RespVis:
A Low-Level Component-Based Framework for Creating Responsive SVG Charts

Peter Oberrauner

RespVis:
A Low-Level Component-Based Framework for Creating Responsive SVG Charts

Peter Oberrauner B.Sc.

Master's Thesis

to achieve the university degree of

Diplom-Ingenieur

Master's Degree Programme: Software Engineering and Management

submitted to

Graz University of Technology

Supervisor

Ao.Univ.-Prof. Dr. Keith Andrews
Institute of Interactive Systems and Data Science (ISDS)

Graz, 03 Dec 2021

RespVis:
Ein Low-Level Komponenten-Basiertes Framework zum Erstellen von Responsiven SVG Diagrammen

Peter Oberrauner B.Sc.

Masterarbeit

für den akademischen Grad

Diplom-Ingenieur

Masterstudium: Software Engineering and Management

an der

Technischen Universität Graz

Begutachter

Ao.Univ.-Prof. Dr. Keith Andrews
Institute of Interactive Systems and Data Science (ISDS)

Graz, 03 Dec 2021

Diese Arbeit ist in englischer Sprache verfasst.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The document uploaded to TUGRAZonline is identical to the present thesis.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Dokument ist mit der vorliegenden Arbeit identisch.

Date/Datum

Signature/Unterschrift

Abstract

[TODO: Write Abstract]

keywords:

- responsive, visualisation, component-based, low-level, framework
- bar chart, line chart, scatterplot, ... [parcoord]
- JavaScript, TypeScript, D3
- SVG, Canvas, WebGL
- Node, gulp, rollup

Kurzfassung

[TODO: Translate abstract into german]

Contents

List of Figures

List of Tables

List of Listings

Acknowledgements

[TODO: Write acknoledgements]

Peter Oberrauner
Graz, Austria, 03 Dec 2021

Credits

I would like to thank the following individuals and organisations for permission to use their material:

- The thesis was written using Keith Andrews' skeleton thesis [Andrews 2019].

[TODO: Add further credits?]

Chapter 1

Introduction

This thesis introduces RespVis, a component-based framework for creating responsive SVG charts which is built on standard browser technologies like HTML, SVG and JavaScript.

[TODO: Outline the various chapters]

Chapter 2

Web Technologies

2.1 HyperText Markup Language (HTML)

HTML is a document markup language for documents that are meant to be displayed in web browsers. The original proposal and implementation in 1989 came from Tim Berners-Lee who was a contractor at CERN at the time [Berners-Lee 1989]. Over the years, the standard has been developed by a range of different entities like the CERN and the Internet Engineering Task Force (IETF). Today, HTML exists as a continuously evolving living standard without specific version releases that is maintained by the Web Hypertext Application Technology Working Group (WHATWG) and the World Wide Web Consortium (W3C) [WHATWG, W3C 2021].

The primary purpose of HTML is to define the content and structure of web pages. This is achieved with the help of HTML elements, which are composed in a hierarchical tree structure and define modular pieces of content that can be interpreted by web browsers. An example of a basic HTML page can be seen in Figure 2.1.

A strong pillar of HTML's design is extensibility. There are multiple mechanisms in place to ensure applicability to a vast range of use cases. These mechanisms include:

- Specifying classes of elements using the `class` attribute. This effectively creates custom elements while still basing them on the most related, already existing elements.
- Using `data-*` attributes to decorate elements with additional data that can be used by scripts. The HTML standard guarantees that these attributes are ignored by browsers.
- Embedding custom data using `<script type="">` elements that can be accessed by scripts.

Figure 2.1: HTML pages are structured as a hierarchical tree of elements which enables the composition of complex structures. [Image drawn by the author of this thesis.]

2.2 Cascading Style Sheets (CSS)

This section is not meant as a comprehensive guide to CSS but to give an overview and reiterate over the concepts that are important in the context of this thesis. In particular, the history of CSS is briefly summarized, a refresher on selectors and cascades is given and the different modules of CSS-based layouting are compared.

Cascading Style Sheets (CSS) is a style sheet language that is used to specify the presentation of HTML documents. It can either be embedded directly in HTML documents using `<style>` elements or it can be defined externally and linked into them using `<link>` elements. This characteristic of being able to externally describe the presentation of documents yields a lot of flexibility because multiple documents with different content can reuse the same presentation by linking to the same CSS file. This solved the problem of having to individually define the presentation of every page with presentation elements like `<strong>` or `<em>`, as was the case in earlier versions of HTML [WHATWG, W3C 1997].

CSS was initially proposed by Lie 1994 and standardized into CSS1 by the W3C in 1996 [W3C 1996]. Throughout its history, the adoption of CSS by browser vendors was fraught with complications and even though most major browsers soon supported almost the full CSS standard, their implementations sometimes behaved vastly different from their competitor's. This meant that authors of web pages usually had to resort to workarounds and provide different style sheets for different browsers. In recent years, CSS specifications have become much more detailed [W3C 2011a] and browser implementations have become more stable with fewer inconsistencies. It has therefore become much rarer that browser-specific workarounds have to be applied, which drastically improves the development experience.

A CSS style sheet contains a collection of rules and each rule consists of a selector and a block of style declarations. Selectors are defined in a custom syntax and are used to match HTML elements. All elements that are matched by the selector of a rule will have the rule's style declarations applied to them. The selector syntax is fairly simple when merely selecting elements of a certain type, but it also provides the means for selecting elements based on their contexts or attributes. For a summary of the CSS 2.1

Pattern	Matches
*	Any element.
E	Elements of type E.
E F	Any element of type F that is a descendant of elements of type E.
E > F	Any element of type F that is a direct descendant of elements of type E.
E + F	Any element of type F that is a directly preceded by a sibling element of type E.
E:P	Elements of type E that also have the pseudo class P.
.C	Elements that have the class C.
#I	Elements that have the ID I.
[A]	Elements that have an attribute A.
[A=V]	Elements that have an attribute A with a value of V.
S1, S2	Elements that match either the selector S1 or the selector S2.

Table 2.1: A summary of the CSS 2.1 selector syntax. [Table created by the author of this thesis with data from [W3C 2011a]]

selector syntax, see Table 2.1.

