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

RespVis is a web-based framework. As such, it builds on a stack of technologies which are native to the web. The first sections in this chapter introduce the web's core technologies: HTML, CSS, and JavaScript. Then the concepts and importance of TypeScript are introduced, and the different technologies to embed graphics in web pages are discussed. Due to the importance of layouting in this work, three different forms of layout engines are compared. Finally, the concept of responsive web design is summarized. Since there are many things to examine, none of the following sections goes into too much detail, the aim is to give a summary of the concepts that are introduced. For more in-depth information, works referenced in the sections below should be consulted.

2.1 HyperText Markup Language (HTML)

HTML is a document markup language for documents which are 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 was further developed by a range of different entities like the CERN and the Internet Engineering Task Force (IETF). Nowadays, HTML exists as a continuously evolving living standard without specific version releases, which is maintained by the Web Hypertext Application Technology Working Group (WHATWG) and the World Wide Web Consortium (W3C) [Hickson et al. 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, such as `<section>`, `<h1>`, `<p>`, and `<img>`, which are composed into a hierarchical tree structure of modular content, and which is then interpreted by web browsers. A strong pillar of HTML's design is extensibility. There are multiple mechanisms in place to ensure its applicability to a vast range of use cases, including:

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

2.2 Cascading Style Sheets (CSS)

Cascading Style Sheets (CSS) apply styling to HTML elements, effectively separating presentation from content. In earlier versions of HTML [Raggett 1997], elements like `<strong>` and `<em>` prevented the clean separation of presentation from content.

Pattern	Matches
`*`	Any element.
`E`	Elements of type E.
`E F`	Any element of type F which is a descendant of elements of type E.
`E > F`	Any element of type F which is a direct descendant of elements of type E.
`E + F`	Any element of type F which is a directly preceded by a sibling element of type E.
`E:P`	Elements of type E which also have the pseudo class P.
`.C`	Elements which have the class C.
`#I`	Elements which have the ID I.
`[A]`	Elements which have an attribute A.
`[A=V]`	Elements which have an attribute A with a value of V.
`S1, S2`	Elements which 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 [Çelik et al. 2018].]

A CSS style sheet can either be embedded directly in HTML documents using `<style>` elements or can be defined externally and linked to using `<link>` elements. This characteristic of being able to externally describe the presentation of documents brings great flexibility. Multiple documents with different content can reuse the same presentation by linking to the same CSS file. Conversely, alternative style sheets can be applied to the same HTML content to achieve a different styling.

CSS was initially proposed by Lie 1994 and standardized into CSS1 by the W3C in 1996 [Lie and Bos 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 differently. This meant that authors of web pages often had to resort to workarounds, including providing different style sheets for different browsers. In recent years, CSS specifications have become much more detailed [Bos et al. 2011] and browser implementations have become more stable with fewer inconsistencies. It has therefore become much rarer that browser-specific workarounds need to be applied, dramatically improving the developer experience. CSS 2.1 [Bos et al. 2011] was the last CSS standard that has been published as a single, monolithic specification. Since then, the specification has been modularized into different documents, each describing a specific module of the overall CSS specification.

A CSS style sheet contains a collection of rules. 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 which are matched by the selector of a rule will have the rule's style declarations applied to them. The selector syntax is fairly straightforward when selecting elements of a certain type, but also has more sophisticated mechanisms for selecting elements based on their contexts or attributes. Table 2.1 summarizes the selector syntax of Çelik et al. [2018].

Another important characteristic of CSS is the cascading of styles. The exact rules for calculating the final style to be applied to an element are quite involved, and Etemad and Atkins [2021] should be consulted for a detailed description. The most important aspect 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 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.

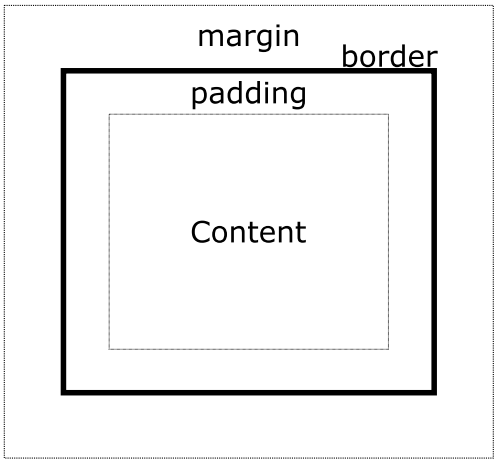


Figure 2.1: The CSS box model defines the properties of boxes which wrap around HTML elements. [Image drawn by the author of this thesis.]

2.2.1 CSS Box Model

All elements in an HTML document are laid out as boxes. The CSS box model specifies how every element is wrapped in a rectangular box and every box is described by its content and optional surrounding margin, border and padding areas. Margins are used to specify 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 properties can be seen in Figure 2.1.

In early versions of CSS, before the introduction of the Flexible Box (Flexbox) layout module [Deakin et al. 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 were very limited, because different configurations for varying screen sizes had to be specified manually using media queries. More complex features, like the filling of available space, required manual implementation via scripting.

2.2.2 CSS Flexbox Layout

CSS Flexible Box layout (Flexbox) [Atkins et al. 2018] is a mechanism for one-dimensional layout of elements in either rows or columns. This one-dimensionality is what separates it from grid-based layout, which is inherently two-dimensional. Even though the first draft of the Flexbox layout module was already published in 2009 [Deakin et al. 2009], implementations by browser vendors have been a slow and bug-ridden process [Deveria 2021a], which held back adoption by users for several years after its inception. More recently though, partly through the deprecation of Internet Explorer [Microsoft 2020], all major browser implementations of current Flexbox standards [Atkins et al. 2018] have reached maturity and, in most cases, fallback styling is not necessary anymore.

Flexbox layouting is enabled for child elements by setting the `display:flex` property on a container 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 layout 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

Figure 2.2: 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.]

with the `justify-content` property, which can take a number of different values and is illustrated in Figure 2.2. Alignment of items on the cross axis is achieved either by the `align-items` property on the container element or the `align-self` property on the items themselves.

This section only grazed the surface of what is possible with the Flexbox layout module. There are many more useful properties like `flex-grow`, `flex-shrink`, and `flex-wrap`. For a more detailed look at this topic it is recommended to review the specification [Atkins et al. 2018] and read the excellent tutorial by Chris Coyier [Coyier 2021].

2.2.3 CSS Grid Layout

The CSS Grid Layout Module [Atkins et al. 2020] defines the layout of elements in a two-dimensional grid. The initial proposal of the CSS Grid layout module was published in 2011 [Mogilevsky et al. 2011] and has been further refined over the years. At the time of writing, even though it still exists as merely a candidate recommendation for standardization [Atkins et al. 2020], many browsers have already adopted it. Similar to the adoption of Flexbox, the history of browser adoption of CSS Grid was initially strewn with inconsistencies and bugs. However, in 2017 the major browsers Chrome, Firefox, Safari and Edge removed the need for vendor prefixes and implementations are now considered stable [Deveria 2021b].

Grid layout of elements is 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. In addition, the `grid-template` property can be used as a shorthand to simultaneously specify both the 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 specifying 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 be freely 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 which control the layout of items within grid cells and the layout of grid cells themselves. Similar to Flexbox, this can be configured with the `align-items` and `justify-items`

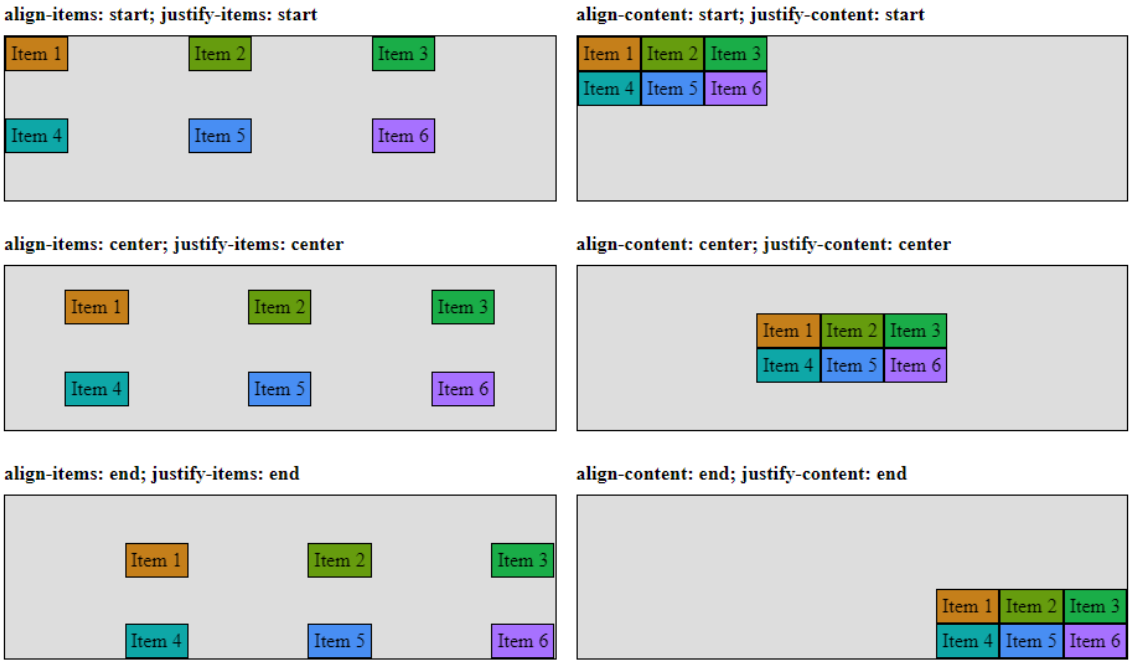


Figure 2.3: The `*-items` properties are used to lay out items within their grid cells, whereas the `*-content` properties are used to lay out the grid cells themselves. [Image created by the author of this thesis.]

properties for laying out within grid cells, and the `align-content` and `justify-content` properties for laying out the grid cells themselves. The latter `*-content` properties only make sense when the cells do not cover the full area of the grid. For a visual comparison between the `*-items` and `*-content` properties, see Figure 2.3.

There is some apparent overlap between the CSS Grid and Flexbox layout modules. At first sight, it seems like Grid layout supersedes Flexbox layout, because everything which 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 syntactic characteristics lead to better suitability of one technology over the other, depending on the context of use. As a general rule [Rendle 2017], top-level layouts which require two-dimensional positioning of elements are usually best implemented using a Grid layout, whereas low-level layouts which merely need laying out on a one-dimensional axis are better implemented using a Flexbox layout.

For more details, the CSS Grid specification [Atkins et al. 2020] and other sources like Meyer [2016] and House [2021] are recommended.

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

It has initially been created by Netscape in 1995 [Netscape Communications Corporation 1995]. Before that, websites were only able to display static content, which drastically limited the usefulness of the web.

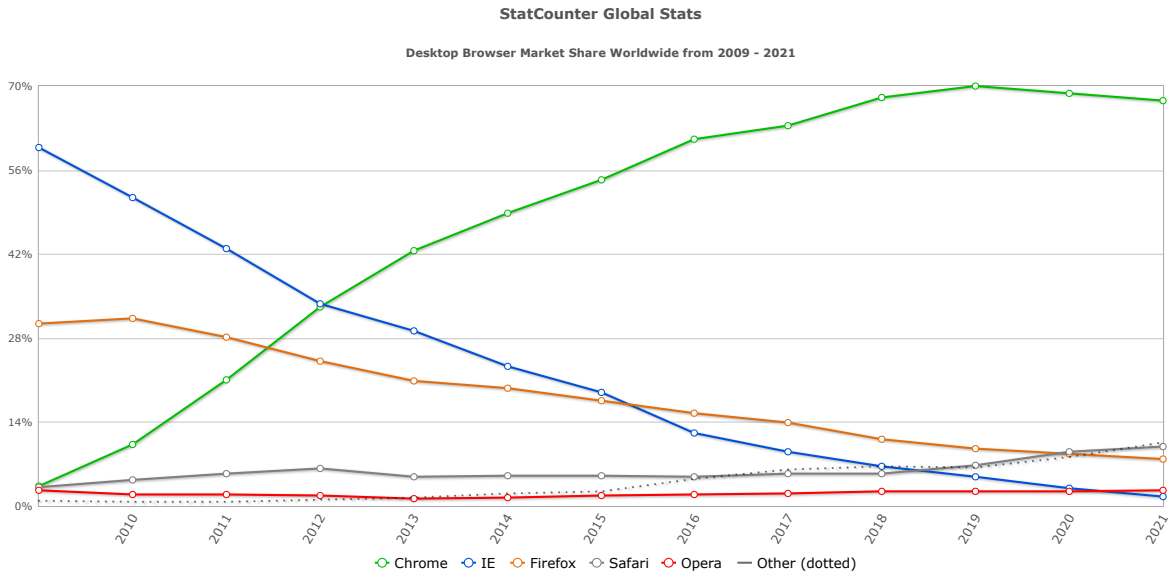


Figure 2.4: Since their release, Firefox and Chrome have contested the monopoly of the Internet Explorer and continuously gained more market share. Recently, Chrome seems to be gaining an increasingly strong position within the market. [Image taken from Statcounter 2021]

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 and the uncontested monopoly of the Internet Explorer [Routley 2020] held back standardization efforts undertaken by Netscape [Ecma International 1997]. When Firefox was released in 2004 [Mozilla 2004] and Chrome in 2008 [Google 2008], they quickly gained a considerable share of the market [Statcounter 2021] (see Figure 2.4). Driven by this new market segmentation, all 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, widely support version, ECMAScript 6, has been released 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 which 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.

The most popular Web API which every web developer is 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 changes of attribute and children in the document tree. The initial publication of the DOM dates back to 1998 [Wood et al. 1998] and currently it is maintained as a living standard by the WHATWG [Kesteren et al. 2021].

Another 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 limitation is fixed by the `ResizeObserver` API, which is already

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 which have to be explicitly written.
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.]

fully supported by all modern browsers, even though it has so far only been published as an editor's draft [Totic and Whitworth 2020].

2.4 TypeScript (TS)

TypeScript (TS) is a strongly-typed programming language which is designed as an extension of JavaScript. Syntactically, it is a superset of JavaScript which enables the annotation of properties, variables, parameters and return values with types. It requires a compiler which transpiles TypeScript code into valid JavaScript code based on specific ECMAScript versions.

Initially, TypeScript has been released by Microsoft in 2012 [Hoban 2012] to extend JavaScript with features which were already present in more mature languages, and whose absence in JavaScript caused difficulties when working on larger codebases. At the time of TypeScript's initial development, it was used to already be able to use features which would later be offered 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 a compiler into standard-conform JavaScript code, which could be interpreted by JavaScript engines at the time. At the time of writing, ECMAScript 6 is already widely 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 which comes with type annotated code. Tools will be able to point out errors early during development and assist developers with automated fixes, improved code completion, and code 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 which 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 TypeScript's type system design are summarized in Table 2.2.

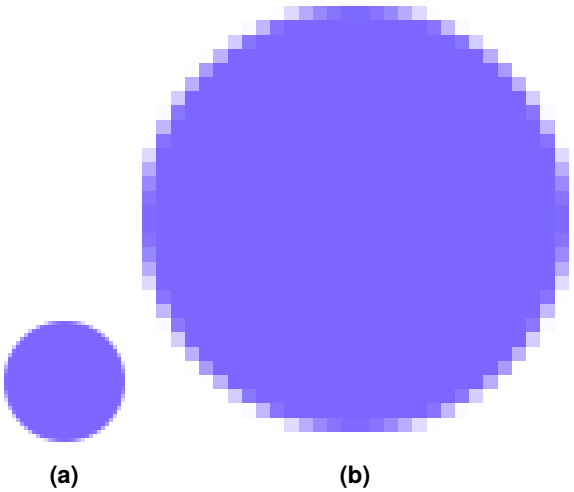


Figure 2.5: This figure demonstrates the artifacts which occur when scaling raster images to a size which is different from the size they were encoded in. (a) shows a raster image of a circle in its original size. (b) shows the same image scaled up considerable. [Image created by the author of this thesis.]

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 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 2.5. Raster images are either created by 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 [Boutell 2003] and optimized for usage on the web. It features a lossless compression and streaming capabilities, which enable progressive rendering of images while they are loaded.

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.

