Tento akademický článek si klade za cíl identifikovat taková omezení specifikovaných vlastností CSS na dobu načítání webových aplikací a celkový výkon a také potenciálně identifikovat alternativy méně náročné na zdroje, jak v konečném důsledku dosáhnout stejných výsledků.

Klíčová slova: CSS, stylizace, analýza, animace, písmo, transformace, webová stránka, načtení, mřížka, zobrazení, čas

Table of Contents

1 Introduction

Cascading Style Sheets (herein “CSS”) has proven to become a crucial tool in modern web development. While Hyper Text Markup Language (herein “HTML”) is employed for the definition of a given document’s markup and structure, and JavaScript is utilized primarily for the addition to its functionality, the addition of CSS ultimately determines all its visual aspects. Its widespread use has grown in previous decades to the extent at which modern web application development can no longer be envisioned without any use of CSS involved. In fact, according to W3Techs [1], external CSS it employed by 95% of existing websites, 93.4% employ inline CSS and 79.5% employ embedded CSS. This is due to the growing widespread understanding of how CSS may affect overall end-user experience, as it allows for the description of how elements can be presented and specific changes in reaction to the end device’s native resolution, allowing for a visually pleasing, responsive overall user experience.

However, although the average end-user’s overall hardware capacity and power in recent years have experienced exponential growth, so has the demands of CSS due to continuous updates and features made available on a frequent basis. In fact, it is not uncommon of an encounter to experience a significant and noticeable drop in performance at present when viewing modern web applications derivative of CSS and its strain on the limited hardware resources available. This comes with no surprise, given how graphical rendering and generation is what accounts for most deterioration of performance, and this fact is not limited merely to applications accessed through Web browsers, but throughout an operating system. And despite the rapid decline in the overall cost of computational hardware, there still remains a global divide in overall access to computers [2]. Yet the end-user’s overall experience is a crucial contributing factor towards the overall success of any given project. Unfortunately, information and research concerning which specific CSS attributes can potentially account for the aforementioned drop in performance is not abundant. Given this, as well as the ever-increasing growth of Internet applications both in popularity and creation in general, the primary purpose behind the selection of this specific thesis and research is to shed light on this vast, yet unexplored realm of the limitations imposed by modern CSS, in hopes that it may contribute in the upcoming future towards decision making during website developmental stages.

2 Objectives and Methodology

2.1 Objectives

The main objective of the thesis is to analyse the influence and identify the limits of selected CSS properties or tools on web applications loading time and performance.

Partial objectives:

- Select relevant CSS properties or tools for further analysis.
- Create an experimental website (or modify an already existing one).
- Design and perform experimental measurements of the website load and performance.

2.2 Methodology

The methodology of the theoretical part is based on the study and analysis of professional information sources.

Within the practical stage, already available development tools will be utilized such as, for example, those already built into individual browsers. These tools will be used for the evaluation of how specific CSS elements reduce and affect the loading of a web page and the overall load on the end user's device. One cascading style sheet element at a time will be measured. These measurements will be repeated each time in order to eliminate potential side effects such as unstable internet connections or current CPU load on other processes.

Based on the findings from the theoretical part and the subsequent evaluation of the results from the practical part, conclusions of the thesis will be formulated.

3 Literature Review

3.1 Fundamental Web Technologies

The ever-expanding prerequisites for merely entering the software development industry alone are vast, with the increasing number frameworks and libraries atop existing numerous programming, scripting, stylesheet and markup languages, as well as other technologies required to develop modern applications. However, as many technologies there are for web development, which collectively make its learning curve particularly daunting, they can mostly be 'boiled down' to a total of three – CSS, JavaScript and HTML, all of which will be covered throughout this section.

3.1.1 Cascading Style Sheets

Cascading Style Sheets, or CSS, was developed by Håkon Wium Lie while working at CERN with Robert Cailliau and Tim Berners-Lee in 1994 [7], with Tim Berners being regarded as the 'father' of HTML. However, the first W3C (World Wide Web Consortium) would not release its first official recommendation until 1996, allowing developers to finally separate their styles from code. The W3C continues to release new recommendations on CSS features to be implemented across browsers to date. CSS itself is a styling language employed for the description and overall presentation of markup language documents, most notably such as HTML. This language is a powerful tool which enables the enhancement of the front end of a given web document, implementing features including fonts, animations, positioning, sizing and colour. The language itself is not compiled and is instead interpreted directly by the browser at execution.

It also enjoys various additional tools to further ease styling development, such as SASS (Syntactically Awesome Style Sheets) which as a pre-processor scripting language notably provides an improved visual representation of CSS code and its hierarchical structure within the document object model [6]. During development and is interpreted by the SASS compiler into "vanilla" CSS.

Unfortunately, not all CSS features are supported entirely by all browsers [8], and it is the responsibility of browser developers to implement the CSS recommendations published by the W3 Consortium. Taking this into account is important to developers to avoid unintended

or non-functional results derivative of cross-browser CSS incompatibility, which will be further elaborated within a dedicated section.

3.1.2 Syntax

CSS definition consists of two factors: the selector and its following declaration enclosed with braces. Furthermore, the declaration itself is composed of properties and values. In CSS, selectors are used to identify the element in question to which declarations are further made [26]. For example, a <p> tag may be identified directly with 'p' as an existing, built-in HTML element type, as is demonstrated in **Source code 1**.

Source Code 1. Example of a CSS declaration.

```
p {
    color: #2a2a2a;
}
```

However, a single document may contain a multiple of said paragraph elements, and by default the browser will apply these declarations to all existing <p> tags within the document which entails unintended consequences. To combat this, as is shown in **Source code 2**, element class attributes allow for identification [52] using dot notation in CSS, allowing for custom classes for intentional differentiation.

Source Code 2. Example of a CSS Class Identification

```
<p class="paragraph">Lorem ipsum dolor sit amet.</p>
<style>
  .paragraph {
     color: rgb(204, 135, 221);
  }
</style>
```

Note that CSS declarations syntactically require a semicolon where the individual declaration ends. CSS class chaining allows for more concrete element identification and involves adhering to the hierarchy defined within the '.html' file [26]. It involves identifying each element by tag type or class with dot notation in hierarchical order, with styling being applied to last identified object by default. Note that although multiple elements may be selected by class, only one element may be selected and receive styling at once with a unique identifier ("id").

3.1.3 Value Types

CSS accepts multiple units of measurement [26] for when defining element declarations, such as for color, margin and padding, to name a few examples, the optimal choice of which is dependent upon the desired outcome. For fonts, among the most common units in use is the pixel, which is an absolute value unit whose dimensions remain consistent across devices regardless of their native display resolution. This applies to attributes related to a given element's distance to/from other elements, the element's size itself (in terms of width and height) or to font sizing, as well as letter spacing and kerning. Yet a core issue with absolute values is how with them, building a web application that is responsive is difficult to achieve, producing undesirable effects on different resolutions. In order to ensure consistency across multiple devices, relative units were introduced to CSS which include em, which are relative to their parent element's dimensions, font or viewport. em is relative to the font size of a parent element, and rem commonly is used for defining font size as it is relative to the root element. Also, noticeably within **Source Code**(s) **2** and **3,** the color property accepts both RGB and hexadecimal values, the difference between which lies simply in how RGB accepts values for red, green and blue of which the desired colour is composed, and hexadecimal values are unique to specific colours across the spectrum.

3.1.4 Media Queries

Expanding on responsive design, another introduction made to CSS to combat resolution constraints is media queries. Media queries are a technique introduced to CSS3 which includes a block of CSS properties which are used only if a specified condition is true [26] as demonstrated in **Figure 1**. Commonly, media queries are employed as a powerful tool for adaptively altering element properties depending upon the current device resolution. The following in **Source Code 3** is a demonstration of the syntax of a media query.

Source Code 3. A media query,

```
@media only screen and (min-width: 600px) {
  .nav-list {
    display: none;
  }
}
```

In the above example is demonstrated how a nav-bar list element may be hidden in the event that the end user's device resolution is above six hundred pixels. Pixels are commonly used

for defining media queries because they represent absolute values across different devices. The value in pixels at which changes in CSS occur due to definitions made within media queries is known as the ‘breakpoint‘. These values may be defined at will, although the generally accepted breakpoints for the following viewports include[24]:

- Mobile devices: 320px - 480px
- Tablet devices: 481px - 768px
- Smaller screens, laptops: 769px -1024px
- Desktops: 1025px -1200px
- Larger screens, TVs: 1201px and above

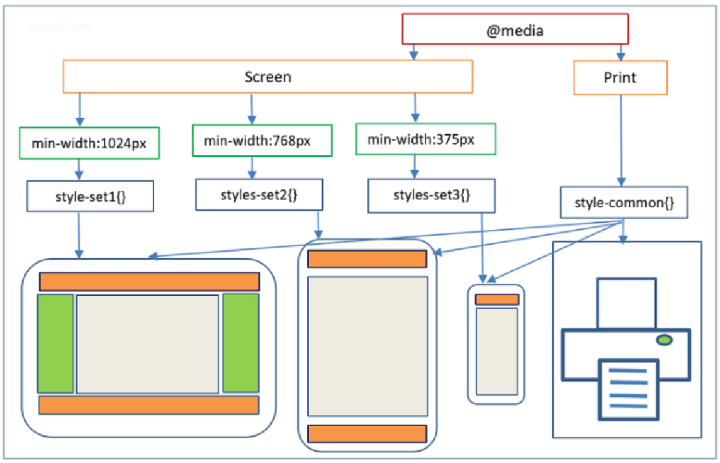


Figure 1. An illustration of media queries. Source:[50]

The importance of media queries is further amplified by the ‘mobile first’ design approach, whereby the UI components of a web application are first implemented for correct intended functionality and rendering on devices with smaller resolutions [52], and only then is implementation expanded to account for the resolution(s) of tablet and desktop devices. This comes to no surprise, given that as of 2022 59.4% of global website traffic originates from a mobile client, which mobile first design aims to avoid neglecting.

3.1.5 Imports

@import allows for importing styling between multiple stylesheets [27]. CSS syntax requires that imports are declared at the top of a CSS document file and imports also apply to media queries. Most commonly, imports are used to provide the browser a location from

which fonts can be downloaded at run-time if the required font files cannot be located on the machine locally. However, some fonts may affect loading performance due to their overall weight and the fact that they are being downloaded over the Web when the server provides the website source code to the end client. It is for this reason that fonts are to be downloaded only at the weights and in the styles that are intended for actual use. It is also recommended against downloading and using multiple font styles for a single project for this reason.

3.1.6 CSS Variables

Variables in CSS are commonly employed when a specific value is used multiple times throughout a project. **Source code 4** demonstrates their implementation. They contribute towards overall code reusability, efficiency and offer convenience in allowing for pre-defining a commonly used reference to the same value. Variables hold global scope [20], meaning that they can be accessed beyond their file of origin, contrastingly to local variables whose use is only possible within their declarative selector. A use case may involve declaring a globally accessible reference to a specific colour in hexadecimal format to be reused within components in accordance with user interface requirements.

Source Code 4. Use of variables in CSS.

```
:root {
  --primary: #2285e9;
  --white: #ffffff;
}

.note {
  background: linear-gradient(90deg, var(--primary) 31%, var(--white)
  100%);
  padding: 1.3em;
  color: var(--white);
}
```

In order to declare variables with global scope, they must be written within a `:root` selector [20], which matches the document's root element. Otherwise, local variables may be declared within their same selector of current use.

3.1.7 Pseudo Elements

In modern web applications the state of a given HTML or JSX element may hold influence over its overall styling. For instance, with a sign-up form, errors may be displayed to the user when entering a password duplicate which does not match or if the password entered does not meet specific criteria for registration. But for code reusability and overall modularity another message box may utilize a class [28] containing the same attributes excluding one (for instance, its background and border colours), depending on the component's current state or other criteria. In this case, a background-colour of red may be used to display issues with the password entered and of green to notify of successful registration into the system. A class becomes a pseudo-class once defined but not in use immediately until the changing of certain conditions, which are often triggered with JavaScript or its frameworks.