Another important characteristic of CSS is the cascading of styles. There's a lot of depth to how the final style of an element is calculated and W3C 2011a should be consulted for detailed notes on this topic. The most important thing to state in the context of this work, is that styles can be overwritten. When multiple rules match an element and define different values for the same style property, the values of the rule with higher specificity will be applied. If multiple rules have the same specificity, the one defined last in the document tree will overwrite all previous ones.

2.2.1 Box Layout

All elements in an HTML document are laid out as boxes, which is defined as the CSS box model. The box model states that every element is wrapped in a rectangular box and every box is described by its content and the optional surrounding margin, border and padding areas. Margins are used to specify the invisible spacing between boxes, whereas the border is meant as a visible containment around the content of a box and the padding describes the invisible spacing between the content and the border. A visual representation of these concepts can be seen in Figure 2.2.

In early versions of CSS, before the introduction of the Flexible Box (Flexbox) layout module [W3C 2009], the box model was the only way to lay out elements. Style sheet authors had to meticulously define margins of elements and their relative (or absolute) positions in the document tree. The responsive capabilities of this kind of layouting are very limited because different configurations for varying screen sizes have to be done manually using media queries and more complex features, like filling the remaining space available, had to be implemented via scripting.

2.2.2 Flexible Box Layout (Flexbox)

Even though the first draft of the Flexbox layout module was already published in 2009 [W3C 2009], browser implementations have been a slow and bug ridden process [Use 2021a] that held back adoption by users for the first couple of years after its inception. Over the past few years, partly through the deprecation of Internet Explorer [Microsoft 2020], all major browser implementations of current Flexbox standards [W3C 2018] have become stable and, in most cases, fallback styling is not necessary anymore.

Flexbox is a mechanism for one-dimensionally laying out elements in either rows or columns. This one-dimensionality is what separates it from grid-based layouting, which is inherently two-dimensional. Flexbox layouting can be enabled for child elements via setting the `display: flex` property on a container

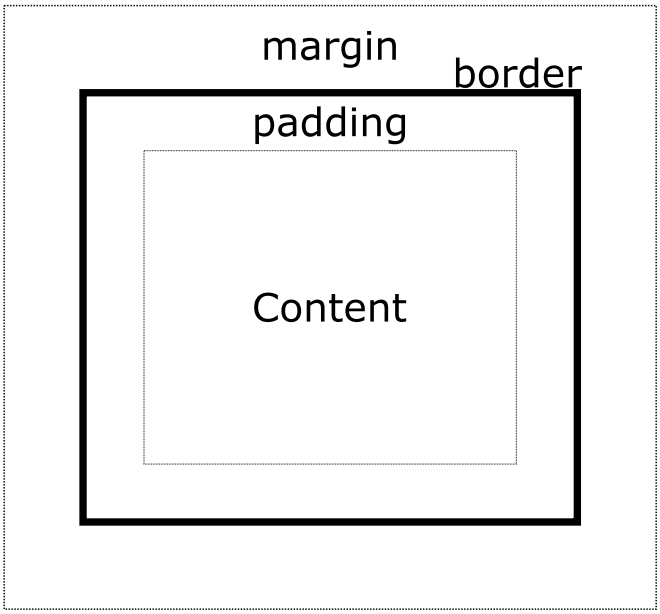

Figure 2.2: The CSS box model is used to define the properties of boxes that wrap around HTML elements. [Image drawn by the author of this thesis.]

Figure 2.3: The `justify-content` property is used to distribute items along the main axis of a Flexbox container. [Image created by the author of this thesis.]

element. The direction of the layout can then be specified using the `flex-direction` property which can be set to either `row` or `column`.

The items inside a Flexbox container can have either a fixed or a relative size. When items should be sized relative to the size of their containers, the proportions of how the available space should be divided can be controlled using ratios. These ratios can be set on item elements via the `flex` property.

Another important feature of Flexbox layouting is the controllable spacing of items which can be specified separately for both the main and the cross axis of the layout. Spacing along the main axis can be configured with the `justify-content` property which can take a number of different values that are demonstrated in Figure 2.3. The property that controls alignment of items on the cross axis is either the `align-items` property on container elements or the `align-self` property on the items themselves.

This section only graced the surface of what is possible with the Flexbox layout module. Those properties that were discussed, were only discussed in a very broad sense and there are many useful properties like `flex-grow`, `flex-shrink` or `flex-wrap` that weren't included in this overview. For a more

detailed look at this topic it is recommended to review W3C 2018.

2.2.3 Grid Layout

Similar to the previous section about Flexbox layout, the goal of this section is not to be a comprehensive guide on the CSS Grid layout module but to provide a rough overview over its general characteristics. For more detailed information, other sources like Meyer 2016 and W3C 2020a are recommended.

The initial proposal of the CSS Grid layout module has already been published in 2011 [W3C 2011b] and it has been further refined over the years. At the time of writing, even though it still exists as merely a candidate recommendation for standardization [W3C 2020a], many browsers have already adopted it. The history of browser adoption has been, similar o the one of Flexbox layout, filled with inconsistencies and bugs. However, browser support has been considered stable and safe to use since 2017 when Chrome, Firefox, Safari and Edge removed the need for vendor prefixes and the deprecation of IE was announced [Use 2021b].

The Grid layout module defines laying out elements in a two-dimensional grid, which differences it from the one-dimensional layouting that can be achieved with a Flexbox layout. Grid layouting of elements can be enabled by setting the `display:grid` property on their container. The grid in which items shall be laid out is then defined using the `grid-template-rows` and `grid-template-columns` properties. There also exists the `grid-template` property, which can be used as a shorthand to simultaneously specify both rows and columns of a grid.

Item elements need to specify the cell of the grid into which they shall be positioned. This is done with the `grid-row` and `grid-column` properties, which take the corresponding row and column indices as values. Items can also be configured to span multiple cells by not only specifying single indices but whole index ranges as the values of those properties.

