RespVis:
A Browser-Based, D3 Extension Library for Creating Responsive SVG Charts

Peter Oberrauner

RespVis:
A Browser-Based, D3 Extension Library for Creating Responsive SVG Charts

Peter Oberrauner B.Sc.

Master's Thesis

to achieve the university degree of

Master of Science

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)

Villach, 02 May 2022

RespVis:
Eine Browser-Basierte, D3 Erweiterungsbibliothek zur Erstellung von Responsiven SVG Diagrammen

Peter Oberrauner B.Sc.

Masterarbeit

für den akademischen Grad

[TODO: Should this also be Master of Science?] 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)

Villach, 02 May 2022

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

[TODO: Sign only printed version or add signature to the digital version?]

Abstract

RespVis is an open-source, browser-based library for rendering responsive information visualizations and charts as composite SVG documents, whose elements are styled and positioned via CSS. It is implemented as a NodeJS package in TypeScript and is designed as an extension of D3, thus offering the flexibility of D3's convenient document manipulation API to its users. RespVis consists of various packages containing modules to render lower-level visualization components like axes, legends, and series of graphical primitives and higher-level premade visualizations for common chart types like bar charts, line charts, and point charts (scatterplots). The modules provided by RespVis can be used to create responsive visualizations by either composing them from the lower-level components or by responsively changing the configurations of the premade, higher-level visualizations. One of the main contributions of RespVis is its custom layouter, which enables the various components of a chart to be positioned via any browser-supported CSS layout mechanism like Grid and Flexbox.

Kurzfassung

[TODO: Translate abstract into german]

Contents

List of Figures

List of Tables

List of Listings

Acknowledgements

[TODO: Write acknoledgements]

Peter Oberrauner
Villach, Austria, 02 May 2022

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

The web is an indispensable part of modern society, and with the ever-growing amounts of available data, visualizations that effectively communicate this data are an essential part of it. Motivated by the increasing variety of devices used to access the web, responsive design has long since been established as one of the core pillars of designing web content to ensure that it is easily accessible to users regardless of the characteristics of their devices. Even though most web content is already designed responsively, visualizations are often only embedded in a static form that does not or only minimally adapt to different device characteristics. The academic field of responsive visualizations is still in its early stages, but some research from various authors regarding scalable visualizations [Hinderman 2015; Körner 2016], design patterns [Andrews 2018; Hoffswell et al. 2020; Kim et al. 2021a], and tools [Hoffswell et al. 2020] was already done and continues to emerge.

Many different software libraries to create visualizations for the web exist, but they all have their shortcomings regarding usability, extensibility, and responsiveness. One of the most popular libraries for creating visualizations is D3, which provides a low-level API to transform HTML and SVG documents based on data. This document-based approach is quite powerful as it allows users to create whatever visualizations they wish for without being hindered by the limitations of a custom renderer. However, building visualizations by manually setting up their entire structure and behavior can be tedious and requires deep knowledge about D3 and the underlying rendering standard like SVG. Other visualization libraries like Vega [IDL 2021] focus on a grammar-based approach to rendering visualizations. Visualization grammars are very expressive and allow users to focus on a visualization's high-level specification, but they tend to be rather complex and are not always easy to understand. Furthermore, since the actual rendering is abstracted away, visualization authors are limited to the capabilities offered by a library's high-level API, which can lead to configurability restrictions. Yet other types of visualization libraries are based on template-based configuration, meaning that visualization authors only need to provide data in a predefined format and the library then renders predefined visualizations using this data. These template-based visualization libraries are usually easy to use, but same as with grammar-based libraries, it can be hard to extend visualizations beyond the intended configuration capabilities.

This thesis introduces RespVis, a new software library for creating presentational information visualizations for the web with a strong emphasis on responsiveness. RespVis is an open-source library [Oberrauner 2022] that has been designed as an extension of D3 and focuses on rendering visualizations as SVG documents. Its API has intentionally been kept as minimal as possible and only allows the configuration of data that affects a visualization's content or behavior. Instead, CSS is used to style and largely position a visualization's content. The main contribution of this work lies in a custom layouter that uses the browser's own layout engine to enable the positioning of SVG-based visualization components, which would otherwise be unaffected by CSS layouting. Allowing visualization authors to configure the layout of visualization components with CSS layouting mechanisms like Flexbox and Grid leads to better responsive capabilities than merely allowing them to style components with CSS. Additionally, the usage

of CSS for styling and positioning allows the application of other tools frequently used for responsive design like media queries, and it also means that styles can easily be configured and overwritten via the CSS cascade. Since RespVis mainly renders and configures visualizations using standardized web technologies such as SVG and CSS, visualization authors can work with technologies all web developers are already familiar with and do not have to learn a complex domain-specific language. Furthermore, this focus on standardized web technologies also means that visualizations can easily be extended beyond foreseen use cases, meaning that it is less likely that visualization authors are limited by restrictions of the library's API.

The first part of this thesis, Chapters 2 to 4, offers a broad view on other works in the field into which this work is embedded. Chapter 2 introduces the various web technologies onto which RespVis has been built, the different ways of embedding graphics into web pages, and the different layout engines that have been considered for laying out SVG elements. Chapter 3 gives an overview of the field of information visualization and its history. Furthermore, some of the more popular software libraries like D3, grammar-based libraries like Vega, and template-based libraries like Highcharts used to create information visualizations for the web are examined and compared regarding their capabilities to make visualizations responsive. Chapter 4 further specializes on the research around responsive visualizations. Specifically, the topic of responsive patterns is introduced, and their application is demonstrated with concrete examples from both academic and other sources.

The second part of this thesis, Chapters 5 to 8, discusses the technical details of RespVis. Chapter 5 introduces the library, its design pillars, naming conventions, and project setup. Chapter 6 describes RespVis' implementation by examining the different packages and modules into which the library has been split. This chapter also discusses the implementation and implications of the custom layouter that enables laying out the elements of SVG-based visualizations using CSS layout mechanisms. Furthermore, Chapter 7 demonstrates the usage of RespVis' modules to create responsive visualizations and explains how different responsive patterns can be applied to them. Finally, Chapter 8 gives an outlook on upcoming topics and potential future work to be done on the library.

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 for content, CSS for presentation, and JavaScript (JS) for behavior. Next, TypeScript is 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 great 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>` muddied the boundary between presentation and 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 a `<style>` element or can be defined externally and linked to using a `<link>` element. This characteristic of being able to externally describe the presentation of documents brings great flexibility because 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 published as a single, monolithic specification. Since then, the specification has been modularized into different documents [Jr. et al. 2020], 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 CSS Selectors Level 3 [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.

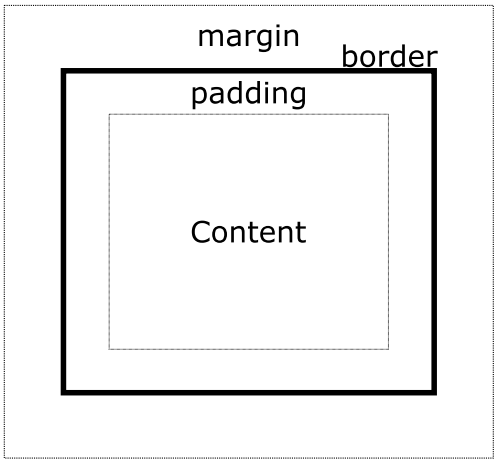


Figure 2.1: The CSS box model defines the properties of boxes which wrap around HTML elements.
[Image drawn by the author of this thesis.]

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. The border provides a visible frame around the content of a box. The padding provides 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 browsers have mature implementations of current Flexbox standards [Atkins et al. 2018], and, in most cases, fallback styling is no longer necessary.

Flexbox layouting is enabled for child elements by setting the CSS `display` property to `flex` on a container element. The direction of the layout can then be specified using the CSS `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 CSS `flex` property.

Another important feature of Flexbox layout is the controllable spacing of items, which can be specified separately for both the main axis and the cross axis of the layout. Spacing along the main axis can be configured with the CSS `justify-content` property, which can take a number of different values and is

Figure 2.2: The CSS `justify-content` property is used to distribute items along the main axis of a Flexbox container. [Image created by the author of this thesis.]

illustrated in Figure 2.2. Alignment of items on the cross axis is achieved either by the CSS `align-items` property on the container element or the CSS `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 CSS 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 CSS `display` property to `grid` on their container. The grid in which items shall be laid out is then defined using the CSS `grid-template-rows` and `grid-template-columns` properties. In addition, the CSS `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 CSS `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 CSS `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 CSS `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 CSS `align-items` and `justify-items` properties for laying out within grid cells, and the CSS `align-content` and `justify-content` properties for laying out the grid cells themselves. The latter `*-content` properties only make sense when the cells

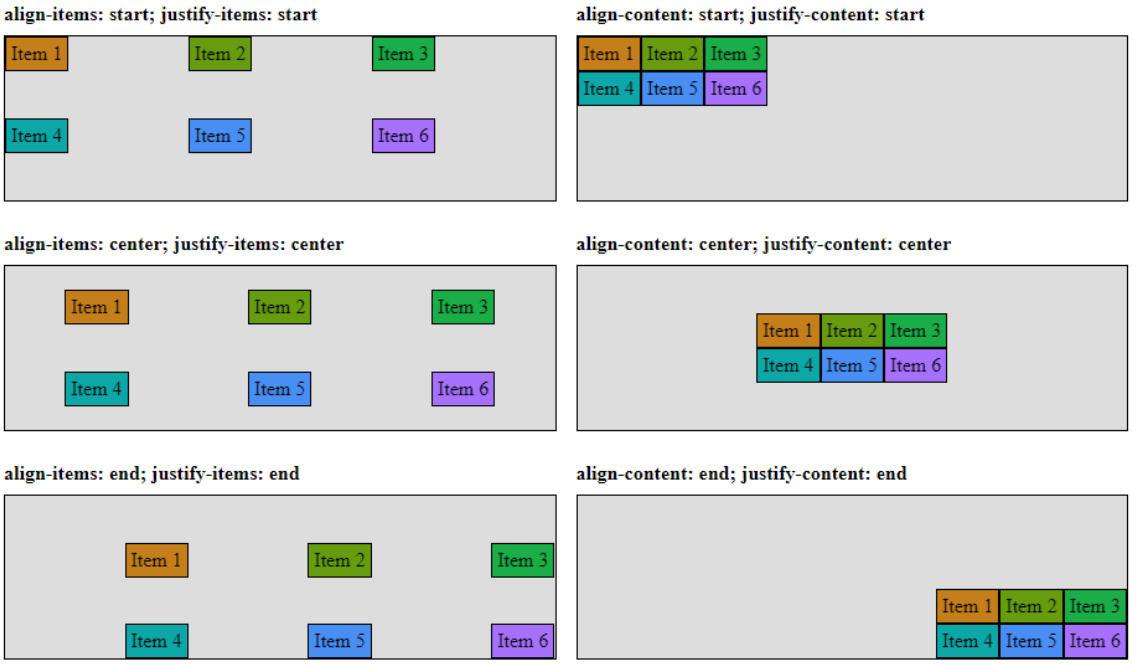


Figure 2.3: The `*-items` properties are used to lay out items within their grid cells, whereas the `*-content` properties are used to lay out the grid cells themselves. [Image created by the author of this thesis.]

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 was originally developed as a client-side scripting language run by an interpreter (engine) inside the web browser. Nowadays, there are also standalone JavaScript engines and environments like NodeJS [OpenJS 2021]. JavaScript is a multi-paradigm language which supports event-driven, as well as functional and imperative programming. Driven by the popularity of the web, JavaScript is currently the most used programming language worldwide [Liu 2021].

JavaScript was initially created by Netscape in 1995 [Netscape 1995]. Before that, websites were only able to display static content, which drastically limited the usefulness of the web. Microsoft seemingly saw JavaScript as a potentially revolutionary development, because they reverse-engineered Netscape's implementation and published their own version of the language for Internet Explorer in 1996 [Microsoft 1996]. The two implementations were noticeably different from one another and

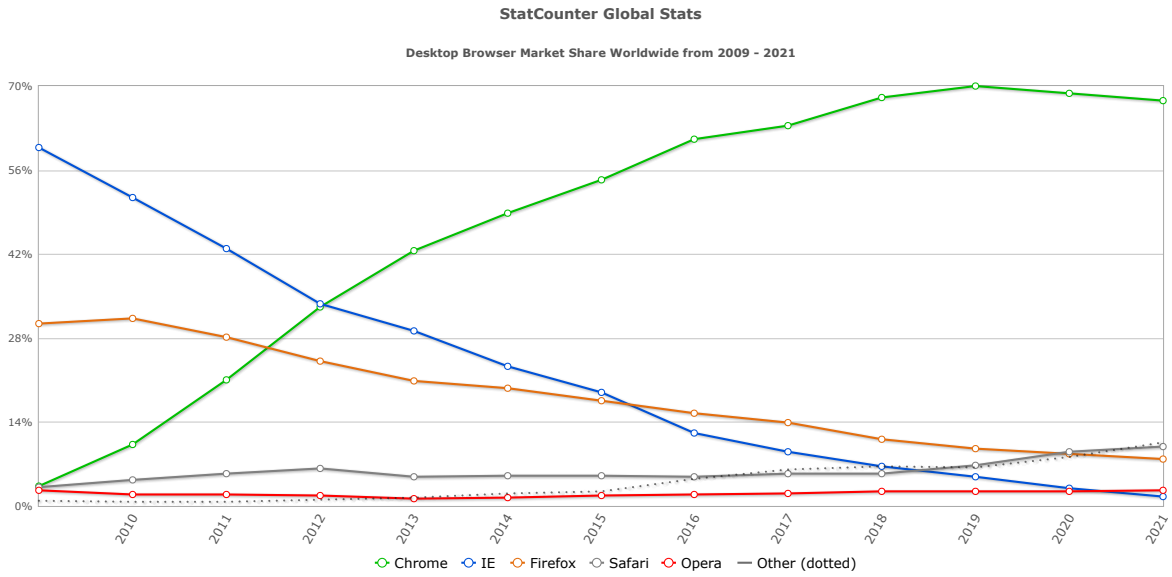


Figure 2.4: Since their release, Firefox and Chrome have contested the monopoly of the Internet Explorer and continuously gained more market share. Recently, Chrome seems to be gaining an increasingly strong position within the market. [Image taken from StatCounter [2021]]

the uncontested monopoly of the Internet Explorer [Routley 2020] held back standardization efforts undertaken by Netscape [ECMA 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], as shown in Figure 2.4. Galvanized by this new market reality, all major browser vendors collaborated on the standardization of JavaScript as ECMAScript 5 in 2009 [ECMA 2009]. Since then, JavaScript has been continuously developed and its later versions ECMAScript 2015 to 2021 [ECMA 2015; ECMA 2016; ECMA 2017; ECMA 2018; ECMA 2019; ECMA 2020; ECMA 2021] are widely supported.

RespVis is a browser-based library which is designed to run within the JavaScript engine of a browser. It builds heavily on widely supported Web APIs, which are JavaScript modules specifically meant for 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 web page or document. Internally, a document is modeled as a tree of objects, where each object corresponds to a specific HTML or SVG element in the document hierarchy and its associated data and functions. In addition to the querying of elements, the DOM also defines functionality to mutate them and their attributes, as well as functionality for handling and dispatching events. It also exposes the mechanism of MutationObservers, which are used to observe changes of attributes and children in the document tree. The initial specification of the DOM was published in 1997 [Wood et al. 1997]. It is currently 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. It provides 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 with the ResizeObserver API, which is already fully supported by all modern browsers, even though it has so far only been published as an editor's draft [Totic and Whitworth 2020].

Design Property	Description
Full erasure	Types are completely removed by the compiler, there is no type checking at runtime.
Type inference	Many types can be inferred from usage, minimizing the number of types which have to be explicitly stated.
Gradual typing	Type checking can be selectively prevented using the dynamic type any.
Structural types	Types are defined via their structure as opposed to via their names. This better fits JavaScript, where 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 thus TypeScript needs to support their typing.

Table 2.2: A summary of 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].]

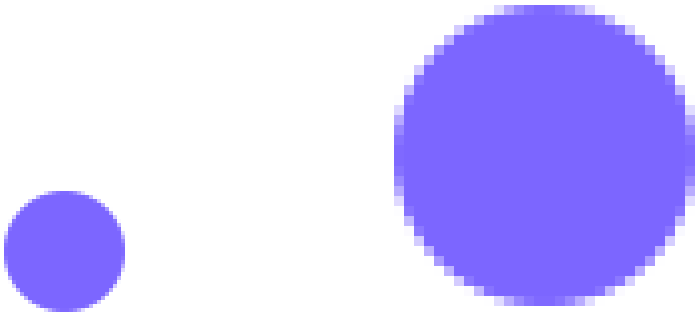
2.4 TypeScript (TS)

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 transpiler (compiler) to convert the TypeScript code into valid JavaScript code for a specific ECMAScript version.

Initially, TypeScript was 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 provided features which would later be offered by ECMAScript 6, including a module system to be able to split source code into reusable chunks and a class system to aid object-oriented development. TypeScript code using these features could then be transpiled into standard-conformant JavaScript code, which could be interpreted by JavaScript engines of the time. At the time of writing, ECMAScript 6 is widely supported by all modern browsers and therefore the main benefit of TypeScript over JavaScript lies in its provision of a static type system.

The extension of JavaScript with a static type system brings many benefits, including the improved tooling which comes with type-annotated code. Tools such as linters [OpenJS Foundation 2022b] are able to point out errors early in development and assist developers with automated fixes, improved code completion, and code navigation. Additionally, studies like Gao et al. [2017] looked at software bugs in publicly available codebases and found that 15% of them could have been prevented with static type checking.

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



(a) Intended size. **(b)** Two and a half times intended size.

Figure 2.5: A raster image of a circle. Pixelation artifacts become very apparent when a raster image is scaled to a different size. [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 many fields of application like the integration of maps, photographs, or charts in a web page. Multiple complementary technologies exist for web graphics, each with particular strengths and weaknesses depending on the use case. These technologies include pixel-based raster images, Scalable Vector Graphics (SVG), and 2d and 3d graphics through the `<canvas>` element.

2.5.1 Raster Images

A raster image represents a graphic as a rectangular, two-dimensional grid of pixels with a fixed size (resolution) in each dimension. Whenever a raster image is scaled up or down to a different size, visual artifacts become very apparent, 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 formats for raster images are JPEG [JPEG 1994] and PNG (Portable Network Graphics) [Boutell 2003]. JPEG has lossy compression, which achieves low file sizes whilst retaining reasonable image quality, and is typically used for photographs. PNG has lossless compression, which compresses well whilst preserving every original pixel as is, and also supports transparency. Both formats support progressive rendering as an image is loaded.