2.5.2 Scalable Vector Graphics (SVG)

The Scalable Vector Graphics (SVG) format 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 [Ferraiolo 1999] and the most recent version SVG 1.1 was released in 2011 [Dahlström et al. 2011]. SVG files are based on XML and therefore can be created with any text editor, but there is also specialized software available which helps with the composition of more complex images. A simple example of an SVG document containing a single circle can be seen in Listing 2.1 with its visualization being shown in Figure 2.6.

```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 2.6

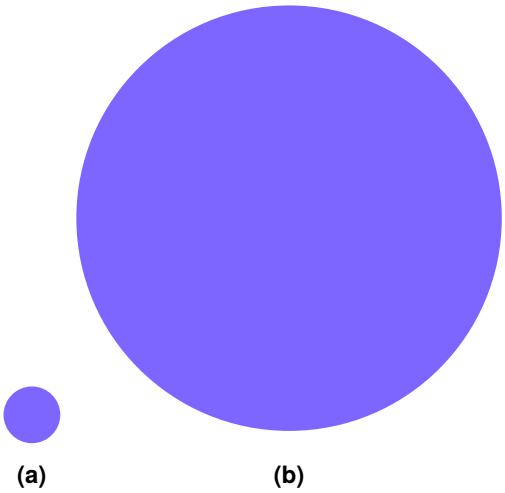


Figure 2.6: SVG documents can be scaled freely without any artifacts occurring. This figure displays the visual representation of the SVG document in Listing 2.1 scaled to different sizes. [Image created by the author of this thesis.]

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 which 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 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 which were already described in Section 2.2. Unfortunately it is not possible to style every SVG attribute with CSS because only the so-called presentation attributes like `fill` and `stroke-width` are available in CSS. These presentation attributes are taxonomically listed in the SVG specification [Dahlström et al. 2011] and will be extended by additional attributes like `x`, `y`, `width` and `height` in upcoming releases [Bellamy-Royds et al. 2018].

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
<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 which render circles relative to the canvas size. The visual representation of this document is shown in Figure 2.7.

2.5.3 Canvas (2D)

The canvas element has been introduced in HTML5 [Hickson et al. 2021] and is used to define a two-dimensional, rectangular region in a document which 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 possibilities for accessibility.

Rendering is programmed 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 is created for canvases which have the `2d` context mode set on them. It enables platform-independent rendering via a software renderer, whose API is standardized directly in the canvas specification. 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 2.7.

2.5.4 Canvas (WebGL)

A three-dimensional render context is created for canvas elements which have their context mode set to `webgl` . Three-dimensional rendering is even faster than two-dimensional rendering via the platform-independent software renderer provided for two-dimensional render contexts. The reason for this is that three-dimensional rendering is performed via a hardware-accelerated WebGL renderer. Even though the API of the WebGL renderer is also standardized [Jackson and Gilbert 2021], the availability of individual features depends on the client's hardware. It is therefore recommendable to only use three-dimensional rendering when the requirements do not allow for any of the alternatives.

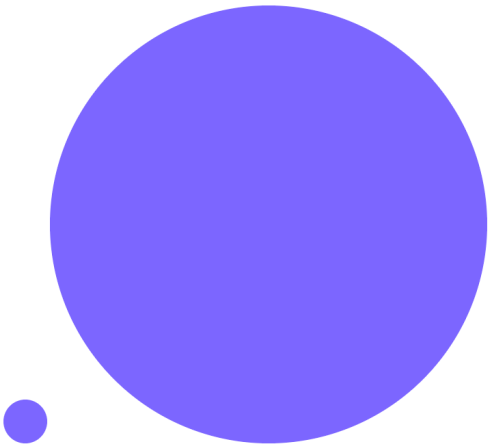


Figure 2.7: Responsive rendering of graphics inside canvas elements has to be implemented manually by calculating everything as ratios of the canvas' dimensions. This figure represents the visual representation of the canvas example in Listing 2.2. [Image created by the author of this thesis.]

2.6 Layout Engines

A layout engine is used to calculate the boundary coordinates of visual components based on some form of input components which are annotated with layout constraints. These layout constraints describe the size and position of components and their relationships between each other in a syntax which 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 which are 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 [Apple Inc. 2021], Blink [Chromium Project 2021] and Gecko [Sikorski and Peters 1999].

In a browser engine the layout of elements is constrained with CSS, which yields great 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 do not 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 they can be parsed, laid out and rendered by the browser engine. These replicated constructs do not have to be visible, and they could also be removed from the document after the layout has been acquired, meaning 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 which enable development of desktop applications using web technologies, this limitation forces applications into a very specific stack of technologies.

2.6.2 Yoga

Yoga [Facebook 2021d] is a layout engine which enables the computation of layouts which are constrained using the grammar defined in the Flexbox layout module (see Section 2.2.2). It is maintained by Facebook as an open-source project since 2016 [Sjölander 2016] with the goal of providing a small and high-performance library which 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 [Facebook 2021c], Litho [Facebook 2021b] and ComponentKit [Facebook 2021a].

2.6.3 FaberJS

FaberJS [FusionCharts, Inc. 2021] is a layout engine which 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 which has originally been created for CSS. In contrast to Yoga, which is used to create one-dimensional layouts using the Flexbox layout grammar, FaberJS implements a two-dimensional layout algorithm which is built on the grammar of the Grid layout module (see Section 2.2.3). This inherently two-dimensional approach to layouting is more suited to information visualization than trying to achieve two-dimensionality using a one-dimensional one. FaberJS is an open-source JavaScript project and 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 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.

2.7 Responsive Web Design

Influenced by the increasing use of mobile devices and their vastly varying screen sizes, responsive web design has established itself as the predominant way of designing web pages. The core idea of responsive web design is that instead of designing pages for different types of devices, website authors create a single design for a page which adapts to the characteristics of the consuming device. The term "Responsive Web Design" has been defined by Marcotte 2014 where the author differentiates between flexible and responsive web designs. It states that a flexible web design, which merely fluidly scales blocks of content to make them fit into the width of a browser window, is not enough to provide a good experience for users. Such designs will work well enough for similarly sized viewports to the one they were created for, but they will lead to noticeable artifacts on lower resolutions. These problems can be avoided by positioning the individual components of a page in a manner which provides them with enough space to render correctly. This can be achieved by using CSS media queries to adapt the overall layout of a page to the dimensions of the consuming device. Another crucial part of responsive web design is to support the different modes of interaction inherent to the various types of devices used to access the web. Desktop users might access a website using a mouse, mobile device users are usually interacting via touchscreens, and yet others might consume a page in a purely textual form with a screen reader which requires interaction via keyboard. It is one of the mantras of responsive web design to provide smooth and complete access to information to all users, regardless of the device they are using.

Chapter 3

Information Visualization

Information visualization is the science of representing abstract information as interactive graphics to present and analyze it more efficiently. These two goals of presenting and analyzing abstract information in a visual form are built on the properties of human visual perception, which include the rapid scanning, recognition and recollection of visual information as well as the automatic detection of patterns in it. In contrast to textual representations of data, the processing of well-designed visualizations requires much less cognitive effort because it leverages features of the human visual processing system. One of these features is preattentive processing, which means that certain visual attributes can be processed very quickly and without any conscious effort [Treisman 1985].

In addition to visuals being easier to assimilate by humans, a purely textual and statistical view on data can also lead to erroneous assumptions as demonstrated by Anscombe 1973 in the infamous visualization of four completely different datasets which have identical summary statistics, called Anscombe's Quartet. An observer trying to understand these sets of data purely from their statistics would mistakenly deem them to be identical. Their inequality will only become obvious after carefully examining and comparing the individual entries in the datasets themselves. This is a tedious and error-prone task when not aided by visual representations like Figure 3.1. Even though Anscombe's Quartet is very likely the most famous example to demonstrate this characteristic, it is certainly not the only set of datasets which possesses it, as has been shown by Chatterjee and Firat 2007.

This thesis adheres to the separation of the field defined by Andrews 2021. It is stated that the field of visualization is segmented into the three main subfields of information visualization, geographic visualization and scientific visualization. Furthermore, the often used term "data visualization" is defined as the intersection of geographic visualization and information visualization.

1. Information Visualization: Deals with abstract data, which has no inherent presentation and for which a suitable type of visualization has to be chosen.

2. Geographic Visualization: Deals with map-based data which has an inherent spatial dimension.

3. Scientific Visualization: Deals with object-related data which has inherent presentation, which is usually related to the object's real-world representation.

Visualizations presented in an interactive medium do not merely consist of visual representations. It is equally important to provide means for interacting with these representations to analyze more complex datasets. Without interactions, a visualization is just a static image and has only very limited use when dealing with large and multidimensional data. Even though the majority of the attention in the field of information visualization has been put on the presentational aspect of visualizations, much research has also been conducted on their interactive aspects. Many taxonomies have been formulated with the goal of defining the design space of interactions to support analytic reasoning, but they vary greatly depending on the concepts they are focusing on. Some taxonomies have been defined on the concept of

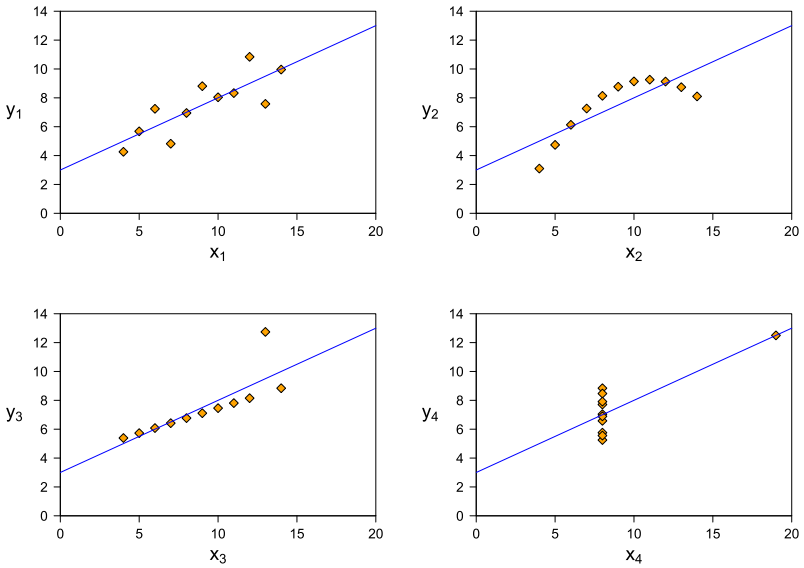


Figure 3.1: Anscombe's Quartet consists of four distinct sets of data which share identical summary statistics. The difference between these datasets is only obvious by carefully examining the textual data or by plotting it. [Image extracted from Andrews 2021. Used with kind permission by Keith Andrews.]

low-level interaction techniques [Shneiderman 1996; Wilkinson 2005], providing a very system-centric view on interaction. Other taxonomies focus on user tasks [Amar et al. 2005] which are not necessarily strongly related to interacting with visualizations. Yi et al. 2007 aims to provide a view in between the purely system-centric and purely user-centric extremes by defining a taxonomy based on user intent. The categories of this taxonomy are listed in Table 3.1 They form a good framework for the discussion of interactivity in the context of information visualization.

Category	Description	Examples
Select	Mark items as interesting.	
Explore	Show different items.	Panning, direct-walk
Reconfigure	Show a different arrangement.	Dimension configuration, position adjustments
Encode	Show a different representation.	Change chart type, orientation, colors, shapes, ...
Abstract/Elaborate	Show more or less detail.	Details-on-demand (drill-down, sunburst, tooltips, zooming)
Filter	Show items based on conditions.	Dynamic query controls
Connect	Show related items.	Highlight connected items, highlight item in different representations

Table 3.1: This table shows the categories of interacting with visualizations based on user intent. [Table adapted from Yi et al. 2007]

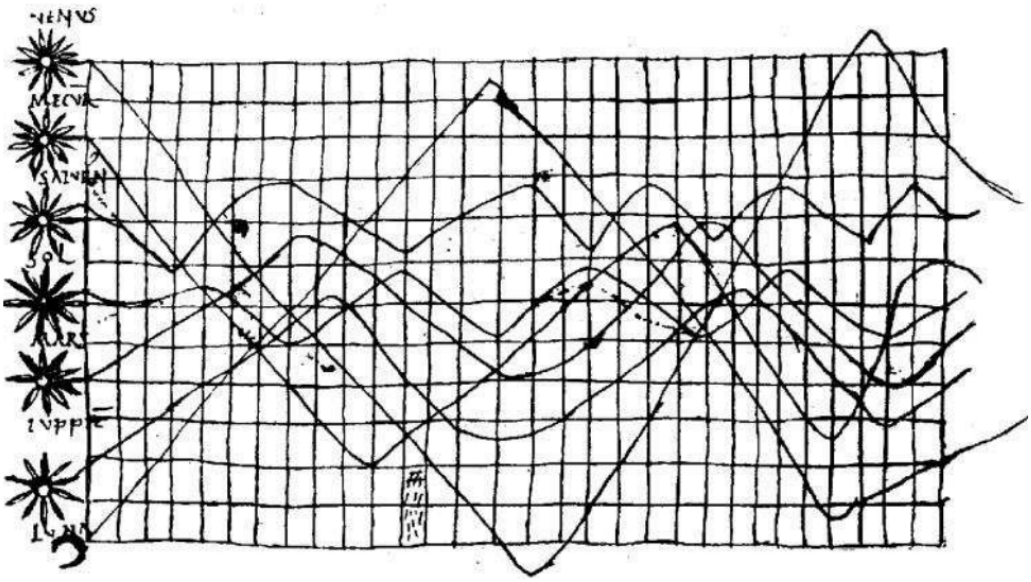


Figure 3.2: A chart created by an unknown astronomer in the tenth century depicting the movements of seven prominent planets. [Image extracted from Friendly 2008. Original appearance in Macrobius 1175.]

3.1 History of Information Visualization

The history of information visualization goes back a long time. One of its earliest examples dates back to the 10th century, when an unknown astronomer created the chart about the movement of prominent planets [Macrobius 1175] shown in Figure 3.2. Other noteworthy early visualizations include the first occurrence of the principle Tufte and Graves-Morris 1983 later coined "small multiples" in the 1626 chart demonstrating sunspot changes seen in Figure 3.3 by Scheiner 1630, and the 1644 chart displaying longitudinal distance determinations between Toledo and Rome seen in Figure 3.4 by Michael Florent van Langren.

William Playfair (1759 - 1823) is by many considered to be one of the forefathers of modern visualiza-

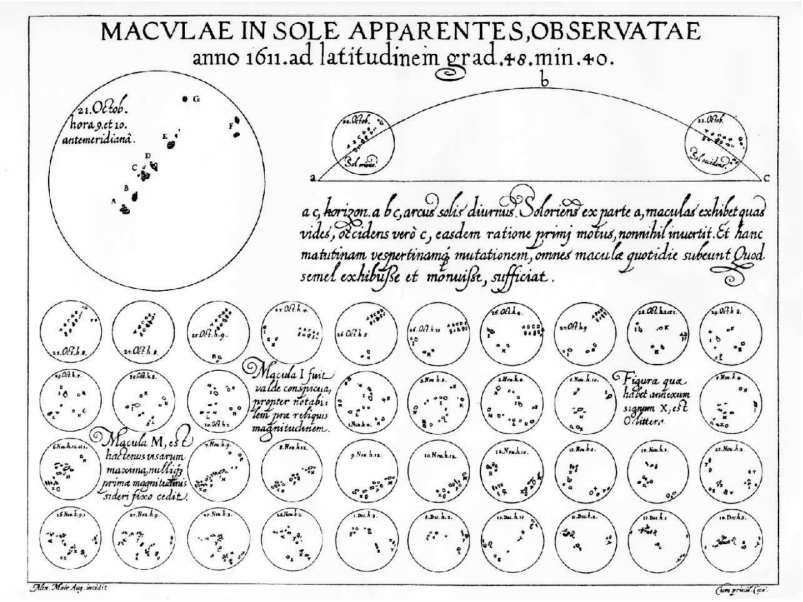


Figure 3.3: This chart shows the observed changes in sunspots based on recordings of two months of data from 1611. It is the first occurrence of the principle later called "small multiples" by Tufte and Graves-Morris 1983. [Image extracted from Friendly 2008. Original appearance in Scheiner 1630.]

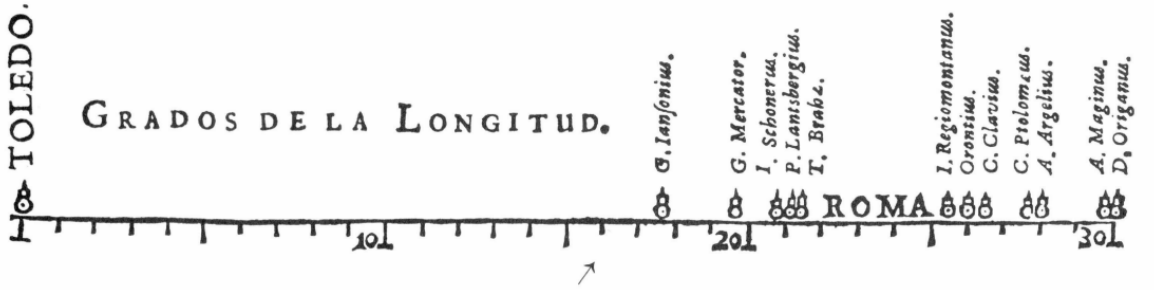


Figure 3.4: This chart compares the twelve known estimates in longitudinal distance between Rome and Toledo by various astronomers. The correct distance is marked by the arrow beneath. It is considered to be the first visual representation of statistical data. [Image extracted from Friendly 2008. Original appearance in Tufte 1997.]

tions. His published works contain the first occurrences of many graphical forms still widely used today. In one of his earlier works [Playfair 1786] he introduced the concepts of line (Figure 3.5), bar (Figure 3.6) and area charts (Figure 3.7) to communicate economic factors of England during the eighteenth century. In a related later work [Playfair 1801] he uses the first ever published pie and circle charts to show and compare the resources of states and kingdoms in Europe. The charts he created are very similar to modern ones because they already contain major concepts found in today's visualizations, such as labeled axes, grids, titles and color-based categorization.

The dot map created by Snow 1855 in 1855 to trace cholera outbreaks in London (Figure 3.8) is undoubtedly one of the most famous and influential visualizations in history. Even though it is not directly an information visualization but rather a geographic one, it has been included here because of its historic relevance. This iconic dot map was used to identify a cluster of cholera-related deaths near a contaminated water pump on Broad Street, leading to the recognition of cholera as a waterborne disease.

It would go amiss not to mention Florence Nightingale (1820 - 1910) [Cohen 1984] when talking about

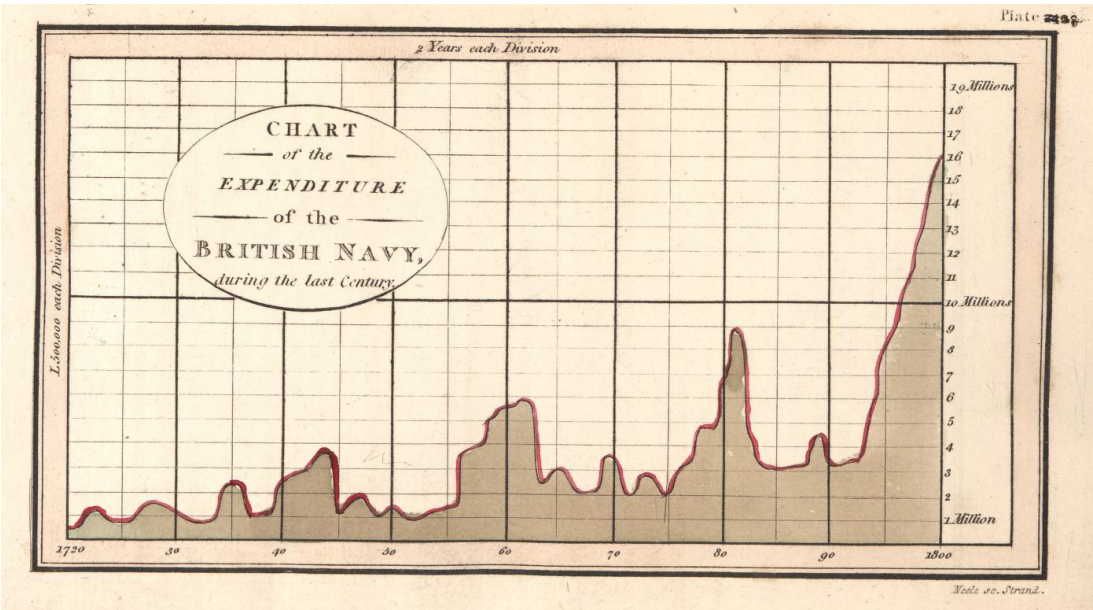


Figure 3.5: This chart shows the expenditure of the British navy during the eighteenth century as a line chart. It was published in 1786 and is considered to be one of the first occurrences of a line chart containing most components found in modern visualizations. [Image extracted from Schoenberg Center for Electronic Text and Image (SCETI). Used under the terms of Creative Commons CC BY 2.5.]

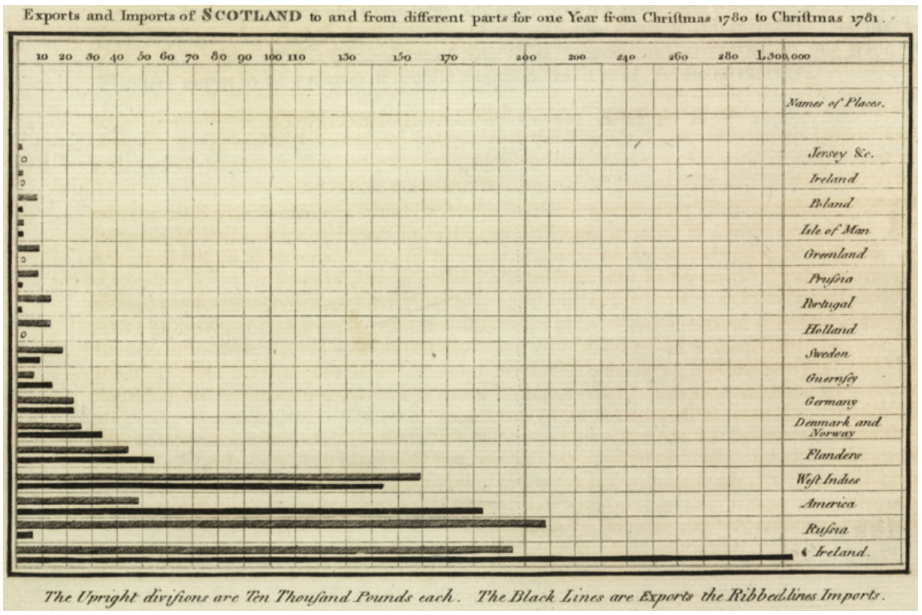


Figure 3.6: This chart shows England's exports and imports from and to Scotland in 1781 visualized as a bar chart. It was published in 1786 and is considered to be one of the first occurrences of a bar chart containing most components found in modern visualizations. [Image extracted from Schoenberg Center for Electronic Text and Image (SCETI). Used under the terms of Creative Commons CC BY 2.5.]

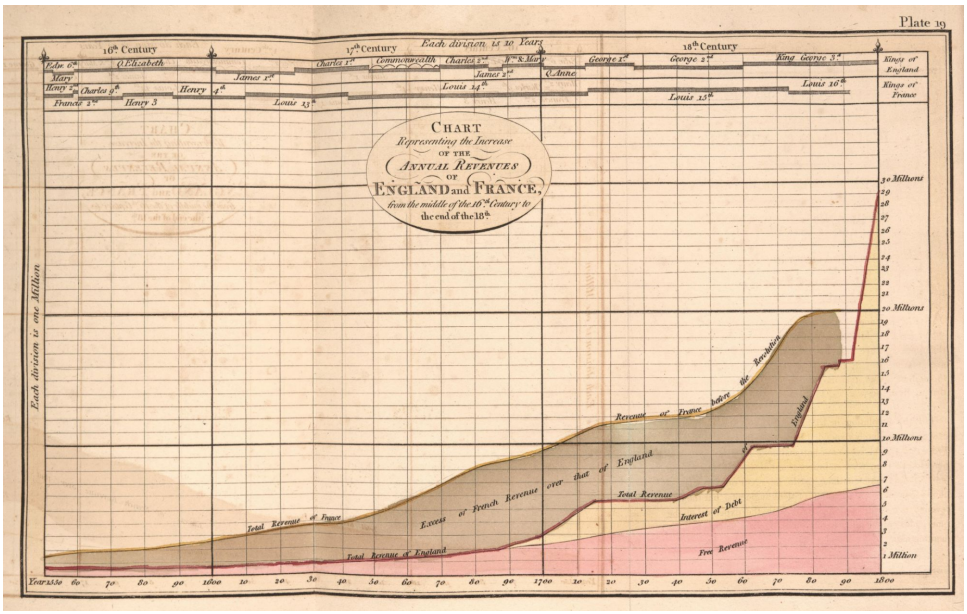


Figure 3.7: This chart shows the annual revenues of England and France between 1550 and 1800 visualized as an area chart. It was published in 1786 and is considered to be one of the first occurrences of an area chart containing most components found in modern visualizations. [Image extracted from Schoenberg Center for Electronic Text and Image (SCETI). Used under the terms of Creative Commons CC BY 2.5.]

Figure 3.8: This iconic chart was created by Dr. John Snow in 1855 to identify a clustering of cholera-related deaths near Broad Street in London. It was used to identify a contaminated water pump and to illustrate the waterborne nature of the disease. Since the data is map-based, this chart is an example of a geographic visualization rather than an information visualization. [Image extracted from Andrews 2021. Original appearance in Snow 1855.]

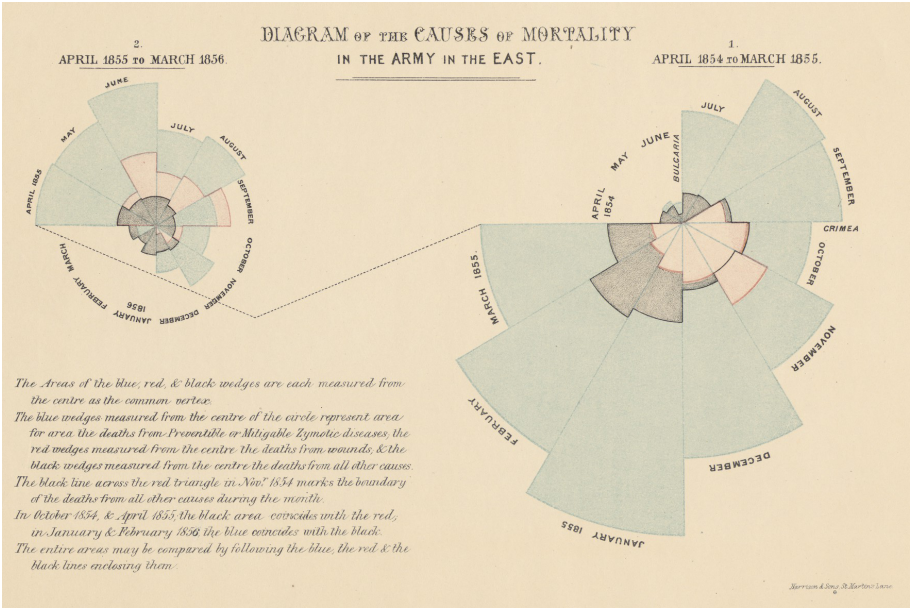


Figure 3.9: This is one of the polar-area charts that Florence Nightingale created in 1859 to convince people of a need for more sanitary conditions in hospitals. It visualizes the causes of mortality for soldiers during the Crimean War and demonstrates that a large percentage of patients died from preventable diseases which are linked to unsanitary environments. [Image extracted from Harvard Library. Used under the terms of Creative Commons Attribution 4.0.]

the history of information visualization. She was a British statistician, social reformer, founder of modern nursing and might be the first person who used visualizations to persuade others of a need for change. During her service as a superintendent of nurses in the Crimean War, she realized that a large share of deaths in hospitals resulted from preventable diseases which originated in poor sanitary conditions. One of her contributions to the field of information visualization was the creation of a new type of diagram, called a rose or polar-area chart. She used these charts to communicate data she collected on the mortality of soldiers during the war and to grab the attention of politicians and the public. One of these charts can be seen in Figure 3.9.

Modern visualizations benefit from the interactive nature of the devices used to consume them. They are allowed to be much more complex than static visualizations because various interaction techniques enable users to navigate large amounts of data and make sense of it. High-D by the company Macrofocus [Macrofocus GmbH 2021] has been chosen as a representative example of such visual analytics tools, and a screenshot of its interface during the analysis of a sample dataset can be seen in Figure 3.10.

It is out of the scope of this work to provide a full account over the long and eventful history of information visualization. This section only provides a brief and very selective view on the topic and much more comprehensive works exist which go significantly deeper into details of the various actors and the intricate influences they had on each other. Recommendable sources for further reading on the history of visualization include Friendly 2008, Friendly and Wainer 2021 and Rendgen 2019.

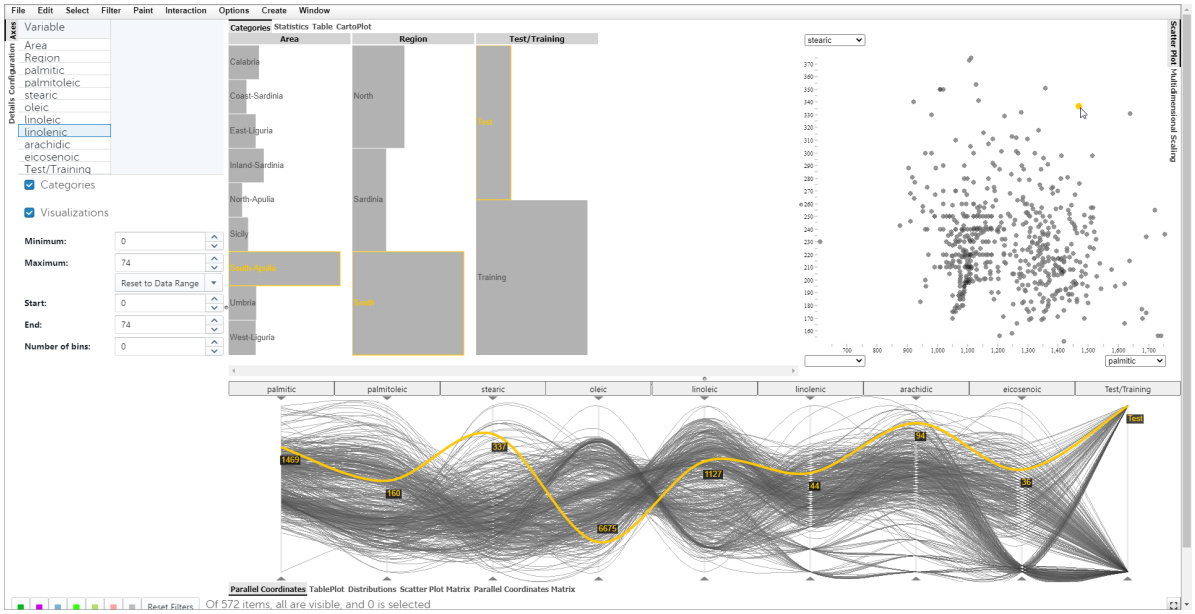


Figure 3.10: High-D from the company Macrofocus is a visual analytics tool specialized in analyzing multidimensional data. [Screenshot of Macrofocus GmbH 2021 taken by the author of this work.] [TODO: How to cite this? Copyright?]

3.2 Information Visualization Libraries for the Web

There are many web-based libraries which simplify the rendering of interactive visualizations. The approaches used to create and update visualization can be vastly different between libraries. D3 is a low-level library which enables data-driven transformations of documents, Vega and Vega-Lite provide a declarative grammar which enables the expression of visual and interactive characteristics of visualizations, and template-based visualization libraries provide a template-based interface which can easily be configured. These libraries and some of their characteristics will be summarized in the following sections.

3.2.1 Data-Driven Documents (D3)

D3 [Bostock et al. 2011] is a free and open-source document manipulation library built in JavaScript. It is maintained by an active GitHub community and its core maintainer Mike Bostock who is also the creator of Observable [Observable, Inc 2021] and the deprecated Protovis visualization library [Bostock and Heer 2009].

D3 enables data-driven document transformations to allow developers to describe documents as functions of data. As an example, developers can define transformations which receive a dataset and transform it into a basic HTML table or into a more sophisticated visualization as an SVG chart. This focus on explicitly defining transformations is better suited for dynamic visualizations because developers have complete control over the creation, modification and removal of elements. It also sets D3 apart from other visualization libraries where developers usually define the desired state of a representation using a declarative domain specific language.

In contrast to other visualization libraries, D3 contains no proprietary visual primitives and relies on well established web standards like HTML, SVG and CSS to provide visual representations. This yields great flexibility because developers work directly with web standards that are implemented by their browsers and do not need to wait for D3 to implement support for new features as standards evolve. If developers chose to switch to a different library, the knowledge of web standards they gained during their work with D3 might be applicable in their future work. The reliance on web standards also makes it