Every cell in a grid can also be assigned a specific name via the `grid-template-areas` property on the grid container element. The items within the grid can then position themselves in specifically named grid cells using the `grid-area` property instead of directly setting the row and column indices. The benefit of positioning items this way, is that the structure of the grid can freely be changed without having to respecify the cells in which items belong. As long as the new layout still specifies the same names of cells somewhere in the grid, the items will be automatically placed at their new positions.

There are also properties that allow to control the layout of items within grid cells and the layout of grid cells themselves. Similar to how it is described in Section 2.2.2, this can be configured with the `align-items` and `justify-items` properties for laying out within grid cells, and the `align-content` and `justify-content` props for laying out the grid cells themselves. The latter `*-content` props only make sense when the cells don't cover the full area of the grid. For a visual comparison between the `*-items` and `*-content` properties, see Figure 2.4.

There is a lot of overlap between the CSS Grid and Flexbox layout modules and, at first sight, it seems like Grid layout supersedes Flexbox layout because everything that can be done using Flexbox layout can also be done with Grid layout. While that is true, the inherent difference in dimensionality and the resulting syntactical characteristics lead to better suitability of one technology over the other, depending on the situational requirements. As a general rule [Rendle 2017], top-level layouts that require two-dimensional positioning of elements are usually implemented using a Grid layout, whereas low-level layouts that merely need laying out on a one-dimensional axis are more expressively described using a Flexbox layout.

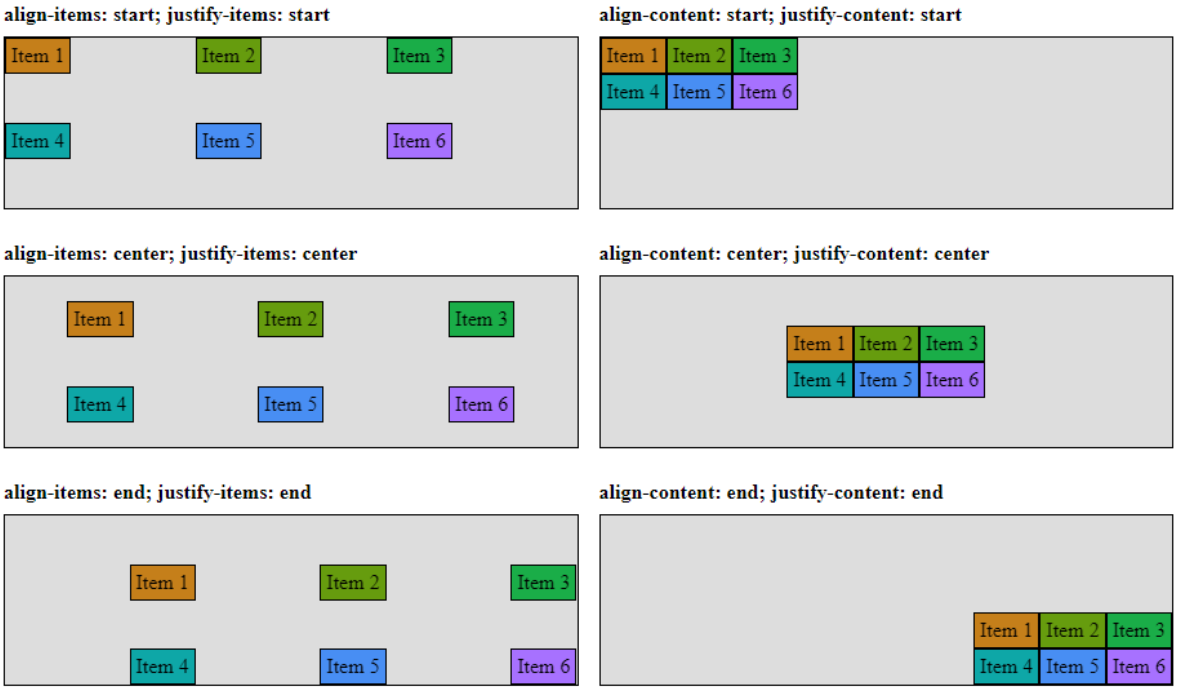

Figure 2.4: The `*-items` properties are used to lay out items within their grid cells, whereas the `*-content` properties are used to lay out the grid cells themselves. [Image created by the author of this thesis.]

2.3 JavaScript (JS)

JavaScript (JS) is one of the three core technologies of the web: HTML for content, CSS for presentation and JS for behavior. the goal of this section is to briefly summarize JS, its history and the specific Web APIs that are relevant for this work.

JavaScript is a client-side scripting language, which requires an engine that interprets it. These engines are usually provided by browsers, but there are also standalone engines, like NodeJS, available. It is a multi-paradigm language that supports event-driven, as well as functional and imperative programming and undoubtedly influenced by the popularity of the web, JavaScript is currently the most used programming language worldwide [Liu 2021].

JavaScript has initially been created by Netscape in 1995 [Netscape 1995] and before that, websites were only able to display static content, which drastically limited the usefulness of the web. Seemingly, Microsoft saw JavaScript as a potentially revolutionary development because they reverse-engineered the implementation of Netscape and published their own version of the language for Internet Explorer in 1996 [Microsoft 1996]. Both implementations were noticeably different from one another but the uncontested monopoly of the Internet Explorer [Routley 2020] held back the standardization efforts undertaken by Netscape [Ecma International 1997]. When Firefox was released in 2004 [Mozilla 2004] and Chrome in 2008 [Google 2008], they gained a considerable share of the market [Statcounter 2021] (see Figure 2.5) and all the major browser vendors collaborated on the standardization of JavaScript as ECMAScript 5 in 2009 [Ecma International 2009]. Since then, JavaScript has been continuously developed, and its latest version was released as ECMAScript 6 in 2015 [Ecma International 2015].

RespVis is a browser-based library, which is designed to run within the JavaScript engine of a browser. Therefore, it builds heavily on widely supported Web APIs, which are JavaScript modules that are specifically meant for the development of web pages. These Web APIs are standardized by the W3C and each browser has to individually implement them in their JavaScript engine.