Source Code 5. A pseudo element.

```
p::first-line {
  color: #3a5dcf;
}
```

3.1.8 Selectors

Different patterns are used in CSS to target or select an element for the purpose of further styling it, depending on the type of element in question. This was implemented for the purpose of avoiding the targeting of elements with styling rules not intended for them due to how scoping in CSS works [14]. For instance, a specific `<p>` element should be selected differently for the prevention of applying the same styles to all other paragraphs within a given document or `<div>` element. There are five categories of such selectors, which are [14]:

1. Simple selectors, which as previously mentioned, rely on targeting a specific element based off of their class, ID or element type defined with HTML.
2. Combinator which explains the relationship between the selectors provided. Element descendants are defined using a space between the targets provided, element children are targeted more specifically using the ">" operator, adjacent siblings which target elements defined directly after the one selected using the "+" operator and the general

sibling operator which selects elements following the elements of the first selector specified, defined with a "~" .

3. Pseudo-class selectors which target elements based on their current state. For instance, if an `<a href="#">` link was accessed its anchor text's styling may be subjected to change, which in this instance received altered text decoration and colour.
4. Pseudo-elements which are utilized for the purpose of styling a specific part of an element for additional effect. Pseudo elements are defined using double semicolons following the element selector, along with a specific pseudo-element itself. For example, the first line of an element may be styles differently the following way as demonstrated in **Source code 5**.
5. Attribute selectors, which targets elements whose attribute(s) defined with HTML meet a specified criteria. This can be used to target elements which contain a specified attribute, or one featuring a specific value.

3.1.9 Positioning

Positioning elements with the aid of CSS is a fundamental aspect of the styling language itself. It allows to define the location of elements on multiple axis across a web document [29]. Determining a given element's positioning can be achieved using the `position` property which accepts one of the following values:

1. `static`, the default positioning value. Elements which are static are positioned according to the default flow of the web page and are additionally unaffected by the top, bottom, left and right properties.
2. `fixed`, which as the name implies, the element with this positioning remains in one specific location relative to the device's viewport, regardless of window size adjustments. In some cases, fixed elements can be displayed in front of others as they do not adhere to the same positioning rules.
3. `relative`, which works similarly to `static`, with the key difference being the added ability to alter the element's given position. Applying this property alone will not produce any visual results which require the addition of a directional property (top, bottom, left, right) along with a value.

4. `absolute`, which renders elements relative to their first non-static parent. This property notifies the browser that the element does not follow the flow of the document which contains it. Absolutely positioned elements can affect other elements defined before or after them within a HTML document. The main differences from `absolute` lie in that relative positioning relies on their closest non-static parent.
5. `sticky`, a hybrid between `fixed` and `relative`. Sticky elements are treated as relative until they cross a specified threshold, beyond which the position is treated as fixed. An example may be a navbar which follows the user as they scroll through a webpage.

3.1.10 Mixins

Mixins are a feature available strictly to pre-processors such as Sass, Less and Stylus, and are subsequently not native to CSS, nor are they valid CSS syntax. They are instead compiled to valid CSS from pre-processors with the use of dedicated compilers. Mixins are widely used for the purpose of writing cleaner code by the means of pre-defined declarations to be used repeatedly on the styles of different elements. Mixins can also be passed arguments related to the element(s) in question. The use of mixins generally involves the declaration of a mixing itself and its application to an element via the insertion of its name using the `@include` key word [32], as shown within the following snippet.

Source Code 6. A Sass mixin.

```
@mixin grid($cols, $mgn) {
  float: left;
  width: ((100% - (($cols - 1) * $mgn)) / $cols );
  margin-right: $mgn;
  margin-bottom: $mgn;
  &:nth-child(#{$cols}n){
    margin-right: 0;
  }
}

li {
      @include grid(6, 2%);
      img {
            width: 100%;
      }
```

```
}
```

3.1.11 CSS Functions

Although some functions are available natively to CSS, including the functions `max()` and `max()`, customized functions may be defined with the use of a CSS pre-processor. Whereas a mixin is used to apply the same pre-defined properties to multiple elements, the purpose of functions is to return single values for further use within property definitions. Also, similarly to mixins, the use of functions lies in the definition of a function with its name and arguments being applied to the element's property in question. The arguments with which functions are handed can be changed at will, although the same number of arguments as within its definition is required. Optional values are supported also to substitute values which have not been passed with a default setting. Arbitrary functions (or functions capable of accepting any number of arguments) can also be declared with the addition of '`...`' as demonstrated in **Source code 9** to the end of an argument name. Although Sass functions may be used for side effects, it is generally advised to instead use mixins for this specific purpose.

Source Code 7. A function written in CSS.

```
@function sum($arguments...) {
  $sum: 0;
  @each $number in $arguments {
    $sum: $sum + $number;
  }
  @return $sum;
}

.micro {
  width: sum(40px, 35px, 90px);
}
```

3.1.12 Fonts

Fonts are and have been a crucial part of CSS since their implementation in 1997[12] following the creation of the styling language. With the importance of typography being recognized even at a time at which websites mostly consisted of text alone, one of CSS' oldest properties is font-weight. Fonts can be used for web applications and websites alike using one of either two methods: downloading the font to the system for local use or defining

a source from which to automatically download the font when the web page is accessed online.

For local access, as shown in **Source code 6,** it is only necessary to define the `font-family` attribute as well as its value for the selector concerned (implying that the font itself has already been downloaded).

Source Code 8. A CSS font declaration.

```
.errorMsg {
      font-family: roboto-regular;
}
```

Bearing in mind that it is important to consider scope, as the same font definitions will apply to an element's children which will inherit the same typographic styling [12]. However, the approach demonstrated above is, by in large, practical only for the purpose of local projects as the font family specified may not be downloaded to the end user's machine. For this reason, it is necessary to import the font the following way:

Source Code 9. A font import.

```
@import
url('https://fonts.googleapis.com/css?family=Open+Sans&display=swap')
;

.errorMsg {
  font-family: Poppins, sans-serif;
}
```

It is also worthy of note that this method may pose a strain upon networking resources, as the same font will be re-downloaded each time the page is reloaded and its respective files are requested from the server which hosts them [12]. In addition, greater variety in font weights may result in negatively affecting web performance due to the increased material downloaded from requests. For this reason it is recommended to avoid implementing fonts which are seldom or not used at all throughout a project's production stages.

3.1.13 Font Types

Due to various compatibility issues between different browsers (which are subsequently built and maintained by different developers), the CSS `font-family` property is designed with a fallback system [13] included which holds several font names by default for the eventual event of a downloaded font being rendered inaccessible, be it due to server-related issues or an ill-performant internet connection, among others. The font used by default is Times New Roman, recognizable for being the first font in use automatically when creating a project which has yet to receive customized styling. Font family declarations should be made ending with a generic font family to which the browser can make a last resort. It is syntactically correct to define families whose names contain spaces with separated commas, and stylistically recommended to include a similar time.

The five font families which CSS features include:

1. Serif fonts whose letters include strokes. For example,

   Georgia

2. Sans-serif fonts whose letters consist of cleaner lines without strokes, most often used for a "modern" overall look. For example,

   Arial

3. Monospace fonts whose letters are set at a defined fixed (identical) width, creating a "robotic" or "mechanical" effect. For example,

   Courier New

4. Cursive fonts which mimic human handwriting. For example,

   Brush Script MT

5. Fantasy fonts which are more used for a decorative effect depending on stylistic theme. For example,

   Papyrus

Not only do CSS generic fonts feature the above specified types and non-default or other custom fonts may be used of them, as well.

3.1.14 Z-Index

CSS elements can be arranged not only horizontally and vertically, but also within a 3D plane in which elements can be positioned atop each other depending on their current z-index value [15]. There are various use cases for the inclusion of a z-index, including the positioning of text atop an image for stylistic effect. This can be achieved using the z-index property which accepts both positive and negative integer values. Generally, elements assigned higher z-index values will be positioned above others which hold lower values as a result. Although it is worthy of note that this property may only be applied to currently positioned elements, as well as flex items. By default, elements without this property assigned will have the last HTML-defined element overlap the other.

3.1.15 HTML

HTML is a markup language responsible primarily for the definition the skeletal structure of a given document and its content. It is the widely accepted standard for documents due to be displayed on a web browser. As mentioned previously, its first version, known then as XHTML was created by Tim Berners Lee in 1993 [9] as an iteration of the existing powerful Extensible Markup Language, and subsequently earned the status of becoming a recognised official standard in 1999. No web document can be created without the implementation of HTML, as all modifications on other languages, including both CSS and JavaScript, are made to the DOM (Document Object Model) generated by the browser using HTML.

The markup language is used for the definition of a document's structure and content for further manipulation with other suited technologies [10]. It is an interpreted language, and browsers have been developed to be rather forgiving and permissive of syntactical errors and may continue to parse, as well as display the content defined with incorrect or poorly written HTML. Arguably, the reason behind HTML being parsed permissively by browsers is the priority being placed upon the creation of websites to populate the Web when it was first created, hence why browsers usually feature in-built rules for specific cases on how to handle logical or syntactical errors to ultimately allow for content to be successfully displayed easily.

3.1.16 Syntax and Structure

A HTML document consists of tags of two types: paired and singular. HTML syntax requires that paired tags include a closing tag, defined with a slash ("/"), whereas singular tags are defined only with the opening tag itself. These tags can be further added to using attributes containing additional data relevant to the tag itself [26]. Use cases of attributes include forms which may require the presence of the 'for', 'type' and 'name' attributes for intended functionality. Importantly, valid and well-formed HTML requires a hierarchy of elements defined with tag enclosure, with the most notable example being the <html> tag which encloses the entire document and its contents, as demonstrated in **Source code 10**. Well-formed and valid HTML essentially sets requirements in the definition of HTML in order for browsers to correctly parse the document, although many browse engines allow for certain degrees of minor syntactical error. The definition of the elements it contains allows for direct manipulation with the use of CSS or JavaScript with the use of attributes [26]. Commonly for the application of CSS, a widely accepted industry standard exists whereby a class attribute is commonly applied to elements for identification. Similarly, with JavaScript an 'id' (herein "ID") attribute may be applied for the same purpose. Both classes and IDs may either be static, or dynamic and be subject to future changes depending on the developer's intentions and end goal.

Source Code 10. Example of a HTML file.

```
<!DOCTYPE html>
<html lang="en">
<head>
   <meta charset="UTF-8">
   <meta http-equiv="X-UA-Compatible" content="IE=edge">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
   <title>Example</title>
   <link rel="stylesheet" href="./style.css">
</head>
<body>
   <form class="form">
      <input type="radio" id="html" name="fav_language" value="HTML">
      <label for="html">HTML</label><br>
      <input type="radio" id="css" name="fav_language" value="CSS">
      <label for="css">CSS</label><br>
          <input type="radio" id="javascript" name="fav_language" value="JavaScript">
      <label for="javascript">JavaScript</label>
     </form>
   <script src="./script.js"></script>
</body>
</html>
```

3.1.17 JavaScript

Developed by Brenden Eich of the Netscape Communications Corporation in the United States in 1997, JavaScript is a scripting and/or programming language which is a crucial building block in the construction of robust web applications as it allows for implementing user interactivity and logic, featuring a familiar syntax reminiscent of C [25]. It allows for the dynamic updating of document content, creation of animations and adding of overall functionality to a website to name a few examples. It is continuously updated and added to by the European Computer Manufacturer's Association (or ECMA) [25], which is a recognized neutral association responsible for developing standards for JavaScript which browsers are collectively required to eventually have implemented. The relevance of JavaScript in the context of this article is in how many features which require the functionality which it offers cannot be constructed at least conveniently or practically using CSS alone. It is an interpreted programming language as opposed to being a compiled, as it is not converted directly into code readable to machines, and browsers instead feature a dedicated interpreter active at code execution [5]. Although JavaScript is not "officially" an object-oriented programming language, ECMAScript 6 (or ES6) implemented approaches towards emulating OOP in a syntax familiar to developers, whereby in vanilla JavaScript, classes are compiled instead as functions, whereas they syntactically appear to be in-built on the surface. While HTML and CSS can be employed for structuring and displaying, as well as styling web content, JavaScript is often used to provide the data being displayed with asynchronous methods requesting it from a server located elsewhere. What additionally makes it a powerful programming language is the potential interactivity it offers with HTML and CSS, allowing for the direct change of element content and its CSS properties.