```
d3.select('body')
  .call(s => s.append('h1').text('Method Chaining in D3'))
  .call(s => s.append('p').text('This is a demonstration of method chaining'));
```

Listing 3.1: A simple example of method chaining in D3 which creates an `h1` and `p` element inside a `body`.

possible to use debugging tools which are already natively implemented in browsers.

Other important aspects of D3's design include immediate evaluation, the principle of parsimony, and support for method chaining. Immediately evaluating functions means that operations, such as modifying attributes, are applied instantaneously at the time of calling the respective functions. This achieves a reduction in internal complexity by handing control flow decisions over to invoking code. It also avoids common errors related to missing state changes when state is being modified multiple times between rendering in libraries that build on delayed evaluation. The principle of parsimony, also referred to as Occam's razor, is a problem-solving principle which stems from the field of philosophy [Sober 1979]. It is frequently paraphrased as "entities should not be multiplied beyond necessity" and when applied to API design it means that superfluous functions in an API should be avoided. As an example, the background color of a circle element can already be set with the generic `Selection.attr` method to set the `background-color` attribute of all elements in a Selection. Adding an additional `backgroundColor` method would violate the principle of parsimony because it would introduce a special method to achieve something that was already achievable. Method chaining is a popular syntax which allows functions to be chained after one another. The use of method chaining avoids having to store intermediate method results into variables which would otherwise not be needed. It is implemented in D3 by returning the `Selection` on which a modifying method is called as a result of that method. Methods that insert new elements into the DOM, such as `Selection.append` and `Selection.insert`, return a Selection of the newly added elements to enable the creation of nested structures. This method chaining syntax is further aided by the `Selection.call` method, which invokes a callback receiving the current Selection as a parameter and returns the original Selection to further chain methods on it after the callback has been executed. The `Selection.call` method enables the creation of complex method chaining structures and is widely used by developers. A simple example of method chaining in D3 and the application of the `Selection.call` method can be seen in Listing 3.1.

Selections have already been mentioned in the previous paragraphs. They are the atomic building blocks of D3 and are used to access almost any functionality. Selections are created using the `d3.select` or `d3.selectAll` methods. These methods are built on the DOM Selectors API, namely the `querySelector` and `querySelectorAll` methods, which allow the selection of elements via CSS selectors (see Section 2.2). The `d3.select` and `d3.selectAll` methods respectively create a Selection containing either a single element matching the provided selector, or multiple elements matching it. A Selection acts as a wrapper container around selected elements to perform frequently performed DOM operations on them. Among others, the element operations provided by a Selection include the setting and getting of: Attributes using the `Selection.attr` method, styles using the `Selection.style` method, properties using the `Selection.property` method, text or HTML content using the `Selection.text` or `Selection.html` methods, and event listeners using the `Selection.on` method. Selections also provide wrapper methods to insert additional elements using the `Selection.append` or `Selection.insert` methods as well as to remove them using the `Selection.remove` method. Accessing the DOM via this wrapper is less tedious than accessing it directly because the native DOM API is very verbose, and also because the method chaining API provided by D3 does not require the storage of unnecessary intermediate variables.