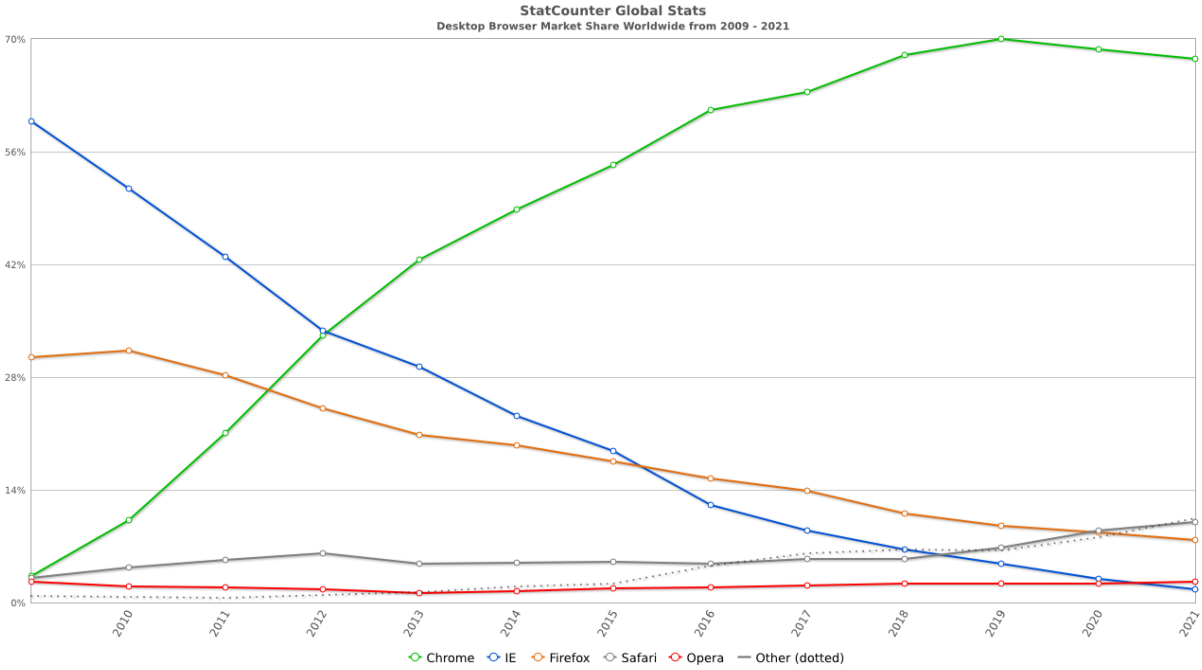

Figure 2.5: Since their release, Firefox and Chrome have contested the monopoly of the Internet Explorer and continuously gained more market share. Recently, Chrome seems to be gaining an increasingly strong position within the market. [Image taken from Statcounter 2021]

Undoubtedly the most popular Web API, that every web developer must be familiar with, is the Document Object Model (DOM). The DOM is the programming interface and data representation of a document. Internally, a document is modeled as a tree of objects, where each object corresponds to a specific element in the document hierarchy and its associated data and functions. In addition to the querying of elements, the DOM also defines the functionality to mutate them and their attributes, as well as the functionality for handling and dispatching of events. It also exposes the mechanism of `MutationObservers`, which are used to observe attribute or children changes in the document tree. The initial publication of the DOM dates back to 1998 [W3C 1998] and currently it is maintained as a living standard by the WHATWG [W3C 2021].

A second important Web API in the context of this work is the `ResizeObserver` API. Its purpose is the ability to observe an element's size and respond to changes, which increases the responsive capabilities of websites. Previously, scripts could only respond to changes in the overall viewport size via the `resize` event on the `window` object, but this meant, that changes of an individual element's size through attribute changes could not be detected. This is fixed by the `ResizeObserver` API, which is already fully supported by all modern browsers, even though it has so far only been published as an editor's draft [W3C 2020b].

2.4 TypeScript (TS)

TypeScript (TS) is a strongly-typed programming language, that is designed as an extension of JavaScript. Syntactically, it is a superset of JavaScript that enables the annotation of properties, variables, parameters and return values with types. Because it is a superset of JavaScript, TypeScript requires a compiler that transpiles the code into valid JavaScript code, which can either be based on ECMAScript 5 [Ecma International 2009] or ECMAScript 6 [Ecma International 2015].

Initially, TypeScript has been released by Microsoft in 2012 [IDG News Service 2012] to extend JavaScript with features that were already present in more mature languages, and whose absence in JavaScript caused difficulties when working on larger codebases. At the time of the initial development,

Design Property	Description
Full erasure	Types are completely removed by the compiler and therefore there is also no type checking at runtime.
Type inference	A big portion of types can be inferred from usage, which minimizes the number of types that have to be explicitly written. E.
Gradual typing	Type checking can be selectively prevented by using the dynamic any type.
Structural types	Types are defined via their structure as opposed to nominal type systems, which define types via their names. This better fits to JavaScript because here, objects are usually custom-built and used based on their shapes.
Unified object types	A type can simultaneously describe objects, functions and arrays. These constructs are common in JavaScript and therefore TypeScript needs to enable their typing.

Table 2.2: A summary over the major design properties on which TypeScript's type system is built. [Table created by the author of this thesis with data from [Bierman et al. 2014]]

ECMAScript 6 hasn't been published yet and TypeScript was used to already be able to use features that would later be implemented by ECMAScript 6. These features included a module system to be able to split source code into reusable chunks and a class system to aid object-oriented development. The TypeScript code using these features could then be translated via the compiler into standard-conform ECMAScript 5 code, which could be interpreted by current JavaScript engines at the time. At the time of writing, ECMAScript 6 is already supported by all modern browsers and therefore the main benefit of TypeScript over JavaScript resides primarily in the addition of a static type system.

The extension of JavaScript with a type system brings many benefits. One such a benefit is the improved tooling that comes with type annotated code. Tools will be able to point out errors during development and assist you with automated fixes and improved code completion and navigation. Additionally, studies like Gao et al. 2017 have evaluated software bugs in publicly available codebases and found that 15% of them could have been prevented with static type checking.

The TypeScript type system has been designed to support the constructs that are possible in JavaScript as completely as possible, which is achieved via structural types and unified object types. Another goal has also been to make the type annotation of JavaScript code as effortless as possible to improve adoption by already existing projects. This is done by consciously allowing the type system to be statically unsound via gradual typing and also by employing type inference to reduce the amount of necessary annotations. The major properties of the type system's design are summarized in Table 2.2.