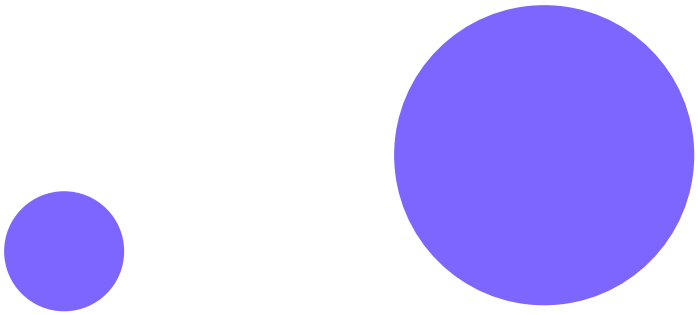
Raster images are embedded into documents in binary format. This means that the contents of the graphic are not accessible in a non-visual representation. To make the information accessible to visually impaired people, an additional textual description of the graphic's content must be provided via the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

2.5.2 Scalable Vector Graphics (SVG)

Vector graphics describe an image in terms of objects and shapes, such as lines, circles, polygons, and text. They can be scaled freely without loss of quality. Scalable Vector Graphics (SVG) is an XML-based format for vector graphics. It was initially published by the W3C in 1999 [Ferraiolo 1999], SVG 1.1 [Dahlström et al. 2011] is the latest version that is widely supported by browsers, and support for SVG 2 [Bellamy-Royds et al. 2018] is currently on its way. Graphics in an SVG file can be specified in a normalised coordinate space (inside a viewBox), enabling them to be freely scaled. Since SVG files are XML, they can be created with any text editor, but numerous tools and editors such as Inkscape [The Inkscape Project 2022] and Illustrator [Adobe 2022] exist to create or export SVG. A simple example of an SVG document containing a single circle can be seen in Listing 2.1, with its visualization shown in Figure 2.6.

```
<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.



(a) Intended size. **(b)** Two and a half times intended size.

Figure 2.6: SVG documents can be scaled freely without pixelation artifacts. Here, the SVG document from Listing 2.1 is shown. [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 to aid comprehension when consumed in a non-visual way.

SVG files are XML documents whose meta format is described in a special SVG namespace. Web browsers support mixing of HTML and SVG elements in a web page, and the SVG elements can be accessed by scripts via the DOM Web API just like HTML elements.

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, only so-called presentation attributes like `fill` and `stroke-width` are available through CSS. These presentation attributes are 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].

2.5.3 Canvas (2D)

The `<canvas>` element was introduced in HTML5 [Hickson and Hyatt 2008] and is used to define a two-dimensional, rectangular region in a document which can be drawn into by scripts. Even though rendering of dynamic graphics as `<canvas>` elements is often faster than representing them as SVG documents, their use is explicitly discouraged by the WHATWG [Hickson et al. 2021] when another suitable representation is possible. The reasons for this are that `<canvas>` elements are not compatible

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
<body>
  <canvas width="100" height="100"></canvas>
  <canvas width="250" height="250"></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.

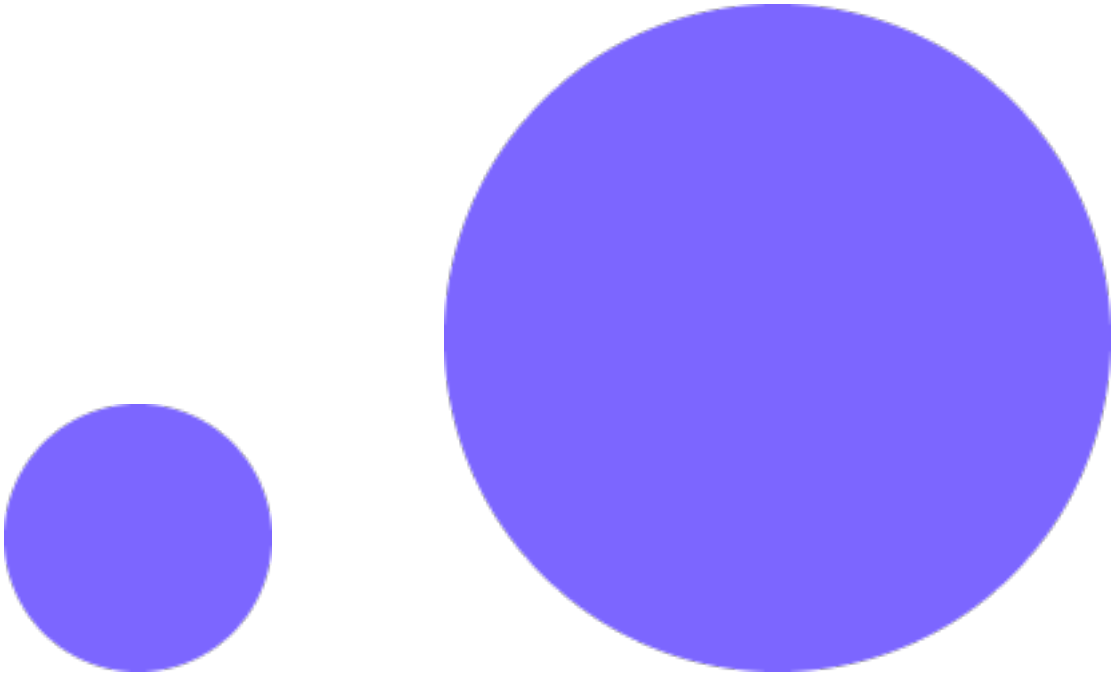
with other web technologies like CSS or the DOM Web API and because the resulting rendering provides only very limited possibilities for accessibility.

The graphics are drawn via a low-level API provided by the rendering context of a particular canvas. The two most significant rendering contexts are `2d` and `webgl`.

The `2d` rendering context enables platform-independent 2d rendering via a software renderer, whose API is standardized directly in the canvas specification [WHATWG 2021]. An example of an HTML document containing two differently sized canvases into which responsive circles are drawn using a 2d rendering context can be seen in Listing 2.2 with the corresponding visual output in Figure 2.7.

2.5.4 Canvas (WebGL)

The `webgl` rendering context enables 3d drawing through the WebGL version 1 API [Jackson and Gilbert 2014]. The `webgl2` rendering context enables 3d drawing through the WebGL version 2 API [Jackson and Gilbert 2017]. WebGL-based rendering is hardware-accelerated and often much faster than rendering via a 2d canvas or SVG. It is also possible to render 2d graphics using a WebGL render context, but the necessary setup and rendering API is rather complex.



(a) 100-pixel canvas. **(b)** 250-pixel canvas.

Figure 2.7: Responsive rendering of graphics inside <canvas> elements has to be implemented manually by calculating everything relative to its dimensions. This figure shows the visual output 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 input components annotated with layout constraints. These layout constraints describe the size and position of components and their relationships between each other in a syntax understood by the layout engine. For browser-based layout engines, the 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 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 document and any additional resources, like CSS, into a visual representation. 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 ill-defined, 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 2021], Blink [Chromium 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. A range of mechanisms is available to precisely control the layout of elements, like the Flexible Box and Grid layout modules, which 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 necessarily 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 [OpenJS Foundation 2022a] 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 constrained using the grammar defined in the CSS Flexible Box layout module (see Section 2.2.2). It has been 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. Yoga is implemented in C/C++, which works on a myriad of devices, with bindings available for other platforms like JavaScript, Android, and iOS. It has been widely 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 2021] is a layout engine very similar to the Yoga layout engine in that it enables the computation of layouts for constructs other than HTML documents, using a layout grammar originally 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 built on the grammar of the CSS Grid layout module (see Section 2.2.3). This inherently two-dimensional approach to layouting is more suited to information visualization than a one-dimensional approach. FaberJS is an open-source JavaScript project developed since 2019 by Idera. Even though the layouts it computes are constrained with the Grid Layout grammar, it only supports a subset the functionality defined in the original CSS module. Some examples of missing functionaly include missing support for margins, gaps, and the `*-content` and `grid-auto-*` properties. Working around the limitations caused by these missing features is non-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” was initially defined by Marcotte [2010] and later compiled into a book [Marcotte 2014], in which the author differentiates between flexible and responsive web designs. 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 typically interact via a touchscreen, and yet others might consume a page in a purely textual form with a screen reader and interact via a 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 seeks to use interactive graphics to assist in the analysis and presentation of abstract information. Information visualization builds on capabilities of human visual perception, including the rapid scanning, recognition, and recall of visual information, as well as the automatic detection of visual patterns. In contrast to textual representations of data, the processing of well-designed visualizations requires less cognitive effort, because it leverages features of the human visual processing system. One of these features is *preattentive processing*, whereby certain visual attributes such as colors, shapes, and sizes can be processed very quickly and without conscious effort [Treisman 1985].

In addition to visuals being easier to assimilate by humans, a purely textual and statistical view of data can also lead to erroneous assumptions. This is demonstrated in Anscombe's famous example of four completely different datasets (variables in x and y) having identical summary statistics (mean and standard deviation), called Anscombe's Quartet [Anscombe 1973], shown in Table 3.1. An observer trying to understand these datasets from their summary statistics alone might mistakenly deem them to be identical. Their inequality only becomes obvious after carefully examining and comparing the individual entries in the datasets themselves. However, the differences in the four datasets are immediately obvious when plotted graphically, as can be seen in Figure 3.1. Even though Anscombe's Quartet is likely the most famous example demonstrating this characteristic, it is certainly not the only example, as has been shown by Chatterjee and Firat [2007].

This thesis adheres to the characterization of the field of visualization as having three main subfields, as defined by Andrews [2021]:

1. *Information Visualization (InfoVis)*: Deals with abstract data, which has no inherent geometry or visual form and for which a suitable type of visualization has to be chosen.

2. *Geographic Visualization (GeoVis)*: Deals with map-based data which has inherent 2d or 3d spatial geometry.

3. *Scientific Visualization (SciVis)*: Deals with real-world objects having inherent 2d or 3d geometry, which is used as the basis for visualization.

The often-used term "Data Visualization" (*DataVis*) is defined as the combination (union) of information visualization and geographic visualization.

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 datasets. Even though the majority of attention in the field of information visualization has been directed towards the presentational aspect of visualizations, research has also been done on their interactive aspects. Numerous taxonomies have been formulated with the goal

	$\mathbf{v_1}$		$\mathbf{v_2}$		$\mathbf{v_3}$		$\mathbf{v_4}$	
	x_1	y_1	x_2	y_2	x_3	y_3	x_4	y_4
	10.00	8.04	10.00	9.14	10.00	7.46	8.00	6.58
	8.00	6.95	8.00	8.14	8.00	6.77	8.00	5.76
	13.00	7.58	13.00	8.74	13.00	12.74	8.00	7.71
	9.00	8.81	9.00	8.77	9.00	7.11	8.00	8.84
	11.00	8.33	11.00	9.26	11.00	7.81	8.00	8.47
	14.00	9.96	14.00	8.10	14.00	8.84	8.00	7.04
	6.00	7.24	6.00	6.13	6.00	6.08	8.00	5.25
	4.00	4.26	4.00	3.10	4.00	5.39	19.00	12.50
	12.00	10.84	12.00	9.13	12.00	8.15	8.00	5.56
	7.00	4.82	7.00	7.26	7.00	6.42	8.00	7.91
	5.00	5.68	5.00	4.74	5.00	5.73	8.00	6.89
mean	**9.00**	**7.50**	**9.00**	**7.50**	**9.00**	**7.50**	**9.00**	**7.50**
sd	**3.3166**	**2.0316**	**3.3166**	**2.0317**	**3.3166**	**2.0304**	**3.3166**	**2.0306**

Table 3.1: The four datasets (variables) in Anscombe's Quartet look identical if only standard summary statistics like mean and standard deviation are considered. The difference between the datasets is only apparent after careful examination of the numbers.

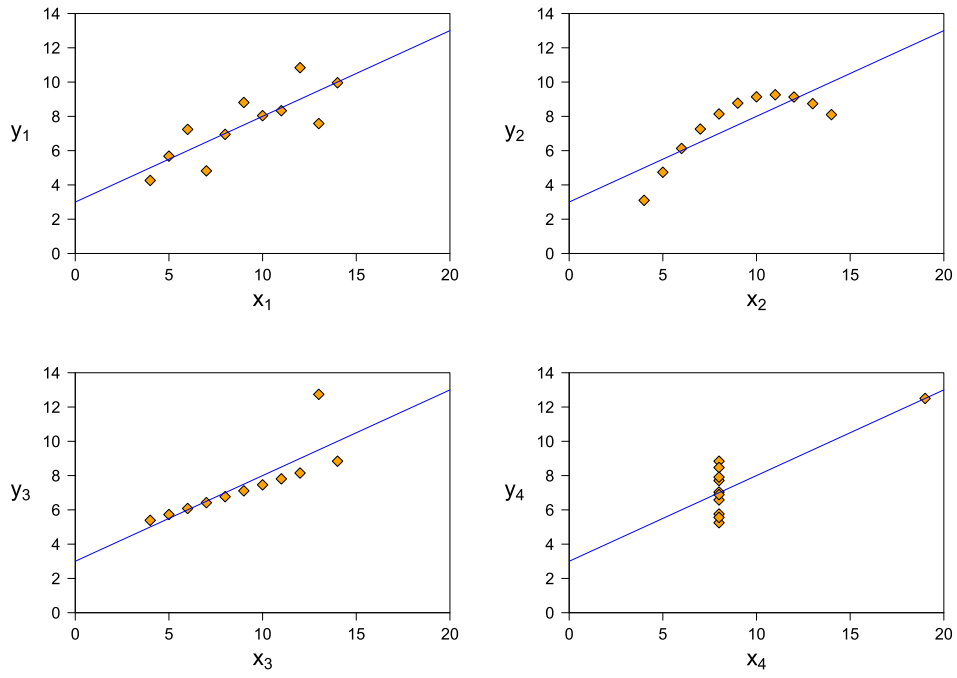


Figure 3.1: When plotted graphically, it is immediately apparent that the four datasets in Anscombe's Quartet are very different. [Image extracted from Andrews [2021]. Used with kind permission by Keith Andrews.]

Category	Description	Examples
Select	Mark something as interesting.	Highlighted selections, placemarks, assigning classes.
Explore	Show me something else.	Different subset of data, panning, direct-walk.
Reconfigure	Show me a different arrangement.	Sorting, rearranging columns, plotting different dimensions, using an alternative projection.
Encode	Show me a different representation.	Changing visual encoding (color, size, shape), or even chart type.
Abstract / Elaborate	Show me more or less detail.	Details-on-demand, drill-down and roll-up, tooltips, zooming in and out.
Filter	Show me something conditionally.	Dynamic queries, range sliders, toggle buttons, query by example.
Connect	Show me related items.	Brushing across views, highlighting connected items on mouseover.

Table 3.2: Categories of interaction with visualizations based on what a user wants to achieve (user intent). [From Yi et al. [2007]]

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 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 what a user wants to achieve, also known as *user intent*. The categories of this taxonomy are shown in Table 3.2. They provide a good framework for the discussion of interactivity in the context of information visualization.

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 added a chart showing the movements of prominent planets to the appendix of [Macrobius 0400]. Other noteworthy early visualizations include the first occurrence of the principle Tufte [1983] later called "small multiples" in Scheiner [1630]'s chart demonstrating observed changes in sunspots, and what Tufte [1997, page 15] deems to be the first visualization of statistical data in Florent [1644]'s chart displaying twelve estimates of the longitudinal distance between Toledo and Rome by various astronomers.

William Playfair (1759–1823) is considered by many to be one of the forefathers of modern information visualization. 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 charts (Figure 3.2), bar charts (Figure 3.3), and area charts (Figure 3.4) to communicate economic factors of England during the eighteenth century. In a related later work [Playfair 1801], he used 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, containing familiar concepts such as labeled axes, grids, titles, and color-coding.

It would be amiss not to mention Florence Nightingale (1820–1910) [Cohen 1984; Nightingale 1859] when talking about the history of information visualization. She was a British statistician, social reformer, and the 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 number of deaths in hospitals resulted from preventable diseases which originated

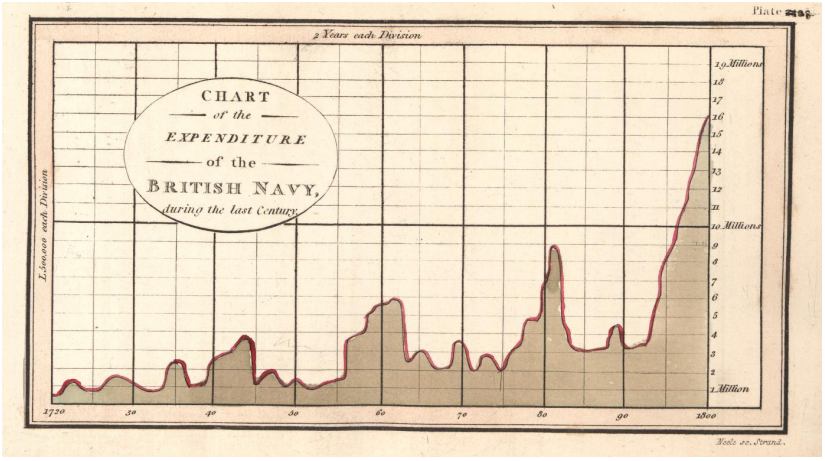


Figure 3.2: Line chart of the expenditure of the British Navy during the 18th Century. It was published in 1786 and is considered to be one of the first occurrences of a line chart containing components found in modern visualizations. [Original appearance in Playfair [1786]. Image extracted from University of Pennsylvania [2022] and used under the terms of Creative Commons CC BY 2.5.]

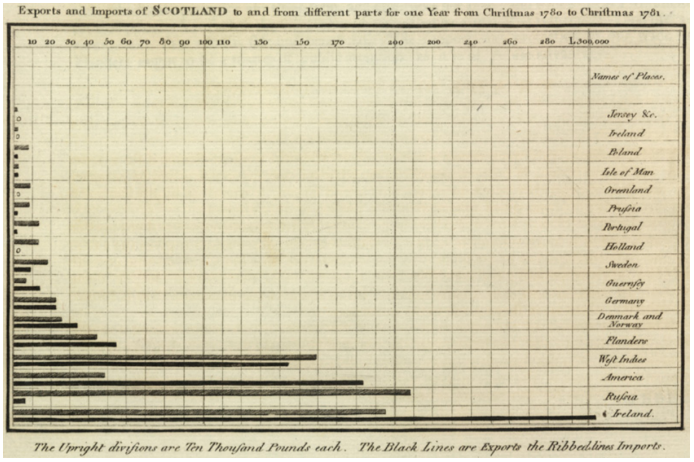


Figure 3.3: Bar chart of England's exports and imports to and from Scotland in 1781. Published in 1786, it is considered to be one of the first occurrences of a bar chart containing most components found in modern visualizations. [Original appearance in Playfair [1786]. Image extracted from University of Pennsylvania [2022] and used under the terms of Creative Commons CC BY 2.5.]

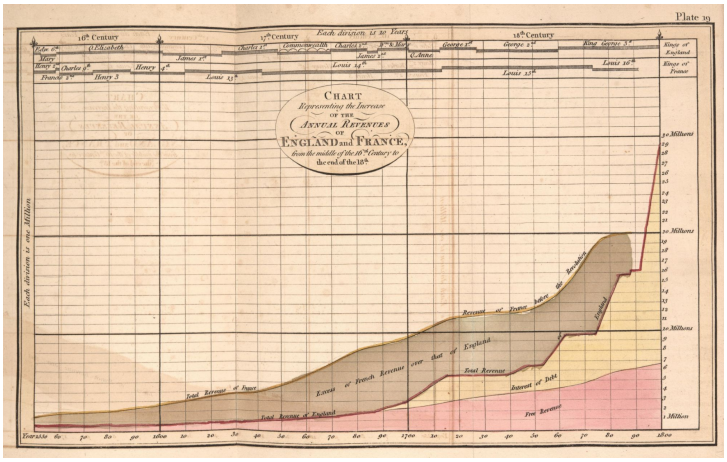


Figure 3.4: Area chart of annual revenues of England and France between 1550 and 1800. Published in 1786, it is considered to be one of the first occurrences of an area chart containing most components found in modern visualizations. [Original appearance in Playfair [1786]. Image extracted from University of Pennsylvania [2022] and used under the terms of Creative Commons CC BY 2.5.]

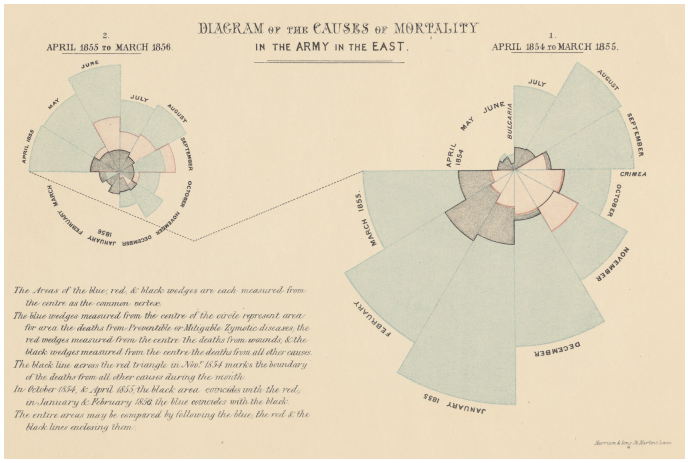


Figure 3.5: One of the polar-area charts created by Florence Nightingale in 1859 to convince people of a need for more sanitary conditions in hospitals. It visualizes the causes of mortality for soldiers during the Crimean War and demonstrates that a large percentage of patients died from preventable diseases linked to unsanitary environments. [Original appearance in Nightingale [1859]. Image extracted from Harvard Library [2022] and used under the terms of Creative Commons Attribution 4.0.]

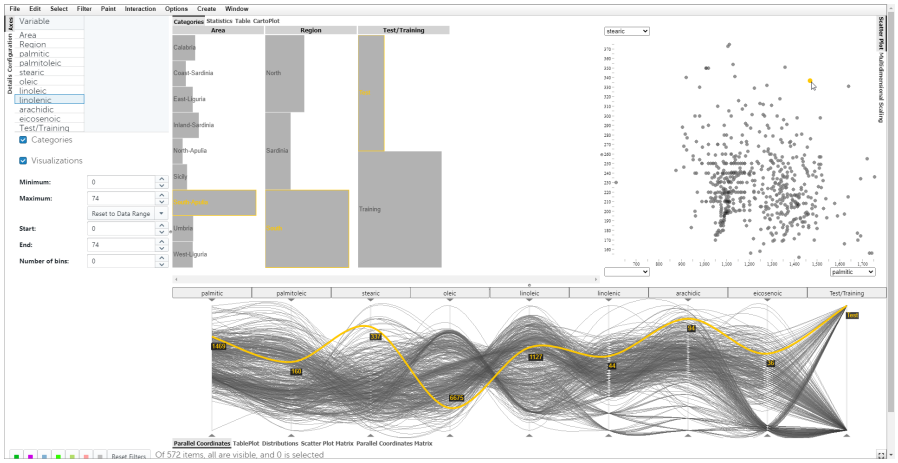


Figure 3.6: High-D by Macrofocus is a visual analytics tool specialized in analyzing multidimensional data. [Screenshot of High-D [Macrofocus 2021] taken by the author of this work.]

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 diagram or polar-area chart, shown in Figure 3.5. 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.

Modern visualizations benefit from the interactive nature of the devices used to consume them. They can be 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 Macrofocus [Macrofocus 2021] is a representative example of a modern interactive visual analytics tool, and is shown in Figure 3.6.

It is out of the scope of this work to provide a full account of the long and eventful history of information visualization. This section only provides a brief and very selective view of the topic. More comprehensive works for further reading include Friendly [2008], Meirelles [2013], Rendgen [2019], and Friendly and Wainer [2021].

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 a visualization differ widely between libraries. D3 is a low-level library which enables data-driven transformations of documents. Vega and other grammar-based libraries provide a declarative grammar to express the visual and interactive characteristics of a visualization. Template-based visualization libraries provide higher-level template-based interfaces that visualization authors can simply fill with data, which makes them very easy to use but can lead to limitations.

3.2.1 Data-Driven Documents (D3)

D3 [Bostock et al. 2011] is a free, open-source document manipulation library built in JavaScript by Mike Bostock and actively maintained by him and a community on GitHub [Bostock 2021]. Mike Bostock is also the creator of Observable [Observable 2021] and was one of the authors of the now deprecated Protovis visualization library [Bostock and Heer 2009]. Wattenberger [2019] is a great introduction to D3.

D3 enables data-driven document transformations allowing developers to describe documents as functions of data. As an example, developers can define transformations which take 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 well suited to 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 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 implement its visual representations. This yields great flexibility, because developers work directly with web standards implemented in browsers and do not need to wait for D3 to support new features as standards evolve. If developers chose to switch to a different library, the knowledge of web standards gained during their work with D3 might be applicable to their future work. The reliance on web standards also makes it possible to use the native debugging tools available in web browsers.

Other important aspects of D3's design include immediate evaluation of functions, the principle of parsimony, and support for method chaining. Immediate evaluate of functions means that operations, such as modifying attributes, are applied instantaneously at the time of calling the respective functions. This reduces internal complexity by handing control flow decisions over to invoking code. It also avoids errors related to missing state changes when state is modified multiple times between rendering, which commonly occur in libraries which use delayed evaluation of functions.

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 in variables which would not otherwise be needed. It is implemented in D3 by returning the D3 Selection on which a modifying method is called as a result of that method. Methods which insert new elements into the DOM, such as `Selection.append` and `Selection.insert`, return a D3 Selection of the newly added elements to enable the creation of nested structures. This method chaining syntax is further aided by the `Selection.call`

```
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. A <h1> element and a <p> element are created inside an existing <body> element.

method, which invokes a callback receiving the current Selection as a parameter and returns the original Selection to chain further 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 with the `Selection.call` method can be seen in Listing 3.1.