```
function renderCircles(container, positions) {
  container.selectAll('circle').data(positions).join(
    (enter) => enter.append('circle')
      .attr('r', '50')
      .attr('fill', 'lightgray')
      .attr('stroke', 'darkgray')
      .attr('cx', d => d.x)
      .attr('cy', d => d.y),
    (update) => update.attr('cx', d => d.x).attr('cy', d => d.y),
    (exit) => exit.remove()
  );
}
```

Listing 3.2: A demonstration of D3's general update pattern and how it can be used to specify different transformations for entering, updating and exiting elements. This code is merely meant as an example of how the join selections can be transformed. The full utility of this pattern will only be appearant in more complex scenarios that involve transitions.

An additional feature that D3 adds is the ability to bind data to elements using the `Selection.data` and `Selection.datum` methods. The `Selection.datum` method binds a single provided data record to all elements in the Selection, whereas the `Selection.data` method receives an array of data records and binds each individual data record to exactly one element. The `Selection.data` method performs a join operation between data and elements to ensure that exactly one element per data record exists. This data join results in three separate Selections: the enter Selection, containing the elements which were newly created, the update Selection, containing the elements which merely receive new data, and the exit Selection, containing the elements which are being removed. Each of these Selections can be individually transformed using the `Selection.join` method, which can receive three callbacks each being respectively invoked with the enter, update and exit Selections of the data join. This ability of individually controlling updates of entering, updating and exiting elements is being referred to as the general update pattern in D3 and a simple demonstration of how it is used can be seen in Figure 3.2. All the previously mentioned DOM wrapper methods can receive either constant values or dynamic ones, which are defined as functions. These functions receive the bound element data, the element's index in the group of nodes represented by the Selection, and the group of nodes themselves as input and calculate a dynamic value based on these parameters, which is then forwarded to the corresponding DOM methods.

D3 also offers a convenient and optional API to perform JavaScript-based animations via Transitions, which wrap a Selection and allow the animation of various element characteristics. Transitions are created using the `Selection.transition` method which creates a Transition wrapping the Selection on which it has been called. The duration of a Transition is defined using the `Transition.duration` method and its easing can be configured using the `Transition.ease` method. There is also the possibility of interrupting and chaining transitions, but explaining these things would be to go into too many details. Transitions provide an almost identical API to Selections. The major change is that the wrapping methods interpolate towards their target values using the set easing function over the set duration instead of setting the target value directly. Using D3 Transitions is completely optional and users can also choose to instead use other animation technologies, like CSS transitions and animations.

At the core, D3 is simply a low-level library to perform data-driven document transformations. Even though this generic core technology is applicable to a wide range of use cases, D3 has been created with a focus on creating visualizations. There are many additional modules which simplify performing the

higher-level tasks necessary for the rendering of visualizations. All modules follow the same patterns inherent to D3, like method chaining and configurable functions. Therefore, despite these higher-level functionalities being split on multiple modules, a consistent experience is provided to developers. Listing all available modules here would be out of the scope of this work, but some noteworthy and characteristic ones include: `d3-shape` to create visual primitives like lines and areas, `d3-scale` to encode abstract data dimensions, `d3-axis` to render scales as human-readable axes, and many more such as `d3-array`, `d3-layout` and `d3-zoom`.

3.2.2 Vega and Vega-Lite

Vega [UW Interactive Data Lab 2021] is a library which consists of a grammar to describe interactive graphics and a parser which translates specifications written in this grammar into static images or web-based views built on SVG documents or the Canvas Web API. An interactive visualization in Vega is fully described by a specification written in Vega's grammar. This grammar is essentially a domain specific language designed for the declarative denotation of interactive graphics. Its syntax is based on the easy-to-read JavaScript Object Notation (JSON), which is among the most frequently used textual serialization formats. Vega builds on previous research in the field of declarative visualization design [Wilkinson 2005]. Compared to previous works, it contains powerful capabilities to declaratively describe interactions [Satyanarayan et al. 2015] in addition to describing visual appearances.

The visual aspects of a visualization are described in a grammar similar to the one defined by Wilkinson 2005. At its top level, a Vega specification contains properties to configure sizing and padding of the container of a visualization. Every specification will also contain a data section which either defines data or specifies where to load it from. The Vega grammar also supports various forms of data transformation which can successively be applied to a dataset to perform various transformations like filtering, deriving additional fields or deriving additional datasets. In a majority of cases, the defined data will consist of abstract information that requires being mapped to visual properties. This mapping is configured and performed using scales. Vega already contains a variety of scales to help with mapping abstract values to visual properties. They can broadly be categorized into quantitative scales which map quantitative inputs to quantitative outputs, discrete scales which map discrete inputs to discrete outputs, and discretizing scales which map quantitative inputs to discrete outputs. For spatially encoded dimensions, these scales can be visualized as axes, whereas non-spatial encodings such as encodings as colors, sizes or shapes can be visualized as legends. At the core of every visualization lies the encoding of data as visual primitives, which is achieved in Vega via marks. Marks use scales to encode data fields as properties of their shapes. Based on the general update pattern of the underlying D3 library, the encoding of marks can be separately controlled for newly created (entering) marks, existing and not exiting (updating) marks, and to-be-removed (exiting) marks. In addition to these basic visualization components, the Vega grammar contains further capabilities to describe interactions (via signals, triggers and event streams), cartographic projections, sequential or layered views (via mark groups), layouts and color schemes. To demonstrate how the various aspects of a Vega specification are defined, an example which represents a static bar chart can be seen in Listing 3.3.

In template-based visualization libraries, interactions are typically defined by configuring premade interaction templates, which is easy but limiting, or by manually modifying the visualization in various callbacks, which is flexible but tedious and not serializable. The ability to describe custom interactions using a serializable, data-driven grammar is what sets Vega apart from other declarative visualization libraries. This approach offers a comparable flexibility to callback-driven interactions while still remaining fully serializable and declarative. The grammar to define interactions is based on the syntax of event-driven functional reactive programming [Wan et al. 2001], a high-level grammar which resembles mathematical equations to describe reactive systems. In Vega, the primitives to express interactions are called signals. Signals can be seen as dynamic variables which change their values based on input events or other signals. These signals and the way their values change are declaratively defined, and they can be used as

```
{
  "$schema": "https://vega.github.io/schema/vega/v5.json",
  "width": 600,
  "height": 300,
  "data": [{
    "name": "data",
    "values": [
      { "category": "A", "value": 16 },
      { "category": "B", "value": 23 },
      { "category": "C", "value": 32 }
    ]
  }],
  "scales": [
    {
      "name": "x",
      "type": "band",
      "domain": { "data": "data", "field": "category" },
      "range": "width",
      "padding": 0.05
    },
    {
      "name": "y",
      "domain": { "data": "data", "field": "value" },
      "range": "height"
    }
  ],
  "axes": [
    { "orient": "bottom", "scale": "x" },
    { "orient": "left", "scale": "y" }
  ],
  "marks": [{
    "type": "rect",
    "from": { "data": "data" },
    "encode": {
      "enter": {
        "x": { "scale": "x", "field": "category" },
        "width": { "scale": "x", "band": 1 },
        "y": { "scale": "y", "field": "value" },
        "y2": { "scale": "y", "value": 0 }
      },
      "update": { "fill": { "value": "green" } }
    }
  }]
}
```

Listing 3.3: The Vega specification of a static bar chart. Demonstrates the principle of data, scales, axes and marks.

```
{
  "...": "...",
  "signals": [{
    "name": "tooltip",
    "value": {},
    "on": [
      { "events": "rect:mouseover", "update": "datum" },
      { "events": "rect:mouseout", "update": "{}" }
    ]
  }],
  "marks": [
    { "...": "..." },
    {
      "type": "text",
      "encode": {
        "enter": {
          "align": { "value": "center" },
          "baseline": { "value": "bottom" }
        },
        "update": {
          "x": { "scale": "x", "signal": "tooltip.category", "band": 0.5 },
          "y": { "scale": "y", "signal": "tooltip.value", "offset": -5 },
          "text": { "signal": "tooltip.value" },
          "opacity": [{ "test": "datum === tooltip", "value": 0 }, { "value": 1 }]
        }
      }
    }
  ]
}
```

Listing 3.4: The necessary additions to the static bar chart specification in Listing 3.3 to display a tooltip when hovering over bars. It demonstrates the basic functionality of signals in Vega. When the mouse hovers over a rect mark, the tooltip signal will receive the value of the rect's bound data record. The tooltip signal will be reset to an empty object when the mouse leaves the rect mark. It is then used in the newly added text mark section of the specification to define the position, text and visibility of it whenever an update occurs.

dynamic variables throughout most places in a Vega specification to change various characteristics of a visualization dynamically. An example of how the previously demonstrated static bar chart specification can be extended with signals to display a tooltip when hovering over bars can be seen in Listing 3.4.

Visualizations created with Vega closely follow their specifications and minimal assumptions are being made in the compilation process. This results in very verbose specifications because all configurations for all parts of the visualization need to be explicitly defined in them. It also means that specification authors have full control over the resulting graphics, making Vega a good base on which to build further libraries and tools. Many tools [Wongsuphasawat et al. 2015; Satyanarayan and Heer 2014; Wongsuphasawat et al. 2016] have already been built on top of Vega. The tool worth mentioning most is Vega-Lite [Satyanarayan et al. 2016]. Vega-Lite is described as a "high-level grammar of interactive graphics", which summarizes its difference to Vega fairly well. Vega-Lite is a higher-level grammar than Vega, which allows authors to write specifications for common visualizations in a much more concise form. Specifications written in Vega-Lite are compiled into Vega specifications. During compilation, the compiler automatically derives default configurations for axes, legends and scales from a set of carefully designed rules. This makes

```
{
  "$schema": "https://vega.github.io/schema/vega-lite/v5.json",
  "width": 600,
  "height": 300,
  "data": {
    "values": [
      { "category": "A", "value": 16 },
      { "category": "B", "value": 23 },
      { "category": "C", "value": 32 }
    ]
  },
  "mark": "bar",
  "encoding": {
    "x": { "field": "category", "type": "ordinal" },
    "y": { "field": "value", "type": "quantitative" },
    "tooltip": [{ "field": "value" }]
  }
}
```

Listing 3.5: This is a Vega-Lite specification of the Vega bar chart specification seen in Listing 3.3 combined with Listing 3.4.

Vega-Lite more convenient to use for quick authoring of visualizations because many details which need to be explicitly stated in a Vega specification can be omitted. In those cases where the derived default configurations are not suitable, Vega-Lite also offers the possibility to override them. Since Vega-Lite specifications are simply being compiled into Vega ones, it is a sensible choice to use Vega-Lite as a primary tool to describe visualizations, and switch to Vega for more exotic cases which are not easily achievable in Vega-Lite. To properly compare the difference between Vega and Vega-Lite specifications, a Vega-Lite version of the Vega bar chart specification seen in Listing 3.3 combined with Listing 3.4 can be seen in Listing 3.5.

3.2.3 Template-Based Visualization Libraries

Template-based visualization libraries work by providing templates for possible types of visualizations and allowing users to customize them. These types of visualization libraries are easier to use than D3 or Vega because they offer a concise form of configuration which does not require users to have detailed knowledge over the underlying rendering technology or complex, non-standardized domain specific languages. Even though these types of libraries are usually flexible enough to create a huge range of visualizations, at one point users may run into limitations. Some of these limitations can not be worked around and require the writing of custom source code which requires a deep understanding of the underlying library. This effectively eliminates the ease-of-use benefit of these types of libraries for users that run into these limitations.

For this thesis, a total of 20 template-based JavaScript visualization libraries have been collected and compared by different factors such as their rendering technology, usage popularity, open-source popularity, license, and recent development activity. In terms of rendering technology, most libraries render visualizations as either SVG documents or canvas elements, though some implement a hybrid renderer which can be configured to render as either one of them. Usage popularity has been measured by the cumulative package downloads from the npm package manager over the last twelve months. This has been deemed one of the most relevant metrics for the comparison because it reflects actual user behavior and gives an indication on how widespread a library is used in practice. The 20 libraries found

in the initial collection phase were filtered by their usage popularity and recent development activity to remove those which were not sufficiently used or maintained anymore. This filtering step yielded the following ten libraries: (1) ChartJS [*Chart.js* 2021], (2) Highcharts [Highsoft 2021], (3) ECharts [Li et al. 2018], (4) ApexCharts [Chhipa and Lagunas 2021], (5) PlotlyJS [Plotly 2021], (6) C3JS [*C3.js* 2021], (7) Chartist [Kunz 2021], (8) amCharts [amCharts 2021], (9) billboardJS [NAVER Corp. 2021], and (10) D3FC [Scott Logic 2021]. These libraries have been selected for deeper evaluation because they are heavily used and, for the most part, still actively maintained. The focus of this more detailed analysis is on summarizing these libraries based on high-level features and responsive capabilities.

Eight of those ten libraries are completely free to use without restrictions, amCharts has a free license for users who are comfortable with an attribution logo on their visualizations, and Highcharts offers a free license option for non-profit, educational and personal applications. Nine libraries implement an SVG-based renderer, two of which (ECharts, D3FC) offer alternative rendering to canvas elements for high-performance scenarios, and only ChartJS purely targets canvas-based rendering. Eight libraries are very actively maintained with most of them showing development activity within the last month. Only C3JS and Chartist do not seem to be actively maintained anymore, but they nonetheless have been included in the deeper evaluation because of their historic and thematic relevance and because they are still widely used.

Template-based visualization libraries have a strong inclination towards designing their APIs according to principles of declarative programming. APIs following these principles allow users to describe a desired state they want the underlying system to be in. This is in strong contrast to the typical imperative way of designing APIs in which users are instead given a set of tools to query and modify a system's state. The difference can be summarized in simple terms as: Using declarative APIs, users specify what state shall be achieved, whereas when using imperative APIs, users specify how a certain state is achieved. Declarative APIs are typically built on top of lower-level imperative APIs and can therefore be seen as a higher level of abstraction over them. They are popular among developers because they are expressive, easy to use and effectively encapsulate complexity which would otherwise have to be handled by users. An often overlooked disadvantage of declarative APIs is that they frequently only provide high-level access to a system and that more specific use cases might not be achievable if they can not be expressed in the domain specific language defined by the API. In many cases it makes sense to provide additional imperative APIs for users which require a lower level of access to the system to implement functionality not achievable via the declarative parts of the interface.

All evaluated libraries, except D3FC, expose declarative interfaces in the form of nested configuration objects which are used to specify the characteristics of individual visualizations. Apart from Chartist, all those libraries feature generic high-level creation functions. These functions create charts from declarative configuration objects, which allow the specification of different forms of visualization for different data dimensions. This type of interface is demonstrated by the Highcharts code seen in Listing 3.6. Generic chart creation functions seem to be correlated with the ability of dynamically changing the type of visualization. In Chartist, for instance, which provides separate chart creation functions for each type of chart, it is not possible to alter the type of chart after it has been created. Another limitation which may originate in this interface partitioning via chart types is that mixed charts which combine multiple forms of visualization in one composite visualization can not be expressed. The only library in the deeper evaluation which does not provide a high-level declarative configuration API is D3FC. The design philosophy of D3FC is based on the idea of "unboxing" D3. Even though many visualization libraries are implemented on top of D3, it is usually hidden behind public APIs which are easier to work with but do not provide the full flexibility of D3. D3FC exposes a component-based interface which closely follows design patterns frequently encountered when working with D3. These components form higher-level building blocks on which advanced visualizations can be built. They are also highly configurable and in those cases where the options for configuration are not sufficient, a decorator pattern is used to allow users to hook into the underlying D3 functionality and inject custom code into the various stages of the general update pattern at the core of D3. Code which demonstrates the usage of D3FC can be seen in Listing 3.7.