2.5 Web Graphics

Graphics are used as a medium for visual expression to enhance the representation of information on the web. There are versatile fields of application like the integration of maps, photographs or charts in a website. Multiple complementary technologies exist and each solves particular use cases of web authors. The different ways of embedding graphics in a document are raster images, Scalable Vector Graphics (SVG) and through the Canvas API. These technologies are described in more detailed in the following sections.

2.5.1 Raster Images

Raster images represent graphics as a rectangular, two-dimensional grid of pixels with a fixed size. This fixed grid size results in limited scalability and, whenever an image is displayed in a different size, visual scaling artifacts will be noticeable as can be seen in Figure 2.6. Raster images are either created by

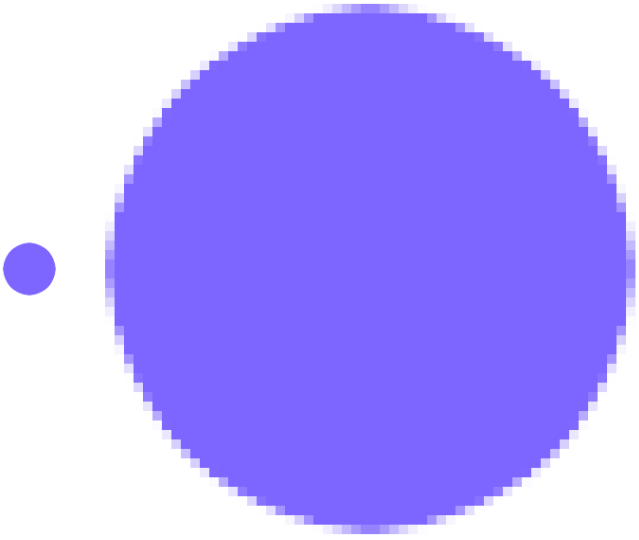

Figure 2.6: This figure demonstrates the artifacts that occur when scaling raster images to a size that is different from the size they were encoded in. The left shows a raster image of a circle in its original size and the right shows the same image scaled to ten times its size. [Image created by the author of this thesis.]

image capturing devices or special editing software and saved as binary files in varying formats. The most widely used format for raster images is Portable Network Graphics (PNG), which is standardized by the W3C [**PNG**] and optimized for usage on the web. It features a lossless compression and streaming capabilities, which enable progressive rendering of images during loading.

Raster images are embedded into documents in binary format. This means that the contents of the graphic are not accessible in a non-visual representation. To make the information accessible to visually impaired people it is required to provide an additional textual description of the graphic's content on the embedding element via the `alt` and `longdesc` attributes.

[TODO: Create these figures using subfigures. See thesis-details.tex in template]

2.5.2 Scalable Vector Graphics (SVG)

Scalable Vector Graphics (SVG) is an XML-based format for vector graphics, which describe images as sets of shapes that can be scaled without loss of quality. It was initially published by the W3C in 1999 [**SVG1.0**] and the most recent version SVG 1.1 was released in 2011 [**SVG1.1**]. Because SVG is based on XML, SVG files can be created with any text editor and a simple example of an SVG document can be seen in Listing 2.1 with its visualization being shown in Figure 2.7. In addition to being able to write SVG documents manually, there is also specialized software available that helps with the composition of more complex images.

The encoding in XML leads to SVG being the best format to represent graphics in terms of accessibility. Graphics are directly saved in a hierarchical and textual form, which describes their shapes and how they are composed. In addition to the shapes being inherently accessible, the various elements of an SVG document can be annotated with further information that aids comprehension when consumed in a non-visual way.

SVG files are XML documents whose meta format is described in special SVG namespaces. Therefore, the elements of SVG documents can be accessed in scripts via the DOM Web API. This means that elements can be completely controlled by scripts, which enables similar levels of interactivity than for

```svg
<svg viewBox="0, 0, 64, 64" xmlns="http://www.w3.org/2000/svg">
  <circle cx="32" cy="32" r="30" fill="#7c66ff" />
</svg>
```

Listing 2.1: A simple SVG document containing a circle element. The visual representation of this document in different sizes is shown in Figure 2.7

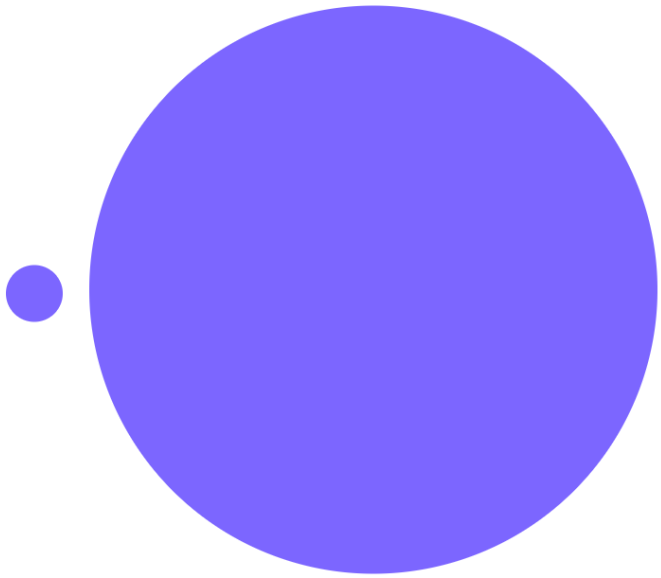

Figure 2.7: SVG documents can be scaled freely without any artifacts occurring. This figure represents the visual representation of the SVG document in Listing 2.1 scaled to different sizes. [Image created by the author of this thesis.]

HTML documents.

The most widely supported way of styling SVG elements is via attributes, which is supported by every software dealing with SVG files. However, the specification aims for maximum compatibility with HTML, and therefore it is also possible to use CSS to style and animate SVG elements when they are rendered in a browser. Using CSS to separate presentation from content has many benefits that were already described in Section 2.2, but this can not be done for all attributes because only the so-called presentation attributes like `fill` and `stroke-width` are available in CSS. These attributes are taxonomically listed in the specification [**CSS1.1**] and have been extended by additional attributes like `x`, `y`, `width` and `height` in upcoming releases [**CSS2**].