Selections are the atomic building blocks of D3 and are used to access almost any functionality. They are created using the `d3.select` or `d3.selectAll` methods. These methods are built on the `querySelector` and `querySelectorAll` methods of the DOM Selectors API, which allow the selection of elements via CSS selectors (see Section 2.2). The `d3.select` and `d3.selectAll` methods create a Selection containing either a single element matching the provided selector, or multiple elements matching it, respectively. 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.

An additional feature of D3 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 newly created elements, the update Selection, containing elements merely receiving new data, and the exit Selection, containing elements being removed. Each of these Selections can be individually transformed using the `Selection.join` method, which can receive three callbacks for the enter, update, and exit Selections of the data join, respectively. This ability to individually control changes to entering, updating, and exiting elements is referred to in D3 as the *general update pattern*. A simple demonstration of how it is used can be seen in Listing 3.2. All the previously mentioned DOM wrapper methods can receive either constant values or dynamic values 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 then calculate a dynamic value based on these parameters, which is forwarded to the corresponding DOM method.

D3 also offers a convenient, optional API to perform JavaScript-based animations via D3 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

```
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 simple demonstration of D3's general update pattern being used to specify different transformations for entering, updating, and exiting elements. The full utility of this pattern is only apparent in more complex scenarios involving transitions.

its easing can be configured using the `Transition.ease` method. It is also possibile to interrupt and chain transitions. Transitions provide an almost identical API to Selections. The major change is that the wrapping methods interpolate towards their target values using the given easing function over the given duration instead of setting the target value directly. Using Transitions is completely optional and developers can choose instead to use other animation technologies, like CSS transitions and animations.

At its 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 was created with a focus on creating visualizations. There are many additional packages which simplify the higher-level tasks necessary for creating and rendering visualizations. All D3 packages follow the same inherent patterns, like method chaining and configurable functions. Therefore, despite this higher-level functionality being split over multiple packages, a consistent experience is provided to developers. Listing all available packages here would be out of the scope of this work, but some noteworthy ones include: `d3-shape` to create visual primitives like lines and areas, `d3-scale` to encode abstract data dimensions, and `d3-axis` to render scales as human-readable axes.

3.2.2 Grammar-Based Visualization Libraries

Vega [IDL 2021] is a library consisting 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 specification 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]. In contrast to previous work, it contains powerful capabilities to declaratively describe interactions [Satyanarayan et al. 2015] in addition to describing visual appearance.

The visual aspects of a visualization are described in a grammar similar to the Grammar of Graphics 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 also contains 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 which is then 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, 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 of 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 [Satyanarayan et al. 2015]. This approach offers the flexibility of 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 defined declaratively, and they can be used as dynamic variables in most places in a Vega specification to change various characteristics of a visualization dynamically. Listing 3.4 shows an example of how the previously shown static bar chart specification can be extended with signals to display a tooltip when hovering over bars.

Visualizations created with Vega closely follow their specifications and minimal assumptions are 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 have already been built on top of Vega [Wongsuphasawat et al. 2015; Satyanarayan and Heer 2014; Wongsuphasawat et al. 2016]. Most noteworthy 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, allowing authors to write specifications for common visualizations in a much more concise form. Specifications written in Vega-Lite are then compiled into Vega specifications. During compilation, the compiler automatically derives default configurations for axes, legends, and scales by following a set of carefully designed rules. This makes Vega-Lite more convenient for quick authoring of visualizations, since many of the 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 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 illustrate the difference between a Vega and a Vega-Lite specification, Listing 3.5 shows a Vega-Lite version of the Vega bar chart specification from Listings 3.3 and 3.4 combined.

Another very interesting development in the field of visualization grammars that is related to responsive visualizations is the recently developed Cicero [Kim et al. 2022], a declarative visualization grammar

```
{
  "$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. It demonstrates the use of data, scales, axes, and marks to construct the bar chart.

```
{
  "...": "...",
  "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 the tooltip whenever an update occurs.

for responsive visualizations. It is an extension of Vega-Lite and includes a compiler that can compile Cicero specifications into Vega-Lite specifications. Due to this compilation into Vega-Lite, Cicero does not require a custom renderer but can rely on Vega-Lite to do the rendering. The main addition of Cicero to Vega-Lite consists of a declarative grammar to describe responsive transformations that transform a source visualization into a responsive version of itself. Responsive transformations are declared in terms of a *specifier* selecting the elements to be transformed, an *action* that should be performed on these elements, and an *option* further defining the action. Cicero's transformation grammar simplifies reusing responsive transformations across different visualizations and is an improvement to Vega-Lite's workflow of having to create completely separate specifications for each responsive version of a visualization.

3.2.3 Template-Based Visualization Libraries

Template-based visualization libraries work by providing high-level 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

```
{
  "$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: A Vega-Lite specification of the Vega bar chart shown in Listings 3.3 and 3.4 combined.

specific languages. Even though these types of libraries are usually flexible enough to create a huge range of visualizations, at some point users may run into limitations. Some of these limitations can only be worked around by writing 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 who run into these limitations.

For this thesis, a total of 20 template-based JavaScript visualization libraries were examined and compared according to factors such as their rendering technology (SVG or Canvas), usage popularity (last year's npm package downloads), open-source popularity (stars on GitHub), license (free or commercial), and recent development activity (commits on GitHub). Usage popularity was 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 no longer maintained. 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 [Tanaka 2020], (7) Chartist [Kunz 2021], (8) amCharts [amCharts 2021], (9) billboardJS [NAVER 2021], and (10) D3FC [Scott Logic 2021]. These ten libraries were selected for further consideration.

Eight of the 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 of the libraries implement an SVG-based renderer, two of which (ECharts and D3FC) also offer alternative rendering to <canvas> elements (D3FC also via WebGL) for high-performance scenarios, and only ChartJS solely targets <canvas>-based rendering. Eight libraries are very actively maintained with most of them showing development activity within the last month. C3JS and Chartist seem to be no longer actively maintained, but were included nontheless in the deeper evaluation, due to 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

```
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: A basic column (vertical bar) chart defined using Highcharts' generic chart creation API. A high-level, declarative configuration object is passed to the creation function.

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 follows: With declarative APIs, users specify what state shall be achieved, whereas with 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 of the 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 in Listing 3.6. Generic chart creation functions seem to correlate with the ability to dynamically change the type of visualization. Chartist, on the other hand, provides separate chart creation functions for each type of chart, and it is not possible to alter the type of chart after it has been created. Another limitation which may originate from partitioning the API by chart type is that mixed charts which combine multiple forms of visualization in one composite visualization cannot 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 upon 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 allows 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 demonstrating the usage of D3FC

```
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: A basic bar chart defined using D3FC's component-based API.

can be seen in Listing 3.7.

amCharts is 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. Its component-based interface is still rather declarative, with most options being configurable by modifying specific 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 in terms of their responsive configurability, most libraries offer similar capabilities albeit in slightly different ways. Six of the ten libraries (Highcharts, C3JS, Chartist, amCharts, billboardJS, and D3FC) support the styling of elements in their created visualizations with CSS, which requires rendering as SVG documents, since 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 visualization authors can make use of CSS-inherent mechanisms to configure responsive styles. Unfortunately, CSS-based styling is inherently limited because only CSS properties representing presentational SVG attributes can be applied on SVG elements, as described in Section 2.5.2.

To responsively configure other visualization characteristics, such as their type, data, and layout, visualization authors have to resort to configuration mechanisms offered by the libraries. Four libraries

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

Listing 3.8: The declaration of responsive rules in Highcharts. In this example, the x-axis and y-axis titles are removed if the chart is narrower than 500 pixels.

(Highcharts, ApexCharts, Chartist, and amCharts) provide the possibility to specify rule-based responsive configurations as part of their declarative interfaces, illustrated 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, authors 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` event listeners. Nearly all libraries provide a means to dynamically resize visualizations and update their data, type, and options. The exceptions are C3JS, which only supports dynamic changes of some options, and Chartist, which does not support changing a visualization's type at all.

Chapter 4

Responsive Information Visualization

A *responsive* visualization is a visualization which adapts itself to the available display space and properties of the device used to access it. Analogous to responsive web design, the need for responsive visualizations arises from the growing variety of devices used to consume content and the physical differences between them. Visualizations and charts often form significant blocks of content embedded inside web pages. For a web page to be responsive, any embedded content such as visualizations and charts must also be responsive.

Visual elements require proper sizing and spacing to be of value. Merely scaling visualizations to fit into their allocated space is insufficient 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, such as touch and keyboard interaction. For example, to ensure that data points remain selectable on less precise input devices such as touchscreens, a visualisation might adapt by reducing the data density and increasing the size of individual elements. The goal of responsive visualizations is that they should adapt themselves to the characteristics of the consuming device and context so as to remain as effective and usable as possible [Kim et al. 2021a].

The topic of responsive visualization only gained prominence in recent years, as responsive web design became mainstream. Hinderman [2015] used the term responsive visualization, but only descibed how to implement scalable visualizations. Körner [2016] covered scalable visualizations, but also considered interactive selection and touch events. Andrews [2018] was possibly the first academic work to address design patterns for responsive visualization. More recently, [Hoffswell et al. 2020] surveyed the design space of responsive visualizations, created a taxonomy of currently used techniques and recurring patterns, and presented a tool to help design responsive visualisations side-by-side. In addition to surveying design patterns, Kim et al. [2021a] also consider issues around different forms of "message loss" when reducing chart complexity.

4.1 Responsive Visualization Patterns

Patterns are templates for solving recurring problems. Hoffswell et al. [2020] created a comprehensive taxonomy of responsive techniques, as well as a tool to help design responsive visualisations side-by-side. They proposed describing responsive techniques according to 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 refers to leaving a component unchanged, but this is deemed a non-technique and therefore left out here. They also described a non-exhaustive set of eleven *components*, upon which these actions can be performed: (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

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: The targets of responsive visualization patterns identified by Kim et al. [2021a]. [Table adapted from Kim et al. [2021a].]

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 interesting finding was that most visualizations (88.7%) implemented no change at all for their interactions, while some (10%) even removed interactive capabilities completely. Only very 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]. Following Hoffswell et al. [2020], they characterised the responsive visualization strategies according to (the same) two dimensions: *targets*, representing what entity is changed, and *actions*, representing how entities are changed. However, the targets and actions are more finely grained, having a number of sub-categories. Targets are grouped into five distinct categories (Data, Encoding, Interaction, Narrative, and References/Layout), with four of the five categories further divided into sub-categories, as shown in Table 4.1. Actions are also grouped into five distinct categories (Recompose, Rescale, Transpose, Reposition, and Compensate), with four of the five top-level categories again having sub-categories, as shown in Table 4.2. The actions are defined as operations with distinct input and output states to ensure they can be inverted, and thus can be applied to either desktop-first or mobile-first design approaches Categorizing techniques using these dimensions, the authors identified a total of 76 viable strategies, whereby some of them are not used in the visualizations they studied. However, their explorable online gallery [Kim et al. 2021b] contains examples demonstrating all these patterns.

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: The actions of responsive visualization patterns identified by Kim et al. [2021a]. [Table adapted from Kim et al. [2021a].]

4.2 Responsive Visualization Examples

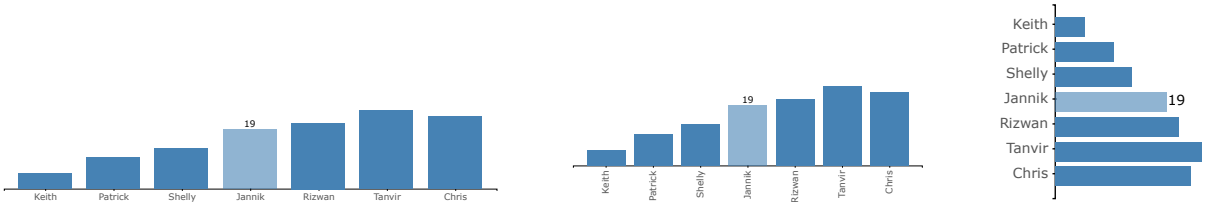
The goal of this section is to provide the reader with some demonstrative examples of responsive visualizations. The figures in this section were taken from external scientific sources which put most of their effort into demonstrating responsive visualization patterns rather than communicating messages 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. 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 is summarized here.

4.2.1 Bar Charts

Bar charts are very widespread, accounting for 135 (= 36%) of the 378 responsive charts in the corpus collected by Kim et al. [2021a]. Bar charts are usually used to visualize two-dimensional data, with one categorical dimension and one quantitative dimension. Two variants of bar charts support the visualization of categorical datasets having subdimensions: grouped bar charts [Ferdio 2021a] compare subdimensions with each other, and stacked bar charts [Ferdio 2021b] 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 created as static images [NYT 2018a; NYT 2020b; Bui 2019; NYT 2020a].

A good example of a responsive bar chart can be seen in Figure 4.1 [Andrews 2018]. 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 to overlap. 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 2017]. Another effective pattern for avoiding overlapping tick labels is to rotate the labels by up to 90 degrees tso they 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 is required necessary [Andrews 2018]. Doing this is more advisable than simply extending the width of the chart past the viewport, since vertical scrolling is easier than horizontal scrolling. When reducing the width of charts containing annotations, a number of patterns can be applied to avoid annotations overlapping. For example, annotations can be removed [Bui 2021; Aisch et al. 2017], simplified, or relocated [WSJ 2017].



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

Figure 4.1: An example of a responsive bar chart at different display widths. (a) At 70rem, axis tick labels are aligned horizontally. (b) At 50rem, axis tick labels are aligned vertically. (c) At 30rem, the chart is transposed. [Screenshots of Andrews [2018] created by the author of this thesis. Used with kind permission by Keith Andrews.]

4.2.2 Line Charts

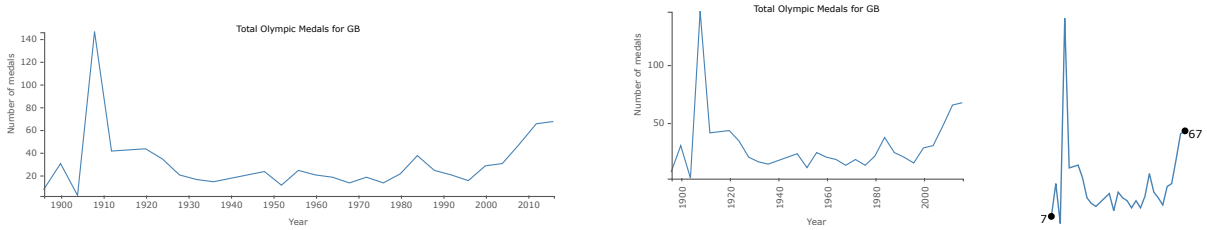
The second most frequent type of responsive charts according to the responsive visualization gallery [Kim et al. 2021a] are line charts, which number 98 (= 26%) out of the 378 responsive visualizations in the gallery. Line charts are used to show trends in two-dimensional datasets by plotting them as points connected by lines (a polyline). They can be extended to compare trends in an additional categorical dimension by drawing additional polylines for each category. Many line charts on the web are published in non-responsive forms [NYT 2019b; NYT 2019a], although some authors take 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] on narrower screens 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 loses information about 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, although transposition could sometimes 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 accomodate annotations without requiring the consumer to scroll horizontally.

4.2.3 Scatterplots

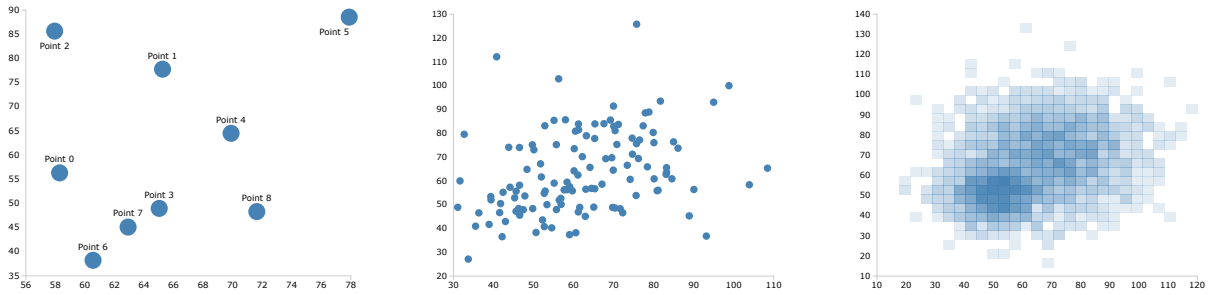
Scatterplots are also quite frequently encountered among responsive charts, numbering 26 (= 7%) of the 378 responsive charts contained in the gallery by Kim et al. [2021a]. A scatterplot represents two-dimensional data as points in a 2d Cartesian coordinate system. There are many examples of scatterplots published as static images [NYT 2018b; NYT 2018c], with responsive versions starting to emerge.

The first step to making scatterplots responsive is to reduce their width to fit them into the space available. As for other types of chart, 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 density 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



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

Figure 4.2: An example of a responsive line chart at different display widths. (a) At 65rem, the x axis labels are horizontal. (b) At 40rem, the x axis ticks have been thinned out and the labels fully rotated by 90°. (c) At 20rem, both axes have been removed, and the chart has become a sparkline. [Screenshots of Andrews [2018] created by the author of this thesis. Used with kind permission by Keith Andrews.]



(a) Small number of data points. **(b)** Medium number of data points. **(c)** Large number of data points.