```
Highcharts.chart('container', {
  chart: { type: 'column' },
  title: { text: 'Highcharts API Demonstration' },
  xAxis: { categories: ['A', 'B', 'C', 'D', 'E'], title: { text: 'Categories ' } },
  yAxis: { type: 'linear', title: { text: 'Values' } },
  series: [{
    name: 'data',
    data: [107, 31, 635, 203, 50],
    color: 'green',
    borderColor: 'black',
  }],
});
```

Listing 3.6: This example demonstrates how a basic column (vertical bar) chart is defined using the generic chart creation API provided by Highcharts.

The only evaluated library which exposes a hybrid API with the possibility of configuring visualizations using both declarative configuration objects and manually composing higher-level visualizations from lower-level components, such as axes and series, is amCharts. Its component-based interface is still rather declarative, with most options being configurable by modifying certain properties on the components. However, modifying only the properties which require changing instead of processing a full configuration object and figuring out the necessary changes from it is less costly in terms of performance. In addition to these performance benefits, the components provide additional functions to perform operations which would not be available using a purely declarative API.

When comparing the evaluated libraries by their responsive configurability, most libraries offer similar capabilities albeit in slightly different ways. Six libraries (Highcharts, C3JS, Chartist, amCharts, billboardJS, D3FC) support the styling of elements in their created visualizations with CSS, which requires rendering as SVG documents because only document-based visualizations can be affected by CSS. The styling of visualizations with CSS is powerful because it leads to a separation of concerns and designers can make use of CSS-inherent mechanisms to configure responsive styles. Unfortunately, CSS-based styling is only of limited applicability because not all CSS properties affect SVG elements, which has already been described in Section 2.5.2. To responsively configure other visualization characteristics, such as their type, data, and layout, designers have to resort to configuration mechanisms offered by the libraries. Four libraries (Highcharts, ApexCharts, Chartist, amCharts) provide the possibility to specify rule-based responsive configurations as part of their declarative interfaces, demonstrated in the Highcharts example in Listing 3.8. These declarative rules consist of a condition part which specifies when to apply the rule, and a configuration part which specifies the configuration options which should be set when applying the rule. Even though this is a convenient form of responsive configuration, if the desired conditions can not be expressed via the provided declarative properties, designers have to fall back to more generic mechanisms which are also applicable to other libraries. The mechanisms for responsive configuration in the other libraries are more generic because they do not offer these configurations as part of their declarative interfaces. This means that developers need to trigger responsive configurations themselves by manually reconfiguring visualizations via their APIs in custom resize or media query event listeners. Nearly all libraries provide means to dynamically resize visualizations and update their data, type and options. The exceptions from this are C3JS, which only supports dynamic changes of some options, and Chartist, which does not support changing of a visualization's type.

```
const data = [
  { category: 'A', value: 107 },
  { category: 'B', value: 31 },
  { category: 'C', value: 635 }
];

const bar = fc
  .autoBandwidth(fc.seriesSvgBar())
  .crossValue((d) => d.category)
  .mainValue((d) => d.value)
  .align('left')
  .decorate((selection) => {
    selection.attr('fill', 'green');
  });

const chart = fc
  .chartCartesian(d3.scaleBand(), d3.scaleLinear())
  .chartLabel('D3FC API Demonstration')
  .xDomain(data.map((d) => d.category))
  .yDomain([0, Math.max(...data.map((d) => d.value))])
  .xPadding(0.1)
  .xLabel('Categories')
  .yLabel('Values')
  .yOrient('left')
  .yNice()
  .svgPlotArea(bar);

d3.select('#container').datum(data).call(chart);
```

Listing 3.7: This example demonstrates how a basic bar chart is defined using the component-based API provided by D3FC.