2.5.3 Canvas

The canvas element has been introduced in HTML5 [WHATWG, W3C 2021] and is used to define a two-dimensional, rectangular region in a document that can be drawn into by scripts. Even though the rendering of dynamic graphics as canvas elements is faster than representing them as SVG documents, their use is explicitly discouraged by the WHATWG when another suitable representation is possible. The reasons for this are that canvas elements are not compatible with other web technologies like CSS or the DOM Web API and because the resulting rendering as a raster image provides only very limited

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
  <head>
    <title>Canvas</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta charset="UTF-8" />
    <style>
      body {
        display: flex;
        flex-direction: row;
        align-items: center;
        justify-items: center;
        gap: 1rem;
      }
    </style>
  </head>
  <body>
    <canvas width="64" height="64"></canvas>
    <canvas width="640" height="640"></canvas>
    <script>
      const canvases = Array.from(document.getElementsByTagName('canvas'));
      canvases.forEach((canvas) => {
        const width = canvas.clientWidth;
        const height = canvas.clientHeight;
        const context = canvas.getContext('2d');
        context.fillStyle = '#7c66ff';
        context.beginPath();
        context.arc(width / 2, height / 2, width / 2, 0, Math.PI * 2);
        context.fill();
        context.closePath();
      });
    </script>
  </body>
</html>
```

Listing 2.2: A basic HTML document containing two canvases of different sizes that render circles relative to the canvas size. The visual representation of this document is shown in Figure 2.8

possibilities for accessibility.

Rendering is done via a low-level API, which is provided by the rendering context of a particular canvas. The type of render context depends on the context mode of a canvas with the two most significant ones being `2d` and `webgl`. A two-dimensional render context (created with the `2d` context mode) enables platform-independent rendering via a software renderer, whose API is standardized directly in the canvas specification. If even faster or three-dimensional rendering is required, a WebGL render context (created with the `webgl` context mode) provides access to a hardware-accelerated renderer, whose API is standardized in its own specification [**WebGL**]. The availability of individual features on a WebGL render context depends on the client's hardware and therefore, it should only be used if the requirements don't allow for any of the alternatives. An example of an HTML document containing two differently sized canvases into which responsive circles are drawn using a two-dimensional rendering context can be seen in Listing 2.2 with the corresponding visualization displayed in Figure 2.8.

[TODO: Split canvas section into Canvas 2D and Canvas 3D]

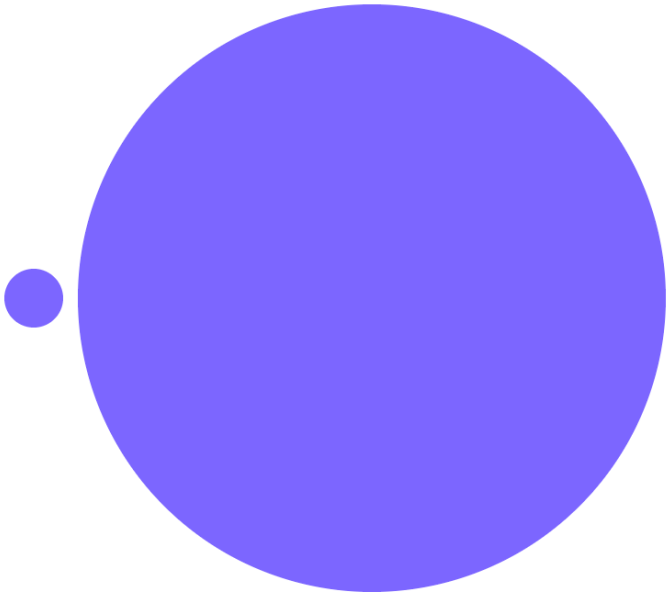

Figure 2.8: Responsive rendering of graphics inside canvas elements has to be implemented manually by calculating everything as ratios of the canvas' dimensions. This figure represents the visual representation of the canvas example in Listing 2.2. [Image created by the author of this thesis.]

2.6 Layout Engines

A layout engine is used to calculate the boundary coordinates of visual components based on some form of input components that are annotated with layout constraints. These layout constraints describe the size and position of components and their relationships between each other in a syntax that is understood by the layout engine. For browser-based layout engines these input components are normally declared as HTML documents which are constrained using CSS. More low-level layout engines require custom formats, which usually involve a hierarchy of objects that is constrained using specific properties. The most relevant layout engines in the context of this work are summarized in the following sections.

2.6.1 Browser Engines

The purpose of a browser engine is to transform documents including their additional resources, like CSS, into visual representations. A browser engine is a core component of every web browser, and it is responsible for laying out elements and rendering them. The terminology of browser engines is very vague with them sometimes also being referred to as layout or render engines. Theoretically, the layout and render processes could be separated into different components but in practice they are tightly-coupled into a combined component, which will be referred to as a browser engine in this work. Some notable browser engines are WebKit [**WebKit**], Blink [**Blink**] and Gecko [**Gecko**].

In a browser engine the layout of elements is constrained with CSS, which gives website authors a lot of flexibility as already described in Section 2.2. They are offered a range of mechanisms to precisely control the layout of elements, like the flexible box and grid layout modules, which don't have to be applied exclusively but can also be used in combination.

The layout module of a browser engine can only be invoked directly by browsers to position HTML elements in actively rendered documents. To use it for calculating layouts of non-HTML constructs, they must be replicated in active documents so that they can be parsed, laid out and rendered by the browser engine. These replicated constructs do not have to be visible, and they can be removed from the document after the layout has been acquired, which means they do not need to be noticeable at all.

A strong limitation of using browser engines to calculate layouts is that it requires a browser runtime to work and, even though there are solutions like Electron available that enable development of native applications using web technologies, this forces applications into this very specific stack of technologies.