JavaScript plays an essential role in web development due to its sole responsibility of overall functionality. Although this thesis is primarily concerned with the performance of CSS attributes, the following practical part will involve a test application built with JavaScript functional components. Furthermore, CSS can be directly altered with JavaScript as will be made apparent further into this section [53].

3.1.18 Variables in JavaScript

First and foremost, the values supported by the language can be divided into literals and variables, the difference between which being how literals are static fixed values,

whereas values which may be subject to change are variables. Generally, the two literal types in JavaScript are strings and numbers [34]. Unlike other programming languages like C# or Scala, JavaScript does not differentiate between numeric types (such as Integers and Longs) at least at definition. Strict typing is instead a solution offered by TypeScript. The variables used to store data can be assigned to the `const`, `var` and `let` key terms. Constants are static variables which cannot be changed, and attempts to do so will subsequently result in an error. It is also an alternative means by which to declare functions as a constant. On the other hand, variables made with the `var` and `let` declarations can be changed, although with a notable difference – in how `var` is function scoped, whereas `let` is block scoped. It is therefore recommended to make declarations with the `let` key term, so as to avoid potential conflict and avoid functional side effects and retain scope restriction. In addition, the language classifies primitive types (Strings Booleans and Numbers) and the three main reference types (Arrays, Object and Functions), the latter of which are referenced by their stored position in memory. For this reason, reference types which hold the same value(s) will pass neither strict, nor loose checks for equality (using the '`==`' and '`===`' operators).

3.1.19 Validity

JavaScript variable names and/or keywords cannot begin with a number, in which case the code would result in an error. This aids the JavaScript engine of the browser to easily differentiate between numbers and unique identifiers [34]. Names considered syntactically legal must have a first character which is either a letter (regardless of case), a dollar sign or an underscore. In previous iterations of the language, due to the `const` key term not yet being implemented, developers would resort to the use of underscores before implied constant variables (as demonstrated in **Source code 11**) in order to signify to others that the specific value in question should remain unaltered, although the compiler would not treat it differently as a result and this was done purely out of general convention. Dollar signs are used to apply template literals, which are strings which contain a variable. Syntactically, template literals require the use of back ticks for the variable to be recognized, which would otherwise be treated as a string.

Source Code 11. Constant Declarations.

```
const constant = "test";
let _oldConstant = "old style";

const getVal = (value) => {
  return `value: ${value}`;
};

  printVal(constant);
```

Like in other programming/scripting languages, JavaScript also supports comments as well. Including inline comments and comments which span multiple lines. Code which is commented out will be ignored at compilation [34]. Syntactically, it is correct to write definitions in both upper and lower camel case or with underscores for separation – although despite its historic significance, the use of hyphens is invalid as they are reserved for subtraction, which suggests greater safety in holding a preference over expressions.

3.1.20 Functions in JavaScript

Functions are used for the purpose of performing operations. Because these pre-defined functions can be 'called' at will, they contribute towards overall code reusability. They offer the advantage of having to write less code in order to perform a common specific task. By default, they can accept one argument, multiple or no arguments at all. Many common methods used in JavaScript, such as `console.log()` are methods belonging to existing objects (in this case, the console). Like in other scripting/programming languages, they can return a value with the `return` key term, which subsequently causes the function to terminate [36]. Invoking a defined function can be done with one of three methods: either from an event listener, with direct invocation or self-invocation. Since functions with return statements return a value to their caller, function themselves can be used for expressive variable value declarations. A function can either be declared with the `function` key term or as a constant arrow function (as demonstrated in **Source code 12**), the difference between which lies in their behaviour in relation to lexical scope. For instance, the block of an arrow function cannot be used with `this` or `arguments`. Furthermore, arrow functions cannot be used as constructors as they lack the required prototype property.

Source Code 12. Function types.

```
const arrowSquareNumber = (x: number) => {
  return x * x;
}
arrowSquareNumber(10); // invocation, returns 100

(function square (x: number) {
  return x * x; // returns 25
})(5); // self-invoking
```

3.1.21 Equality Operator Influence on Performance

Interestingly, according to research conducted by the Twitter software developer Ben Cherry, the strict operators enjoy a marginal speed of "roughly 10%" [33] on average in comparison to lose operators when combined with explicit type conversions. The results were produced from twenty-four different tests including over two million iterations across six internet browsers, although the number of iterations were reduced to five hundred thousand for Internet Explorer 8 Interestingly, a performance reduction of up to 20% [33] was prevalent in tests conducted with Mozilla Firefox 3.6. In addition, it was discovered that converting integers for comparison with another integer, as opposed to simply comparing two pre-existing integers, entails significant performance reduction, concluding that integers should not be stored as strings internally, so as to avoid a performance penalty.

3.1.22 CSS Manipulation with JavaScript

CSS becomes a powerful tool when utilised in combination with JavaScript for direct DOM element manipulation. Among the most common use cases would be setting individual styles in specific scenarios, such as, for instance, changing an element's display property as a result of an altered state. This can be triggered with JavaScript event listeners placed onto DOM elements. For direct DOM manipulation, the element must first be identified beforehand, using the `document.querySelector()` method and likely saved to a variable for compressed code and reuse [37]. Changes to styles can be implemented either by means of direct style manipulation (`element.style`) or via the alteration of its existing class list (`classList.add()` or `remove()`), which involves the addition or removal of existing classes defined within the imported CSS file [37], as is demonstrated in **Source**

code 13. Similarly, classes assigned to elements can be switched as opposed to the removal and/or addition of class names with `classList.toggleClass()`.

Source Code 13. Style manipulation in React.

```
// vanilla JavaScript
  button.addEventListener('click', () => {
      button.style.color = 'red';
      button.classList.toggleClass('.test');
  });

// React
const handleClick = () => {
    button.styles.backgroundColor = '#fff';
    button.styles.color = '';
  }

<ButtonActive input="test" onClick={handleClick} />
```

3.1.23 Asynchronous JavaScript

By default, as JavaScript code is a single-threaded scripting language, code execution is performed in order. This entails the possibility of performance issues when synchronous operations occur which take a considerable amount of time [43]. This issue may come in the form of requesting resources external to the application itself which require a HTTP request, and said requests can produce a perceivable drop in performance [43] derivative not of the browser itself, but instead how resource gathering works over a network, the time of which is dependent upon numerous variables – including network transmission speeds or server workloads and configurations. To circumvent this issue, JavaScript offers asynchronous features to allow code to execute in the background without requiring other functions to await its termination for execution, as demonstrated in **Source code 14**. Asynchronous functions are declared using the `async` keyword and promises to be fulfilled are declared with `await`. Note how in the snippet below, it is demonstrated how code below the asynchronous function is not interrupted.

Source Code 14. An asynchronous function.

```
const getTitle = async () => {
  const endpoint = "https://jsonplaceholder.typicode.com/todos/1";

  try {
    const response = await fetch(endpoint);
```

```
    if (response.ok) {
      const jsonResponse = await response.json();
      console.log(jsonResponse.title); // returned later
    };
  } catch (error) {
    console.log(error);
  };
};

console.log('test'); // returned first

getTitle(); // returns 'test', then the response
```

3.2 Related Technologies

Rarely at present are used the core default (or "vanilla") technologies alone. Although use of HTML, CSS and JavaScript may serve as a solid foundation upon which to develop skills in, and an understanding of web and software development, in practice and most especially with considerably larger enterprise projects, reliance upon them alone leads to elevated time consumption and inefficiency. For this purpose, technologies such as various pre-written external libraries and frameworks are in frequent use for the simplification of tasks and their faster completion.

3.2.1 Frameworks

In web development, frameworks are a set of additional resources and/or tools used in the production or maintenance of web applications [23], whether it be for the sake of convenience or out of necessity in some cases. Some frameworks may feature environments for data flow scripting and API data access, or templating capabilities which determine how components may be presented. This should not be confused with external libraries, which are instead collections of predefined functions, classes or components which ease the development process. For instance, Angular is a TypeScript based framework whereas the Carbon DS is an open source library of web components. In the context of CSS development, frameworks are generally collections which provide tools which ease the process of creating and implementing UI components across different pages or projects. Popular CSS frameworks include Bootstrap, Tailwind CSS and Uikit, to name a few examples. Bootstrap aids in designing websites or components faster as it already includes existing templates

written with HTML and CSS for common components including forms, typography, navigation and buttons.

3.2.2 Sass

Designed originally by Hampton Catlin and developed by Natalie Weizenbaum in 2006, Sass is a superset and pre-processor of CSS also used for styling elements within a document [16]. It stands for "Syntactically Awesome Stylesheet" and is completely compatible with all released versions of CSS. Its main purpose is the overall reduction of definition which leads ultimately to decreased storage use for applications as well as generally cleaner code. In addition, Sass introduced other features not yet available with original CSS itself, including nested rules, variables, inheritance and built-in functions, to name a few examples. Browsers themselves are not designed to directly interpret Sass code, which leads to the responsibility of a Sass pre-processor to convert it into browser-readable CSS, a process referred to as "transpiling". Sass files are stored using the '.scss' format. CSS code written with Sass follows the same pattern in which elements are defined with HTML in descending order; with parent blocks enveloping their children.

3.2.3 Less

Developed by Alexis Sellier and Dmitry Fadeyev in 2009, Linear Style Sheets is an open-source alternative to Sass as a dynamic pre-processor to standalone CSS [31]. It utilises an approach similar to the block formatting syntax of Sass allowing for ease in applying styles while traversing a block hierarchy similar to that of the DOM. Although it was originally written in Ruby, those adaptations were depreciated for the replacement with JavaScript. A primary difference between Less and competing pre-processors that it allows for real-time compilation. Sass also supports the variable, nesting and mixin mechanisms. Similarly to Sass, Less also supports functions. There are many other similarities [56] held with Sass, as Less was mostly based off of this specific technology – although Less aims to more closely mirror valid CSS code. Less can be applied to web applications with a 'less.js' JavaScript file or to render Less code into pure CSS.

3.2.4 Stylus

Stylus is an open-source dynamic CSS pre-processor designed and developed originally by TJ Holowaychuk, which is written in Node.js and JADE [32]. Among the most noticeable differences from CSS itself and competing pre-processors is its syntactical approach to block definition, which contrastingly relies on indentation, as opposed to the braces with which developers are most familiar. Additionally, the use of semicolons to end declarations is entirely optional and the lack thereof does not entail syntactical error(s). This also applies to commas and colons, which similarly are not required. The reason for this lies in its approach towards easing the job of a front-end developer. It was originally based off of a combination of Sass and Less, and thus supports mixins, functions and string interpolation useful for dynamic values subject to change. Stylus enjoys the IDE support of Visual Studio Code via a dedicated extension, as well as WebStorm where it is built-in.

3.3 Browser Components and Functionality

Over the course of this section, a detailed description of the prime components of a browser will be provided, as well as how they interact in order to retrieve, parse and render document resources to the end user. A strong understanding of these concepts provides vivid insight into what affects end user experience, and this section will serve as foundation to the metrics used for the measurement of performance within the practical section of this thesis.