```
Highcharts.chart('container', {
  ...
  responsive: {
    rules: [{
      condition: { maxWidth: 500 },
      chartOptions: {
        chart: { type: 'bar' },
        yAxis: { title: { text: null } },
        xAxis: { title: { text: null } },
      },
    }],
  },
});
```

Listing 3.8: This example demonstrates how responsive rules can be declared to configure various aspects of a visualization in relation to the size of the viewport.

Chapter 4

Responsive Information Visualization

A responsive visualization is a visualization which adapts to the properties of the device used to access it. Similar to responsive design, the need for responsive visualizations arises from the growing variety of devices used to consume content and the physical differences between them. On the web, visualizations are significant blocks of content which are embedded into documents. These blocks of content must not be ignored when a web page adhering to the principles of responsive web design is to be created. Visual elements require proper sizing and spacing to be of value. Merely scaling visualizations to fit into their allocated space is not sufficient to provide a seamless experience to users, as has already been discussed in Section 2.7. Another factor which is often ignored is the different methods of interaction inherent to specific types of devices. In addition to enabling these device-specific types of interaction, web designers need to adjust visualizations accordingly to support them as well as possible. An example for such a necessary adjustment would be to ensure that data points remain selectable on less precise input devices, such as touchscreens, by reducing the data density and increasing the size of individual elements. The goal of responsive visualizations is to alter them depending on the characteristics of the consuming device to ensure an optimal trade-off between the density of graphical elements and the messages they aim to deliver [Kim et al. 2021a].

The topic of responsive visualization only came up in recent years, with Andrews 2018 being one of the first academic works exploring it. Earlier works [Hinderman 2015; Korner 2016] exist, but they strongly focus on the aspect of software development and do not go much into detail on how to properly design responsive visualizations. For design-related fields, it is generally helpful to study existing solutions and better define the design space by creating taxonomies of currently used techniques and recurring patterns. In the case of responsive visualization, some such works already exist [Hoffswell et al. 2020; Kim et al. 2021a; Adler et al. 2021] and the various patterns defined in them, including some representative examples, are discussed in the following sections.

4.1 Responsive Visualization Patterns

Patterns are templates for solving recurring problems. The core problem to solve when designing responsive visualizations is to optimize the trade-off between the visual density of components and the messages which the visualization aims to convey [Kim et al. 2021a]. Many techniques can be applied to solve that problem by using the available screen space as efficiently as possible. Various authors have analyzed many existing visualizations to identify recurring patterns, and they formulated different taxonomies based on their results.

Adler et al. 2021 have conducted a survey under close supervision by Keith Andrews [Andrews 2018] identifying nine common patterns which reoccurred in several solutions. Slightly reworded, these patterns are: (1) rotate axis labels, (2) remove axis ticks, (3) modify strings, (4) transpose chart, (5) reposition

components, (6) zoom, (7) filter, (8) modify data density and (9) modify chart type. Compared to other works, they did not categorize the techniques they found according to multiple dimensions. Rather than that, they created a collection of specific patterns and, even though they are a good collection on which to base further research, they are not comprehensive enough to cover all the techniques which can be applied to increase the responsiveness of a chart. An example of a technique which can not be derived from these patterns is the adding and removing of components, such as in the example of a responsive line chart by Andrews 2018, in which the chart's axes were removed on narrow screens. These patterns also do not consider any adaptations of interactions, which should not be ignored when talking about responsive design.

A more comprehensive categorization of responsive techniques was created by Hoffswell et al. 2020. They state that responsive techniques can be described by a set of five actions which are applied to different components. These actions are: (1) resize, (2) reposition, (3) add, (4) modify and (5) remove. A sixth action which refers to not changing a component has also been defined in their work, but this is deemed a non-technique and therefore left out. They list a collection of eleven components on which these actions can be performed, though they do not claim this list to be exhaustive. For the sake of completeness, the components they identified are: (1) axis, (2) axis labels, (3) axis ticks, (4) gridlines, (5) legend, (6) data, (7) marks, (8) labels, (9) title, (10) view, and (11) interaction. It should be noted that some combinations of actions and components do not make sense and therefore do not occur in practice. It is, for example, not possible to resize interactions or reposition data. Hoffswell et al. 2020 performed their research following a desktop-first approach of responsive design because the interviews they conducted with visualization authors revealed a strong inclination towards this approach. They found that when adapting desktop visualizations for narrow screens, it was much more common to remove elements (37.7%) than to add them (11.3%). Another one of their findings was that most visualizations (88.7%) implemented no change at all for their interactions, while some (10%) even removed interactive capabilities completely. On the other hand, a few visualizations (5.6%) improved the experience of mobile users by adapting interactions accordingly.

The most detailed research on patterns in responsive visualization design was performed by Kim et al. 2021a. Similar to Hoffswell et al. 2020, they formulate the strategies they found in terms of the same two dimensions: targets, representing what entity is changed, and actions, representing how entities are changed. Apart from the grouping of specific targets into five distinct categories shown in Table 4.1, the major difference to the taxonomy defined by Hoffswell et al. 2020 is the increased level of detail which is put into the definition of actions. Instead of describing actions as general editing operations (resize, reposition, add, etc.), they are defined by how exactly they affect their targets. The action dimension consists of five categories which are split into further subcategories, shown in Table 4.2. These subcategories are defined as operations with distinct input and output states. This ensures that actions can be inverted, and that these patterns can be applied in both a desktop-first and a mobile-first design approach. Categorizing techniques using these dimensions, the authors identified a total of 76 viable strategies, with some of them not being used in the visualizations they studied and excluding others which are by definition not possible. Listing all these strategies here is outside the scope of this work, but the explorable online gallery by Kim et al. 2021b contains examples demonstrating all these patterns and shall be referred to for further research.

Category	Description
Data	Data is the information which is encoded in a visualization. This category includes targets such as data records, data fields, or levels of hierarchy in the data.
Encoding	Encodings are the visual forms in which data is represented.
Interaction	Interactions define how users can engage with visualizations. This category includes targets such as interaction triggers, interaction feedback and interaction features.
Narrative	This category groups targets based on the story a visualization should convey. It contains targets such as the presented sequence of information (views and states) and the information itself in the form of annotations, emphases, and texts.
References/Layout	References represent additional information which makes visualizations easier to understand, and a layout describes how the individual visual components are placed.

Table 4.1: This table shows the different target categories of responsive visualization patterns. A target of a responsive visualization pattern specifies the entity which is changed by it. [Table adapted from Kim et al. 2021a]

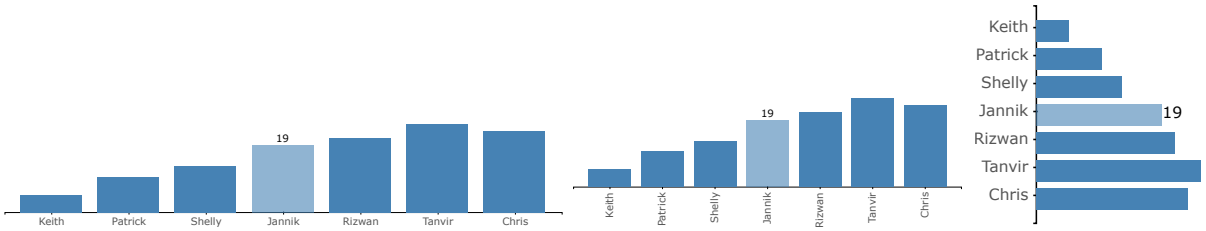
Category	Description
Recompose	Actions which affect the existence of targets. Includes remove, add, replace and aggregate actions.
Rescale	Actions which affect the size of targets. Includes reduce width, simplify labels and elaborate labels actions.
Transpose	Actions which affect the orientation of targets. Includes serialize, parallelize and axis-transpose actions.
Reposition	Actions which affect the position of targets. Includes externalize, internalize, fix, fluid and relocate actions.
Compensate	Actions which compensate for loss of information. Includes toggle and number actions.

Table 4.2: This table shows the different action categories of responsive visualization patterns. The action of a responsive visualization pattern specifies how exactly a pattern affects an entity. [Table adapted from Kim et al. 2021a]

4.2 Responsive Visualization Examples

The goal of this section is to provide the reader with some demonstrative examples of responsive visualizations. The figures in this section were taken from external scientific sources which put most of their effort into demonstrating responsive visualization patterns rather than communicating messages in the data they used. Owing to this, some figures below are lacking essential features, such as titles and axes descriptions, which would usually be present in practice. The examples in this section are organized by chart type, with each paragraph describing some responsive patterns applicable to a certain type of chart. It would be an immense endeavor to bring examples for every pattern used for all types of charts, so only a subset which demonstrates some of the most frequently encountered patterns for frequently used types of charts will be summarized here.

The first types of charts which shall be discussed are a bar charts. They are among the most often encountered types of charts, accounting for 135 (= 36%) of the 378 responsive charts studied by Kim et al. 2021a. Bar charts are usually used to visualize two-dimensional data, with one dimension being categorical and the other one being quantitative. Two additional variants of bar charts exist to visualize

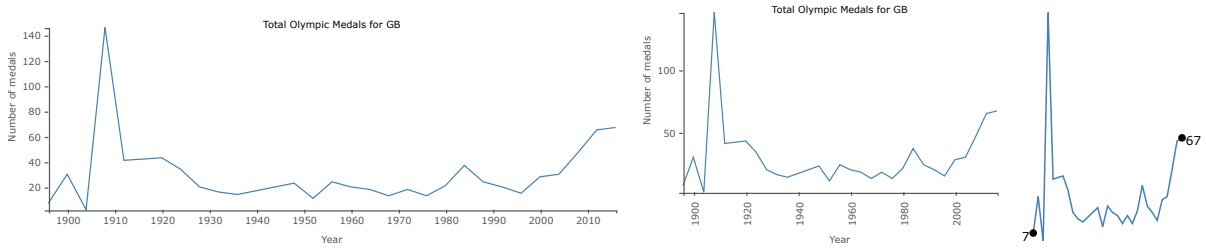


(a) 70rem **(b)** 50rem **(c)** 30rem

Figure 4.1: An example of a responsive bar chart at different display widths. (a) Axis tick labels are aligned horizontally. (b) Axis tick labels are aligned vertically. (c) Chart is transposed. [Screenshots of Andrews 2018 created by the author of this thesis. Used with kind permission by Keith Andrews.]

categorical datasets with multiple subdimensions: grouped bar charts [Ferdio ApS 2021a], to compare subdimensions with each other, and stacked bar charts [Ferdio ApS 2021b], to compare part-to-whole relationships of the subdimensions. Even though responsive design of visualizations is slowly becoming more common, most charts found in today's web articles are still being created as static images [The Learning Network 2018a; The Learning Network 2020b; Bui 2019; The Learning Network 2020a]. A good example of a responsive bar chart has been created by Andrews 2018 and can be seen in Figure 4.1. Bar charts are freely scalable by adjusting the width of individual bars [Barnett et al. 2016; Francis 2017; Minczeski et al. 2017], so they all can fit into their allocated space. When reducing the width of any type of chart past a certain point, the tick labels of the horizontal axis may start overlapping each other. This is why the reducing width pattern usually occurs together with the recompose axis ticks and simplify/elaborate axis labels patterns [Minczeski et al. 2017; Francis 2017; WSJ Graphics 2017]. Another effective pattern for solving overlapping of tick labels is to rotate them by up to 90 degrees to make them take up less horizontal space [Andrews 2018]. If there is too much data to fit into the available width, the chart can be transposed and grown to as much height as necessary [Andrews 2018]. Doing this is more advisable than simply extending the width of the chart past the viewport because vertical scrolling is easier to achieve than horizontal scrolling. When reducing the size of charts which contain annotations, similar patterns than those targeting tick labels can be applied to avoid the annotations from overlapping. Annotations can simply be removed [Bui 2021; Aisch et al. 2017], or they can be simplified and relocated [WSJ Graphics 2017].

The second most frequent types of responsive charts according to the responsive visualization gallery created by Kim et al. 2021a are line charts, which amount to 98 (= 26%) out of the 378 responsive visualizations in the gallery. Line charts are used to show trends in two-dimensional datasets by plotting them as points which are connected by lines. They can be extended to compare trends in multiple dimensions with each other by drawing additional points and lines for every additional dimension which shall be compared. Many line charts on the web are still published in non-responsive forms [The Learning Network 2019b; The Learning Network 2019a], though some web authors already took the extra effort to make their charts responsive. The minimum which can be done to make a line chart responsive is to reduce their width [Barton and Recht 2018] by shrinking the horizontal distance between neighboring points. This usually occurs together with the recomposition and simplification of horizontal ticks. If the chart contains annotations, it may also be necessary to recompose, relocate, and simplify them as well [Fessenden and Park 2016; Katz and Sanger-Katz 2021; Francis 2017; Aisch et al. 2017]. A good demonstration of which responsive patterns can be applied to make a line chart responsive is shown in the responsive line chart created by Andrews 2018 which can be seen in Figure 4.2. In addition to the recomposition of ticks, tick labels are rotated to reduce their required horizontal space. For exceptionally limited space, it can make sense to remove the axes of a line chart entirely and turn it into a sparkline. However, it should be noted that by doing this, the consumer of the visualization looses information about



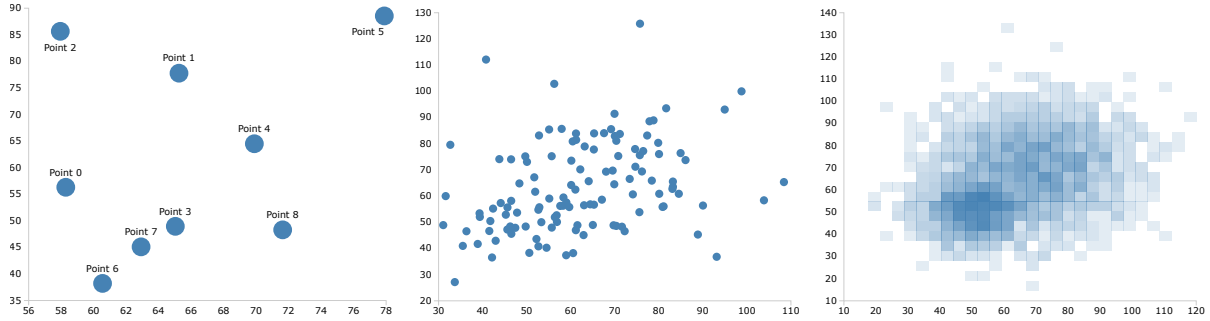
(a) 65rem **(b)** 40rem **(c)** 20rem

Figure 4.2: An example of a responsive line chart at different display widths. (a) Maximum width configuration is shown. (b) Axis ticks are thinned out and labels are rotated. (c) Axes are removed, turning the chart into a sparkline. [Screenshots of Andrews 2018 created by the author of this thesis. Used with kind permission by Keith Andrews.]

the type and scale of the chart's dimensions. This technique should therefore only be applied in cases where no other pattern is applicable or if the trend in the data is the most important message to convey. It is rare to encounter transposed versions of line charts, though the transposing could benefit heavily annotated line charts [Munroe 2021]. Applying a transpose pattern would allow the chart to take up as much vertical space as necessary to neatly fit all the annotations without requiring the consumer to scroll horizontally.

Scatterplots are also among the rather frequently encountered types of responsive charts, amounting to 26 (= 7%) of the 378 responsive charts contained in the gallery by Kim et al. 2021a. A scatterplot is a visualization which represents two-dimensional data as points in a Cartesian coordinate system. There are plenty of examples of scatterplots which are merely being published as static images [The Learning Network 2018b; The Learning Network 2018c], even though responsive versions can also already be observed occasionally. The first step to making scatterplots responsive is to reduce their width to fit them into the space available for them. As for other types of charts, care must be taken to avoid overlapping of labels and annotations by applying recomposition, relocation and simplification patterns [Canipe and Yeip 2017; Shifflett 2016]. To counteract the increased cramming of points when reducing the size of their container, various interaction features are usually implemented in scatterplots which help consumers in making sense of the represented data. The most useful interaction features in these charts are elaborative zooming interactions and the explorative panning interactions. In addition to zooming and panning, Andrews 2018 employs additional methods which reduce overlapping of individual points, such as fisheye distortion, Cartesian distortion or temporary displacements of points. An interesting technique for responsive visualizations which bases responsive configurations on a visualization's density rather than on its size has been introduced by Rabinowitz 2021. The benefit of this approach is that charts will also adapt to changing amounts of data and reconfigure their appearance accordingly. The patterns applied in the responsive scatterplot by Rabinowitz 2021 shown in Figure 4.3 are the recomposition of annotations to only show them for selected data records, and the switching of the encoding from a scatterplot to a heatmap for high densities. A good number of other techniques, such as for example the recomposition of data records, are also applicable to improve the responsiveness of scatterplots, but no examples for such patterns could be found. If the data which shall be encoded is inherently cyclic, a radial scatterplot, which is a polar coordinate system variant of a scatterplot, can be applied to better visualize this cyclic nature of the data [Barton and Recht 2018].

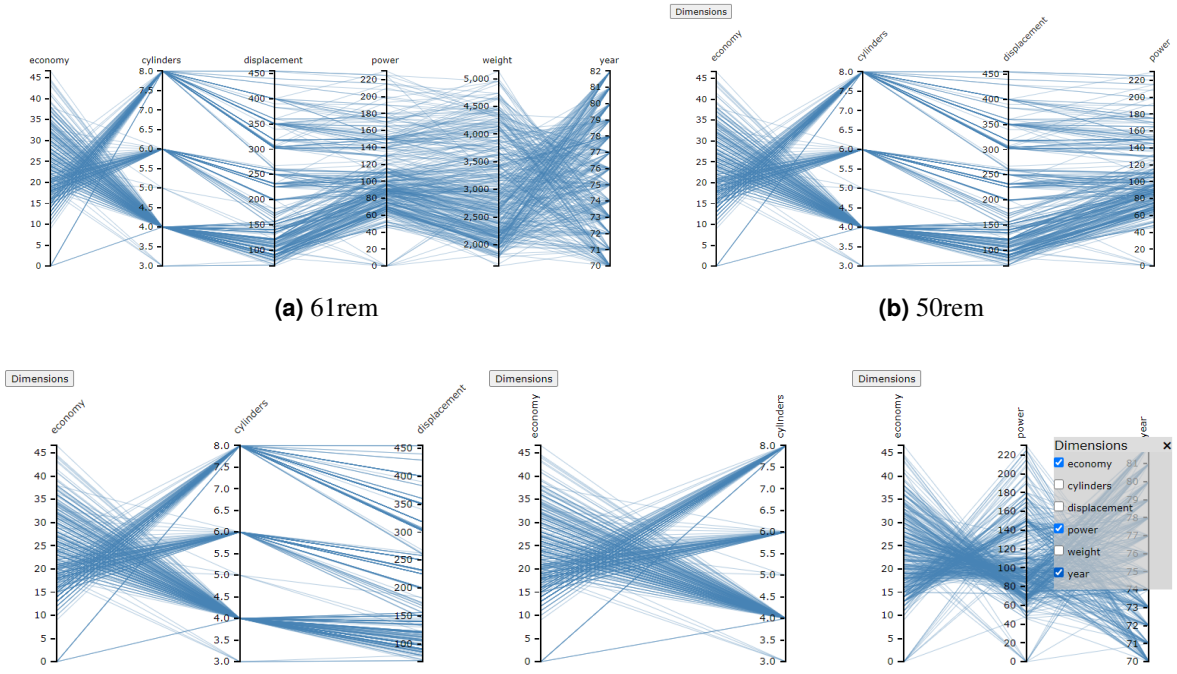
Even though parallel coordinates charts are rarely encountered in non-technical contexts, they are very popular when it comes to visualizing multidimensional data in visual analytics systems [Macrofocus GmbH 2021]. In these kinds of charts, multiple dimensions are rendered as parallel axes which are connected via paths. Each path represents an individual data record and its values in the corresponding dimensions. The axes of a parallel coordinates chart are typically laid out horizontally, meaning that



(a) 0.00005 points per pixel **(b)** 0.0007 points per pixel **(c)** 0.017 points per pixel

Figure 4.3: An example of a responsive scatterplot with increasing amount of data. The breakpoints on which the design changes are defined on the metric of data density (points per pixel) instead of on the width of the viewport as is usually the case. (a) All points and their corresponding labels are shown. (b) Point labels have been removed and are only shown for selected points. (c) The scatterplot has been replaced by a heatmap to more efficiently display the large amount of data. [Screenshots created by the author of this thesis. Visualization created by Rabinowitz 2021]