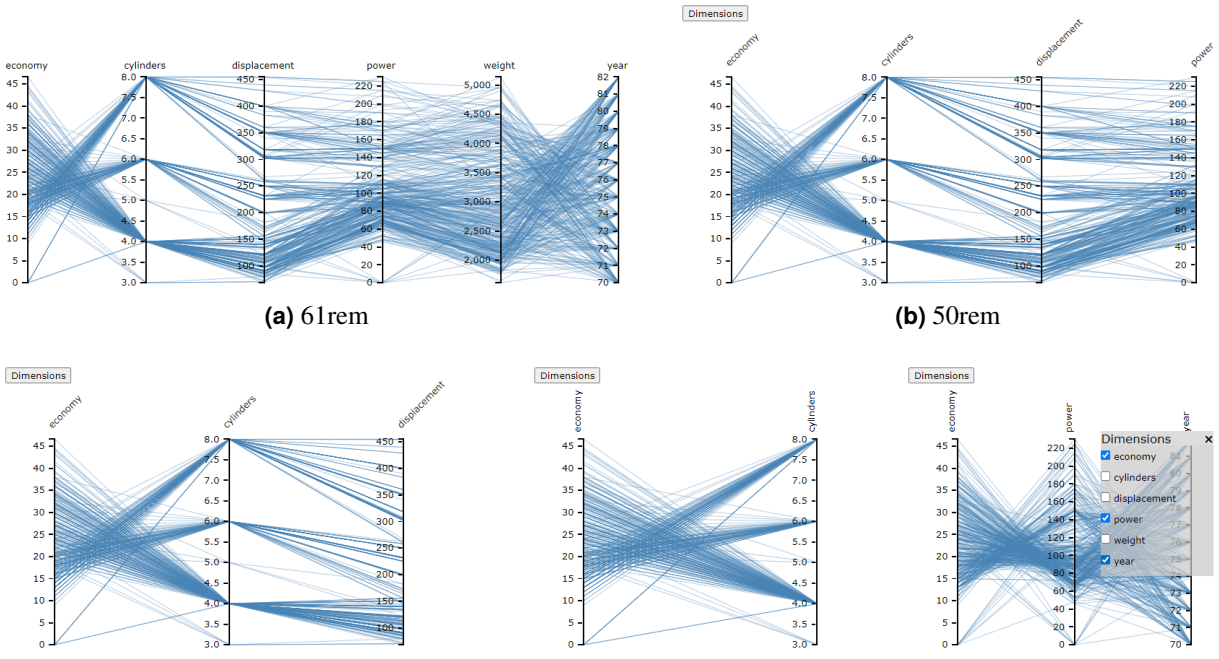
Figure 4.3: An example of a responsive scatterplot based on data density (data points per pixel). (a) With a small number of data points, all points and their corresponding labels are shown. (b) At a certain density, labels are only shown for selected points. (c) At very large densities, the scatterplot is 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 [2014]]

explorative panning interactions. In addition to zooming and panning, Andrews [2018] employs additional methods to ameliorate the overlapping of individual points, including fisheye distortion, Cartesian distortion, and temporary displacements of points.

An interesting technique for responsive scatterplots based on the visualization's density (data points per pixel) rather than its width was introduced by Rabinowitz [2014]. The benefit of this approach is that charts adapt to changing amounts of data and reconfigure their appearance accordingly. The patterns applied in the responsive scatterplot shown in Figure 4.3 are the recomposition of annotations to only show annotations for selected data records, and the switching of the encoding from a scatterplot to a heatmap for high point densities. Other techniques, such as the recomposition of data records, would also be applicable to responsive scatterplots, but no examples for such patterns could be found. If the data to be encoded is inherently cyclic, a radial scatterplot, using polar coordinates, can be used to better reflect the cyclic nature of the data [Barton and Recht 2018].

4.2.4 Parallel Coordinates

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 2021]. In these kinds of charts, multiple dimensions are rendered as parallel axes, upon which points are connected via paths (polylines). Each polyline represents a data record and its values at the corresponding dimensions. The axes of a parallel coordinates chart are typically laid out horizontally, meaning that the



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

(c) 40rem **(d)** 30rem **(e)** 30rem, user-configured dimensions

Figure 4.4: A responsive parallel coordinates chart at different display widths. (a) At larger widths, 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) The dimension configuration panel has been opened, and the user has taken control over which dimensions to show. [Screenshots of Andrews [2018] created by the author of this thesis. Used with kind permission by Keith Andrews.]

chart can be made narrower by reducing the distance between individual axes. Previously mentioned axis-related responsive patterns, such as rotating labels and recomposing ticks, can also be applied.

Another technique is to temporarily hide some dimensions, based on some criteria. When automatically hiding dimensions, it is necessary to apply compensation patterns, giving the user additional controls to configure which dimensions are displayed and override the system's hiding behavior. An example of a responsive parallel coordinate chart incorporating some of these patterns can be seen in Figure 4.4. If reducing the chart's complexity is not appropriate, an alternative is to transpose the chart, so its dimensions are laid out vertically and vertical scrolling can be used to explore the full chart.

Chapter 5

The RespVis Library

RespVis is an open-source D3 extension library for creating responsive SVG charts [Oberrauner 2022]. It enables the use of CSS for the responsive styling and layout of visualizations. RespVis renders visualizations as pure and complete SVG documents, meaning that the whole visualization is contained in one SVG document and includes no elements of other XML namespaces. RespVis is designed as an extension to D3, rather than a wrapper around it. Unlike most other visualization libraries built on top of D3, RespVis does not hide it behind a custom API. Rather, users invoke RespVis functionality by binding specially structured data to D3 Selections, which render visual components using render functions to transform the bound data into some form of visual representation. Furthermore, RespVis espouses the strong separation between data and code and applies strong static type-checking through the use of TypeScript.

5.1 Design

The design of the RespVis library is guided by six principles:

1. Responsive Styling and Layout via CSS.
2. Visualizations are Pure, Complete SVG Documents.
3. Extend Rather Than Wrap D3.
4. Separation of Data and Code.
5. Strong Static Type-Checking with TypeScript.
6. Layered Component Hierarchy.

5.1.1 Responsive Styling and Layout via CSS

Every part of a visualization that can be configured with CSS should be configured with CSS. The visual appearance and layout of HTML elements can already be configured by CSS. Many presentation attributes of SVG elements can also be styled with CSS. However, CSS-based layouting cannot be applied to SVG elements, which seriously limits the responsive layout of SVG charts. Without powerful CSS layout technologies like Flexbox and Grid, all the individual components of an SVG chart have to be positioned manually via JavaScript.

The RespVis Layouter enables the responsive layout of visualization components using the syntax of CSS Flexbox or CSS Grid, by calculating the bounding boxes of SVG elements from the CSS configuration of a shadow `<div>` hierarchy. The Layouter offers visualization authors comparable configurability to

what they are used to when laying out HTML elements. For example, a legend can be configured to be placed to the right of a chart in a wider view, and beneath the chart in a narrower view, using familiar CSS constructs.

5.1.2 Visualizations are Pure, Complete SVG Documents

Every RespVis 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, which embed elements of an XML namespace other than SVG, for example to embed HTML elements inside an SVG. RespVis does not support rendering to HTML canvas elements, because graphics rendered there cannot be styled by CSS.

An SVG document representing a visualization is considered *complete*, if it contains the entire visualization within it. Splitting visualization components across multiple SVG documents is considered bad practice, because these components conceptually belong together and should be represented as a whole. Having a full visualization enclosed in a complete SVG document allows the whole visualization to be exported and stored as a standard-compliant SVG file, which can be further processed using a wide range of tools.

5.1.3 Extend Rather Than Wrap D3

RespVis is designed as a library which extends D3, rather than a wrapper around D3. Compared to other visualization libraries which wrap a layer around D3, it does not provide an entirely new interface to users, but uses D3 Selections as the core interface with which to interact. The typical workflow of invoking RespVis functionality is to bind data objects of a specific structure to the elements of a Selection and then to visualize this data by calling a render function to transform it into visual marks. If D3 were hidden behind a custom API, its powerful capabilities would not be directly accessible to users of the library, and would need to be exposed manually through special mechanisms. By designing RespVis as an extension of D3, users can continue to leverage its expressive and concise API and author their documents using data joins and the general update pattern.

5.1.4 Separation of Data and Code

In RespVis, data and code are decoupled from each other. Everything in RespVis is built from functions and objects without using any classes. Classes were avoided, since they are not common when working with D3, and also because they lead to a tight coupling between data and functionality, which was deemed undesirable. The decoupling of data and code 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 a software system which requires less cognitive effort to understand.

Functions are easier to reuse because they only require well-shaped input data to perform their task. Mechanisms like inheritance or composition, which tend to increase the complexity of a system, are unnecessary. Compared to class-based code, where an object needs to be instantiated before testing 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 dependent on other methods which could affect the results of a test case.

Possibly the main benefit of decoupling data and code lies in the reduced complexity of the resulting system. A software system which treats data and code as different entities might be composed of more entities than a system which does not, but the individual entities have fewer dependencies between one another. The reduced number of dependencies between entities results from separating entities into a data entities group and a function entities group, with no relationships between them. The research related to software complexity is hard to convey in simple terms, but one rule of thumb is well summarized

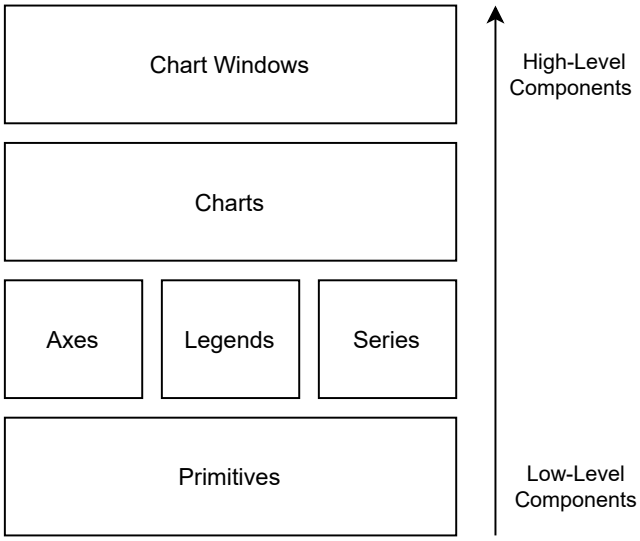


Figure 5.1: The four layers of components in the RespVis library. Higher layers contain increasingly higher-level components. [Image created by the author of this thesis.]

by Sharvit [2022a] 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 drawbacks when designing a system adhering to this concept, but they are not too severe and are therefore not listed here. For further reading on this topic, the reader is directed to Moseley and Marks' classic workshop paper [Moseley and Marks 2006] and Sharvit's forthcoming book [Sharvit 2022b].

5.1.5 Strong Static Type-Checking with TypeScript

The RespVis library is written in TypeScript, with everything being 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, their contents are typed as strongly as possible using the generic type variables available on Selections. Most of the time, it is sufficient to specify only the type of elements contained in a Selection and the structure of the data bound on them. If the element and data types of a Selection are declared, the various functions can assume that parameters passed to them have specific types, and they do not have to worry about dynamic type checking. Applying a strongly typed system has many advantages, such as better development tooling and compile-time identification of type-related bugs. These advantages have already been described in Section 2.4.

5.1.6 Layered Component Hierarchy

Components in RespVis are structured in four layers with increasing levels of abstraction, as shown in Figure 5.1. Components in higher layers make more assumptions about their content than components in lower layers. The bottom-most layer with the lowest level of abstraction consists of visual Primitives, represented by basic SVG elements like `<rect>`, `<circle>`, and `<text>`. These Primitives do not require any data to be bound to them and are simply rendered by setting their attributes to the desired values.

The second layer comprises composite components such as Axes, Legends, and Series. These are usually rendered as `<svg>` or `<g>` elements containing only primitive elements. The components in this layer are the lowest-level components which are configured using structured data bound to their elements. Series components are composite elements which render a collection of underlying elements using a data join and the general update pattern.

The third layer consists of Chart components. Charts are composite components which can also include other composite components. They are the visual entities which represent complete visualizations and are usually composed of axes, series, and legend components.

In many other visualization libraries, charts are the highest-level components users can work with, but RespVis contains an additional layer above them, formed by Chart Window Components. Chart Windows are wrapper components around Charts. Unlike the previously discussed lower layers, Chart Windows are not rendered as SVG elements but as HTML `<div>` elements. Their purpose is to nest Charts into a Layouter Component, render them in a three-phased rendering process, and provide an optional toolbar for them. Toolbars are customizable and can hold different tools for different types of chart.

5.2 Naming Conventions

The naming of entities in RespVis follows the same naming conventions used in D3 modules. The names of entities usually start with the name of the group to which an entity belongs and are then further narrowed down by successively adding more words until the exact entity is accurately described. This convention is referred to as "top-down naming" in this thesis. An example of the top-down naming convention can be seen in the `d3-scale` [Bostock 2022b] and `d3-axis` [Bostock 2022a] modules, in which entities are called things like `scaleLinear`, `scaleOrdinal`, `axisBottom`, and `axisLeft` rather than `linearScale`, `ordinalScale`, `bottomAxis`, and `leftAxis`. Since this is the exact opposite of how these entities would be called in natural English language, using such names can feel odd for the uninitiated. However, experience shows working with APIs which follow such a naming convention is easier than with those which do not, since users of such APIs can easily discover specialized entities by inputting the general entity type and browsing through code completion suggestions provided by their development tools. Hence, entity names in RespVis also follow this convention.

RespVis' public interface is made up of types and functions. Types are usually written as interfaces and represent the shape of an object. Type names are written in `PascalCase` [TechTerms 2022b] and adhere to the top-down naming convention. They always start with the group a type belongs to, and further words are appended to distinguish ever more specialized types. The naming of types can best be demonstrated by the different names given to interfaces describing data objects to configure different kinds of Bar Charts. Data objects for the configuration of Basic Bar Charts are described by the `ChartBar` interface, those of Grouped Bar Charts are described by the `ChartBarGrouped` interface, and those of Stacked Bar Charts are described by the `ChartBarStacked` interface.

The API of RespVis is largely composed of functions. Function names are always written in `camelCase` [TechTerms 2022a] and also follow the top-down naming convention. Function names always start with the type of object on which they operate, followed by the operation they perform. A component in RespVis always consists of a data object which describes it, an element to which the data object is bound, and a render function which transforms the bound data into some form of visual representation. The names of functions to create data objects for the configuration of components are always in the form of `componentNameData`, such as `chartBarData` or `chartBarGroupedData`. Functions which transform bound data into a visual representation are always named in the form of `componentNameRender`, such as `chartBarRender` or `chartBarGroupedRender`.

5.3 Project Structure

RespVis is a NodeJS [OpenJS 2021] project hosted as an open-source project on GitHub [Oberrauner 2022]. It is implemented in TypeScript and grouped into different modules by thematic affinity. The TypeScript source files are compiled (transpiled) to JavaScript and bundled into one combined library package, which users can then import into their projects. The Rollup module bundler [Rollup 2022] is used to perform compilation and bundling. In addition to the bundled JavaScript library, users are

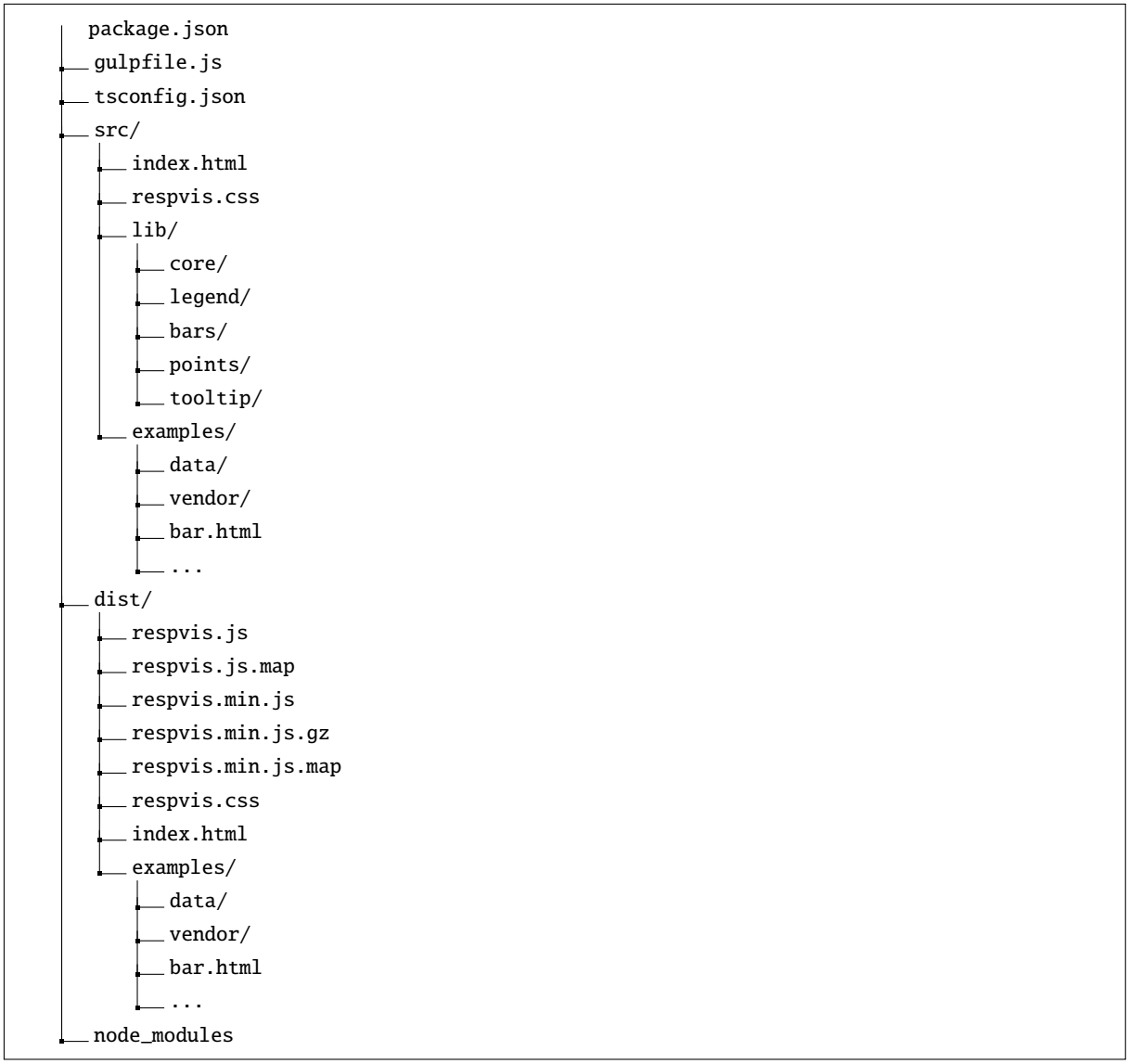


Figure 5.2: The most important files and directories of the RespVis project. [Figure created by the author of this thesis.]

required to import an accompanying CSS file containing default styling for the generated visualizations. The project also contains examples demonstrating usage of the library to create various charts. These examples are HTML files which import the required files and contain JavaScript to invoke RespVis functionality to create and update visualizations. The Gulp [Gulp 2022] task runner is used to automate the build process of the library, including the preparation of the output directory, the bundling of library code, and the copying of various files to the correct locations in the output directory.

The RespVis 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, later sections go into more detail about the setup of the various tools. An overview of the most important files and directories can be seen in Figure 5.2.

The root directory of the RespVis project contains 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 2022]. 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 are partitioned into modules formed around the 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, Chart base functionality, Chart Window base functionality, and various utility functions. The `legend` module contains a Legend Component, which renders a Legend consisting of a title and configurable labeled symbols. The `tooltip` module contains functions to show and hide tooltips, modify their contents, and position them. It also contains helper functions for Series Components to prevent the replication of tooltip-related code in their data creation and rendering functions. 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 built into a combined package, but there are plans to also distribute them separately to allow users of the library to import only those packages they need.

The `src/examples/` directory holds the source files of the developed examples. These examples are distributed alongside the library files, and are copied over to the `dist/examples/` directory when building the project. Every example consists of an HTML file which 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 embedded in the body of the HTML document. In addition to individual example files, the `examples` directory also contains a `vendor` directory, which contains third-party dependencies, and a `data` directory containing data imported by individual examples.

In addition to configuration files and the `src/` directory, the root directory also contains two directories which are automatically generated during the build process. These are the `node_modules/` and `dist/` directories. The `node_modules/` directory 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 RespVis library.

The code of RespVis is distributed as JavaScript bundles in different formats which can be used depending on the situation. These formats are based on both Immediately Invoked Function Expressions (IIFE) and the more modern ES modules format, both of which are explained in more detail in Section 5.5. Bundles containing the `.js` extension in their file name contain IIFE source code, whereas bundles containing the `.mjs` extension contain ES module source code. Furthermore, these bundles are also distributed in minimized versions. The `dist/respvis.[m]js` file contains the unmodified JavaScript bundle that can be used by library consumers who require readable code, `dist/respvis.min.[m]js` contains the minified JavaScript bundle, and `dist/respvis.min.[m]js.gz` contains the minified JavaScript bundle that has additionally been compressed in the GZIP format [Deutsch 1996]. Alongside these code bundles, the Rollup creates source maps for the `dist/respvis.[m]js` and `dist/respvis.min.[m]js` bundles: `dist/respvis.[m]js.map` and `dist/respvis.min.[m]js.map`, respectively. These source maps are 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 and are immensely helpful for debugging.

Since RespVis aims to perform all possible styling in CSS, the distribution also contains a `dist/respvis.css` file with the default styles for 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 could be improved by employing a CSS preprocessing tool

such as SASS [O'Donnell 2019], so that the styles can be split into multiple files during development. Besides the bundled library source code and stylesheet, the `dist/` directory also contains usage examples of RespVis in the `dist/examples/` directory, copied over from the `src/examples/` directory.

5.4 NodeJS

NodeJS is a standalone JavaScript runtime [OpenJS 2021], built on top of the V8 JavaScript engine [Google 2022], which is an open-source and multi-platform runtime for the execution of JavaScript code. NodeJS allows JavaScript code to be run outside of web browsers. NodeJS is heavily used for server-side development to unify the technology stack of web developers and allow them to use JavaScript for both client-side and server-side development. However, with the appropriate project setup, NodeJS can be used for any kind of development, and it can be set up as a powerful framework to develop client-side applications as done in this project. One of the most important tools in the NodeJS environment is the npm package manager [npm 2022], which exists to simplify the sharing of source code modules and the dependency management of a module. The npm package registry hosts a huge number of open-source modules for NodeJS projects, which can easily be imported and used to create new projects.