2.6.2 Yoga

Yoga [**Yoga**] is a layout engine that enables the computation of layouts that are constrained using the grammar of the flexible box layout CSS module (see Section 2.2.2). It is maintained by Facebook as an open source project since 2016 [**YogaRelease**] with the goal of providing a small and high performance library that can be used across all platforms. This is achieved through the implementation being programmed in the C/C++ programming language, which works on a myriad of devices, with bindings available for other platforms like JavaScript, Android and iOS. Yoga has been very well adopted and is used to perform layouting in major frameworks, such as React Native [**ReactNative**], Litho [**Litho**] and ComponentKit [**ComponentKit**].

2.6.3 FaberJS

FaberJS [**FaberJS**] is a layout engine that is very similar to the Yoga layout engine in the fact that it enables the computation of layouts for constructs other than HTML documents using a layout grammar that has originally been created for CSS. In contrast to Yoga, which is used to create one-dimensional layouts using the flexible box layout grammar, FaberJS implements a two-dimensional layout algorithm that is built on the grammar of the grid layout CSS module (see Section 2.2.3). In the context of two-dimensional information visualization, this inherently two-dimensional approach to layouting makes more sense than trying to achieve two-dimensionality using a one-dimensional one. FaberJS is an open source JavaScript project that has been developed since 2019 by Idera, Inc. Even though the layouts it computes are constrained with the grid layout grammar, it only supports a subset of all the functionalities that are defined in the original CSS module. Some examples of missing functionalities include missing support for margins, gaps and the `*-content` and `grid-auto-*` properties. Working around the limitations caused by these missing features is not trivial, and it seems unlikely that support for them will be added by the FaberJS maintainers in the near future because, at the time of writing, the project has not been updated in nearly two years.

Chapter 3

Information Visualization

[TODO: Explain InfoVis]

[TODO: InfoVis vs. SciVis vs. GeoVis vs. DataVis]

[TODO: Mention purposes of InfoVis] [TODO: Mention Anscombe's Quartet] [TODO: Mention human visual perception] [TODO: Mention preattentive processing]

[TODO: Explain why interactivity is such an important concept in visualization]

[TODO: Mention the history of information visualization] [TODO: Supposedly, there is a good book on this topic. Find it and cite it. History of data visualization or something like that.] [TODO: Add large image with timeline and visualizations]

3.1 Responsive Web Design

[TODO: Define responsive web design] [TODO: Mention adapting to characteristics of consuming device] [TODO: Mention text wrapping] [TODO: Mention]

[TODO: Mention importance of responsive web design] [TODO: Add chart showing the rise of mobile devices]

3.2 Responsive Information Visualization

[TODO: Mention necessity of responsive visualizations in context of responsive web design] [TODO: Mention visualizations being important blocks of content that can not be ignored when designing responsive web pages]

[TODO: Mention scalable vs responsive visualizations] [TODO: Mention responsive layout] [TODO: Mention responsive display density] [TODO: Mention responsive interaction]

[TODO: Mention the book Building responsive data visualizations for the web] [TODO: Mention focus on scalable visualizations rather than responsive visualizations]

3.2.1 Responsive Visualization Patterns

[TODO: Mention patterns in software development] [TODO: Mention patterns in responsive web design] [TODO: Mention patterns in visualizations]

[TODO: Mention patterns defined by Adler et. al] [TODO: 1. Rotating axis labels 2. Removing axis ticks 3. Shortening strings 4. Flipping visualizations by 90 degrees 5. Repositioning visual components 6. Zooming 7. Filtering 8. Reducing data density 9. Changing chart types]

[TODO: Mention high-level actions for responsive adaptations of visualization as defined by Hoffswell et. all] [TODO: Identified by analyzing 231 visualizations from various sources] [TODO: 1. No change 2. Resizing visualizations 3. Repositioning visual components 4. Adding visual components 5. Modifying visual components 6. Removing visual components]

[TODO: Mention patterns as defined by Kim et. al (2021 paper)]

3.2.2 Responsive Visualization Examples

3.3 Information Visualization Libraries

3.3.1 Chartist

3.3.2 Highcharts

3.3.3 ECharts

3.3.4 ...?

3.3.5 D3

[TODO: Mention that D3 is successor of Protovis]

3.3.6 D3FC

[TODO: Mention the mixed HTML/SVG approach when it comes to layout]

Chapter 4

Software Architecture

[TODO: Add software architecture diagram]

[TODO: Describe relationship to D3]

[TODO: Describe storing data on elements]

[TODO: Describe using DOM events for callbacks]

[TODO: Describe components]

4.1 Primitives

4.1.1 Text

4.1.2 Rectangle

4.1.3 Circle

4.2 Series

[TODO: Describe series extension mechanism (enter/update/exit events)]

4.2.1 Bar Series

4.2.2 Grouped Bar Series

4.2.3 Stacked Bar Series

4.2.4 Point Series

4.2.5 Line Series

4.3 Charts

4.3.1 Bar Chart

4.3.2 Grouped Bar Chart

4.3.3 Stacked Bar Chart

4.3.4 Point Chart

4.3.5 Line Chart

4.4 Chart Windows

4.4.1 Bar Chart Window

4.4.2 Grouped Bar Chart Window

4.4.3 Stacked Bar Chart Window

4.4.4 Point Chart Window

4.4.5 Line Chart Window

4.5 Components

4.5.1 Lifecycle

events

updating on data change

updating on bounds change

4.6 Layouter

Chapter 5