3.3.1 The Document Object Model

The DOM (Document Object Model) is a logical tree-like structure consisting of nodes which mirrors the hierarchical structure of a web document in-memory [21]. It is relied heavily upon by all three major web engineering related technologies and their libraries, transpilers and frameworks. Its primary purpose is to represent the document‘s existing objects, as well as their data for the purposes of organisation, presentation and modification. The DOM is not static, and can be manipulated at will as changes are made to itself and the nodes it contains, as well as their current properties (such as their class names or inner text). It is therefore the means by which programming and styling languages such as JavaScript and CSS may achieve direct interaction. The root node of this structure is the document,

which itself is a class featuring multiple methods whose execution is possible with JavaScript, as the DOM itself is written in this specific language. The DOM itself consists of multiple API's such as the DOM API [22]. Furthermore, direct dynamic DOM manipulation is possible with thanks to handles such as `HTMLAnchorElement` with HTML.

3.3.2 Essential Components of The Browser

At present, the four most widely used Internet browsers are Google Chrome, Mozilla Firefox, Apple's Safari and Opera. Microsoft Internet Explorer has been depreciated as of June 15th 2022, meaning an end to future security updates and patches to the browser. In essence, the function of a browser is to acquire a specified resource and present it to the end user. Typically, a browser's main components consist of the following [11] and as depicted in **Figure 2**:

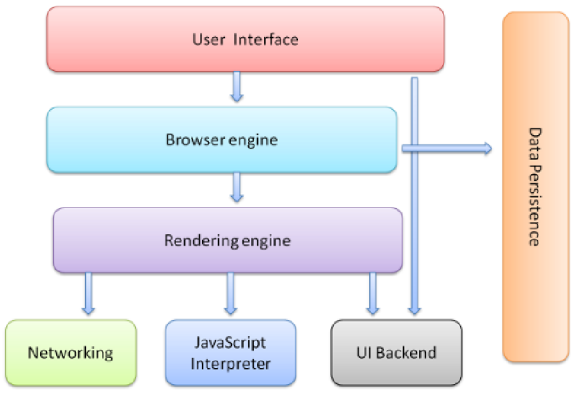


Figure 2. Vital componets of a modern browser. Source: [11].

1. **Its User Interface**: The component which most user interactions occur with.
2. **A Browser Engine**: Engines are also commonly referred to as a layout or rendering engine. The browser engine is the core software component of a browser whose primary purpose is to transform documents and additional resources into a visual representation with which the end user can directly interact. Examples of such engines include Blink used by Opera and Chrome, WebKit by Safari and Gecko by Firefox browsers.
3. **User Interface Backend**: Which is responsible for drawing browser default components such as widgets and buttons. It is not specific to browsers and it instead utilizes the UI methods of the operating system onto which the browser was installed.

This is why, for instance, default button elements without modified styling appear noticeably visually different across different operating systems.

4. **Local Storage**: Which is the persistence layer utilized by browsers as local cache storage. Web pages by default are reset to their initial state as programmed in source code once refreshed, with any changes being reverted as a result – although the use of local storage is a common method of circumventing that. Browsers are allocated a relatively small amount of local memory which is commonly used to store required values. For Google Chrome, approximately 5 megabytes are allocated. Although it worth drawing attention to that this storage is indeed local, and its data contents will be specific to the machine on which the Web document is opened.
5. **Networking**: This component allows for communication between the end user's device and other devices within a local network or beyond. Because document recourses are acquired with requests to a host which in turn responds with the document source files to be rendered across a WAN (Wide Area Network), it is a crucial component in itself.
6. **JavaScript Interpreter**: A browser component responsible for performing the execution of JavaScript code. When the browser receives a response with document source code, a parser converts JavaScript code into machine-readable representative objects. In turn, the parser generates the Abstract Syntax Tree (AST), from which the objects returned are passed into the interpreter and converted by it into machine readable bytecode. Depending on various factors, either the bytecode is used itself or instead is passed onto the compiler which generates more optimized machine code for the CPU.

3.3.3 Functionality of Resource Retrieval and Presentation

Internet browsers employ the above components sequentially prior to ultimately displaying the document and its content to the end user, as shown in **Figure 3**. To elaborate further on the major steps of loading a document, the following is a list of such steps browsers take whenever the end user accesses a document from their browser:

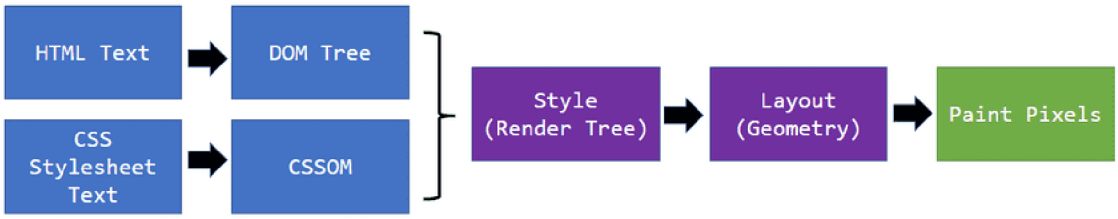


Figure 3. Browser rendering pipeline. Soure: [42].

1. **Resource gathering:** The process by which network requests are made by the browsers across the Internet in order to gather the files required to display a document [42]. This may include HTML files themselves, CSS files, script files, images, URL's, libraries and fonts to name a few – all of which may be scattered across different servers to which requests are made by the browser.
2. **HTML Parsing:** The process by which HTML markup is parsed and is generated what is known as a parse (or DOM) tree, the generic specification with which the manipulation of documents is possible [42]. HTML cannot be parsed using conventional methods, so browsers feature in-built custom parsers for this purpose. Upon completion, the browser marks the document as interactive and scripts are permitted to run, after which the "load" event is fired.
3. **Render Tree Creation:** The browser then creates a render tree based upon the recently created DOM tree, which is responsible for the styling of a given document and allowing for the painting of its contents in order [42]. It is here in which the styles downloaded by the networking component of the browser are used. Unlike the DOM tree, the render tree only features elements which do not contain the `display: none` property, meaning the element can exist within the DOM tree but not the render tree at once. The render tree assigns each element a 'box', or defined area of occupation for later use in the layout and paint stages.
4. **Layout:** A critical milestone to rendering, layout is responsible for calculating the geometry of each node and determining its dimensions. In order to determine the size and location of the current object, the browser traverses the recently created render tree from its root [42]. Almost everything is viewed as a box on a web page, defined either with coordinates (points) or dimensions, using the current viewport as its base.

Layout occurs from the body to its descendants, as well as theirs in turn. There is a difference between *layout* and *reflow*, as the former occurs when node dimensions are re-calculated beyond the initial first layout operation.

5. **Paint:** The final stage for presentation, which involves painting individual nodes onto the document [42]. During this stage, node layout is provided a value in pixels representing the boundary onto which the element is rendered. Paint breaks elements into layers for performance optimization to be handled by the GPU [40], although these layers require being assigned specific CSS properties.

3.3.4 Cross Browser Compatibility

A common issue which often arises during initial developmental phases of a website is cross-browser compatibility [19]. Although it is the end goal for browsers to be developed in close accordance with ever-changing and continuously advancing standards across different web technologies, browsers may take time to adequately adapt. As a result, various CSS declarations may not function as intended when compiled with a browser different to the one on which they were written and initially executed. This does not apply to CSS solely either, as code written in either HTML or JavaScript may suffer a similar fate. This is of particular relevance when taking older browser versions into account, which are by in large installed onto older systems which do not meet the system requirements to update to newer releases. It is generally the end goal of every project to be compatible across all browsers in widespread current use, and if possible, older versions of them as well so as to ensure the same functionality regardless of the machine in use (or at least within reason).

There exist various tools with which to verify the compatibility of specific CSS code. Among them is ‘Can I Use?’ (https://caniuse.com/?search=css) which as input receives the name of an entered property and displays a table of browser versions from which it is officially compatible, as well as additional information. Myriad other websites exist featuring a table of all CSS properties and supported browser versions, including one from W3Schools (https://www.w3schools.com/cssref/css3_browsersupport.asp).

3.4 Performance

With the basic underlying principles of CSS, HTML, JavaScript, the DOM and browser functionality having been established, at this stage it is worth considering the valid attainable approaches by which performance can be maximised to compliment end user experience. In general, it is worth pointing out that CSS performance as a subject seldom concerns the overall size of CSS files due to it being relatively negligible as they are, in essence, files of executable text. Performance instead focuses on key areas involving how the code itself is interpreted and rendered onto a web page which attributes towards most cases of its own reduction.

3.4.1 Content Visibility

Due to layout and painting accounting for a significant portion of a possible performance impact, this causes preventable strain on resources dedicated for rendering elements which are not yet within the field of view of the end user's current resolution. This impact may be partially mitigated with the use of the content visibility property. By default, the browser engine calculates page layout and renders components when first loaded, although this process may be restricted until these steps are necessary, relieving the browser engine of a what may be a large load. The content visibility property may accept one of three values –

1. **visible**: The default value at which content visibility is set. Assigning this property value manually is not necessary, as by default, all content is displayed.
2. **hidden**: Similar to the `display: none` property (which the browser engine ignores at the layout and painting stages), the element's contents are skipped. For this to function properly the content cannot be accessed by user-agent features.
3. **auto**: Layout, styling and paint are contained. If the element in question is not visible to the user, its rendering is skipped. However, unlike with hidden content visibility, the content skipped must still remain available for user-agent features.

Content visibility can be of extreme use in combination with the CSS `contain` property as a means of reducing overall rendering cost, especially when it comes to particularly long documents [37].

3.4.2 CSS Contain Property

In CSS, the `contain` property provides the designer manual control over the relationships between elements and their extent of independence from others in the document tree. This property is essential to the browser for calculations related to layout, painting, styling and/or sizing within a specific region of the DOM [36]. Rules defined for a contained area will not apply beyond its own confines and the remainder of the page will remain unaffected, which provides a significant performance improvement with the help of reducing the number of component re-renders. The contain property may prove to be particularly useful within a document which features multiple independent widgets and may prevent them from causing an external side effect beyond their assigned bounding box. The property itself can be assigned with one or multiple values. If handed a multiple of values, new context for stacking and block formatting, as well as a new containing block will be created. The acceptable values may include:

1. `none`: The render block assumes normal behaviour and containment is not applied.
2. `strict`: States that all rules for containment apply to the element in question, with the exception of the size value.
3. `inline-size`: Containment is applied to the element, although it is inline. This property ignores child elements and cannot be combined with the size value.
4. `layout`: The element in question features a layout isolated from the rest of the document, meaning that beyond the element's scope, no other element may affect its layout and vice-versa.
5. `style`: CSS property definitions which may possibly apply to multiple elements and/or their descendants will not apply to an element beyond that, whose style containment is restricted when defined within their restricted scope.
6. `paint`: Whereby an element whose contain property is set to this value will not allow its children to be rendered and displayed beyond its set bounds. In the event that the element itself is not within the end user's field of view, the element is not rendered at all. However, for this to function as intended, these elements are required to be contained.

3.4.3 Render Blocking Optimization

CSS features a technique known as render blocking which prevents unnecessary paint operations until stylesheets are completely parsed and the CSS-DOM is completely constructed. Otherwise, elements would be rendered initially only to be rendered once more when their styles have been calculated, resulting in a negative user experience and subsequent drop in performance [40]. It must be noted that the browser still downloads all assets initially regardless of whether or not render blocking applies. A good means to improve performance with render blocking and the CSS -DOM's construction it is advised to remove and compress unnecessary styles, as well as isolate them into a separate file to relieve the browser of calculating excess render blocking. This also results in a reduction of the CSS which is initially required for the browser to parse and construct a CSS DOM. However, the prevention of render blocking may yield faster rendering performance if applied to styles that will not be rendered by the browser. As mentioned previously, media queries in CSS are a crucial part of responsive web design, allowing web pages to respond dynamically to the end user's current resolution, although they also offer additional benefit if employed to prevent render blocking by means of using media queries in combination with the addition of a `"print"` media attribute value in the HTML markup. When the browser sees that a stylesheet which features a media query, it understands that the stylesheet would only apply in a specific scenario, and thus, the time taken for render blocking is reduced [40].