RespVis is developed as an npm package. Every npm package is configured via a `package.json` file. This file contains all the necessary meta-data for a package to make it identifiable and provide information about what the package contains. The `package.json` file also lists all the dependencies of a package, so they can easily be updated and downloaded during an installation process. A package can rely upon both normal dependencies and development dependencies. Normal dependencies of a package are required for it to work at run time and need to be installed alongside it. Development dependencies are only required during development and are only installed when installing a local package. The `package.json` file is located in the root directory of the RespVis package and can be seen in Listing 5.1.

```
{
  "name": "respvis",
  "version": "0.2.0",
  "description": "A library to build responsive SVG-based visualizations.",
  "main": "index.js",
  "scripts": {
    "build": "npx gulp build",
    "start": "npx gulp"
  },
  "repository": {
    "type": "git",
    "url": "git+https://github.com/AlmostBearded/respvis.git"
  },
  "keywords": [
    "..."
  ],
  "author": "Peter Oberrauner",
  "license": "MIT",
  "bugs": {
    "url": "https://github.com/AlmostBearded/respvis/issues"
  },
  "homepage": "https://github.com/AlmostBearded/respvis#readme",
  "devDependencies": {
    "rollup": "^2.45.2",
    "rollup-plugin-gzip": "^2.5.0",
    "rollup-plugin-terser": "^7.0.2",
    "@rollup/plugin-commonjs": "^18.0.0",
    "@rollup/plugin-node-resolve": "^11.2.1",
    "@rollup/plugin-typescript": "^8.2.1",
    "@types/...": "...",
    "browser-sync": "^2.26.14",
    "del": "^6.0.0",
    "gulp": "^4.0.2",
    "gulp-cli": "^2.3.0",
    "gulp-rename": "^2.0.0",
    "tslib": "^2.2.0",
    "typescript": "^4.2.4"
  },
  "dependencies": {
    "d3": "^7.2.1",
    "debounce": "^1.2.1",
    "to-px": "^1.1.0",
    "uuid": "^8.3.2"
  }
}
```

Listing 5.1: The `package.json` file of the RespVis library. This file contains all the meta-data describing the package and its dependencies. Keywords and type dependencies have been omitted for readability.

```
// do-something.js
export function doSomething() {
  console.log('something incredible was done!');
}


// module.js
var someModule = (function () {
  function doSomething() {
    console.log('something incredible was done!');
  }
  return { doSomething };
})();


// application.js
someModule.doSomething();
```

Listing 5.2: Immediately Invoked Function Expression (IIFE) modules wrap the module code inside a function which is executed immediately after declaring it and returns the public interface of the module. `do-something.js` contains the original code that should be wrapped as an IIFE module, `module.js` contains the code of the IIFE module, and `application.js` demonstrates usage of the module.

5.5 Rollup

The Rollup module bundler [Rollup 2022] is used to bundle the source code of the RespVis library into bundles of different kinds. Bundling combines code written as multiple smaller modules into one combined package to make it easier to distribute. Developers do not have to worry about the details of how their code will be packaged, as Rollup takes care of all the necessary transformations. In addition to bundling source code, Rollup also performs *tree shaking* on the bundled code, eliminating unused code from the resulting bundle by statically analyzing dependencies between modules. Rollup supports the creation of bundles in several common module formats, such as CommonJS, Asynchronous Module Definition (AMD), Universal Module Definition (UMD), Immediately Invoked Function Expressions (IIFE), and ES. RespVis is distributed as both IIFE and ES modules.

IIFE modules have been used for a long time, as they were used to support modular software designs in JavaScript before more elaborate module formats were defined. They are anonymous functions which are executed directly after declaring them. These functions contain the full logic of the module and return an object representing its publicly accessible interface. This object is usually stored in a variable to allow interactions with the module after its creation. IIFE modules are plain JavaScript and do not require any modern features to be supported by browsers. They are simply loaded in web documents like any other JavaScript resource via a `<script>` element. The example in Listing 5.2 illustrates the IIFE module format.

ES modules are a more recent addition to JavaScript, introduced in ECMAScript 6 [ECMA 2015]. They are a native module system built around the `import` and `export` statements, which are widely supported by modern browsers. Since the individual modules of the RespVis library are built as ES modules anyway, Rollup mostly only has to merge them to create a valid, combined ES module. ES modules are natively supported in browsers, so they can be loaded directly in a HTML document using a `<script>` element. However, it is necessary to mark them as modules via the `type="module"` attribute on the loading `<script>` element, so that browsers can interpret them accordingly.

The core package of Rollup is only able to create mostly unmodified bundles from JavaScript source files. Various plugins add frequently required additional functionality. There are two kinds of Rollup plugins: bundle plugins, which affect the bundling process, and output plugins, which transform the already bundled code.

The Rollup bundle plugins used for the bundling of RespVis are `@rollup/plugin-node-resolve`, `@rollup/plugin-commonjs`, and `@rollup/plugin-typescript`. The `@rollup/plugin-node-resolve` plugin is used to resolve imports from other NodeJS packages that reside in the `node_modules` directory. Since many NodeJS packages are still implemented as CommonJS modules, which are not natively supported by Rollup, the `@rollup/plugin-commonjs` plugin is used to interpret them. Lastly, the `@rollup/plugin-typescript` plugin is used to compile TypeScript source files to JavaScript before bundling them. The configuration for the TypeScript compiler is taken from the `tsconfig.json` file at the root directory of the project.

The Rollup output plugins used during the bundling process are `rollup-plugin-terser` and `rollup-plugin-gzip`. These plugins do not affect every created bundle, but are used to selectively transform the contents of specific bundles. The `rollup-plugin-terser` plugin is used to create minified versions of the RespVis bundles denoted by the term `.min` in their file names. Logically, they are equivalent to non-minified bundles, but are compressed as much as possible to reduce their file size while still containing valid, but unreadable, JavaScript code. The `rollup-plugin-gzip` plugin is used to create compressed gzipped versions of the RespVis bundles denoted by the term `.gz` in their file extensions [Deutsch 1996].

Note that D3 is not included in any of the generated RespVis bundles. RespVis is designed to be an extension of D3 and, most of the time, an application wishing to use RespVis will already be using D3. If D3 were to be included in the RespVis bundle, it would unnecessarily be loaded a second time. To prevent D3 from being included in the created bundles, all dependencies on D3-related packages are marked as external.

The actual bundling is performed via the JavaScript API of Rollup in the private `bundleJS` Gulp task. This task is executed in various automation processes set up with Gulp, which are explained in more detail in Section 5.6. The code of the `bundleJS` task can be seen in Listing 5.3. The RespVis library is bundled via the `Rollup.rollup` function which returns the created bundle. This bundle is then written to one or more target destinations via the `Bundle.write` method, which allows the specification of the target bundle format and any plugins used to transform the code before writing it.

```
async function bundleJS() {
  const bundle = await rollup.rollup({
    input: 'src/lib/index.ts',
    external: [
      'd3-selection',
      'd3-array',
      'd3-axis',
      'd3-brush',
      'd3-scale',
      'd3-transition',
      'd3-zoom',
    ],
    plugins: [
      rollupNodeResolve({ browser: true }),
      rollupCommonJs(),
      rollupTypescript()
    ],
  });

  const minPlugins = [rollupTerser()];
  const gzPlugins = [rollupTerser(), rollupGzip()];
  const writeConfigurations = [
    { ext: 'js', format: 'iife', plugins: [] },
    { ext: 'min.js', format: 'iife', plugins: minPlugins },
    { ext: 'min.js', format: 'iife', plugins: gzPlugins }, // .gz added by plugin
    { ext: 'mjs', format: 'es', plugins: [] },
    { ext: 'min.mjs', format: 'es', plugins: minPlugins },
    { ext: 'min.mjs', format: 'es', plugins: gzPlugins }, // .gz added by plugin
  ];

  return Promise.all(
    writeConfigurations.map((c) =>
      bundle.write({
        file: `dist/respvis.${c.ext}`,
        format: c.format,
        name: 'respVis',
        globals: {
          'd3-selection': 'd3',
          'd3-array': 'd3',
          'd3-axis': 'd3',
          'd3-brush': 'd3',
          'd3-scale': 'd3',
          'd3-transition': 'd3',
          'd3-zoom': 'd3',
        },
        plugins: c.plugins,
        sourcemap: true,
      })
    )
  );
}
```

Listing 5.3: The private Gulp task which bundles the code of the RespVis libary. The bundle is created once and then written to multiple targets.

```
Tasks for gulpfile.js
|-- clean
|-- cleanAll
| |-- <series>
|   |-- cleanDist
|   |-- cleanNodeModules
|-- build
| |-- <series>
|   |-- cleanDist
|   |-- <parallel>
|     |-- bundleJS
|     |-- bundleCSS
|     |-- copyExamples
|-- serve
|-- default
```

Listing 5.4: A hierarchical representation of the tasks defined in the `gulpfile.js` file, as output by the `gulp --tasks` command.

5.6 Gulp

Gulp is a task runner which automates workflow processes via a set of named tasks [Gulp 2022]. It is used to automate processes like building the library and serving examples on a development server. Tasks are useful for automating operations which need to be carried out repeatedly. They can perform an atomic operation or be composed of other tasks. Composite tasks can execute tasks contained in them in serial or in parallel. The tasks are implemented as JavaScript functions in the Gulp configuration file, `gulpfile.js`, which can be found in the root directory of the project. Gulp's approach of favoring code over declarative configuration files means that the person setting up process automation needs to be familiar with JavaScript. In return, the possibilities of configuration are endless.

Tasks in the `gulpfile.js` file are separated into private and public tasks. Private tasks are simply asynchronous functions which perform a certain action that does not necessarily have to be executed by external entities. The private tasks in the RespVis project are `bundleJS`, `bundleCSS`, `copyExamples`, `cleanDist`, `cleanNodeModules`, and `reloadBrowser`. Public tasks are also asynchronous functions, but they are exported and are therefore available to be executed via the Gulp command-line interface. Most public tasks in the RespVis project are compositions of other tasks. The public tasks in RespVis are `clean`, `cleanAll`, `build`, and `serve`. The default task, `serve`, is executed when no other task is specified on the command line. A hierarchical representation of all the tasks in the `gulpfile.js` file is shown in Listing 5.4.

Bundling the RespVis library's source code is implemented in the private `bundleJS` task. It uses the JavaScript API of Rollup to compile the TypeScript source files into JavaScript and bundle them into IIFE and ES modules of varying levels of minification. This task has already been described in detail in Section 5.5, so it won't be discussed further here. It is executed during the public `build` and `serve` tasks.

The `bundleCSS` task is used to copy the `src/respvis.css` file to the `dist/` directory. Since one of the design pillars of RespVis is to style everything possible with CSS, this file contains all the default styles for visualizations created with RespVis. Currently, this file is one large single file in the `src/` directory and is merely copied over to the `dist/` directory, but there are plans to build this file from different modules using a CSS preprocessor in the future, which will require an additional bundling step. This task is executed as part of the public `build` and `serve` tasks.

The private `copyExamples` task copies all the files from the `src/examples/` directory to the `dist/` directory. This task is required because the examples are developed inside the `src/` directory, but need to be made available in the distributable library packages. Another reason for the copying is that the BrowserSync development server is initialized with the `dist/` directory as its root, and every potentially viewable file must reside somewhere in that directory. The `copyExamples` task is executed during the public `build` and `serve` tasks.

The private `cleanDist` and `cleanNodeModules` tasks are used to delete the `dist/` and `node_modules/` directories, respectively. The `cleanDist` task is exported under a different name as the public `clean` task. This task is necessary because without cleaning the `dist/` directory before every rebuild, redundant files from previous builds that might have disappeared in the meantime would cause littering and confusion. Therefore, this task is executed as the first step of the `build` task. The public `cleanAll` task is composed of the private `cleanDist` and `cleanNodeModules` tasks. It is manually executed by a developer to delete the currently cached dependencies of the project, to then reinstall them from scratch.

The public `build` task is responsible for building all parts of the project. It is a composite task which executes the `clean`, `bundleJS`, `bundleCSS`, and `copyExamples` tasks. The `clean` task is invoked before all of the other tasks, which are then executed in parallel. After this task finishes, the `dir/` directory will contain all distributable JavaScript and CSS files of the library, as well as the distributable `examples/` directory.

To simplify the development of RespVis, a Browsersync [*Browsersync* 2022] development server is used to host the built distributables. Browsersync is a useful tool for synchronized browser testing during development. It has many features like simulated network throttling, interaction synchronization, and file synchronization, which enable simultaneous testing in multiple environments. For RespVis, it is only used for its ability to synchronize and hot-reload files on the fly. The public `serve` task, which is also exported as the default task, initializes a Browsersync development server which serves files from the `dist/` directory. Automatic reloading of the development server is implemented manually via the `Gulp.watch` function. This function enables a task to be executed whenever a change to a file matched by the supplied glob pattern is detected. The `serve` task implements three different cases that cause the development server to reload. Firstly, every time one of the TypeScript files in the `src/lib/` directory changes, the `bundleJS` task is executed, and the browser is reloaded. Secondly, every time the `src/respvis.css` file changes, the `bundleCSS` task is executed, and the browser is reloaded. Thirdly, whenever a file in the `src/examples/` directory is changed, the `copyExamples` task is executed, and the browser is reloaded.

Chapter 6

RespVis Modules

The source code of RespVis is structured into modules written in the ES module format. Currently, all individual modules are combined into a single, monolithic library bundle during the build process, but this will be changed in the future so that users can import only the modules they need. The reason for this is that most users will likely require only a subset of all the features included in the library, and it would unnecessarily increase the size of their bundles to import them all. A good example of this is D3, which also separates its extensive collection of functionality into different modules which can be successively added to a project as the need arises.

At the time of writing, the RespVis library comprises five modules: Core Module, Legend Module, Tooltip Module, Bar Module, and Point Module, each containing various submodules grouped by thematic similarity. The Core Module holds the core functionality of the library which all other modules depend on, including the Layouter and Axis Components, Chart and Chart Window base functionality, and various utility functions and types. The Legend Module contains the implementation of a Legend Component to identify discrete data by rendering a legend of distinct values as labeled symbols. The Tooltip Module holds functions to control the display, placement, and content of Tooltips, as well as utility functions to simplify the configuration and initialization of Tooltips on Series Components. The Bar Module distinguishes between Single-Series, Grouped, and Stacked Bars and includes various low-level and high-level components to render each of those types. Similarly, the Point Module contains low-level and high-level components to visualize Point Charts. All of the different modules and the dependencies between them are shown in Figure 6.1.

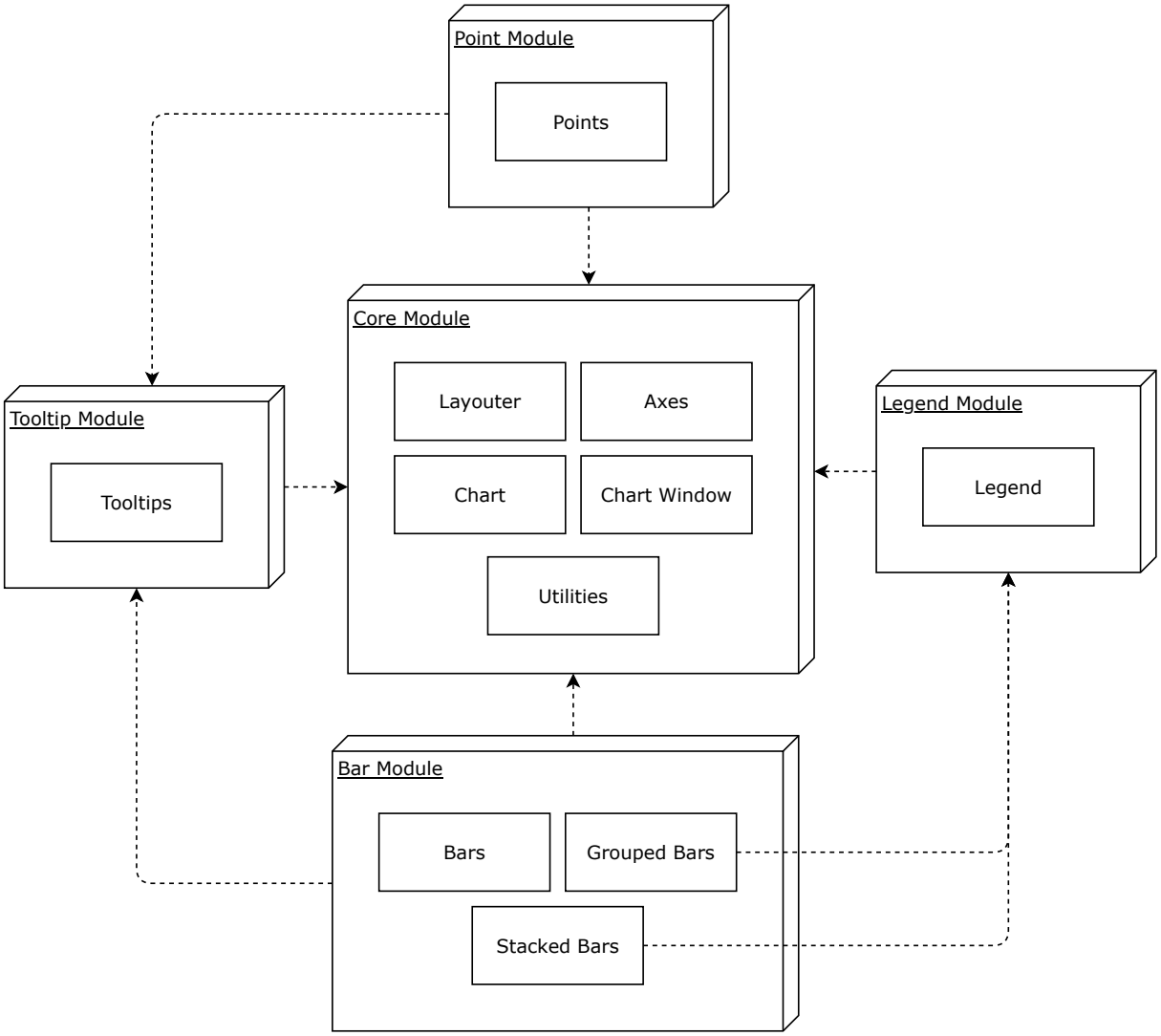


Figure 6.1: The five modules of the RespVis library and their submodules. The directional arrows indicate dependencies between modules. [Image created by the author of this thesis using diagrams.net.]

6.1 Core Module

The implementation of the Core Module is located in the `src/lib/core/` directory of the project and contains the necessary core functionality of the library. It forms the base on which all other modules depend and includes various utility functions, the Layouter, Axes, Chart base functionality, and Chart Window base functionality. RespVis heavily relies on utility functions to reuse recurring operations, and the Core Module contains utilities which simplify the handling of arrays, elements, Selections, texts, positions, sizes, rectangles, circles, and paths. The Layouter is a custom component which enables the layout of SVG elements with CSS. Axis Components have been included in the Core Module because they are important components which occur in nearly all visualizations. Lastly, the Core Module offers base functionality for Charts and Chart Windows to simplify the creation of more specialized Chart and Chart Window Components.

6.1.1 Utilities

The utilities provided by RespVis are split into multiple submodules placed in the `utilities/` directory of the Core Module. These modules include types and functions to perform array, element, selection, and text operations, as well as modules to simplify geometric operations with positions, sizes, rectangles,

circles, and paths. Utility functions are grouped into submodules by the type of entity on which they operate, which is is also reflected in the utility function names. The names of utility functions follow the top-down naming convention described in Section 5.2, which means that the names all begin with the type of entity with which the function is associated.

Array utilities can be found in the Array Utilities Module in the file `utilities/array.ts`. The `Array` class in the JavaScript base implementation already offers a wide variety of convenient methods to work with arrays. These methods form a solid foundation to handle a broad range of situations, but not everything is covered, and some things require manual implementations, which is why the RespVis library offers additional functions to simplify commonly encountered tasks. The `arrayEquals` function is used to verify the equality of two arrays and also works with arbitrary levels of nesting. Array type guard functions are used to determine at runtime whether or not a variable is an array. The function `arrayIs` evaluates to true if the passed parameter is any kind of array, and `arrayIs2D` evaluates to true if the passed parameter is a two-dimensional array. The `arrayIs` function is merely an alias for the `Array.isArray` method and has been added to provide a consistent counterpart to the `arrayIs2D` type guard function. The last function in the Array Utility Module is the `arrayPartition` function, which receives an array and a partition size as parameters and returns a partitioned version of the input array with each chunk containing the number of items specified by the partition size parameter.