Layouter

5.1 CSS Layouting

Chapter 6

Another Chapter; Maybe about Components?

Chapter 7

Selected Details of the Implementation

7.1 D3 Select Function Data Modification

7.2 Save as SVG

Chapter 8

Outlook and Future Work

8.1 Outlook

8.2 Ideas for Future Work

8.2.1 Relative Positioning of Series Items

[TODO: Write about plans to use relative units (%) to position series items which would most likely get rid of the need to update components on bound changes]

8.2.2 Container Queries

Chapter 9

Concluding Remarks

Appendix A

User Guide

Appendix B

Developer Guide

Bibliography

Andrews, Keith [2019]. *Writing a Thesis: Guidelines for Writing a Master's Thesis in Computer Science*. Graz University of Technology, Austria. 24 Jan 2019. http://ftp.iicm.edu/pub/keith/thesis/ (cited on page xiii).

Berners-Lee, Tim [1989]. *Information management: A Proposal*. 1989. http://www.w3.org/History/1989/proposal.html (cited on page 3).

Bierman, Gaving, Martín Abadi, and Mads Torgersen [2014]. *Understanding Typescript*. Volume 8586. Aug 2014, pages 257–281. ISBN 978-3-662-44202-9. doi:10.1007/978-3-662-44202-9_11 (cited on page 10).

Ecma International [1997]. *ECMAScript: A general purpose, cross-platform programming language*. Jun 1997. https://www.ecma-international.org/wp-content/uploads/ECMA-262_1st_edition_june_1997.pdf (cited on page 8).

Ecma International [2009]. *ECMAScript 5th Edition Language Specification*. Dec 2009. https://www.ecma-international.org/wp-content/uploads/ECMA-262_5th_edition_december_2009.pdf (cited on pages 8–9).

Ecma International [2015]. *ECMAScript 6th Edition Language Specification*. Jun 2015. https://262.ecma-international.org/6.0/ (cited on pages 8–9).

Gao, Zheng, Christian Bird, and Earl T. Barr [2017]. *To Type or Not to Type: Quantifying Detectable Bugs in JavaScript*. May 2017, pages 758–769. ISBN 978-1-5386-3869-9. doi:10.1109/ICSE.2017.75 (cited on page 10).

Google [2008]. *A fresh take on the browser*. 01 Sep 2008. https://googleblog.blogspot.com/2008/09/fresh-take-on-browser.html (cited on page 8).

IDG News Service [2012]. *Microsoft Augments JavaScript for Large-Scale Development*. 01 Oct 2012. https://www.infoworld.com/article/2614863/microsoft-augments-javascript-for-large-scale-development.html (cited on page 9).

Lie, Håkon Wium [1994]. *Cacading HTML Style Sheets: A Proposal*. 1994. https://www3.org/People/howcome/p/cascade.html (cited on page 4).

Liu, Shanhong [2021]. *Most used programming languages among developers worldwide, as of 2021*. 05 Aug 2021. https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/ (cited on page 8).

Meyer, Eric A. [2016]. *Grid Layout in CSS: Interface Layout for the Web*. 1st Edition. O'Reilly Media, 18 Apr 2016. ISBN 9781491930212 (cited on page 7).

Microsoft [1996]. *Microsoft Internet Explorer 3.0 Beta Now Available*. 29 May 1996. https://news.microsoft.com/1996/05/29/microsoft-internet-explorer-3-0-beta-now-available/ (cited on page 8).

Microsoft [2020]. *Microsoft 365 apps say farewell to Internet Explorer 11 and Windows 10 sunsets Microsoft Edge Legacy*. 17 Aug 2020. https://techcommunity.microsoft.com/t5/microsoft-365-

blog/microsoft-365-apps-say-farewell-to-internet-explorer-11-and/ba-p/1591666 (cited on page 5).

Mozilla [2004]. *Firefox 1.0 Release Notes*. 09 Nov 2004. https://website-archive.mozilla.org/www.mozilla.org/firefox_releasenotes/en-us/firefox/releases/1.0 (cited on page 8).

Netscape [1995]. *Netscape and Sun Announce Javascript, the Open, Cross-Platform Object Scripting Language for Enterprise Networks and the Internet*. 04 Dec 1995. https://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html (cited on page 8).

Rendle, Robin [2017]. *Does CSS Grid Replace Flexbox?* 31 Mar 2017. https://css-tricks.com/css-grid-replace-flexbox/ (cited on page 7).

Routley, Nick [2020]. *Internet Browser Market Share (1996–2019)*. 20 Jan 2020. https://www.visualcapitalist.com/internet-browser-market-share/ (cited on page 8).

Statcounter [2021]. *Desktop Browser Market Share Worldwide*. 31 Jul 2021. https://gs.statcounter.com/browser-market-share/desktop/worldwide/#yearly-2009-2021 (cited on pages 8–9).

Use, Can I [2021a]. *Can I use CSS Flexible Box Layout Module*. 13 Aug 2021. https://caniuse.com/flexbox (cited on page 5).

Use, Can I [2021b]. *Can I use CSS Grid Layout*. 19 Aug 2021. https://caniuse.com/css-grid (cited on page 7).

W3C [1996]. *Cascading Style Sheets Level 1 (CSS 1) Specification*. 17 Dec 1996. https://www.w3.org/TR/REC-CSS1 (cited on page 4).

W3C [1998]. *Document Object Model Specification*. 16 Apr 1998. https://www.w3.org/TR/1998/WD-DOM-19980416/ (cited on page 9).

W3C [2009]. *CSS Flexible Box Layout Module*. 23 Jul 2009. https://www.w3.org/TR/2009/WD-css3-flexbox-20090723 (cited on page 5).

W3C [2011a]. *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification*. 07 Jun 2011. https://www.w3.org/TR/CSS2 (cited on pages 4–5).

W3C [2011b]. *Grid Layout*. 07 Apr 2011. https://www.w3.org/TR/2011/WD-css3-grid-layout-20110407/ (cited on page 7).

W3C [2018]. *CSS Flexible Box Layout Module Level 1*. 19 Nov 2018. https://www.w3.org/TR/css-flexbox-1 (cited on pages 5, 7).

W3C [2020a]. *CSS Grid Layout Module Level 1*. 18 Dec 2020. https://www.w3.org/TR/css-grid-1/ (cited on page 7).

W3C [2020b]. *Resize Observer - First Public Working Draft*. 11 Feb 2020. https://www.w3.org/TR/2020/WD-resize-observer-1-20200211/ (cited on page 9).

W3C [2021]. *DOM Living Standard*. 02 Aug 2021. https://dom.spec.whatwg.org/ (cited on page 9).

WHATWG, W3C [1997]. *HTML 3.2 Reference Specification*. 14 Jan 1997. https://www.w3.org/TR/2018/SPSD-html32-20180315 (cited on page 4).

WHATWG, W3C [2021]. *HTML Standard*. 11 Aug 2021. https://html.spec.whatwg.org (cited on pages 3, 12).