3.4.4 Animation on the GPU

Modern browsers have evolved to effectively handle CSS animation(s) and their properties which would not result in a reflow. As mentioned previously, reflow is a process handled by the browser which involves the reordering a document's layout either at initial stages of loading or in response to changes to an element's styles. The final stage of rendering by browsers, painting, also occurs immediately. As a means to increase performance, the node in question to be animated may be moved to the main thread of the GPU, which should be significantly more capable of rendering an animation, thus, increasing animation performance [40]. This process is known as compositing, which can be achieved by the means of including 3D transformations and animation transformation, among other properties which include:

1. `transform: translateZ()` (to trigger a 3D plane)
2. `transform: rotate3d()`
3. `position: fixed;`
4. `will-change: auto`
5. `filter`

Other elements by default feature their own layer, including `<video>`, `<canvas>` and `<iframe>`. When elements are composited (promoted), their calculations are performed by the GPU, which also noticeably improves performance in mobile devices which may feature less powerful hardware resources.

3.4.5 Browser Preparations for Upcoming Changes

Animations in CSS can be additionally optimized if the `will-change` property is applied, which notifies the browser of upcoming changes to which an element may be subject, allowing the browser to make preparations beforehand in advance which allows for better optimized code. It must be taken into account that the excess use of this property may also lead to a decline in performance and should thus be prevented [41]. For this reason there is no 'all' option for `will-change` available. An additional measure to circumvent this issue is to toggle an element's will-change value when the anticipated process has already occurred, although when alternative solutions are available it is advised to first resort to them, in the first place. This is because otherwise, if not toggled, the browser will continuously anticipate changes which may not occur, consuming resources. This property can accept one of the following values:

1. `auto`: Where browser optimization remains as standard.
2. `scroll-position`: Which indicates than an expected change in an element's scroll position is due to occur as a result of an animation which may potentially cause a recalculation of its layout and a repaint. This prepares the browser for content which is not yet visible within the element's scroll window.
3. `contents:` Which prevents the caching of an element's content as the element is expected to change, rendering caching as an unnecessary operational expense.

It is also advised to provide the browser sufficient time beforehand to make these preparations and make their necessary calculations [41].

3.4.6 CSS Font Display Property

Web fonts are yet another resource which requires being downloaded for further use within a document. Due to the nature of downloading resources over a network, this may produce an undesirable, visually perceivable outcome if the font face in use has not already been downloaded to the client's machine. For this reason, it is important to control the behaviour of fonts and how they will render depending on the time required for them to be downloaded and rendered. Most web browsers, including Google Chrome, Opera, Mozilla Firefox and Microsoft Edge, with the exception of Apple Safari, implement a timeout to mitigate the risk of exceedingly long font download speeds [44], with thanks to which a fallback font will be resorted to for the rendering of text content. However, different browsers practice this feature differently, and the resulting difference in fonts causes an adverse effect upon layout and font sizing [44]. The CSS `font-display` property and `format()` value are a means to circumvent this issue and of providing deeper control over font behaviour at load time indiscriminate of the client browser in use.

3.4.7 Summary

A considerable number of factors and best practices for the reduction of an undesirable impact on page performance have been successfully identified, which can be summarized as:

- Content containment for the prevention of layout recalculation.
- Content visibility alterations for a reduction of rendered elements beyond what is viewed by the end user.
- Animations can be assigned to the GPU from the CPU in order to relieve if of the strain taken from graphical calculations.
- Browsers can be optimized in advance for changes in layout and paint operations.
- Fallback fonts can be assigned for the improvement of end user experience for when typographic content has yet to be rendered.

Some of these properties and approaches were selected for further use within the practical part of this thesis.

3.5 Related Research

A rather interesting and unique approach encountered was published by H. Natarajan and R. Rashmi, titled "Improving a website's First Meaningful Paint (FMP) by optimizing render blocking resources – An experimental case study" [55]. In this study, the researchers similarly also emphasised the importance of a web page's load performance for first-time users. Natarajan also used Google PageSpeed Insights for the analysis of load performance, and was concerned with First Contentful Paint as the metric of choice for demonstrating changes. The researcher conducted tests on an existing application of the company Axis Communications AB, and encountered that it took 2.95 seconds for it to fully load, which expectedly raised concern. A large portion of this load time was attributed to rendering and scripting, the combined duration of which produced over half of the load time recorded in total. In using tools such as Dust-Me Selectors, JCLAIM and Telerik Fiddler 4б [55], it was discovered that eighty seven percent of the site's stylesheets were unused by the landing page, making the removal of which a sufficient means of optimization and reducing page FMP. Concerned with issues related to application security, the researcher was tasked with producing a solution without reliance on the product(s) of a third party.

One of the objectives of the thesis' practical solution was to implement features similar to that of the tools listed, however which instead saves logs of the unused CSS rules locally for secure use. The application developed could identify and separately store unused CSS rules in their respective files, including that of media queries containing unused code at some resolutions, and used Fiddler to obtain these results as a proxy. To test this feature, a link tag with the media attribute of none was used to emulate disregarded CSS. Tests were also conducted to determine any difference in FMP before and after the non-essential CSS was render blocked. The resulting data suggested a reduction in FMP to 447.1ms from 1351ms [55], as well as a considerable reduction from 358.3ms to 160.61 [55]. It was also determined that over ninety nine percent of the unused CSS file generated was confirmed to have not been in use for the home page of the application, as initially anticipated.

This work provided a considerable solution for the mitigation of negatively impacted end user experience from poor document load performance. The authors created and tested a solution which delivered on demonstrating a notable reduction in FMP resulting from the isolation of CSS files and the removal of stylesheet rules which were not used for a specific web page. It additionally demonstrated means of optimizing JavaScript code, which also

provides a meaningful contribution, considering the frequent combination of CSS with JavaScript in web development.

Another study titled “A Complexity Metrics Suite for Cascading Style Sheets” raises concerns for overall code maintainability and its long-term effects upon future efforts and implementation(s). The researchers explored means by which the complexity of applications, including their stylesheets, could be quantified for analysis. Within a previous paper the researchers also suggested the following metrics [57]:

1. Number of Rule Blocks (NORB)
2. Entropy Metric (E)
3. Number of Extended Rule Blocks (NERB)
4. Number of Attributes Defined per Rule Block (NADRB)
5. Number of Cohesive Rule Blocks (NCRB)

– and the purpose of this study was ultimately to validate these aforementioned metrics. It was suggested that these metrics may be applied for the purpose of evaluating applicational complexity for increased improvement to overall future scalability and modularity.

For the theoretical validation of these proposed metrics, they were evaluated against Weyuker’s properties. Further proving the extent at which these metrics may be of use was done via empirical validation with quantile-quantile plot graphs, violin plots, Chi-square tests and Lilliefors corrected Kolmogorov-Smirnov tests. In conclusion of the study, fifty six percent [57] of metrics sufficiently satisfied the Weyuker properties, and the NADRB metric satisfied sixty seven percent [57], which suggested that these properties were confirmed viable metrics. Results from the empirical and statistical studies suggested positive characteristics of statistical distribution. The correlation between the overall size of a document’s given stylesheets and metric values proved to be insignificant, thus suggesting that these metrics measure the complexity, rather than size of a given CSS file.

This study differs in that its aim primarily was concerned with introducing, as well as proving additional metrics for the measurement of the physical complexity of a given CSS file. These metrics may be further employed in order to correct code considered excessively complex. However, this study does not focus on specific CSS properties and how they may contribute towards deterioration in performance. It instead proposes means by which CSS code can be analysed for preventable complexity for ease of use.

The study titled “Defect prediction for Cascading Style Sheets” shed light onto the difficulty in exhaustively testing software due to implications imposed by its allocated budget and time. The researchers suggested that defect prediction techniques would prove to aid with this issue they would allow software testers with better identifying which modules would require more focus. The paper’s aim was to improve the overall performance of software defect prediction with CSS code with use of machine learning algorithms.

This study also proposed a set of metrics [58] for the prediction of proneness of defects of CSS code, this time which concerned factors such as the number of classes selectors have and the number of qualified selectors. Data sets were generated from multiple web applications, and defect prediction was performed with use of machine learning algorithms. It was revealed that defect prediction via the use of metrics may lead to an overall improvement in the quality of CSS code [58], which would thus allow for a reduction in costs for testing. It is unclear how performance and load time could be affected with this proposal.

Although this study involves the improvement of CSS code and its tests involved were conducted on the stylesheets of multiple open-source projects, its primary purpose is not related directly to end-user experienced and perceived performance – but instead how application quality analysis and its entailed costs may be reduced, by virtue of algorithmically predicting which modules of an application are more likely prone to defects. However, it is worth acknowledging the importance of QA testing for the identification of potential issues which could affect performance.

4 Practical Part

Over the course of this section of this thesis, a detailed description of the tools, methodology, selected measurement metrics and experiment conditions is first stated, which is subsequently followed by a description of the tests conducted, themselves, with corresponding code samples.

4.1 Selected CSS Properties for Testing

Below is a list of the CSS properties and approaches selected for testing, as well as a statement of the reason(s) for this choice:

1. `transform: translate3d()`: This property was selected for the analysis of any difference in performance between animation on the GPU and the CPU. This property was selected for the identification of any performance improvement on mobile devices with restricted hardware or any reduction in performance from throttling. Some animations may be taxing on the CPU, the limited cores of which could instead be freed for different processes as mentioned in Section **3.4.4**.
2. `will-change`: For underpinning any performance improvement from the pre-emptive notification of changes to layout and opacity to the browser. Selected as abrupt changes in layout and repaint operations may visibly affect user experience as mentioned in Section **3.4.5**.
3. **Stylesheet separation**: To identify if page load speeds may be reduced by virtue of preventing unused CSS from being loaded. This was attempted as the likelihood of an end user exiting a web page is considerably high if it takes over three seconds to perform its First Contentful Paint, which in theory, could be partially mitigated with use of this approach, as mentioned in Section **3.4.3**.
4. **Font Face Downloads**: To identify differences in page load speeds from larger font downloads made using the `@import` statement. This was selected for the relationship between external downloads and the document Total Blocking Time and because larger network payloads directly prevent user interaction with the page as underlined in Section **3.1.12**.

4.2 Tools

For conducting the necessary tests, the following tools will be employed for the creation of the web application and for the measurement of performance differences derivative of changes to the document stylesheet(s), as well as for the application's deployment.

4.2.1 Visual Studio Code

VS (Visual Studio) Code is an integrated development environment application and among the most widely used by developers and within the software development industry as a whole. It is proprietary software released and owned by Microsoft with support of an extensive variety of programming, scripting, and styling languages, some of which were used for the production of the application to be subjected to the testing phases of this thesis. It is additionally widely expandable with the use of extensions available via a central repository which may be installed from the application itself [45]. Additionally, its terminal shell feature will be used to run node commands which this test application for this thesis requires. VS Code itself was used for the purpose of producing the test application in use, as well as making changes to its source code to cause possible differences in load performance.

4.2.2 Live Sass Compiler

The Sass Live Compiler is a VS Code extension capable interpreting Sass code and transpiling it into valid CSS in real-time [49]. This tool provides particular use in that changes to CSS made take immediate effect and may be visibly observed once the edited file is saved. This process requires a separate file of the .scss format from which another .css file is generated automatically with the corresponding code. It was used for altering the test application's various stylesheets.

4.2.3 React

React is an increasingly popular open-source functional library built in JavaScript which was released by the then Facebook (Meta) corporation and is used widely for the construction of user interfaces [47]. With functional programming in mind, it is declarative, and much like languages such as SQL it abstracts the complexity of functions, which thus

encourages ease in development, as well as encourages the composition of applications into smaller components which are easier to maintain and understand as a result. It additionally supports high-order components which may accept or return another component, allowing for highly robust applications. React components consist of JSX (JavaScript XML, or TSX with TypeScript) [54] which is an extension of JavaScript returning a component comprised of its markup, state, and functionality in one package. State can be passed throughout these components or accessed universally via context. React utilises JSX for the generation of the React DOM, a virtual DOM tree stored in-memory used for the necessary generation of the DOM itself, as the browser can only parse HTML, CSS, and JavaScript. Considering that React is merely concerned with user interfaces and state management, it commonly would not suffice for the making of a complete application of high complexity, which requires additional libraries for functionality such as routing. The test application itself was built with React, using TSX components.