the chart can be scaled down by reducing the distance between the individual axes. It might also be beneficial to apply previously mentioned axis-related patterns, such as rotating labels and recomposing ticks. Another technique which can be applied to make parallel coordinates charts even more responsive is the successive hiding of dimensions based on their priorities. When automatically hiding dimensions, it is necessary to apply compensation patterns which give users additional controls which allow configuration of the displayed dimensions to override the system's hiding behavior. If reducing the chart's complexity is not a desirable approach, it can again be recommended to transpose the chart and expand it to whatever height necessary rather than cram too much information into the limited width available. An example of a responsive parallel coordinate chart incorporating some of these patterns can be seen in Figure 4.4.



(a) 61rem **(b)** 50rem

(c) 40rem **(d)** 30rem **(e)** 30rem with re-added dimensions

Figure 4.4: An example of a responsive parallel coordinates chart at different display widths. (a) All dimensions are shown. (b) Dimensions are removed based on their priority, dimension labels are rotated by 45 degrees, and a dimensions toggle is shown which enables the configuration of dimensions. (c) Further dimensions are removed. (d) Further dimensions are removed, and dimension labels are rotated by 90 degrees. (e) Dimension configuration panel is opened, and a dimension has been manually re-added. [Screenshots of Andrews 2018 created by the author of this thesis. Used with kind permission by Keith Andrews.]

Chapter 5

The RespVis Library

RespVis is an open-source D3 library for creating responsive SVG charts. It enables the use of CSS, which is a core pillar of designing responsive HTML documents, for the design of visualizations. Since CSS can only be applied to documents, RespVis focuses on rendering visualizations as pure and complete SVG documents, meaning that the whole visualization is contained in one SVG document that includes no elements of other XML namespaces. RespVis is designed as an extension library of D3. Unlike most other visualization libraries that are built on top of D3, RespVis does not hide it behind a custom API. Rather than that, users invoke RespVis functionalities by working directly with D3 Selections. Using D3 Selections, specially structured data is set, and visual components are rendered on elements with render functions that transform the set data into some form of visual representation. This separation between data and code and the application of strongly-typed TypeScript are the main principles guiding the software design of RespVis.

5.1 Design

The design of the RespVis library is guided by five main concepts, which are further discussed in the later paragraphs of this section:

1. CSS should be used as much as possible for describing the style and layout of visualizations.
2. Visualizations should be rendered as pure and complete SVG documents.
3. RespVis is an extension of D3 rather than a wrapper around it.
4. Data and code are treated as separate entities.
5. Every aspect of the library should be as strongly-typed as possible.

Firstly, everything that can be configured using CSS should be done so. CSS already has the capability to modify the visual appearance of elements and to lay them out. Unfortunately, CSS-based layouting does not affect SVG elements. This seriously limits responsive possibilities of styling SVG charts with CSS. Without powerful CSS layout technologies like Flexbox or Grid, all the individual components of an SVG chart would have to be positioned manually via JavaScript. To enable laying out of SVG elements with CSS, a special Layouter component has been developed which calculates positions of SVG elements via their CSS configuration and applies them. This offers visualization authors comparable configurability to what they are used to when styling HTML documents for when they are styling SVG charts. With this approach, visualization authors are not required to understand any library specific APIs and can simply apply the knowledge of CSS-based styling they most likely already possess. Since CSS can only be applied to documents, RespVis does not support rendering to HTML canvas elements because graphics rendered there are not exposed to the document and therefore can not be affected by CSS.

Secondly, every visualization should be rendered as a pure and complete SVG document. An SVG document is considered pure if it contains only elements defined in the SVG namespace. This means that it must not contain any `<foreignObject>` elements that nest elements of an XML namespace other than SVG. When an SVG document represents a visualization, it is considered to be complete if it contains all the components of the visualization within it. Different components must not be split into multiple SVG documents because they conceptually belong together and should be represented as a whole. This allows complete visualizations to be exported and stored as standard-compliant SVG files that can be further processed using the wide range of tools supporting them.

Thirdly, RespVis has been designed as a library that extends D3. Compared to other visualization libraries that are built on top of D3, RespVis does not represent a wrapping layer around it. Instead of providing an entirely new interface to consumers of the library, the core interface they interact with are D3 Selections. The typical workflow of invoking RespVis functionality is to bind data objects of a specific structure onto the elements of a Selection, and visualize this data by calling a render function that transforms it into visual marks. This design decision has been made because D3 offers powerful capabilities for the rendering of documents that would be lost when hiding it behind a custom API. By designing RespVis as an extension of D3, users can continue to leverage its expressive and concise API and design their documents using data joins and the general update pattern.

Fourthly, data and code are separated from each other. Everything in RespVis is built from basic functions and objects, without using any classes at all. Classes have been avoided because they are not common when working with D3, and also because they lead to a tight coupling between data and functionality, which has been deemed undesirable. When data and code are treated as separate entities, it results in various benefits compared to the prevalent object-oriented way of building software. Among these benefits are easier reuse and testing of functions, and less complexity in terms of effort to understand a system. Functions are easier to reuse because they only require input data of a certain shape to perform their task, and no mechanisms like inheritance or composition, which tend to dramatically increase the complexity of a system, are required. Compared to class-based code, where an object needs to be instantiated before being able to test its methods, it is easier to test functions in isolation when they are not coupled to their data. The reason for this is that the instantiation of an object might be a complex operation that depends on other methods and could affect the results of a test case. Possibly the greatest benefit of such a separation lies in the reduced complexity of the resulting system. A system that treats data and code as different entities might be composed of more entities than a system that does not, but individual entities have fewer dependencies between one another. This is because, on the highest level, entities of such a system are separated into at least two groups with no relationships between them. The research related to software complexity is difficult to convey in simple terms. However, one rule that is related to this concept of data and code separation is well summarized by “DataCodeSeparation” as: "A system made of disjoint simple parts is less complex than a system made of a single complex part." Of course, there are also various drawbacks when designing a system adhering to this concept, but they are not too severe and are therefore not listed here. For further research on this topic, readers are advised to review “DataCodeSeparation” and “OutOfTarPit”, which have acted as references for this paragraph.

Fifthly, the library is written in TypeScript and everything is as strongly-typed as possible. For the most part, interfaces are used to describe the structure of data objects and function parameters are annotated with types. Whenever working with D3 Selections, all required type contracts of a Selection are specified using the generic type variables available on them. Most of the time, it is sufficient to only specify the type of elements contained in a Selection and the structure of the data bound on them. Using these annotations, the various functions can assume that parameters passed to them are of specific types, and they do not have to worry about dynamic type checking. The application of a strongly-typed type system has many advantages like better development tooling and the compile-time identification of type-related bugs. These advantages have already been described in Section 2.4.

5.2 Project Setup

RespVis is set up as a NodeJS [OpenJS Foundation 2021] project that is hosted as an open-source project on GitHub [**RespVisGitHub**]. The implementation is written in TypeScript and grouped into different modules by thematic affinity. These TypeScript source files must be compiled to JavaScript and bundled into one combined package, so that users can import the library in their projects. To perform this compilation and bundling, the Rollup module bundler [**Rollup**] is used. In addition to the bundled JavaScript library, users are required to import an accompanying CSS file containing default styling for the generated visualizations. The project also contains examples to demonstrate usage of the library by creating various charts. These examples are HTML files that import required files and contain JavaScript that invokes RespVis functionality to create and update visualizations. The build process of the library contains multiple steps involving output directory preparation, bundling of library code and copying of various files to the correct locations in the output directory. It would be tedious to manually perform all these steps every time the library needs to be rebuilt, and therefore this process is automated using Gulp [**Gulp**], a task-based workflow automation tool. The following sections will briefly introduce the setup of RespVis and the tools used in the development process.

5.2.1 Directory Structure

The goal of this section is to give an overview over the directory structure of the RespVis project. Roughly summarized, the project contains configuration files for various tools, a `src/` directory containing the source code for the whole library and accompanying examples, a `node_modules` directory containing the project's cached NodeJS dependencies, and a `dist/` directory containing built versions of the library and examples ready for distribution. The configuration files are only discussed broadly here, as later sections go into more details about the setup of the various tools. A tree visualization of the whole directory structure, including all important files and directories but excluding individual source files, can be seen in Figure 5.1.

At the root directory of the RespVis project reside the necessary project configuration files for NodeJS, TypeScript and Gulp. The NodeJS configuration file, `package.json`, describes the meta-data of the NodeJS project. It is used to specify the project's dependencies to other packages and is required for every NodeJS project so that it can be uploaded to the npm package registry [**npm**]. The TypeScript configuration file, `tsconfig.json`, specifies the configuration the TypeScript compiler uses to compile the libraries' TypeScript source files into their JavaScript counterparts. The Gulp configuration file, `gulpfile.js`, is used to describe atomic, recurring tasks and compositions of them. These tasks can then be invoked via the Gulp command line tool to automate otherwise tedious workflow processes.

The `src/` directory at the root of the project contains all the implementation files of the library in the `src/lib/` directory and examples in the `src/examples/` directory. The `src/lib/` directory contains all TypeScript source files of the library. They have been partitioned into modules formed around thematic affinity of the various components. The `core` module contains the core functionality of the library and is a prerequisite for all the other modules. It includes the layouter implementation, D3 selection extensions, chart base components, and assorted utility functions that simplify diverse tasks when creating visualizations. The `legend` module contains a basic legend component, which renders a color legend consisting of a title, colored rectangles and corresponding labels. The `tooltip` module contains functions to show and hide tooltips, modify tooltip contents, and position tooltips. It also contains helper functions for series components that render tooltips, to simplify data creation and rendering of those series, so that tooltip-related code does not have to be repeated in various places. The `bars` and `points` modules contain the necessary series, chart and chart window components to render bar, grouped bar, stacked bar and point visualizations. At the moment, all these modules are being built into a combined package, but there are plans to distribute them separately to allow users of the library to only import those packages they need to not unnecessarily increase their own bundle sizes with code they do not require.

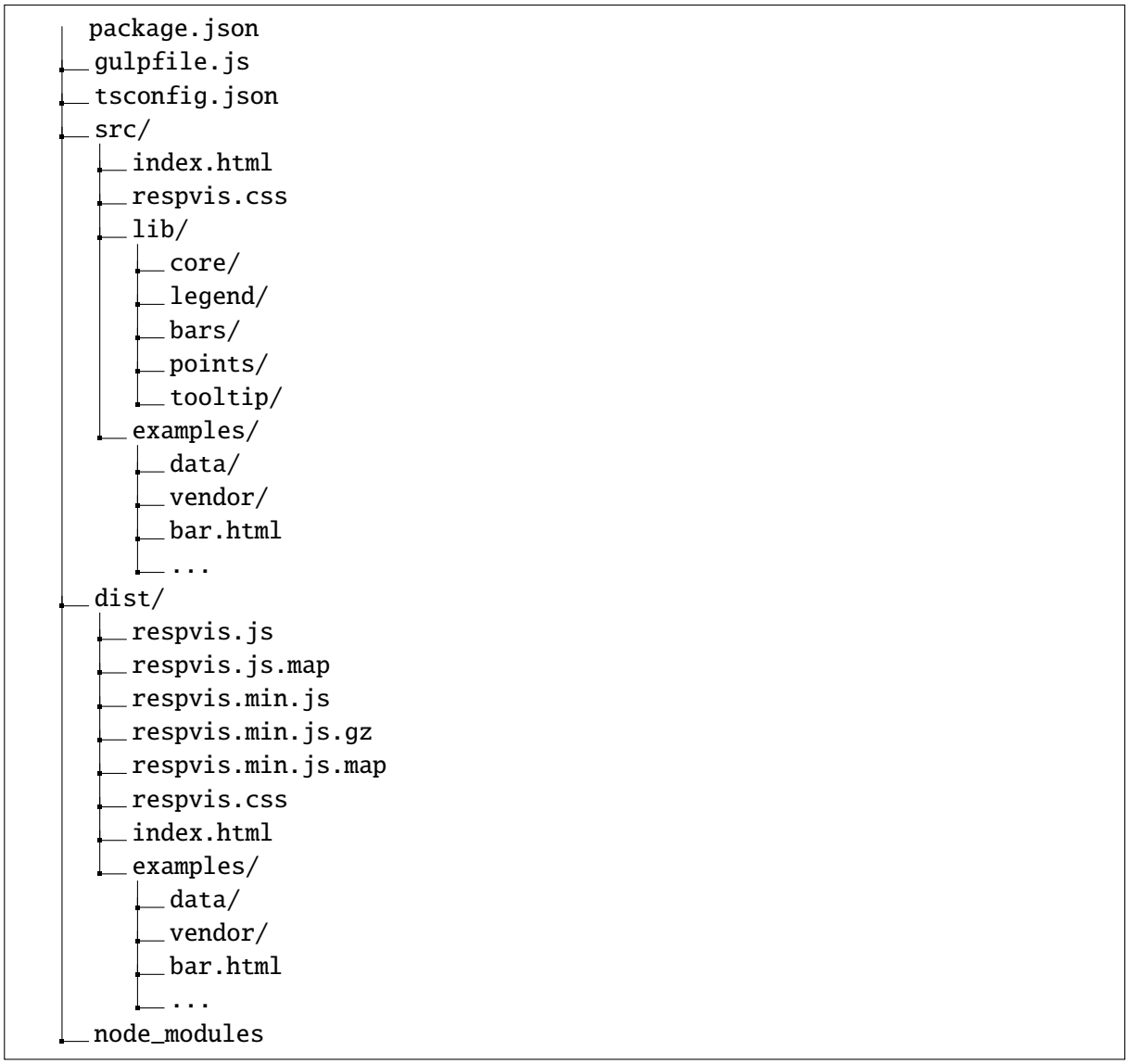


Figure 5.1: The directory structure of RespVis project. Only important files are shown here for readability reasons. [Figure created by the author of this thesis.]

Beside the `src/lib/` directory, the `src/` directory also contains the `src/examples/` directory, which holds the source files of the developed examples. These examples are distributed alongside the library files, so they are copied to the `dist/examples/` directory upon building the project. Every example consists of an HTML file that imports all the requirements such as `respvis.js` and `respvis.css` as well as external dependencies such as D3. It then invokes the necessary RespVis functionality within a `<script>` tag, which is embedded in the body of the document. In addition to the individual example files, the `examples` directory also contains a `vendor` directory, which contains third-party dependencies, and a `data` directory containing data, which is imported by individual examples to make it reusable.

In addition to configuration files and the `src/` directory, the root directory also contains two directories that are automatically generated during the build process. These are the `node_modules/` and `dist/` directories. The `node_modules/` directory is a directory that exists in every NodeJS project. It is created when installing the dependencies of a NodeJS project and contains a cached copy of every direct and indirect dependency. The `dist/` directory is generated by the Gulp build tasks and contains all the files necessary to distribute a built version of the library.

The code of RespVis is distributed as JavaScript bundles of different formats that can be used depending on the situation. Currently, all these bundles are based on Immediately Invoked Function Expressions (IIFE), which are explained in more detail in Section **??**. These bundles are also distributed in gradually more minimized versions. The `dist/respvis.js` file contains the unmodified JavaScript bundle that can be used by library consumers who require readable code, `dist/respvis.min.js` contains the minified JavaScript bundle, and `dist/respvis.min.js.gz` contains the minified JavaScript bundle that has additionally been compressed in the GZIP format [**GZIP**]. Beside these code bundles, the Rollup module bundler has been configured to create source maps for the `dist/respvis.js` and `dist/respvis.min.js` bundles: `dist/respvis.js.map` and `dist/respvis.min.js.map`. These source maps can be interpreted by developer tools in browsers to map from certain instructions in the bundled JavaScript code to the exact instruction in the original TypeScript code. They are an immense help when developing the library because, without them, debugging in browsers would be virtually impossible. Since RespVis aims to perform all possible styling in CSS, the distribution also contains a `dist/respvis.css` file which contains all the default styles of visualizations created with RespVis. Currently, this file is written manually as a whole in the `src/` directory and merely copied to the `dist/` directory during the build process. In the future, this process should be improved by employing a CSS preprocessing tool such as SASS [**SASS**] so that the CSS can be split into multiple files during development. Beside the bundled library source code and stylesheets, the `dist/` directory also contains usage examples of the library within the `dist/examples/` directory. This directory is identical to the one under `src/examples/` because it is merely being copied to the `dist/` folder during the build process.

5.2.2 NodeJS

5.2.3 Rollup

5.2.4 Gulp

Chapter 6

Modules

Chapter 7

Examples

Chapter 8

Selected Details of the Implementation

8.1 D3 Select Function Data Modification

8.2 Save as SVG

Chapter 9

Outlook and Future Work

9.1 Outlook

9.2 Ideas for Future Work

9.2.1 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]

9.2.2 Container Queries

Chapter 10

Concluding Remarks

Appendix A

User Guide

Appendix B

Developer Guide

Bibliography

Adler, Valentin, Ledio Jahaj, Markus Petritz, and Pooja Yeli [2021]. *Information Visualization Survey - Responsive Data Visualization*. Graz University of Technology, Austria. 10 May 2021. https://courses.isds.tugraz.at/ivis/surveys/ss2021/ivis-ss2021-g2-survey-resp-datavis.pdf (cited on page 33).

Aisch, Gregor, Larry Buchanan, Amanda Cox, and Kevin Quealy [2017]. *Some Colleges Have More Students From the Top 1 Percent Than the Bottom 60. Find Yours.* The New York Times. 18 Jan 2017. https://www.nytimes.com/interactive/2017/01/18/upshot/some-colleges-have-more-students-from-the-top-1-percent-than-the-bottom-60.html (cited on page 36).

Amar, Robert, James Eagan, and John Stasko [2005]. *Low-level Components of Analytic Activity in Information Visualization*. Proceedings of the 2005 IEEE Symposium on Information Visualization (InfoVis 05). IEEE, 2005, pages 111–117. doi:10.1109/INFVIS.2005.1532136 (cited on page 16).

amCharts [2021]. *amCharts*. 21 Oct 2021. https://www.amcharts.com/ (cited on page 29).

Andrews, Keith [2018]. *Responsive Visualisation*. CHI 2018 Workshop on Data Visualization on Mobile Devices (MobileVis 2018). 21 Apr 2018 (cited on pages 33–34, 36–37, 39).

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).

Andrews, Keith [2021]. *Information Visualisation - Course Notes*. Graz University of Technology, Austria. 01 Apr 2021. https://courses.isds.tugraz.at/ivis/ivis.pdf (cited on pages 15–16, 20).

Anscombe, Francis J [1973]. *Graphs in Statistical Analysis*. The American Statistician 27.1 (Feb 1973), pages 17–21. doi:10.1080/00031305.1973.10478966 (cited on page 15).