The Element Utility Module located at `utilities/elements.ts` in the Core Module contains functions and constants related to elements in a document. The `elementRelativeBounds` function is used to calculate the bounding box of an element relative to the bounding box of its parent in viewport coordinates. Internally, it uses the `getBoundingClientRect` function, which returns the actual bounding box of an element in viewport coordinates and, as opposed to other ways of accessing this information, this function also takes transformations into account. Every element has a set of CSS styles applied to them, and the `Window.getComputedStyle` method is used to query the active styles of elements. The style declaration object returned by this method contains all possible CSS properties and their values, regardless of whether or not they are set to default values. Sometimes this behavior may be desired, but in this library, the computed style is mainly used for the preparation of a downloadable SVG document to transform styling information set in CSS to attributes on the individual elements. If every possible CSS property on every element would be mapped to an attribute, the resulting SVG document would be unnecessarily bloated and hard-to-read, because only those properties that are not set to their default values actually have an effect. For this reason, the `elementComputedStyleWithoutDefaults` function has been implemented to calculate the computed style of an element and remove all default-valued properties from the returned style declaration object. This is implemented by adding a `<style-dummy>` element as a sibling of the element of interest, getting the computed styles of both elements, and calculating the difference between them. To accelerate these calculations, the `elementComputedStyleWithoutDefaults` function accepts an array of property names as its second parameter and will only consider the properties listed in this array. The constant `elementSVGPresentationAttrs` array contains the names of all SVG presentation attributes listed in the SVG 1.1 specification [Dahlström et al. 2011]. As soon as support for SVG 2 [Bellamy-Royds et al. 2018] by most major browsers has reached maturity, this array will be extended to include any newly added presentation attributes. Since only these SVG attributes can be styled via CSS, only CSS properties representing presentation attributes have to be considered when preparing downloadable SVG documents.

Selection utilities are implemented in the Selection Utilities Module in file `utilities/selection.ts` and include typing improvements for the D3 `Selection`, `Transition`, and `SelectionOrTransition` interfaces and type guards to distinguish between them. The `Selection`, `Transition`, and `SelectionOrTransition` interfaces allow the specification of four type variables: the type of elements contained in the Selection or Transition, the type of data bound to those elements, the type of the parents of those elements, and the type of data bound to those parents. In most cases, the type variables related to parent elements do not influence the logic of code using these interfaces and could be omitted to keep it more concise. For this reason, these interfaces have been reexported with default types set on all of the type variables,

which means that whenever type variables need to be manually specified, only those that need to be set to specific types need to be explicitly stated. Further typing improvements have been made to the `attr` and `dispatch` methods of the `Selection` interface. The D3 type declarations of the `Selection.attr` method do not include `null` as a possible return value, which is wrong because this method will result in a `null` value when reading an attribute that does not exist. To fix this inconsistency and catch potential bugs related to it during compilation, the type declaration of the `Selection.attr` method has been overwritten in the Selection Utility Module to include `null` as a possible return value. A less important but convenient improvement has been made to the type declaration of the `Selection.dispatch` method, which allows the dispatching of custom events with certain parameters that control different aspects of how this event is dispatched and the data bound to it. In practice, not all parameters need to be specified at every invocation because the implementation of the `Selection.dispatch` method will provide default values for all of them, but this is not reflected in the type declaration of the function, which requires every parameter to be set every time the function is called. To fix this, the Selection Utility Module provides a type declaration overwrite for the `Selection.dispatch` function that wraps the type of the parameters parameter into the `Partial` utility type. Apart from these typing improvements, this module also provides the `isSelection` and `isTransition` type guard functions that are used to distiguish between Selections and Transitions.

Utilities for dealing with `<text>` elements can be found in the Text Utilities Module in file `utilities/text.ts`, which contains basic functionality to set specific `data-*` attributes to specific values on `<text>` elements. The Text Utility Module holds functions that set `data-*` attributes controlling the horizontal and vertical alignment of `<text>` elements, as well as their orientation. Horizontal and vertical alignment is configured using the `textAlignHorizontal` and `textAlignVertical` functions, which respectively set the `data-align-h` and `data-align-v` attribute on `<text>` elements to the value passed into either function as a string enum parameter of type `HorizontalAlignment` or `VerticalAlignment`. The `HorizontalAlignment` enum represents the string values `"left"`, `"center"` and `"right"`, while the `VerticalAlignment` enum represents the values `"top"`, `"center"` and `"bottom"`. The distinct `data-align-h` and `data-align-v` attribute values are then used in Selectors of various CSS rules to declare different values for the `text-anchor` and `dominant-baseline` properties. Text orientation is set using the `textOrientation` function, which sets the `data-orientation` attribute on `<text>` elements to the value specified via the `Orientation` string enum parameter. The `Orientation` enum represents the values `"horizontal"` and `"vertical"`. These `data-orientation` attribute values are then used in CSS to set the `text-anchor`, `dominant-baseline`, and `transform` properties of `<text>` elements, in order to rotate them accordingly and position them correctly inside their bounding box calculated by the Layouter.

The Core Module also contains utilities to simplify geometric operations. One of these utility modules is the Position Utility Module located in the `utilities/position.ts` file, which contains the `Position` interface and various functions to perform operations related to it. The `Position` interface consists of the `x` and `y` number properties. Rounding these properties is necessary to be able to correctly compare the equality of two `Position` objects and to not render unnecessarily long strings when transforming them into string representations. This rounding is performed with the `positionRound` function, which allows the specification of the number of decimals the properties should be rounded to. Equality comparision between two `Position` objects can be done with the `positionEquals` function, which evaluates to `true` if all properties of both `Position` objects are equal and `false` if not. The `positionToString` function can be used to transform a `Position` object into its `"x, y"` string representation, and its counterpart, the `positionFromString` function, can be used to transform a correctly-formatted string into a `Position` object. A large part of RespVis consists of modifying the attributes of elements. Therefore, the `positionToAttrs` function can be used to set the `x` and `y` attributes of elements to the values of the `x` and `y` members of a `Position` object, and similarly, the `positionToTransformAttr` function can be used to set the `transform` attribute of elements to a translation representing a `Position` object. The Position Utility Module also contains the `positionFromAttrs` function, which can be used to create a `Position` object from an element's `x` and `y` attributes.

The Size Utility Module located in the `utilities/size.ts` file in the Core Module is very similar

to the Position Utility Module. It contains the `Size` interface, which consists of the `width` and `height` number properties, the `sizeRound` function to round the properties of a `Size` object to a certain number of decimals, and the `sizeEquals` function to compare two `Size` objects for equality. Similar to the equivalent functions in the Position Utility Module, the `sizeToString` and `sizeFromString` functions can be used to convert between `Size` objects and their string representations, and the `sizeToAttrs` and `sizeFromAttrs` functions can be used to convert between `Size` objects and `width` and `height` attributes of elements.

Utilities for dealing with rectangles can be found in the Rectangle Utility Module, which is located in the `utilities/rect.ts` file of the Core Module. This module contains the `Rect` interface, which is the union of the `Position` and `Size` interfaces and therefore describes an object with the `x`, `y`, `width`, and `height` number properties. Similar to the Position and Size Utility Modules, this module contains the `rectRound` function to round `Rect` objects, the `rectEquals` function to compare two of them for equality, the `rectToString` and `rectFromString` functions to convert between `Rect` objects and their string representations, and the `rectToAttrs` and `rectFromAttrs` functions to convert between objects and `x`, `y`, `width`, and `height` attributes of elements. Since the `Rect` interface is a combination of the `Position` and `Size` interfaces, most of the functions in this module internally use the functions provided by the Position and Size Utility Modules. The `rectMinimized` function creates a minimized version of the passed `Rect` object, which is infinitely small and positioned at the original `Rect` object's center. Minimized rectangles are used in transitions that grow or shrink `<rect>` elements from or to their centers. When declaring a stroke for SVG elements, it is drawn exactly on the outline of an element's shape, which means that a stroke will extend outside the original bounds of an element by half the stroke width. This can lead to unwanted artifacts like the stroke of bars in a Bar Chart overlapping over the Chart's Axes. To counteract this, the `rectFitStroke` function is offered by the Rect Utility Module to adjust the properties of `Rect` objects to account for a specific stroke width around them. Lastly, the Rectangle Utility Module provides functions to calculate specific positions inside rectangles. The most generic of these functions is the `rectPosition` function, which enables the calculation of a position inside a rectangle via a two-dimensional parameter which expresses a position as the percentual width and height distance from a rectangle's top-left corner. All other position-calculating rectangle utility functions are simply shorthand functions that internally call the `rectPosition` function. The `rectCenter` function returns a `Position` object representing the center position of a `Rect` object. The `rectLeft`, `rectRight`, `rectTop`, and `rectBottom` functions return `Position` objects that represent the middle position of the corresponding edge of a `Rect` object. Similarly, The `rectTopLeft`, `rectTopRight`, `rectBottomRight`, `rectBottomLeft` functions can be used to calculate the corner positions of a rectangle.

The Circle Utility Module can be found in the `utilities/circle.ts` file in the Core Module. It contains the `Circle` interface, which describes a circle object via a `center` `Position` property and a `radius` number property. This module also contains equivalent functions to those found in previously-mentioned utility modules: `circleRound`, `circleEquals`, `circleToString`, `circleFromString`, `circleToAttrs`, `circleFromAttrs`, `circleMinimized`, and `circleFitStroke`. Furthermore, the `circlePosition` function can be used to calculate positions inside a circle using an angle that defines an offset direction and an offset distance from a circle's center as a percentage of the circle's radius. The Circle Utility Module also contains functions to create circles from rectangles, which are the `circleInsideRect` function to calculate the largest circle that can fit inside of a rectangle and the `circleOutsideRect` function to calculate the smallest circle that encloses a rectangle.

The Path Utility Module is located in the `utilities/path.ts` file in the Core Module and provides functions to simplify the creation of path definitions that can be set as `d` attributes on `<path>` elements. The `pathRect` function uses a `Rect` object to create a rectangle path definition that can be set on `<path>` elements instead of using `<rect>` elements. Similarly, the `pathCircle` function uses a `Circle` element to create a circle path definition that can be set on `<path>` elements instead of using a `<circle>` elements. The reasons for using `<path>` elements rather than more descriptive shape elements is that their shapes can be changed dynamically and it is possible to smoothly transition between shapes by interpolating their path definition strings.

6.1.2 Layouter

The Layouter is the most novel contribution of this work. It is a component which wraps around an SVG document and allows configuration of the layout of elements in this document with CSS constructs like Grid and Flexbox. Instead of implementing a custom layout algorithm, the Layouter utilises the layout engine already built in to the browser, which were summarized in Section 2.6.1. Earlier proof of concept implementations used the FaberJS [FusionCharts 2021] and Yoga [Facebook 2021d] layout engines to compute layouts, but these implementations limited layouting to either Grid or Flexbox-based constraints. Furthermore, the use of built-in browser functionality in the current implementation leads to a reduced bundle size and to visualization authors being able to use all the layouting capabilities natively offered by browsers.

CSS has always been the foundation of responsive web design for HTML-based websites because of its ability to adapt an element's presentation and the possibility of defining different presentations for different contexts via media queries. A large part of the responsive power of CSS comes from its ability to change the positioning and layout of elements. As already mentioned in previous chapters, CSS can style certain aspects of SVG documents, but it is not possible to use CSS layouting techniques to position SVG elements. Even though there are already other visualization libraries such as Chartist [Kunz 2021] and Highcharts [Highsoft 2021] which allow the use of CSS to style visualizations, none of them offer the possibility to modify the layout of visualizations via CSS, which means that visualization authors have to learn and use custom APIs to position elements, limiting the range of possible layouts to those supported by the individual libraries.

The RespVis Layouter distinguishes between laid-out and non-laid-out elements, since not every element in a visualization profits from being laid out. The positions and sizes of laid-out elements are calculated by the Layouter, whereas non-laid-out elements are ignored during the layout process. Theoretically, the Layouter could be used to position all visualization elements, since all that is necessary is to determine a good mapping for each element that maps the rectangular bounding box calculated by the Layouter to the desired SVG shape of the element. However, the positioning of elements in a visualization is constrained more strictly than element positioning in typical HTML documents, because the content of a visualization is communicated through visual features such as position, size, shape, and poximity of elements rather than simply through text which can be positioned much more freely. For this reason, many elements of a visualization must be positioned at specific locations with specific dimensions, which means there is very little profit in laying them out with an elaborate layout algorithm. Instead, exactly-positioned elements like the `<rect>` elements of Bar Series and the `<circle>` elements of Point Series are usually positioned directly via their SVG attributes.

The Layouter Module can be found in the `layouter.ts` file of the Core Module. The main function of this module is the `layouterCompute` function which implements the three-phased layout process shown in Figure 6.2. The three phases are:

1. Replication: The structure of the SVG document that shall be laid out is replicated with HTML `<div>` elements, because only HTML elements can be affected by CSS-based positioning. These elements are referred to as "layout elements" and have the same classes and `data-*` attributes as the SVG elements they are replicating.

2. Layout: The replicated layout elements are affected by CSS rules to configure their positioning and are automatically laid out by browsers. If the selectors of CSS rules used to style SVG elements only select them using classes and `data-*` attributes, the layout of these elements can be directly configured in these rules, because the corresponding layout elements have the same classes and `data-*` attributes and these CSS rules will also be applied to them.

3. Synchronization: In this phase, the positions of layout elements are synchronized with their respective SVG elements. The calculated bounding boxes of layout elements are set as `bounds` attributes on SVG elements to make the boundary information available in subsequent renderings for the positioning

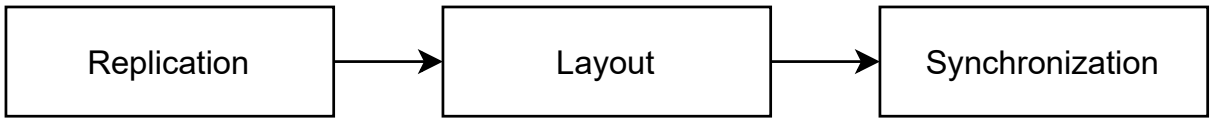


Figure 6.2: The three phases of the layout process of the RespVis Layouter. During the replication phase, the SVG document is replicated with HTML `<div>` elements. Afterwards, these HTML elements are laid out by the browser in the layout phase, and the positions of the laid-out HTML elements are applied to their respective SVG elements during the synchronization phase. [Image created by the author of this thesis using diagrams.net.]

of nested elements. In addition, the Layouter sets different default attributes on different types of SVG elements that aim to represent the boundaries of individual elements.

During the replication process, the structure of an SVG document is replicated with HTML `<div>` elements, which is implemented via a hierarchical D3 data join, in which the original SVG elements are bound as data objects to layout elements. The hierarchical data join results in a counterpart in the hierarchy of layout elements for each SVG element that should be affected by the Layouter. Since not every SVG element should be positioned via the Layouter, the Layouter must be told which elements to ignore. For this, the `data-ignore-layout` and `data-ignore-layout-children` attributes have been introduced. Elements having the `data-ignore-layout` attribute or which are children of elements having the `data-ignore-layout-children` attribute will not be replicated by the Layouter.

To configure the layout of such layout elements in CSS, it must be possible to select them uniquely with a CSS selector. This selector should be as similar as possible to the selector of their corresponding SVG elements, to make it as easy as possible to configure the CSS properties of layout elements. For this purpose, the `class` attributes and all `data-*` attributes of SVG elements are copied to their layout elements. In addition to the classes of the replicated SVG element, the `layout` class is set on all layout elements, which makes it possible to specifically select an SVG element's layout element via the same selector extended by the `layout` class. If CSS rules affecting SVG elements use only classes and `data-*` attributes in their selectors, the properties of corresponding layout elements can be directly configured in the same rules, since these selectors will also match them. An example of a replicated layout element tree of an SVG document can be seen in Listing 6.1. An example of CSS rules to set various properties of SVG elements and their layout elements can be seen in Listing 6.2.

The size of dynamically-sized elements depends on the size of their content, and since layout elements exist separately from their SVG elements and can not access their content, a manual solution had to be implemented to set the size of layout elements to the content size of their SVG elements when required. `<text>` elements are a good example of dynamically-sized elements, because their size is rarely explicitly declared and usually depends on the size of their textual content. The custom `--fit-width` and `--fit-height` CSS properties were introduced to activate the manual copying of dimensions from SVG elements to their layout elements. These boolean properties can be set in CSS rules and are checked during the replication phase via the `window.getComputedStyle` method. If at least one of these properties is set to `true`, the dimensions of the SVG element are calculated with the `Element.getBoundingClientRect` method are set as `width` or `height` properties in the `style` attribute on the corresponding layout element. This way, layout elements will have the same sizes as their SVG elements and can be properly used in the calculation of the overall layout.

Layout elements are positioned according to the layout information specified in CSS rules during the layout phase. Since layout elements are merely `<div>` elements which have been styled via CSS rules, the browser can position them automatically via its integrated layout engine, which happens immediately after they have been created or updated in the replication phase. After the layout phase, the final bounding boxes of layout elements can be calculated and used for further operations.

In the synchronization phase of the layout process, the Layouter calculates the bounding boxes of

```
<div class="layouter">
  <svg class="chart">
    <rect class="box" data-index="0" />
    <rect class="box" data-index="1" />
    <rect class="box" data-index="2" />
  </svg>
  <div class="layout chart">
    <div class="layout box" data-index="0" />
    <div class="layout box" data-index="1" />
    <div class="layout box" data-index="2" />
  </div>
</div>
```

Listing 6.1: The replicated layout element structure of an SVG document. Every SVG element has a corresponding layout element that has the same classes and `data-*` attributes. In addition to the classes of the original SVG element, every layout element also has the `layout` class to allow specific targeting of layout elements via CSS Selectors.

```
.chart {
  display: grid;
  grid-template: 25rem 15rem / 50% 50%;
  grid-template-areas:
    'a b'
    'c c';
}

.box[data-index=0] {
  grid-area: a;
  fill: red;
}

.box[data-index=1] {
  grid-area: b;
  fill: green;
}

.box[data-index=2] {
  grid-area: c;
  fill: blue;
}
```

Listing 6.2: These CSS rules are used to configure the layout and style of an SVG document that is being laid out by the Layouter. Since the selectors of these CSS rules only use `class` and `data-*` attributes to match elements, the same rule can be used to configure the properties of an SVG element and its corresponding layout element. The structure of the SVG document and its replicated layout elements can be seen in Listing 6.1.

Figure 6.3: The three phases of the render process when using the RespVis Layouter. During the first render phase, every element that affects the layout needs to be rendered. The layout phase of the render process is equivalent to the layout process described in Figure 6.2. In this phase, the Layouter calculates the final positions and sizes of laid-out elements and stores them in attributes on these elements. During the second render phase, the boundary informations calculated in the previous phase is utilized to rerender all elements of the visualization at their final positions with their final dimensions. [Image created by the author of this thesis using diagrams.net.]

all layout elements and sets this boundary information as attributes on the corresponding SVG elements. Bounding boxes of layout elements are calculated relative to their parent elements using the `elementRelativeBounds` utility function, converted to their string representations via the `rectToString` utility function, and set as `bounds` attributes on corresponding SVG elements. These `bounds` attributes can then be deserialized to `Rect` objects whenever the bounding boxes of SVG elements are needed for calculations in subsequent renderings. In addition to setting `bounds` attributes, the Layouter also sets specific default attributes on different types of SVG elements in an attempt to automatically fit them into their bounding boxes without manually having to set attributes in subsequent renderings. If the Layouter would not set these default attributes, they would have to be set manually on every laid-out element in the rendering functions, which would be less convenient and lead to duplicated code in various places. For those SVG elements which can be mapped directly to rectangular areas, such as `<svg>` and `<rect>` elements, the `x`, `y`, `width`, and `height` attributes are set to the values of the element's bounding boxes. SVG shape elements that have explicit sizes and positions but are not rectangular, such as `<circle>` and `<line>` elements, also receive attributes that fit them into their boundaries in a way that was deemed most sensible. Other SVG elements that are not explicitly sized, such as `<g>` and `<text>` elements, are merely moved to the correct positions by setting their `transform` attributes to translations so that their top-left corners align with the top-left corners of their bounding boxes. The Layouter does not automatically reposition exactly-positioned elements based on the changed boundary of the composite `<svg>` or `<g>` elements containing them, so this has to be implemented manually in the render functions of various components.

Using the Layouter requires a more complex rendering process than would be needed if the boundaries of elements would already be known before rendering them. The way the Layouter works, some elements need to be rendered before calculating the layout, and afterward, when the positions and sizes of elements are known, the visualization needs to be rerendered in its final form. This rendering process consists of three phases shown in Figure 6.3. The three phases of the render process are the first rendering phase to render elements affecting the layout, the layouting phase, and the second rendering phase to render elements affected by the layout. In the first rendering phase, all elements and attributes which affect the layout of a visualization need to be rendered, which mainly includes laid-out container `<svg>` and `<g>` elements containing exactly-positioned child elements. Dynamically-sized elements such as `<text>` elements and axes also need to be fully rendered in this phase because their content affects the calculation of the overall layout. The layouting phase is where the previously-described layout process seen in Figure 6.2 is executed, including the bounding box calculation of laid-out elements and their persistence as attributes which can be accessed during the second rendering phase. In the second rendering phase, the previously-calculated bounding boxes of elements are used to perform a second rendering of the complete visualization. Here, every element affected by the layout, i.e. nearly every element, is rendered at its final position with its final dimensions. In theory, the two rendering phases of components could be implemented as separate functions, but it is more convenient to invoke the same render function twice and perform some operations only if the appropriate `bounds` attribute has already been set.