4.2.4 PageSpeed Insights

PageSpeed Insights is a family of tools created by Google for the identification of faults in the compliance of applications with web performance best practices published by Google themselves. It offers detailed analysis of the page performance based upon numerous metrics, and on the basis of the results generated may even provide suggestions on how a page may be improved. User experience can be reported on for both mobile and desktop viewports. It was used for the generation and analysis of performance data resulting from tests through which the test application was ran.

4.2.5 Node Package Manager

Node Package Manager (“npm”) among the most widely used and largest software registry, containing over eight hundred thousand code packages [46]. It is used for the Node JavaScript platform and used primarily for the publishing, installation and development of node-based programs, be they are open source or commercial and/or proprietary. These packages are defined within a mandatory package.json file containing data in JSON format, requiring at least two fields containing a name and version for intended functionality.

4.2.6 Netlify Edge

It was decided to conduct the tests on an application deployed to the cloud in order to better emulate the real-world conditions of an application accessible over the Web. It should therefore be brought to attention that some of the resulting data may (or may not) have been under the influence of factors beyond the scope of the test application itself. Considering that the test application mostly comprises of relatively simple UI components, it would more than suffice to deploy using Netlify Edge, which allows for fast and surprisingly simple deployment within seconds while making use of a free trial account. It is fast due to automated pre-rendering and global distribution, and supports both client and server side hosting. For the purpose of this thesis, a conventional client-side application will be deployed.

4.3 Environment and Conditions

This section covers the conditions in which the necessary tests were conducted, as well as the structure of said tests and selected metrics for quantifying performance differences. This is the for the purpose of allowing for the replication of results and overall transparency.

4.3.1 Experimental Application

The goal of this front-end application titled ‘eclinic’ as a part of the practical section of this thesis was to serve as a sandbox app to emulate CSS performance effects as they would occur in practice. It was created partially for the purpose of this thesis and to demonstrate how these properties may influence overall user experience. It does not include many complex features, and most of its JavaScript was implemented for animations and base functionality, and instead its primary focus on its core UI components (as demonstrated in **Figure 4**) and pages implemented with the React router.

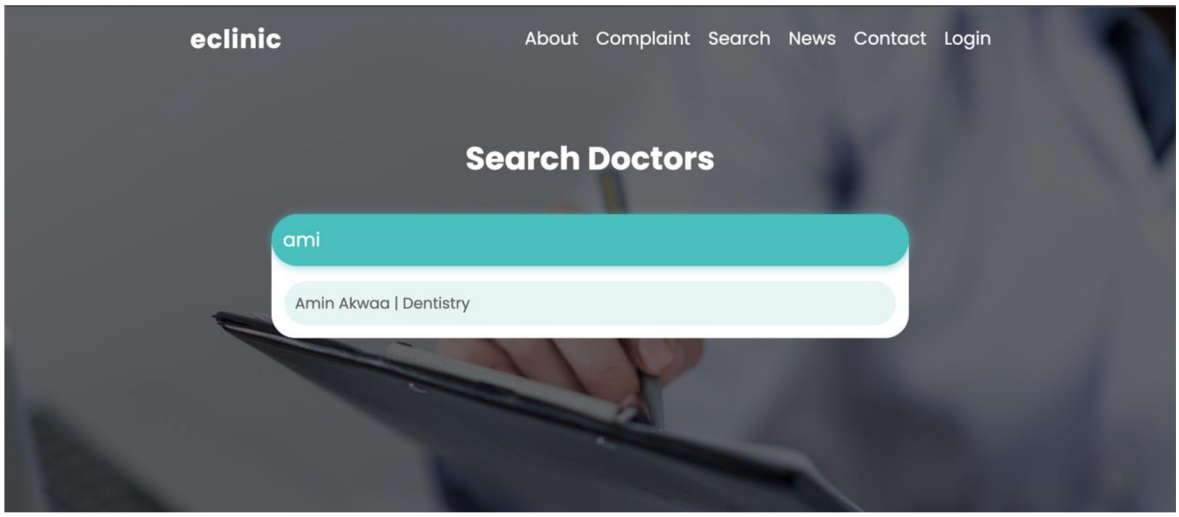


Figure 4. Search page of eclinic test application.

4.3.2 Environment

The following is a table detailing the present versions of the tools due to be employed throughout unit testing phases:

Table 1. List of tools and respective versions.

TOOL	VERSION
Node	16.14.2
React	18.2.0
Google Lighthouse (for PageSpeed Insights)	9.6.8
Live Sass Compiler	5.5.1

4.3.3 Separated Test Applications

Two versions of the same applications were employed; one built with React and another built without Node. To further elaborate on these differences, the version without the Node Package Manager consists of baseline HTML, CSS and JavaScript files linked together via the `<head>` tag of each page's respective HTML file. Applications built with Node, on the other hand, have a large directory containing the modules required for a (for example) React application to build and run. The version without Node was built and deployed in order to test differences in performance resulting from CSS render blocking with

stylesheet separation for different resolutions, as document `<head>` tags cannot be directly accessed with projects created using `npx create-react-app`.

Table 2. Corresponding application weight.

APPLICATION	SIZE
Including Node	29.8 MB (build)
Without Node	36.7 MB

4.3.4 Experiment Structure

The experiments conducted throughout the practical section of this thesis comprised of five total iterations which were made with the following steps:

1. Changes were made to the source code of the experimental application to satisfy the test requirements for the selected property in question.
2. The newly edited source code was deployed using Netlify.
3. The deployed application was analysed with Google PageSpeed Insights for the purpose of collecting performance data.
4. The data was recorded with which an average value was calculated.

Some experiments were made with mobile viewpoints, whereas others were with desktop viewports. Slow 4G throttling was present throughout. For the remainder of this chapter, each individual test is explained in further detail. The resulting data retrieved and its analysis are stated within the following chapter.

4.3.5 List of Chosen Metrics

Below is a list of selected metrics for measurement and their corresponding descriptions [51]:

1. **Speed Index:** A load page load performance metric which measures the speed at which the contents of a page are visibly populated, measured in seconds (s).
2. **Total Blocking Time:** A sum of adding the blocking portion of each expensive task occurring between the First Contentful (FCP) Paint and Time to Interactive (TTI), measured in milliseconds (ms).

3. **First Contentful Paint:** A measurement of when the first content (text or images) is rendered/painted to the end user.

A lower value of all aforementioned metrics is indicative of an improvement to performance.

4.4 Testing Phase

The following is a detailed overview of the individual tests which were made to achieve the resulting data revealed within the next chapter of this thesis. It is divided into sections dedicated to each tested property and its corresponding source code.

4.4.1 GPU Handling of CSS Animations

A CSS login form illustrated in **Figure 5** was animated to have its position and opacity altered over a period of one second at load time. This has been done for the purpose of achieving a smooth effect of appearing as it moves up the y-axis. The former and latter CSS properties for this experiment are defined within **Source code 15**.

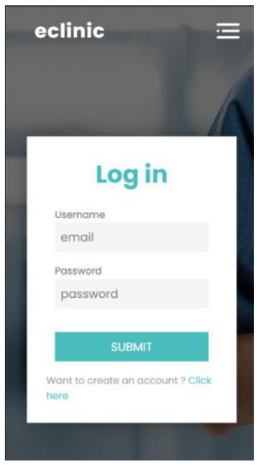


Figure 5. The eclinic Login Page.

Source Code 15. Initial and altered CSS properties.

Before	After

`@keyframes appearUp {`
`  from {`
`  transform: translateY(5em);`
`  }`
 	
`  to {`
`  transform: translateY(0);`
`  }`
`}`
	`@keyframes appearUp {`
	`  from {`
	`  transform: translate3d(0, 5em, 0);`
	`  }`
	
	`  to {`
	`  transform: translate3d(0, 0, 0);`
	`  }`
	`}`

Considering that in theory, allowing CSS animations to be handled by the GPU directly relieves stress from other resources at hand, the `transform: translate3d()` was be applied in order to trigger a three-dimensional plane, which automatically assigns the given animation to be assigned from the CPU to the GPU. Measurements were made using Speed Index on the Login page.

4.4.2 Beforehand Preparation of the Browser

The browser can be optimized to prepare for the same element's upcoming animations beforehand, with use of the `will-change` property to its opacity and transform. It should be noted, however, that it is advised to only make use of this property as a last resort effort to resolve an ongoing issue related to application performance. This property has been combined with the previously added 3D transformations in order to better demonstrate their combined effectiveness, as is demonstrated in **Source code 16**. Note the addition of the `will-change` property to the form and its encapsulation wrapper div element, which were not present in prior recorded iterations. For the series of tests on this combination, the document's Speed Index was measured and evaluated on the Login page.

Source Code 16. The will-change property in practice.

Wrapper div element	Semantic form element

`.wrapper {`
`  background-color: rgba(0, 0, 0, 0.582);`
`  padding: 1em;`
`  color: white;`
`  height: 100%;`
`  width: 100%;`
`  display: flex;`
`  flex-direction: column;`
`  justify-content: center;`
`  align-items: center;`
`  animation: fadeInto 1s;`
`  will-change: opacity;`
`}`
	`form {`
	`  animation: appearUp 1s;`
	`  will-change: opacity, transform;`
	`  display: flex;`
	`  flex-direction: column;`
	`  background-color: #fff;`
	`  width: 90%;`
	`  max-width: 30em;`
	`  padding: 2em;`
	`  box-shadow: 0px 3px 38px -10px rgba(0, 0, 0, 0.55);`
	`}`

4.4.3 Render Blocking With Isolated Stylesheets

Another goal of the application was to achieve mobile-first design, which was implemented via the use of the CSS flexbox, grid and media queries.

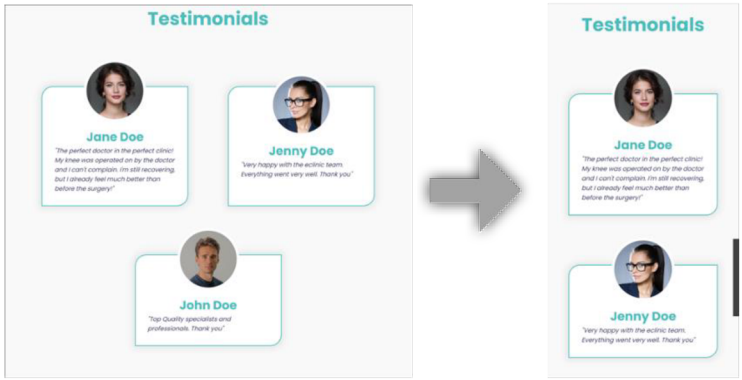


Figure 6. Implemented flexbox for adaptive design.

Source Code 17. Styling for demonstrated card elements.

```
.card {
    // additional code..
  display: flex;
  justify-self: center;
  max-width: 20em;
  justify-content: center;
  align-items: center;
  flex-direction: column;
  border: 2px solid var(--primary);
}
```

These implementations, however, of a responsive web design may potentially come at a cost of web performance. This could be a result of render blocking caused by the loading of CSS files by the browser. These files may include stylesheets with code currently unnecessary for the rendering of the user's document at their given resolution, and as a means of circumventing this, it is recommended to separate code (such as media queries) into their respective stylesheets for use if, and only if, certain criteria are met – such as the viewport as is demonstrated in **Source Code 18**. The corresponding experiments were made on the version of the experimental application without Node for this purpose. Measurements were made of First Contentful Paint (FCP) and on the landing page of the application.

Additionally, the viewport selected was that of a desktop, in order to isolate stylesheets not reserved for smaller resolutions.