Apple Inc. [2021]. *WebKit - A fast, open-source web browser engine*. 22 Oct 2021. https://webkit.org/ (cited on page 13).

Atkins, Tab, Elika J. Etemad, and Rossen Atanassov [2018]. *CSS Flexible Box Layout Module Level 1*. W3C Candidate Recommendation. W3C, 19 Nov 2018. https://www.w3.org/TR/css-flexbox-1 (cited on pages 5–6).

Atkins, Tab, Elika J. Etemad, Rossen Atanassov, and Oriol Brufau [2020]. *CSS Grid Layout Module Level 1*. W3C Candidate Recommendation. W3C, 18 Dec 2020. https://www.w3.org/TR/css-grid-1/ (cited on pages 6–7).

Barnett, Andrew, Jason French, and Robert Wall [2016]. *Comparing the World's Fighter Jets*. The Wall Street Journal. 25 Sep 2016. http://graphics.wsj.com/how-the-worlds-best-fighter-jets-measure-up (cited on page 36).

Barton, Susanne and Hannah Recht [2018]. *The Massive Prize Luring Miners to the Stars*. Bloomberg. 08 Mar 2018. https://bloomberg.com/graphics/2018-asteroid-mining/ (cited on pages 36–37).

Bellamy-Royds, Amelia, Bogdan Brinza, Chris Lilley, Dirk Schulze, David Storey, and Eric Willigers [2018]. *Scalable Vector Graphics (SVG) 2*. W3C Candidate Recommendation. W3C, 04 Oct 2018. https://www.w3.org/TR/SVG2/ (cited on page 11).

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

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

Bos, Bert, Tantek Çelik, Ian Hickson, and Håkon Wium Lie [2011]. *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification*. W3C Recommendation. W3C, 07 Jun 2011. https://www.w3.org/TR/CSS2 (cited on page 4).

Bostock, Michael and Jeffrey Heer [2009]. *Protovis: A Graphical Toolkit for Visualization*. IEEE Transactions on Visualization and Computer Graphics 15.6 (23 Oct 2009), pages 1121–1128. doi:10.1109/TVCG.2009.174 (cited on page 22).

Bostock, Michael, Vadim Ogievetsky, and Jeffrey Heer [2011]. *D^3 Data-Driven Documents*. IEEE Transactions on Visualization and Computer Graphics 17.12 (03 Nov 2011), pages 2301–2309. doi:10.1109/TVCG.2011.185 (cited on page 22).

Boutell, Thomas [2003]. *Portable Network Graphics (PNG) Specification (Second Edition)*. W3C Recommendation. W3C, 10 Nov 2003. https://www.w3.org/TR/PNG/ (cited on page 10).

Bui, Quoctrung [2019]. *Three Months' Salary for an Engagement Ring? For Most People, It's More Like Two Weeks*. The New York Times. 13 Feb 2019. https://www.nytimes.com/interactive/2019/02/13/upshot/engagement-rings-cost-two-weeks-pay.html (cited on page 36).

Bui, Quoctrung [2021]. *Delta Variant Hasn't Yet Changed Many Return-to-Office Plans*. The New York Times. 23 Aug 2021. https://www.nytimes.com/2021/08/12/upshot/covid-return-to-office.html (cited on page 36).

C3.js [2021]. 21 Oct 2021. https://c3js.org/ (cited on page 29).

Canipe, Chris and Randy Yeip [2017]. *Health-Care Holdouts in the House*. The Wall Street Journal. 02 May 2017. https://www.wsj.com/graphics/house-health-care-holdouts-round-two/ (cited on page 37).

Çelik, Tantek, Elika J. Etemad, Daniel Glazman, Ian Hickson, Peter Linss, and John Williams [2018]. *Selectors Level 3*. W3C Recommendation. W3C, 06 Nov 2018. https://www.w3.org/TR/selectors-3/ (cited on page 4).

Chart.js [2021]. GitHub. 21 Oct 2021. https://github.com/chartjs/Chart.js (cited on page 29).

Chatterjee, Sangit and Aykut Firat [2007]. *Generating Data with Identical Statistics but Dissimilar Graphics: A Follow up to the Anscombe Dataset*. The American Statistician 61.3 (Aug 2007), pages 248–254. doi:10.1198/000313007X220057 (cited on page 15).

Chhipa, Juned and Brian Lagunas [2021]. *ApexCharts*. 21 Oct 2021. https://apexcharts.com/ (cited on page 29).

Chromium Project [2021]. *Blink - Rendering Engine*. 22 Oct 2021. https://www.chromium.org/blink/ (cited on page 13).

Cohen, I. Bernard [1984]. *Florence Nightingale*. Scientific American 250.3 (Mar 1984), pages 128–137. doi:10.1038/scientificamerican0384-128 (cited on page 18).

Coyier, Chris [2021]. *A Complete Guide to Flexbox*. 10 Sep 2021. https://css-tricks.com/snippets/css/a-guide-to-flexbox/ (cited on page 6).

Dahlström, Erik, Patrick Dengler, Anthony Grasso, Chris Lilley, Cameron McCormack, Doug Schepers, and Jonathan Watt [2011]. *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*. W3C Recommendation. W3C, 16 Aug 2011. https://www.w3.org/TR/SVG11/ (cited on pages 10–11).

Deakin, Neil, Ian Hickson, and David Hyatt [2009]. *CSS Flexible Box Layout Module*. W3C Working Draft. W3C, 23 Jul 2009. http://www.w3.org/TR/2009/WD-css3-flexbox-20090723/ (cited on page 5).

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

Deveria, Alexis [2021b]. *Can I use CSS Grid Layout*. 19 Aug 2021. https://caniuse.com/css-grid (cited on page 6).

Ecma International [1997]. *ECMAScript: A general purpose, cross-platform programming language*. ECMA-262. Ecma International, 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*. ECMA-262. Ecma International, Dec 2009. https://www.ecma-international.org/wp-content/uploads/ECMA-262_5th_edition_december_2009.pdf (cited on page 8).

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

Etemad, Elika J. and Tab Atkins [2021]. *CSS Cascading and Inheritance Level 3*. W3C Recommendation. W3C, 11 Feb 2021. https://www.w3.org/TR/css-cascade-3/ (cited on page 4).

Facebook [2021a]. *ComponentKit: A declarative UI framework for iOS*. Facebook Open Source. 21 Oct 2021. https://componentkit.org/ (cited on page 14).

Facebook [2021b]. *Litho: A declarative UI framework for Android*. Facebook Open Source. 21 Oct 2021. https://fblitho.com (cited on page 14).

Facebook [2021c]. *React Native*. Facebook Open Source. 21 Oct 2021. https://reactnative.dev (cited on page 14).

Facebook [2021d]. *Yoga Layout*. Facebook Open Source. 21 Oct 2021. https://yogalayout.com (cited on page 14).

Ferdio ApS [2021a]. *Grouped Bar Chart*. Data Viz Project. 22 Oct 2021. https://datavizproject.com/data-type/grouped-bar-chart/ (cited on page 36).

Ferdio ApS [2021b]. *Stacked Bar Chart*. Data Viz Project. 22 Oct 2021. https://datavizproject.com/data-type/stacked-bar-chart/ (cited on page 36).

Ferraiolo, Jon [1999]. *Scalable Vector Graphics (SVG) Specification*. W3C Working Draft. W3C, 11 Feb 1999. https://www.w3.org/TR/1999/WD-SVG-19990211/ (cited on page 10).

Fessenden, Ford and Haeyoun Park [2016]. *Chicago's Murder Problem*. The New York Times. 27 May 2016. https://www.nytimes.com/interactive/2016/05/18/us/chicago-murder-problem.html (cited on page 36).

Francis, Theo [2017]. *Why You Probably Work for a Giant Company, in 20 Charts*. The Wall Street Journal. 06 Apr 2017. https://www.wsj.com/graphics/big-companies-get-bigger/ (cited on page 36).

Friendly, Michael [2008]. *A Brief History of Data Visualization*. In: *Handbook of data visualization*. Springer, 2008, pages 15–56. ISBN 3540330364. doi:10.1007/978-3-540-33037-0_2 (cited on pages 17–18, 21).

Friendly, Michael and Howard Wainer [2021]. *A History of Data Visualization and Graphic Communication*. Harvard University Press, 27 Aug 2021. ISBN 0674975235 (cited on page 21).

FusionCharts, Inc. [2021]. *FaberJS*. GitHub. 21 Oct 2021. https://github.com/fusioncharts/faberjs (cited on page 14).

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. doi:10.1109/ICSE.2017.75 (cited on page 9).

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).

Hickson, Ian, Simon Pieters, Anne van Kesteren, Philip Jägenstedt, and Domenic Denicola [2021]. *HTML Standard*. Living Standard. WHATWG, 11 Aug 2021. https://html.spec.whatwg.org (cited on pages 3, 12).

Highsoft [2021]. *Highcharts*. 21 Oct 2021. https://www.highcharts.com/ (cited on page 29).

Hinderman, Bill [2015]. *Building Responsive Data Visualization for the Web*. Wiley, 02 Nov 2015. ISBN 1119067146 (cited on page 33).

Hoban, Luke [2012]. *Announcing TypeScript 0.8.1*. 15 Nov 2012. https://devblogs.microsoft.com/typescript/announcing-typescript-0-8-1/ (cited on page 9).

Hoffswell, Jane, Wilmot Li, and Zhicheng Liu [2020]. *Techniques for Flexible Responsive Visualization Design*. Proc. Conference on Human Factors in Computing Systems (CHI 2020) (Online). ACM, 25 Apr 2020, pages 1–13. doi:10.1145/3313831.3376777 (cited on pages 33–34).

House, Chris [2021]. *A Complete Guide to Grid*. 09 Nov 2021. https://css-tricks.com/snippets/css/complete-guide-grid/ (cited on page 7).

Jackson, Dean and Jeff Gilbert [2021]. *WebGL 2.0 Specification*. Khronos Editor's Draft. Khronos Group, 12 Aug 2021. https://www.khronos.org/registry/webgl/specs/latest/2.0/ (cited on page 12).

Katz, Josh and Margot Sanger-Katz [2021]. *'It's Huge, It's Historic, It's Unheard-of': Drug Overdose Deaths Spike*. The New York Times. 14 Jul 2021. https://www.nytimes.com/interactive/2021/07/14/upshot/drug-overdose-deaths.html (cited on page 36).

Kesteren, Anne van, Aryeh Gregor, and Ms2ger [2021]. *DOM Standard*. Living Standard. WHATWG, 02 Aug 2021. https://dom.spec.whatwg.org/ (cited on page 8).

Kim, Hyeok, Dominik Moritz, and Jessica Hullman [2021a]. *Design Patterns and Trade-Offs in Responsive Visualization for Communication*. Computer Graphics Forum 40.3 (Jun 2021), pages 459–470. doi:10.1111/cgf.14321. https://arxiv.org/pdf/2104.07724.pdf (cited on pages 33–37).

Kim, Hyeok, Dominik Moritz, and Jessica Hullman [2021b]. *Responsive Visualization Gallery*. 2021. https://mucollective.github.io/responsive-vis-gallery/ (cited on page 34).

Korner, Christoph [2016]. *Learning Responsive Data Visualization*. Packt Publishing, 23 Mar 2016. ISBN 178588378X (cited on page 33).

Kunz, Gion [2021]. *Chartist.js*. 21 Oct 2021. http://gionkunz.github.io/chartist-js/ (cited on page 29).

Li, Deqing, Honghui Mei, Yi Shen, Shuang Su, Wenli Zhang, Junting Wang, Ming Zu, and Wei Chen [2018]. *ECharts: A declarative framework for rapid construction of web-based visualization*. Visual Informatics 2.2 (17 May 2018), pages 136–146. doi:10.1016/j.visinf.2018.04.011 (cited on page 29).

Lie, Håkon Wium [1994]. *Cacading HTML Style Sheets: A Proposal*. 1994. https://www.w3.org/People/howcome/p/cascade.html (cited on page 4).

Lie, Håkon Wium and Bert Bos [1996]. *Cascading Style Sheets Level 1 (CSS 1) Specification*. W3C Recommendation. W3C, 17 Dec 1996. https://www.w3.org/TR/CSS1/ (cited on page 4).

Liu, Shanhong [2021]. *Most used programming languages among developers worldwide*. 05 Aug 2021. https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/ (cited on page 7).

Macrobius, Ambrosius Theodosius [1175]. *Commentarii in somnium Scipionis*. De Gruyter, 1175. ISBN 3598715269. doi:10.1515/9783110951899 (cited on page 17).

Macrofocus GmbH [2021]. *High-D: High Dimensionality Analytics Using Parallel Coordinates*. 21 Oct 2021. https://www.high-d.com (cited on pages 21–22, 37).

Marcotte, Ethan [2014]. *Responsive Web Design*. 2nd Edition. A Book Apart, 02 Dec 2014. ISBN 1937557189 (cited on page 14).

Meyer, Eric A. [2016]. *Grid Layout in CSS: Interface Layout for the Web*. O'Reilly Media, 18 Apr 2016. ISBN 1491930217 (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).

Minczeski, Pat, Donato Paolo Mancini, Colleen McEnaney, and Jason French [2017]. *France's New Political Class*. The Wall Street Journal. 03 Jul 2017. https://www.wsj.com/graphics/french-assembly-2017/ (cited on page 36).

Mogilevsky, Alex, Phil Cupp, Markus Mielke, and Daniel Glazman [2011]. *Grid Layout*. W3C Working Draft. W3C, 07 Apr 2011. https://www.w3.org/TR/2011/WD-css3-grid-layout-20110407/ (cited on page 6).

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).

Munroe, Randall [2021]. *Earth Temperature Timeline*. xkcd. 21 Oct 2021. https://xkcd.com/1732/ (cited on page 37).

NAVER Corp. [2021]. *bilboard.js*. 21 Oct 2021. https://naver.github.io/billboard.js/ (cited on page 29).

Netscape Communications Corporation [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 7).

Observable, Inc [2021]. *Observable: Explore, analyze, and explain data. As a team.* 21 Oct 2021. https://observablehq.com/ (cited on page 22).

OpenJS Foundation [2021]. *Node.js*. 21 Oct 2021. https://nodejs.org (cited on pages 7, 43).

Playfair, William [1786]. *Commercial and Political Atlas: Representing, by Copper-Plate Charts, the Progress of the Commerce, Revenues, Expenditure, and Debts of England, during the Whole of the Eighteenth Century*. Cambridge University Press, 1786. ISBN 0521855543 (cited on page 18).

Playfair, William [1801]. *Statistical Breviary; Shewing, on a Principle Entirely New, the Resources of Every State and Kingdom in Europe*. Cambridge University Press, 1801. ISBN 0521855543 (cited on page 18).

Plotly [2021]. *Plotly JavaScript Open Source Graphing Library*. 21 Oct 2021. https://plotly.com/javascript (cited on page 29).

Rabinowitz, Nick [2021]. *Responsive Data Visualization*. GitHub. 22 Oct 2021. http://nrabinowitz.github.io/rdv/ (cited on pages 37–38).

Raggett, Dave [1997]. *HTML 3.2 Reference Specification*. W3C Recommendation. W3C, 14 Jan 1997. https://www.w3.org/TR/2018/SPSD-html32-20180315 (cited on page 3).

Rendgen, Sandra [2019]. *History of Information Graphics*. TASCHEN, 26 Jun 2019. ISBN 3836567679 (cited on page 21).

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).

Satyanarayan, Arvind and Jeffrey Heer [2014]. *Lyra: An Interactive Visualization Design Environment*. Computer Graphics Forum. Volume 33. 3. Wiley, 12 Jul 2014, pages 351–360. doi:10.1111/cgf.12391 (cited on page 27).

Satyanarayan, Arvind, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer [2016]. *Vega-Lite: A Grammar of Interactive Graphics*. IEEE Transactions on Visualization and Computer Graphics 23.1 (10 Aug 2016), pages 341–350. doi:10.1109/TVCG.2016.2599030 (cited on page 27).

Satyanarayan, Arvind, Ryan Russell, Jane Hoffswell, and Jeffrey Heer [2015]. *Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization*. IEEE Transactions on Visualization and Computer Graphics 22.1 (12 Aug 2015), pages 659–668. doi:10.1109/TVCG.2015.2467091 (cited on page 25).

Scheiner, Christoph [1630]. *Rosa Ursina sive Sol ex Admirando Facularum & Macularum Suarum Phoenomeno Varius*. 1630 (cited on pages 17–18).

Scott Logic [2021]. *D3FC*. 21 Oct 2021. https://d3fc.io/ (cited on page 29).

Shifflett, Shane [2016]. *A Divided America*. The Wall Street Journal. 09 Nov 2016. https://www.wsj.com/graphics/elections/2016/divided-america/ (cited on page 37).

Shneiderman, Ben [1996]. *The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations*. Proceedings 1996 IEEE Symposium on Visual Languages. IEEE, 1996, pages 336–336. doi:10.1109/vl.1996.545307 (cited on page 16).

Sikorski, Robert and Richard Peters [1999]. *Netscape's Gecko and You*. Science 283.5409 (19 Mar 1999), pages 1871–1872. doi:10.1126/science.283.5409.1871b (cited on page 13).

Sjölander, Emil [2016]. *Yoga: A cross-platform layout engine*. Facebook Engineering. 07 Dec 2016. https://engineering.fb.com/2016/12/07/android/yoga-a-cross-platform-layout-engine (cited on page 14).

Snow, John [1855]. *On the Mode of Communication of Cholera*. John Churchill, 1855. ISBN 9354417019 (cited on pages 18, 20).

Sober, Elliott [1979]. *The Principle of Parsimony*. The British Journal for the Philosophy of Science 32.2 (10 Dec 1979), pages 145–156. doi:10.1093/bjps/32.2.145 (cited on page 23).

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

The Learning Network [2018a]. *What's Going On in This Graph? Dec. 5, 2018*. The New York Times. 06 Dec 2018. https://www.nytimes.com/2018/11/29/learning/whats-going-on-in-this-graph-dec-5-2018.html (cited on page 36).

The Learning Network [2018b]. *What's Going On in This Graph? March 13, 2018*. The New York Times. 15 Mar 2018. https://www.nytimes.com/2018/03/08/learning/whats-going-on-in-this-graph-march-13-2018.html (cited on page 37).

The Learning Network [2018c]. *What's Going On in This Graph? Oct. 17, 2018*. The New York Times. 18 Oct 2018. https://www.nytimes.com/2018/10/16/learning/whats-going-on-in-this-graph-oct-17-2018.html (cited on page 37).

The Learning Network [2019a]. *What's Going On in This Graph? March 6, 2019*. The New York Times. 09 Mar 2019. https://www.nytimes.com/2019/09/12/learning/whats-going-on-in-this-graph-sept-18-2019.html (cited on page 36).

The Learning Network [2019b]. *What's Going On in This Graph? Sept. 18, 2019*. The New York Times. 19 Sep 2019. https://www.nytimes.com/2019/09/12/learning/whats-going-on-in-this-graph-sept-18-2019.html (cited on page 36).

The Learning Network [2020a]. *What's Going On in This Graph? North American Bird Populations*. The New York Times. 28 Feb 2020. https://www.nytimes.com/2020/01/09/learning/whats-going-on-in-this-graph-north-american-bird-populations.html (cited on page 36).

The Learning Network [2020b]. *What's Going On in This Graph? Voters by Age Group*. The New York Times. 05 Mar 2020. https://www.nytimes.com/2020/02/27/learning/whats-going-on-in-this-graph-voters-by-age-group.html (cited on page 36).

Totic, Aleks and Greg Whitworth [2020]. *Resize Observer*. W3C Working Draft. W3C, 11 Feb 2020. https://www.w3.org/TR/2020/WD-resize-observer-1-20200211/ (cited on page 9).

Treisman, Anne [1985]. *Preattentive Processing in Vision*. Computer Vision, Graphics, and Image Processing 31.2 (Aug 1985), pages 156–177. doi:10.1016/S0734-189X(85)80004-9 (cited on page 15).

Tufte, Edward R [1997]. *Visual Explanations. Cheshire, CT*. Graphics Press 40 (1997), pages 310–12 (cited on page 18).

Tufte, Edward R. and Peter R. Graves-Morris [1983]. *The Visual Display of Quantitative Information*. Volume 2. 9. Graphics press Cheshire, CT, 1983. ISBN 1930824130 (cited on pages 17–18).

UW Interactive Data Lab [2021]. *Vega – A Visualization Grammar*. 21 Oct 2021. https://vega.github.io/vega (cited on page 25).

Wan, Zhanyong, Walid Taha, and Paul Hudak [2001]. *Event-Driven FRP*. International Symposium on Practical Aspects of Declarative Languages. Springer. 20 Dec 2001, pages 155–172. ISBN 354043092X. doi:10.1007/3-540-45587-6_11 (cited on page 25).

Wilkinson, Leland [2005]. *The Grammar of Graphics*. 2nd Edition. Springer, 15 Jul 2005. ISBN 0387245448. doi:10.1007/0-387-28695-0 (cited on pages 16, 25).

Wongsuphasawat, Kanit, Dominik Moritz, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer [2015]. *Voyager: Exploratory Analysis via Faceted Browsing of Visualization Recommendations*. IEEE Transactions on Visualization and Computer Graphics 22.1 (12 Aug 2015), pages 649–658. doi:10.1109/TVCG.2015.2467191 (cited on page 27).

Wongsuphasawat, Kanit, Dominik Moritz, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer [2016]. *Towards a general-purpose query language for visualization recommendation*. Proceedings of the Workshop on Human-In-the-Loop Data Analytics. ACM Digital Library, 26 Jun 2016, pages 1–6. doi:10.1145/2939502.2939506 (cited on page 27).

Wood, Lauren, Vidur Apparao, Steve Byrne, Mike Champion, Scott Isaacs, Gavin Nicol, Jared Sorensen, Robert Sutor, and Chris Wilson [1998]. *Document Object Model Specification*. W3C Working Draft. 16 Apr 1998. https://www.w3.org/TR/1998/WD-DOM-19980416/ (cited on page 8).

WSJ Graphics [2017]. *October's Not as Bleak as Its Reputation for Stock Markets*. The Wall Street Journal. 07 Oct 2017. https://www.wsj.com/articles/octobers-not-as-bleak-as-its-reputation-for-stock-markets-1507384342 (cited on page 36).

Yi, Ji Soo, Youn ah Kang, John Stasko, and Julie A Jacko [2007]. *Toward a Deeper Understanding of the Role of Interaction in Information Visualization*. IEEE Transactions on Visualization and Computer Graphics 13.6 (05 Nov 2007), pages 1224–1231. doi:10.1109/TVCG.2007.70515 (cited on pages 16–17).