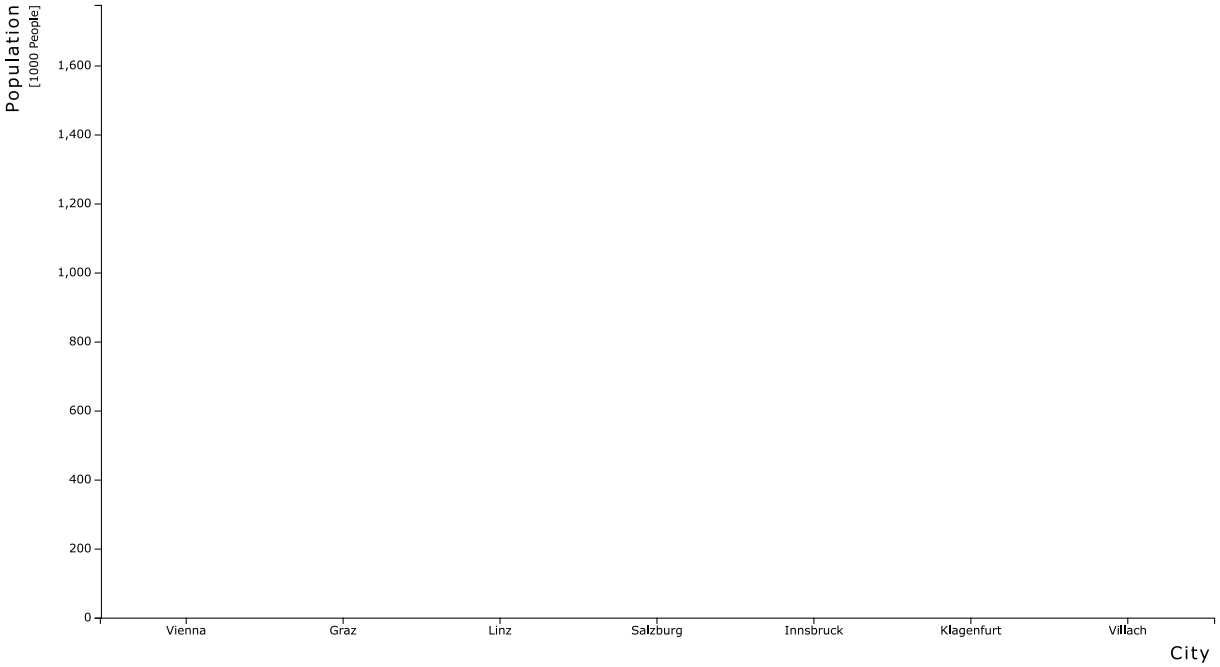


Figure 6.4: A rendered Left Axis with ticks, a title, and a subtitle, and a Bottom Axis with ticks and a title. [Image created by the author of this thesis using RespVis and Inkscape.]

6.1.3 Axes

Axes are implemented in the `axis.ts` file in the Core Module and are used to visualize scales that map abstract values to spatial dimensions. Currently, only Cartesian Axes, distinguished by their position relative to a visualization's draw area, are provided by RespVis, since only Cartesian Charts have been implemented so far. The implementation has so far been focused on Left and Bottom Axes, because they are the most commonly encountered types of Cartesian Axes and cover most use cases. An Axis consists of ticks, an optional title, and an optional subtitle, where the ticks of an Axis are the actual visualization of the Axis' scale, and the title and subtitle can be specified for additional description. An example of what a rendered Left and Bottom Axis might look like can be seen in Figure 6.4.

The `Axis` interface describes the shape of a data object with which an Axis can be configured. It includes a `scale` property, representing the scale to be visualized, the `title` and `subtitle` string properties, and the `configureAxis` function property, which can be used to configure the underlying D3 axis before rendering it. Like most other components, axis components consist of two main functions: a data creation function and a render function. The `axisData` function is used to create an `Axis` data object from a `Partial<Axis>` object parameter, where all non-set but required properties are filled with default values. The `axisBottomRender` and `axisLeftRender` functions are used to render a Left and Bottom Axis in a composite element on which an `Axis` data object has been bound. An Axis' root element is a CSS Grid container and defines the layout of the title, subtitle, and ticks elements. The default configuration of a Left Axis positions these elements in a three-column layout in which the title, subtitle, and ticks elements are placed in this order from left to right. For a Bottom Axis, the default configuration positions the same elements in a three-row layout, in which the ticks, title, and subtitle elements are placed in this order from top to bottom. Furthermore, the title and subtitle elements of a Left Axis are oriented vertically to save horizontal space using the `textOrientation` utility function. Internally, the RespVis Axis Components use the `axisBottom` and `axisLeft` functions from the D3 Axis Module [Bostock 2022a] to render the ticks of an Axis. Since these D3 functions use attributes to position and style elements, as many of these attributes as possible must be removed directly after the ticks have been rendered to enable their configuration via CSS.

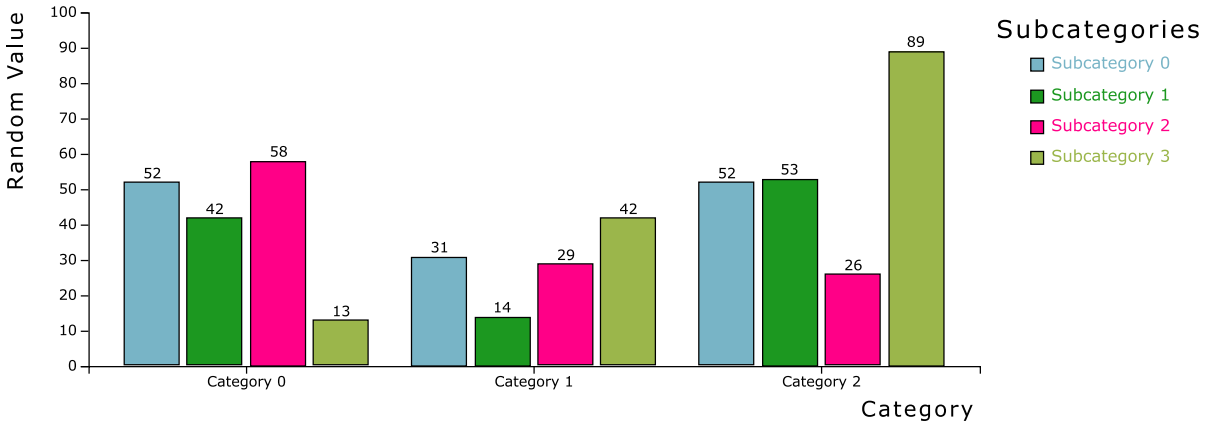


Figure 6.5: An example of a Chart containing a Left Axis, a Bottom Axis, a Grouped Bar Series, a Label Series, and a Legend. [Image created by the author of this thesis.]

6.1.4 Chart

Charts are high-level components which represent a complete visualization with Axes, Legends, and Series. An example of a rendered RespVis Chart containing a Left Axis, a Bottom Axis, a Grouped Bar Series, a Label Series, and a Legend can be seen in Figure 6.5. A Chart is typically rendered in the root `<svg>` element of an SVG document that has at least the `chart` class set in its `class` attribute and the appropriate SVG namespace set in its `xmlns` attribute. These attributes can be set in more specific Chart Components either manually or via the `chartRender` function from the `chart.ts` file in the Core Module, which only sets these attributes.

As mentioned previously, RespVis currently only supports Cartesian Charts, which visualize data in a Cartesian coordinate system. The implementation of the base functionality of Cartesian Charts is located in the `chart-cartesian.ts` file in the Core Module. The `ChartCartesian` interface describes a data object for the configuration of Cartesian Charts via the `xAxis` and `yAxis` `Axis` properties which describe the X and Y Axes of the Chart respectively. Transposing Axes is a useful pattern to improve the responsiveness of visualizations and can be configured using the `flipped` boolean property of `ChartCartesian` data objects. If the `flipped` property is set to `false`, the `xAxis` object is used to configure the Bottom Axis and the `yAxis` object is used to configure the Left Axis. If it is set to `true`, it is the other way around.

The `chartCartesianData` function is used to create a `ChartCartesian` data object. This function gets a partial data object with only those properties set which are of interest to the calling code, and all non-set properties are filled with default values. The default values of the `xAxis` and `yAxis` properties are set by the `axisData` function from the Axis Module. The `flipped` property is initialized to `false`.

The rendering of Cartesian Charts is split into two functions which have to be called separately, because not all parts of a Cartesian Chart can be rendered simultaneously. The general structure of a Chart must be rendered before anything else can be rendered, since this includes the draw area container element into which individual Series Components are rendered. A Chart's Axes need fully initialized scales to be rendered correctly. However, the range of a scale, i.e. the range of values into which abstract values are mapped, depends on the size of the draw area and is only set during the render function of the individual Series Components. Therefore, the Axes of a Cartesian Chart must be rendered after Series Components in order to ensure fully initialized scales.

The structure of a Cartesian Chart is rendered with the `chartCartesianRender` function, which sets the necessary attributes and classes on the root element and attaches the draw area `<svg>` element to it. The draw area is the container element into which the Series Components of a Chart are rendered. An `<svg>` element without actual content is not able to receive input events, which means that it would, for example, not be possible to capture scroll events to control a zoom interaction when the cursor is over the

empty area of the draw area. To counter this, a transparent <rect> background element filling the whole draw area is added, allows input events to be received even in empty areas.

The chartCartesianAxesRender function is used to render the Axes of a Cartesian Chart. This function must only be called on elements with a bound ChartCartesian data object and only after the scales that are to be visualized by the Axes have been fully initialized. Charts must first render the Chart's structure using the chartCartesianRender function, followed by the desired Series Components, and only then can the Chart's Axes be rendered using the chartCartesianAxesRender function. The chartCartesianAxesRender function creates two <g> elements and renders a Left Axis and a Bottom Axes in them. Depending on whether the flipped property in the bound data object is set to true or false, the xAxis data object is used to configure the Bottom Axis or Left Axis and the yAxis data object is used to configure the other. After the Axes are rendered, the x-axis class is set on the one the x-axis data object is bound to, and the y-axis class is set on the other one.

The elements of a Cartesian Chart are positioned using a CSS Grid layout. By default, a grid is created which defines the axis-left, axis-bottom, draw-area, and legend areas. Most rows and columns of this grid are sized to fit their content, with the only exception being the row and column containing the draw area, which is set to fill the remaining space not occupied by the other rows and columns. The default CSS configuration of a Cartesian Chart positions any legend to the right of the draw area, but this can be changed by either adjusting the grid directly via CSS or activating one of the preconfigured positions via the data-legend-position attribute. To simplify setting the data-legend-position attribute, the chartLegendPosition function can be used, which sets this attribute to the value of the passed LegendPosition enum parameter.

6.1.5 Chart Window

Chart Windows are implemented in the chart-window.ts file in the Core Module and are wrapper components around Charts that render a Chart inside a Layouter and decorate it with a Toolbar. These components represent an even higher-level layer of abstraction than Charts and are used to manage their rendering process and configuration. In most other visualization libraries, Charts are provided as the highest level of components that can be configured, which typically means that additional HTML elements for the runtime configuration of Charts have to be created and managed by the embedding web page itself. Chart Windows are rendered with the chartWindowRender function on HTML elements containing a Chart's SVG document. Their structure consists of a element in which the Toolbar is rendered, and of another element on which a Layouter is initialized and which holds the wrapped Chart's SVG document. An example of a Chart Window with an expanded Tool Menu containing two Nominal Filtering Tools and an SVG Download Tool can be seen in Figure 6.6.

Currently, the Toolbar only contains the Tool Menu, a dropdown menu into which individual tools are added as menu items or as submenus. Dropdown menus are created with the menuDropdownRender function and consist of a title and a container for menu items. The current dropdown menu implementation uses no JavaScript and simply shows menu items via CSS when there is a hover interaction. The Tool Menu is created with the menuToolsRender function, which internally uses the menuDropdownRender function to initialize it as a dropdown menu.

The Core Module provides various tools which can be added to the Tool Menu of Chart Windows. One of these tools is the Nominal Filtering Tool located in the tools/tool-filter-nominal.ts file of the Core Module. This tool is used to filter a nominal (or categorical) data dimension of a visualized dataset via a dropdown menu that includes a Checkbox Series. Nominal data, in contrast to ordinal or quantitative data, consists only of labels that do not have a quantitative value assigned to them and therefore have no inherent ordering. The data object for the configuration of a Nominal Filter Tool is described by the ToolFilterNominal interface, which contains properties to specify the title of the dropdown menu, the individual options to be filtered, and the keys of these options. The toolFilterNominalData

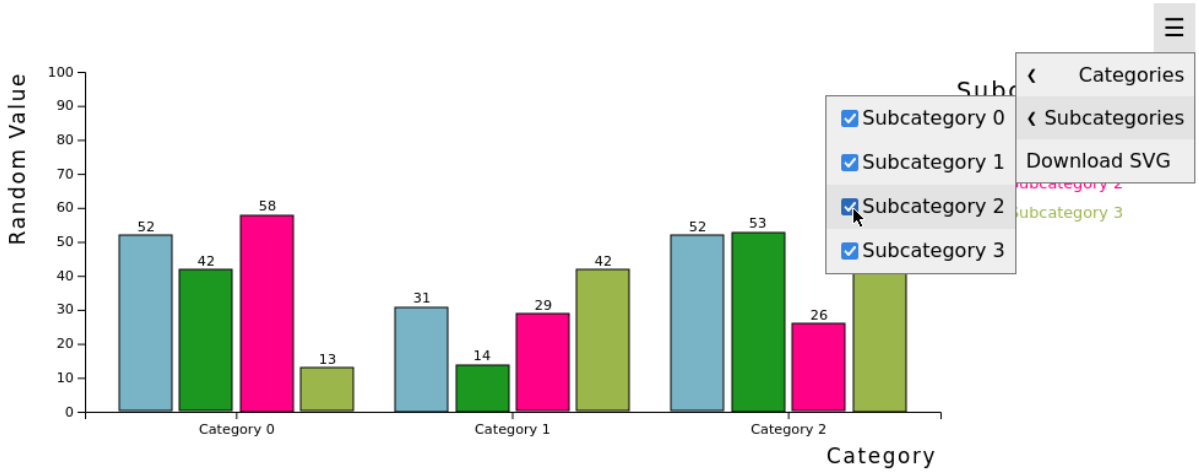


Figure 6.6: An example of a Chart wrapped in a Chart Window. The Tool Menu has been expanded by hovering over it, and the menu entries of two Nominal Filtering Tools and the SVG Download Tool can be seen inside. [Image created by the author of this thesis.]

function is used to create a data object of type `ToolFilterNominal` from a partial input object where undefined properties are being filled with default values. Nominal Filtering Tools can then be rendered on elements with bound `ToolFilterNominal` data objects using the `toolFilterNominalRender` function, which internally uses the `menuDropdownRender` function to initialize a dropdown menu into which the checkboxes representing the individual options of the filter are rendered as menu items.

Checkbox Series are implemented in the `series-checkbox.ts` file in the Core Module and render a series of checkboxes consisting of checkbox `<input>` elements with associated `<label>` elements. They are configured using data objects in the form of the `SeriesCheckbox` interface, which contains properties to determine the type of checkbox container elements and the labels of checkboxes. A data object of this type can be created with the `seriesCheckboxData` function, which creates a complete object from a partial one. Checkbox Series are rendered using the `seriesCheckboxRender` function, which generates individual checkboxes using a data join requiring an array of `Checkbox` data objects so that a single data object can be bound to each checkbox. Individual `Checkbox` data objects contain all the data needed to render a single checkbox and are created by transforming the `SeriesCheckbox` data objects bound on the Series' root element. Single checkboxes consist of a container element, an `<input>` checkbox element, and a `<label>` element. In order to semantically assign the `<label>` elements to their `<input>` elements, the `for` attributes on `<label>` elements must be set to the ids of their corresponding `<input>` elements. This requires assigning unique ids to `<input>` elements, which are generated via the `uuid` function and set as `id` attributes on the `<input>` elements and as `for` attributes on `<label>` elements when the checkbox is first created. The `uuid` function is an alias for the `v4` function from the `uuid` npm package [UUID 2022] and is used to generate UUIDs (Universally Unique IDentifiers) [Leach et al. 2005] of the fourth version, which are very likely to be unique and therefore can be safely used as values for `id` attributes.

The SVG Download Tool is implemented in the `tools/tool-download-svg.ts` file of the Core Module and is included by default in every Chart Window. SVG documents embedded in HTML documents cannot be downloaded by users as easily as raster images, which can simply be downloaded with native browser tools. Instead, SVG documents must be encoded in `Blob` objects and then set as as object URLs in the `href` attributes of `<a>` elements. However, since the presentation of RespVis visualizations is mainly configured using CSS, the active CSS properties must first be converted to attributes before the SVG document can be downloaded. For this, a clone of the whole SVG document is made to set attributes on the cloned elements without affecting the rendered visualization. After the document has been cloned, attributes reflecting the active CSS configuration of the original elements are set on cloned elements. The active CSS configuration of the original elements is calculated using the `elementComputedStyleWithoutDefaults`

Figure 6.7: The three Legends resulting from the source code in Listing 6.3. The first Legend has been configured to have rectangles as symbols, the second to have circles as symbols, and the third with rectangle symbols horizontally laid out. [Image created by the author of this thesis.]

utility function, which yields a list of CSS properties and their values, only containing properties that are not set to defaults. After setting all the necessary attributes on cloned elements, the string representation of the cloned document is calculated using the `Element.innerHTML` property and encoded in a `Blob` object with the content type `image/svg+xml`. At present, the string representation of the SVG document is not processed further or prettily formatted, which results in a rather difficult-to-read file and which will be improved in future work. The `Blob` object containing the SVG document is further transformed into an object URL via the `URL.createObjectURL` method and set in the `href` attribute of a newly created `<a>` element. This newly created `<a>` element is then briefly attached to the `<body>` element of the active HTML document and clicked using the `Element.click` method, which initiates the download of the final prepared SVG document. Download SVG Tools can be rendered as menu items in a Chart Window's Tool Menu with the `toolDownloadSVGRender` function, which initializes them and triggers the download of the SVG document embedded in the Chart Window when the user clicks on the menu item.

6.2 Legend Module

The Legend Module in the `src/lib/legend/` directory currently comprises a sigle file `legend.ts`, containing the implementation of the Legend Component. Legends are used to visualize scales whose abstract values are not mapped to spatial dimensions in a coordinate system, but to other visual properties such as colors, shapes, or sizes. The Legend implemented in this module illustrates such scales by creating labeled, configurable symbols for every mapping and, therefore, this component is best suited for the visualization of discrete value mappings. Continuous data dimensions can still be visualized with this component, but they must be approximated by dividing the continuous domain of values into equal discrete steps. An example demonstrating the use of the Legend Module can be found in the `legend.html` file in the `src/examples/` folder. An excerpt from this example can be seen in Listing 6.3, with its rendering shown in Figure 6.7.

A data object for the configuration of a Legend is defined by the `Legend` interface, which contains properties to describe the title, labels, and symbols of a Legend. Symbols are configured via functions which take the boundaries calculated by the Layouter into account to set the `d` attributes of `<path>` elements. Legend symbols are rendered as `<path>` elements because these enable the rendering of arbitrary symbols that can be changed dynamically, whereas the usage of `<rect>` or `<circle>` elements would be much more restrictive. A disadvantage of using `<path>` elements is that, since they require manual configuration of exact shapes via path definition strings, their usage is more tedious than the usage of more restricted SVG elements, and they require slightly more annotation effort when it comes to accessibility. Colors of individual items in a Legend are indirectly configured via `data-style` attributes on items whose values are specified in the `Legend` data objects. Some style classes, such as the `categorical-x` classes for categorical styling, are already provided by RespVis and handled in the library's distributable CSS file.

```
<html>
<head>
  <style>
    /* ... */
    .horizontal-legend .items {
      flex-direction: row;
    }
  </style>
</head>
<body>
  <div id="chart"></div>
  <script type="module">
    // imports, init chart window, init chart, ...

    const labels = [0, 1, 2, 3].map((n) => `Label ${n}`);

    // legend with rectangle symbols
    const rectLegendData = legendData({
      title: 'Legend',
      labels,
      symbols: (p, s) => pathRect(p, rectFromSize(s)),
    });

    // legend with circle symbols
    const circleLegendData = legendData({
      title: 'Legend',
      labels,
      symbols: (p, s) => pathCircle(p, circleInsideRect(rectFromSize(s))),
    });

    // horizontal legend with rectangle symbols
    const horizontalLegendData = rectLegendData;

    // handle render process, resize, ...
  </script>
</body>
</html>
```

Listing 6.3: The source code of the example implemented in the `legend.html` file in the `src/examples/` directory. When executed, this code results in the three Legends shown in Figure 6.7. Non-essential parts of the source code have been removed to focus on Legend-related configurations. The horizontal Legend is configured with the same data object as the rectangle symbol Legend, but the items of the horizontal Legend are laid out horizontally via the `flex-direction: row` CSS property.