Source Code 18. Separated stylesheets.

```
<link rel="stylesheet" href="./style.css">
<link rel="stylesheet" href="./mobile.css" media="screen and (max-width:
900px)">
<link rel="stylesheet" href="./non-mobile.css" media="screen and (min-width:
600px)">
```

4.4.4 External Font Downloads

Although typography is an essential element to UX design, theme and identity, it should also be considered that web fonts may pose additional strain on the overall performance of a web application – most particularly if downloaded online from external source. A series of tests were conducted with the restricted downloads of the popular font face Poppins, as well as the excessive downloads of it including font weights and styles not used throughout the application, as demonstrated in **Source Code 19**. A fallback font was intentionally not included in order to isolate and amplify the resulting data. Measurements made in Total Blocking Time (TBT) and on the landing page.

Source Code 19. Initial and latter font implementation.

Before	After
`@import url("https://fonts.googleapis.com/css2?family=Barlow:wght@200;400;700&family=Poppins:wght@300;400;700&display=swap");`	`@import url('https://fonts.googleapis.com/css2?family=Poppins:ital,wght@0,100;0,200;0,300;0,400;0,500;0,600;0,700;0,800;0,900;1,100;1,200;1,300;1,400;1,500;1,600;1,700;1,800;1,900&display=swap');`

5 Results and Discussion

5.1 Data Analysis

The following is a collection of the retrieved results, as well as some proposals for why they may have occurred. All results presented are legitimate and their accuracy was not altered to convey a specific narrative.

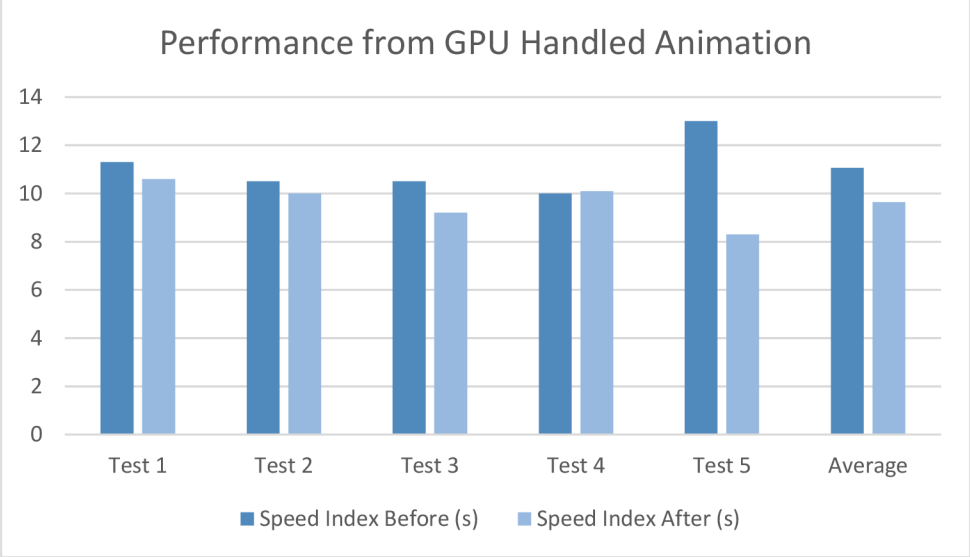
5.1.1 Animation Handled by the GPU

In analysing the data collected, it can be observed that a notable difference in performance can be achieved at a 14.7% improvement, or a 1.42 second reduction of the document's overall speed index as shown in **Table of Results 1**. This suggests that allowing a system's GPU to handle animations results in a slight visually noticeable increase in performance to the end user. This could be attributed to that the GPU is specifically designed with graphics-oriented calculations in mind, as well as the fact of other hardware resourced being freed for the handling of other tasks. Although, given that the tests were conducted with a mobile device, the resulting data may have been under the influence of mobile throttling, as mobile devices are generally equipped with less powerful hardware resources.

Table 3. Table of Results 1.

	Speed Index Before (s)	**Speed Index After (s)**
Test 1	11.3	10.6
Test 2	10.5	10
Test 3	10.5	9.2
Test 4	10.0	10.1
Test 5	13.0	8.3
Average	11.06	9.64

**Speed index of each experiment before and after 3d animations were applied. Measurements made in seconds.*



Graph 1. Effects of 3D Animation Transformations. Speed Index in Seconds.

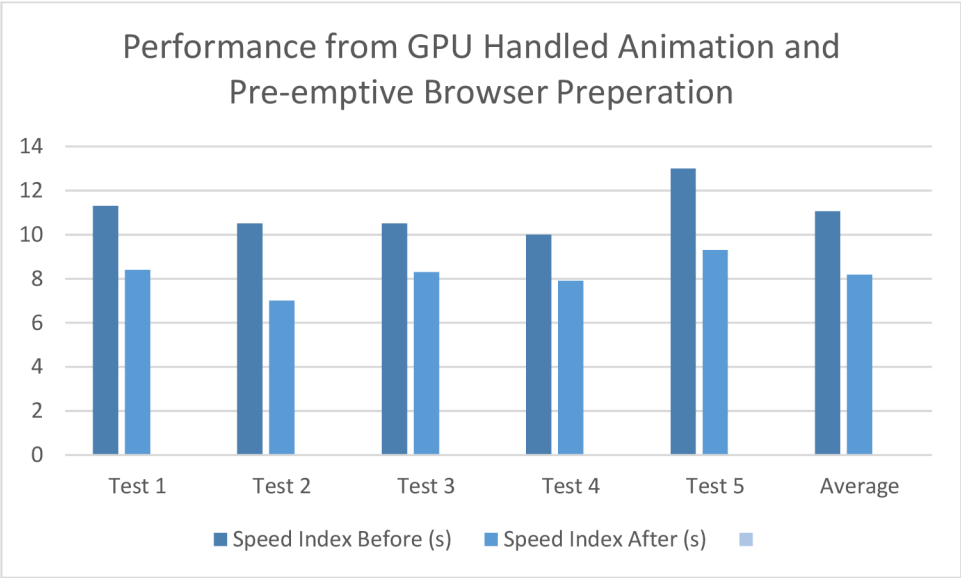
5.1.2 Combined GPU Animation with Pre-Emptive Browser Preparations

Given the proven effectiveness of assigning the responsibility of animation to a dedicated graphics processing unit, it was decided to further test the limits of performance optimisation via the introduction of pre-emptive preparation for the client browser. According to the resulting data displayed in **Table of Results 2**, a performance index reduction of 2.88 seconds, or a performance increase of 35.2% was achieved in combining both properties. These results can be attributed to how the `will-change` property notifies the browser of an imminent alteration to a given element's properties, thus, allowing it to perform optimizations to increase the page responsiveness by virtue of conducting possibly expensive operations beforehand.

Table 4. Table of Results 2.

	Speed Index Before (s)	**Speed Index After (s)**
Test 1	11.3	8.4
Test 2	10.5	7.0
Test 3	10.5	8.3
Test 4	10.0	7.9
Test 5	13.0	9.3
Average	11.06	8.18

Speed index of each experiment before and after the changes stated in **Source Code 15. Measurements made in seconds. Note that this experiment involves some of the code from the previous example.*



Graph 2. Effects of 3D Animation Transformations and will-change property. Speed Index in Seconds.

5.1.3 Render Blocking with Isolated Stylesheets

The end results of the conducted experiments concerning the First Contentful Paint before and after stylesheet separation and the intentional render blocking of some properties prove to be negligible and inconclusive as indicated in **Table of Results 3**. However, this may be a result of intentional testing on desktop devices, as the majority of media queries present were defined for mobile resolutions. It is worth pointing out that desktop devices tend to be better equipped in terms of hardware resources in comparison to their mobile counterparts. This was done for the purpose of potentially demonstrating a difference in performance with temporarily reduced stylesheets. However, results may differ on larger scales, as the project employed is of an insignificant size too inadequate to confirm a reliable result at least for this metric.

Table 5. Table of Results 3.

	First Contentful Paint (s)	First Contentful Paint (s)
Test 1	0.7	0.7
Test 2	0.7	0.7
Test 3	0.7	0.7
Test 4	0.7	0.7
Test 5	0.7	0.7
Average	0.7	0.7

**Differences in first contentful paint with and without render blocking. Results measured using an application without Node and on a desktop device.*

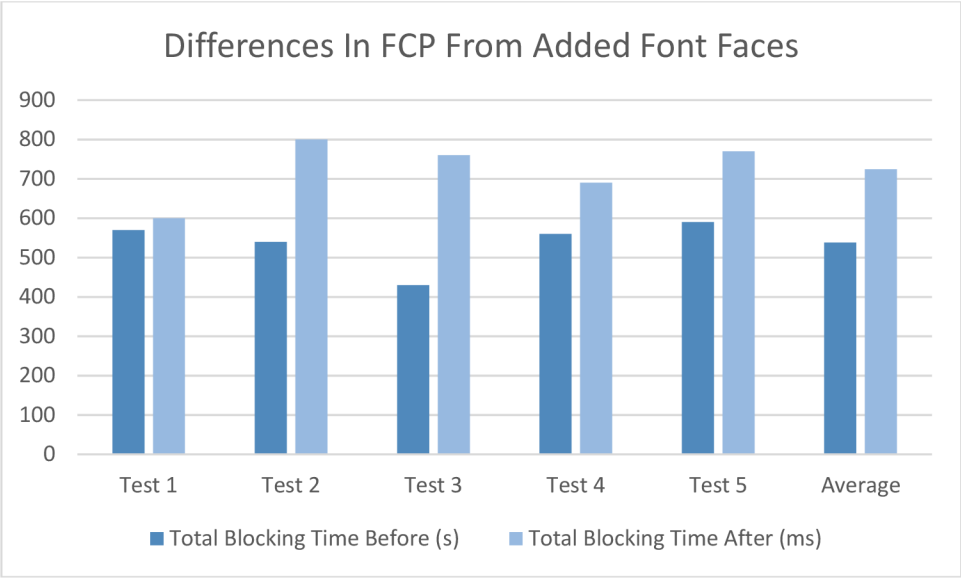
5.1.4 Loading of Excess Font Faces

In analysing Total Blocking Time prior to and after the additional inclusion of a large number of font faces, a notable difference was be observed. As shown in **Table of Results 4**, an average difference of 186ms had been uncovered, accounting for an overall 34.6% additional count to blocking time, which consequentially can likely be noticeable to the end user. These results may suggest the importance of the selective downloads of font faces whose are source is external, or at least do so to an extent.

Table 6. Table of Results 4.

	Total Blocking Time (ms)	Total Blocking Time (ms)
Test 1	570	600
Test 2	530	800
Test 3	430	760
Test 4	560	690
Test 5	590	770
Average	538	724

**Differences in Total Blocking Time before and after the excessive use of unnecessary font faces, in microseconds.*



Graph 3. TBT resulting from added font faces. Measured in microseconds.

5.1.5 Future Recommendations

Further recommendations for building upon this existing research may include:

- Testing for and evaluating a performance increase/decrease of a selected metric resulting from the use of the CSS `contain` property in order to achieve an enhanced understanding of how it affects document layout.
- Exploring other means of reducing document network payloads, such as with the compression of graphical content including images and video, as downloading document resources account for a considerable portion of initial page load time.
- Testing the influence of fallback fonts upon end user experience with the measurement of FCP before and after their implementation, so as to confirm or deny if this could be a means of preventing the end user from exiting a web page.

6 Conclusion

In conclusion, the theoretical part covered throughout the literature review of this thesis served as a substantial foundation to understanding the criteria and conditions necessary for the practical section. The theoretical section provided a considerable understanding of fundamental web and related technologies, including CSS itself, HTML and JavaScript, as well as of the functionality of conventional browsers and their primary components and which best practices for performance are recommended to developers.

The first partial objective was addressed via the selection of multiple CSS properties and practices, which were subject to further analysis throughout the practical section. For addressing the second partial objective, as an experimental application with technologies noted within the literature review was created to serve as the environment for practical testing and data collection. The third partial objective, which concerned designing and performing experimental measurements on the aforementioned test application selected properties, was addressed with tests successfully conducted under the conditions specified on the page's overall differences in performance in response to changes to its respective stylesheet(s).

Concerning the main objective of this thesis, some limits to selected CSS properties were identified through the data resulting from the tests performed. A slight improvement in performance could be observed through assigning animations to the GPU, which could be further improved with use of pre-emptive browser preparation. Render blocking via the separation of stylesheets with media queries proved to produce insufficiently significant results to suggest a change in document load performance. A notable difference in total blocking time was observed from subjecting the document to a large network payload with the excessive downloading of external font faces.

7 References

1. Usage Statistics of CSS for websites. W3Techs [online]. [Accessed 13 July 2022]. Available from: https://w3techs.com/technologies/details/ce-css
2. The Global Digital Divide. Khan Academy [online]. Accessed 13 July 2022]. Available from: https://www.khanacademy.org/computing/computers-and-internet/xcae6f4a7ff015e7d:the-internet/xcae6f4a7ff015e7d:the-digital-divide/a/the-global-digital-divide
3. Metadata in HTML. MDN Web Docs [online]. [Accessed 14 July 2022]. Available from: https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/The_head_metadata_in_HTML
4. CSS Values and Units. MDN Web Docs [online]. [Accessed 15 July 2022]. Available from: https://developer.mozilla.org/en-US/docs/Learn/CSS/Building_blocks/Values_and_units
5. A Brief History of JavaScript. Learner Academy [online]. [Accessed 15 July 2022] Available from: https://www.learnacademy.org/blog/who-introduced-javascript-originally-called-why-created/#:~:text=Brendan%20Eich%20created%20JavaScript%20in,legendary%20Netscape%20Navigator%20web%20browser.
6. SASS [online]. [Accessed 21 July 2022]. Available from: https://sass-lang.com/
7. Introduction to CSS. TechnologyUK [online]. [Accessed 21 July 2022]. Available from: https://www.technologyuk.net/computing/website-development/introduction-to-css/introduction.shtml
8. CSS Browser Support Reference. W3Techs [online]. [Accessed 21 July 2022]. Available from: https://www.w3schools.com/cssref/css3_browsersupport.asp
9. A Brief History of HTML. University of Washington. [Accessed 21 July 2022]. Available from: https://www.washington.edu/accesscomputing/webd2/student/unit1/module3/html_history.html#:~:text=The%20first%20version%20of%20HTML,HTML%20as%20an%20XML%20language
10. HTML Basics. MDN Web Docs [online]. [Accessed 14 July 2022]. Available from: https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/HTML_basics
11. How Browsers Work. web.Dev (Google Developers) [online]. [Accessed 22 July 2022]. Available from: https://web.dev/howbrowserswork/
12. The Decade-Long Path of Web Fonts. The History of The Web [online]. [Accessed 19 September 2022]. Available from: https://thehistoryoftheweb.com/web-fonts/#:~:text=The%20First%20Font%20Rules&text=So%20even%20the%20first%20version,%2Dvariant%20and%20font%2Dweight%20
    The Performance Cost of Custom Web Fonts [online]. [Accessed 20 September 2022]. Available from: https://www.wholegraindigital.com/blog/performant-web-fonts/
13. CSS Fonts. W3Schools [online]. [Accessed 22 September 2022]. Available from: https://www.w3schools.com/css/css_font.asp
14. CSS Selectors. W3Schools [online]. [Accessed 22 September 2022]. Available from: https://www.w3schools.com/css/css_selectors.asp
15. Z-Index. W3Schools [online]. [Accessed 23 September 2022] Available from: https://www.w3schools.com/css/css_z-index.asp

16. Sass Introduction. W3Schools [online]. [Accessed 23 September 2022]. Available from: https://www.w3schools.com/sass/sass_intro.php
17. Working with JSON. MDN Web Docs [online]. [Accessed 27 September 2022]. Available from: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON
Schema.org Markup. MDN Web Docs [online]. [Accessed 27 September 2022]. Available from: https://moz.com/learn/seo/schema-structured-data#:~:text=Schema.org%20(often%20called%20schema,represent%20your%20page%20in%20SERPs.
18. How to Assure a Well-Formed Website. InformIT [online]. [Accessed 27 September 2022]. Available from: https://www.informit.com/articles/article.aspx?p=1193471
19. ANDREW. Rachel. A Guide to CSS Support in Browsers. [online]. 4 February 2019. [Accessed 1 October 2022]. Available from: https://www.smashingmagazine.com/2019/02/css-browser-support/
20. CSS Variables. W3Schools [online]. [Accessed 2 October 2022]. Available from: https://www.w3schools.com/css/css3_variables.asp
21. Document Object Model. MDN Web Docs [online]. [Accessed 12 October 2022]. Available from: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model
22. MONUS, Anna. What is the DOM API (and How is it Used to Write JavaScript for the Web). [online]. 1 December 2021. [Accessed 12 October 2022]. Available from: https://webdesign.tutsplus.com/articles/what-is-the-dom-api-for-javascript--cms-35650
23. Web Development Framework. TechTarget [online]. [Accessed 12 October 2022]. Available from: https://www.techtarget.com/searchcontentmanagement/definition/web-development-framework-WDF
24. Media Query CSS Tutorial. freeCodeCamp [online]. [Accessed 25 October 2022]. Available from: https://www.freecodecamp.org/news/css-media-queries-breakpoints-media-types-standard-resolutions-and-more/
25. Introduction to Programming in JavaScript. Launchschool [online]. [Accessed 25 October 2022]. Available from: https://launchschool.com/books/javascript/read/introduction
26. DUCKETT, Jon. HTML & CSS: Design and Build Websites. Wiley, 2011. ISBN 1118008189.
27. CSS Imports. MDN Web Docs [online]. [Accessed 25 October 2022]. Available from: https://developer.mozilla.org/en-US/docs/Web/CSS/@import
28. Pseudo Elements. MDN Web Docs [online]. [Accessed 25 October 2022]. Available from: https://developer.mozilla.org/en-US/docs/Web/CSS/Pseudo-elements
29. CSS Layout. W3Techs [online]. [Accessed 25 October 2022]. Available from: https://www.w3schools.com/css/css_positioning.asp
30. How To Use the LESS CSS Preprocessor [online]. [Accessed 31 October 2022]. Available from: https://www.smashingmagazine.com/2010/12/using-the-less-css-preprocessor-for-smarter-style-sheets/
31. Stylus Language. Stylus [online]. [Accessed 1 November 2022]. Available from: https://stylus-lang.com/

32. Mixins. Sass [online]. [Accessed 1 November 2022]. Available from: https://sass-lang.com/documentation/at-rules/mixin
33. CHERRY, Ben. Performance of strict vs. loose equality. [online]. 8 March 2010. [Accessed 2 November 2022]. Available from: http://www.adequatelygood.com/Performance-of-vs-.html
34. JavaScript Syntax. W3schools [online]. [Accessed 2 November 2022]. Available from: https://www.w3schools.com/js/js_syntax.asp
35. Expression vs. statement. F# for Fun and Profit [online]. [Accessed 3 November 2022]. Available from: https://fsharpforfunandprofit.com/posts/expressions-vs-statements/#:~:text=In%20programming%20language%20terminology%2C%20an,and%20doesn't%20return%20anything.
36. Functions. JavaScript Info [online]. [Accessed 4 November 2022]. Available from: https://javascript.info/function-basics
37. ALVARO, Trigo. How to Change CSS with JavaScript. [online]. 29 March 2022 [Accessed 4 November 2022]. Available from: https://alvarotrigo.com/blog/change-css-javascript/
38. CSS Contain property. MDN Web Docs [online]. [Accessed 11 November 2022]. Available from: https://developer.mozilla.org/en-US/docs/Web/CSS/contain
39. CSS Content visibility property. MDN Web Docs [online]. [Accessed 11 November 2022]. Available from: https://developer.mozilla.org/en-US/docs/Web/CSS/content-visibility
40. CSS performance optimization. MDN Web Docs [online]. [Accessed 12 November 2022]. Available from: https://developer.mozilla.org/en-US/docs/Learn/Performance/CSS
41. CSS will-change. MDN Web Docs [online]. [Accessed 18 November 2022]. Available from: https://developer.mozilla.org/en-US/docs/Web/CSS/will-change
42. How Browsers Work. MDN Web Docs [online]. [Accessed 25 November 2022]. Available from: https://developer.mozilla.org/en-US/docs/Web/Performance/How_browsers_work
43. FAN, Yan. Synchronous and Blocking JavaScript. [online]. 16 December 2022. [Accessed 5 January 2023]. Available from: https://www.codechrysalis.io/blog/synchronous-blocking-javascript-tutorial
44. DODSON, Rob. Controlling Font Performance with font-display. Chrome Developers. [online]. 31 January 2016. [Accessed 27 November 2022]. Available from: https://developer.chrome.com/blog/font-display/
45. Visual Studio Code. [online]. [Accessed 1 January 2023]. Available from: https://code.visualstudio.com/
46. Node Package Manager. npm [online]. [Accessed 1 January 2023]. Available from: https://www.npmjs.com/
47. React – A JavaScript Library for building user interfaces [online]. [Accessed 1 January 2023]. Available from: https://reactjs.org/
48. About PageSpeed Insights [online]. Google Developers [Accessed 1 January 2023]. Available from: https://developers.google.com/speed/docs/insights/v5/about
49. Live Sass Compiler [online]. [Accessed 1 January 2023]. Available from: https://marketplace.visualstudio.com/items?itemName=ritwickdey.live-sass
50. Mobile Internet Traffic. Oberlo [online]. [Accessed 4 January 2023]. Available from: https://www.oberlo.com/statistics/mobile-internet-traffic
51. Google LightHouse. Google Developers [online]. [Accessed 20 January]. Available from: https://developer.chrome.com/docs/lighthouse/overview/

52. FRAIN, Ben. 2012. "Responsive Web Design with HTML5 and CSS". s.l. : Packt, 2012. ISBN 1839219793.
53. FRAIN, Ben. 2017. "Enduring CSS". s.l. : Packt, 2017. ISBN 9781787282803.
54. GRIFFITHS, David. 2021. „React Cookbook: Recipes for Mastering the React Framework“. s.l. : O'Reilly, 2021. 1492085847.
55. NATARAJAN, H. and RASHMI, R. Improving a website's First Meaningful Paint (FMP) by optimizing render blocking resources – An experimental case study [online]. [Accessed 25 January 2023]. Available from: https://www.diva-portal.org/smash/get/diva2:1479947/FULLTEXT01.pdf
56. EPSTEIN, Chris. Less/Sass Comparison. [online]. 2010. [Accessed 31 October 2022]. Available from: https://gist.github.com/chriseppstein/674726
57. ADEWUMI, A.; MISRA, S.; DAMAŠEVIČIUS, R. A Complexity Metrics Suite for Cascading Style Sheets. [online]. [Accessed 2 March 2023]. Available from: https://doi.org/10.3390/computers8030054
58. BIÇER, M.; DIRI, B. Defect Prediction for Cascading Style Sheets. [online]. [Accessed 2 March 2023]. Available from: https://www.sciencedirect.com/science/article/abs/pii/S1568494616302484

List of pictures, tables, graphs and abbreviations

7.1 List of pictures

7.2 List of tables

7.3 List of graphs

7.4 List of abbreviations

1. CSS: Cascading Style Sheets
2. W3C: World Wide Web Consortium
3. HTML: Hyper Text Markup Language
4. RGB: Red, Green, Blue
5. ECMA: European Computer Manufacturer's Association
6. API: Application Program Interface

7. UI: User Interface
8. Sass: Syntactically Awesome Stylesheets
9. DOM: Document Object Model
10. AST: Abstract Syntax Tree
11. GPU: Graphical Processing Unit
12. CPU: Central Processing Unit
13. TBT: Total Blocking Time
14. FCP: First Contentful Paint

7.5 List of Source Code Snippets