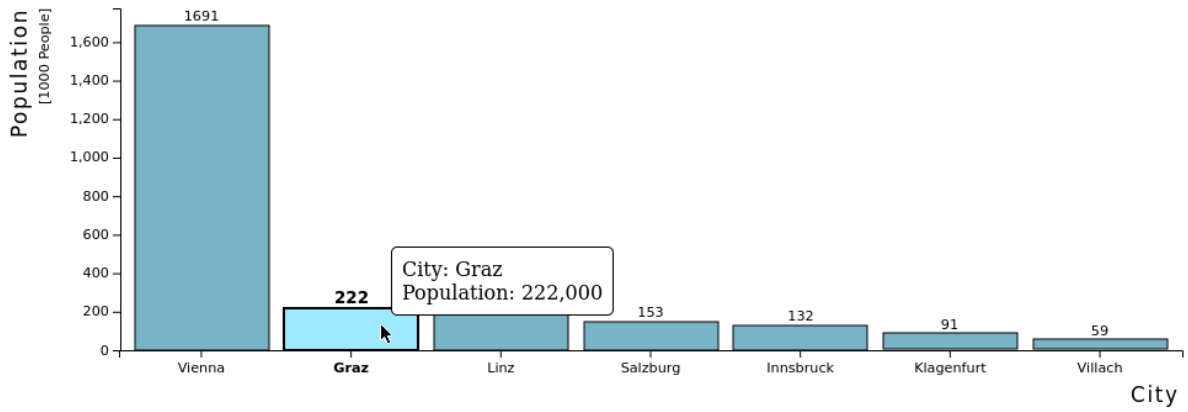


Figure 6.8: A Bar Chart with an active Tooltip displaying additional information from the data record associated with an individual bar. Since a Tooltip is only visible during interaction with a particular element, it may sometimes overlap and cover other important parts of a Chart. [Image created by the author of this thesis.]

Furthermore, custom style classes can easily be added by simply handling them in user-specified CSS rules. `Legend` data objects can be created with the `legendData` function, which receives a partial input object in which the properties which are not of interest to the calling code are not required to be set and are populated with default values.

A Legend is rendered into elements which have a `Legend` data object bound on them using the `legendRender` function. This function sets the `legend` class on the root element, attaches a `<text>` element with which the title of the Legend is shown, and attaches a `<g>` element into which the individual items of the Legend are rendered via a data join. To perform the data join with which the items of the Legend are created, one `LegendItem` data object per item to be rendered is needed to describe individual legend items, and these are generated via transformation of the bound `Legend` data object. Via a data join with these data objects, one `<g>` element with the `legend-item` class is created for every legend item, to which a `<path>` element for the symbol and a `<text>` element for the associated label are attached. The operations performed during this data join can be directly modified via the custom `enter`, `update`, and `exit` events, which are dispatched on the root element of the Legend and which respectively contain the data join's enter, update, or exit selection in a property of the event object. Also, the `legendRender` function sets the `data-style` and `data-key` attributes on the legend item elements to values configured on the bound `Legend` data object.

6.3 Tooltip Module

Tooltips display additional contextual information which is too expansive to be shown all the time. The more information a visualization shows simultaneously, the more cognitive effort is required to interpret it. Detailed information which may be valuable but is not absolutely necessary for understanding a visualization's core message is only shown via a Tooltip when the user interacts with an element in whose context this information stands. Since Tooltips are only shown explicitly after user interaction, they may overlap and cover other elements of a Chart which would otherwise be too important to hide. An example of a Bar Chart in which additional information is displayed via a Tooltip can be seen in Figure 6.8.

The Tooltip Module is located in the `src/lib/tooltip/` directory of the RespVis library and contains the implementation of Tooltips and utility functions to simplify the configuration of Tooltips on Series Components. Tooltips are merely HTML `<div>` elements with the `tooltip` class, which are styled via CSS and which can be initialized with the `tooltip` function. Their visibility, position, and content is controlled via the `tooltipShow`, `tooltipHide`, `tooltipPosition`, and `tooltipContent` functions in the `tooltip.ts`

file of the Tooltip Module. Since RespVis supports multiple simultaneous Tooltips, all of these functions have to be passed the Tooltip element they should affect. Alternatively, passing a `null` value operates on the default Tooltip. Tooltips are shown and hidden via the `tooltipShow` and `tooltipHide` functions, which respectively add or remove the `show` class to the classes of the Tooltip element. The actual showing and hiding is done in CSS by setting the `opacity` property of the element depending on whether or not the `show` class is set. Positions of Tooltips are configured via the `tooltipPosition` function. This function calculates positions through an anchor position in viewport coordinates and a directional offset from that anchor. Specifying the anchor position is necessary, but the directional offset can be omitted to have a sensible one chosen by the `tooltipPosition` function, so that the Tooltip is always placed inside the visible area of the browser. The actual positioning is again done in CSS by setting a combination of the `left`, `right`, `bottom`, and `top` properties depending on the offset direction of the Tooltip. A Tooltip's content can be set with the `tooltipContent` function, which sets the Tooltip's inner HTML to the HTML string passed to the function.

Apart from Tooltips, the Tooltip Module also contains utility functions to simplify the setup and handling of Tooltips on different Series Components such as Bar or Point Series. Interfaces describing the data objects of Series Components can inherit from the `SeriesConfigTooltips` interface to add additional properties for the configuration of Tooltips to these data objects. Among these properties is a property to enable the Tooltips of the Series and properties to specify the contents and positions of individual Tooltips based on their associated context-element and the current cursor position. In addition to the `SeriesConfigTooltips` interface, the `seriesConfigTooltipsHandleEvents` function is provided to automatically set `mouseover`, `mousemove`, and `mouseout` event listeners on Series to automatically update the visibility, content, and position of Tooltips based on the `SeriesConfigTooltips` properties stored in the Series' data object. The usage of these utilities is optional, and Series Components are free to provide their own properties for the configuration of Tooltips and their handling. However, for consistency reasons, the way of configuring Tooltips should not differ too much between different types of Series Components, and therefore it is recommended to use the utilities provided here unless there is a good reason not to.

6.4 Bar Module

A bar chart is used to compare the values of a quantitative variable (values) uniquely associated with the distinct values of a qualitative variable (categories), using labelled rectangles whose lengths are proportional to the values. Examples of datasets suitable for plotting with a bar chart would be countries with their populations, people with their ages, and web browsers with their market shares. Bar charts have been in use for many centuries, with one of the earliest documented occurrences being Playfair [1786], and are among the most frequently encountered types of visualizations in the modern web.

The bars of a bar chart can be either horizontally or vertically oriented. In horizontal bar charts, sometimes also called row charts, categories are spread in equal intervals across the y-axis to define the positions and heights of bars, and the associated values of categories are mapped on the x-axis to determine their widths. Conversely, in vertical bar charts, sometimes also called column charts, categories are spread in equal intervals across the x-axis to determine the positions and widths of bars, and the values of categories are mapped on the y-axis to determine their heights. Horizontal bar charts are better suited for display in narrow contexts than vertical bar charts, because category labels can be positioned more easily without having to rotate them and because these charts can be vertically extended by having the user scroll vertically, which is strongly preferable to scrolling horizontally.

The Bar Module is located in the `src/lib/bars/` directory of the RespVis library and contains components to create Single-Series, Grouped Multi-Series, and Stacked Multi-Series Bar Charts that can be used in different situations. For every type of Bar Chart, the Bar Module provides a respective Series Component to render only the actual bars, a Chart Component to render a full Chart including

bars, axes, and possible legends, and a Chart Window Component to render a full Chart embedded into a Layouter with additional tools provided via a Toolbar. Higher-level components are more convenient to use, but they also impose more assumptions, and therefore restrictions, on lower-level components contained within them. The implementations of the different types of Bar Charts and when to best use which type is described in the following sections.

6.4.1 Single-Series Bars

Single-series bar charts are what most people think of when thinking of bar charts. They consist of only a single series of bars and can therefore only visualize differences between categories associated with a single quantitative value at a time. The categories of a single-series bar chart are mapped to spatial dimensions via a band scale which partitions available space into equal intervals (bands) with configurable padding between them. Thus, the width of bars is calculated via the total number of categories, the range they are spread on, and the padding between them. The mapping of quantitative values to spatial dimensions in a single-series bar chart is performed using a continuous scale which maps quantitative values stored in a dataset into a range of values between two extremes via a continuous interpolation function. In most cases, linear interpolation via a linear scale is used to map quantitative values, but visualization authors can choose other forms of interpolation, such as logarithmic interpolation via a logarithmic scale.

In RespVis, the Bar Series module renders collections of `<rect>` elements representing the bars in a Bar Chart's draw area, and they are the lowest-level components required for rendering Bar Charts. Their implementation is located in the `series-bar.ts` file of the Bar Module and contains the `SeriesBar` interface describing data objects to configure Bar Series, the `seriesBarData` function to create these data objects from partial data objects, and the `seriesBarRender` function to render Bar Series. The `SeriesBar` interface inherits Tooltip configuration properties from the `SeriesConfigTooltips` interface described in Section 6.3 and adds additional properties to configure the used categories and values, the scales to map them to spatial dimensions, whether or not bars should be oriented vertically or horizontally, and properties to configure their colors and keys. After the desired `SeriesBar` data objects have been created and bound to `<svg>` or `<g>` elements, the `seriesBarRender` function can be used to render Bar Series on these elements. During rendering, the bound `SeriesBar` data object is transformed into an array of `Bar` data objects, and a data join with these data objects is performed to render the individual bars. The output range of the scales used to map categories and values to spatial dimensions is set to the dimensions of the Bar Series element's bounding box that has been previously calculated and stored in the `bounds` attribute by the Layouter. With these scales, the dimensions of bars are calculated, and enter, update, and exit transitions are utilized to interpolate them towards these new dimensions, so that changes in visualization are easier to track, which leads to an improved experience for users. As with all other Series, the `enter`, `update`, and `exit` events containing the respective data join selections are dispatched on the Series' root element and allow the injection of custom behavior into the different phases of the Series' data join.

Bar Charts are Cartesian Charts, as discussed in Section 6.1.4, which display a Bar Series with optional labels in their draw area and render the scales used to map categories and values to spatial dimensions as Axes. Their implementation can be found in the `chart-bar.ts` file of the Bar Module, which contains the `ChartBar` interface to describe data objects used for the configuration of Bar Charts, the `chartBarData` function to create a fully initialized `ChartBar` data object from a partial input object, and the `chartBarRender` function to render Bar Charts as `<svg>` or `<g>` elements to which `ChartBar` data objects have been bound. The `ChartBar` interface inherits all properties of the `ChartCartesian` and `SeriesBar` interfaces and adds additional properties for the configuration of bar labels. It is not necessary to manually specify all the offered properties of this interface because the `chartBarData` function initializes all of them with sensible defaults derived from the properties callers are interested in and which have been specified in the partial data object passed to this function. The `chartBarRender` function simply initializes a Cartesian Chart and renders a Bar Series whose configuration is derived

```
select('#chart')
  .append('div')
  .datum(
    chartWindowBarData({
      categories: ['A', 'B', 'C', 'D', 'E', 'F'],
      values: [2, 4, 3, 1, 5, 2],
      xAxis: { title: 'Category' },
      yAxis: { title: 'Value' },
    })
  )
  .call(chartWindowBarRender)
  .call(chartWindowBarAutoResize)
  .call(chartWindowBarAutoFilterCategories);
```

Listing 6.4: The example source code to create the Bar Chart Window shown in Figure 6.9. This Bar Chart Window is configured with the bound data object initialized with the `chartWindowBarData` function and rendered with the `chartWindowBarRender` function. Since no special responsive behavior is desired in this example, the default resize and category filter behavior is attached to the Chart Window via the `chartWindowBarAutoResize` and `chartWindowBarAutoFilterCategories` functions.

from the `ChartBar` data object bound to the Chart's element and an optional series of labels to annotate the bars of the Bar Series into the Cartesian Chart's draw area. Furthermore, the scales used for rendering the Bar Series are visualized as Left and Bottom Axes of the Cartesian Chart, and a `mouseover` event listener to highlight bars and their corresponding ticks on the category axis is attached to bars.

Bar Chart Windows are high-level components which wrap a Bar Chart into a Layouter so its elements can be laid out with CSS. They also add a Toolbar containing tools to filter a Bar Chart's categories and download an SVG version of the current chart. Their implementation can be found in the `chart-window-bar.ts` file of the Bar Module, which contains the `ChartWindowBar` interface to describe data objects for the configuration of Bar Chart Windows, the `chartWindowBarData` function to create a fully initialized `ChartWindowBar` data object from a partial input object containing only relevant properties, and the `chartWindowBarRender` function to render a Bar Chart Window as a `<div>` element onto which a `ChartWindowBar` data object has been bound. The `ChartWindowBar` interface inherits all properties of the `ChartBar` interface to configure the wrapped Bar Chart and adds additional properties needed for the filtering of categories. Bar Chart Windows are rendered with the `chartWindowBarRender` function, which renders a Toolbar including a Nominal Filtering Tool for categories and an SVG Download Tool, initializes the nested Bar Chart with the filtered values, and renders it, according to the render process required by the Layouter defined in Section 6.1.2.

By default, Bar Chart Windows are not automatically updated when the viewport size changes or if their categories are filtered. Instead, a custom `resize` event listener is attached to the Chart Window's element, in which the Chart Window is responsively reconfigured according to the changed viewport size and subsequently rerendered via the `chartWindowBarRender` function. Furthermore, Bar Chart Windows dispatch custom `categoryfilter` events containing the currently active categories whenever categories are activated or deactivated via the Nominal Filtering Tool. These events can be used to responsively reconfigure Bar Chart Windows according to the currently active categories by adding a custom `categoryfilter` event listener which updates the configuration of the Bar Chart Window accordingly and rerenders it. If no special configuration is required when the viewport size or the category filtering changes, then the `chartWindowBarAutoResize` and `chartWindowBarAutoFilterCategories` functions

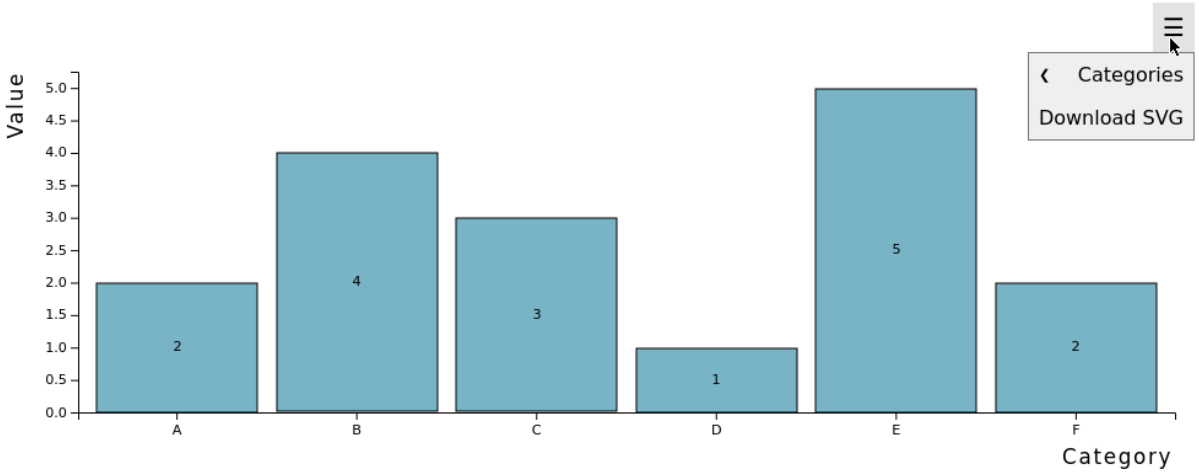


Figure 6.9: The Bar Chart Window resulting from the example code in Listing 6.4. [Image created by the author of this thesis using RespVis.]

can be used to attach custom `resize` and `categoryfilter` event listeners to implement the default behavior. A simple example of how a scalable Bar Chart Window with filterable categories can be created is shown in Listing 6.4 and the resulting visualization is shown in Figure 6.9.

6.4.2 Grouped Bars

Grouped bar charts, sometimes also called clustered or multi-series bar charts, consist of multiple series of bars and are used to visualize differences between categories where each category is associated with a number of quantitative variables. Every category must be associated with the same number of quantitative variables, effectively splitting categories into subcategories, and all values assigned to these subcategories must be comparable with one another, meaning they must have the same units and scales. As with single-series bar charts, categories in grouped bar charts are also mapped to spatial dimensions using band scales, but here, two different band scales have to be applied. The category band scale divides the available space along either the x or y-axis into equal intervals based on the number of categories, whereas the subcategory band scale further divides these intervals into even smaller intervals based on the number of subcategories. Quantitative values are again mapped to spatial dimensions via a single continuous scale which must be used for all of them.

In RespVis, the Bar Module provides the Grouped Bar Series, Grouped Bar Chart, and Grouped Bar Chart Window Components, which are very similar to their Single-Series Bar Chart counterparts, to render grouped bar charts in different levels of hierarchy. Grouped Bar Series are the lowest-level components of these and render a collection of `<rect>` elements meant for display in the draw area of a Grouped Bar Chart. The difference between a Grouped Bar Series and a Single-Series Bar Series is that this one offers additional properties needed for the configuration of subcategories and that other properties specified as one-dimensional arrays in Single-Series Bar Series are here required to hold two-dimensional arrays so that their values can be assigned to the two-dimensionally grouped bars. All bars belonging to the same subcategory have the same style to simplify their comparison across all categories. Grouped Bar Charts are Cartesian Charts which contain a Grouped Bar Series with optional labels in their draw area, Left and Bottom Axes that visualize the scales used by this Grouped Bar Series, and a Legend that describes the different subcategories. These Charts also attach event listeners to various elements to highlight bars, labels, ticks on the category axis, and Legend items that semantically belong together.

Grouped Bar Chart Windows are the highest-level components to render visualizations based on Grouped Bars. They contain Grouped Bar Charts wrapped inside Layouters and render them following the render process defined in Section 6.1.2 to allow positioning of their elements via CSS.

```
select('#chart')
  .append('div')
  .datum(
    chartWindowBarGroupedData({
      categories: ['A', 'B', 'C'],
      subcategories: ['X', 'Y', 'Z'],
      values: [
        [2, 3, 4],
        [3, 4, 2],
        [2, 1, 3],
      ],
      xAxis: { title: 'Category' },
      yAxis: { title: 'Value' },
      legend: { title: 'Subcategories' },
    })
  )
  .call(chartWindowBarGroupedRender)
  .call(chartWindowBarGroupedAutoResize)
  .call(chartWindowBarGroupedAutoFilterCategories)
  .call(chartWindowBarGroupedAutoFilterSubcategories);
```

Listing 6.5: Example source code to create the Grouped Bar Chart Window shown in Figure 6.10. The Grouped Bar Chart Window is configured via a bound data object which is initialized with the `chartWindowBarGroupedData` function and rendered with the `chartWindowBarGroupedRender` function. Since no special responsive behavior is desired in this example, the default resize, category filter, and subcategory filter behavior is attached to the Chart Window via the `chartWindowBarGroupedAutoResize`, `chartWindowBarGroupedAutoFilterCategories`, and `chartWindowBarGroupedAutoFilterSubcategories` functions.

Furthermore, they decorate their nested Charts with Toolbars containing tools to download them as SVG documents and filter their categories and subcategories. Every time the user interacts with these Nominal Filtering Tools to change the configuration of displayed categories and subcategories, the `categoryfilter` and `subcategoryfilter` events are dispatched, containing the newly active categories and subcategories respectively. Visualization authors can either implement special resize and filter behavior by attaching custom listeners to these events, or can activate the default behavior via the `chartWindowBarGroupedAutoResize`, `chartWindowBarGroupedAutoFilterCategories`, and `chartWindowBarGroupedAutoFilterSubcategories` functions. Example code to create a scalable Grouped Bar Chart Window whose categories and subcategories can be filtered via the Toolbar is shown in Listing 6.5 and the resulting visualization is shown in Figure 6.10.

6.4.3 Stacked Bars

Stacked bar charts are multi-series bar charts in which individual series of bars are rendered as stacks rather than as clusters. Since the bars of stacked bar charts do not share a common baseline, these charts are not well-suited for comparing individual subcategories across multiple categories but rather for approximating the contributions of subcategories to the totals of their categories. A bar's color is determined by its subcategory so that all bars that have the same subcategory are colored equally. Percent stacked bar charts are variants of stacked bar charts, in which the totals of all categories are transformed to equal 100% using the quantitative values of subcategories as ratios of a whole. Due to this transformation, all stacks of bars are of equal length, meaning the information about their totals is lost, but the share subcategories contribute to their categories is emphasized more strongly than in ordinary stacked bar

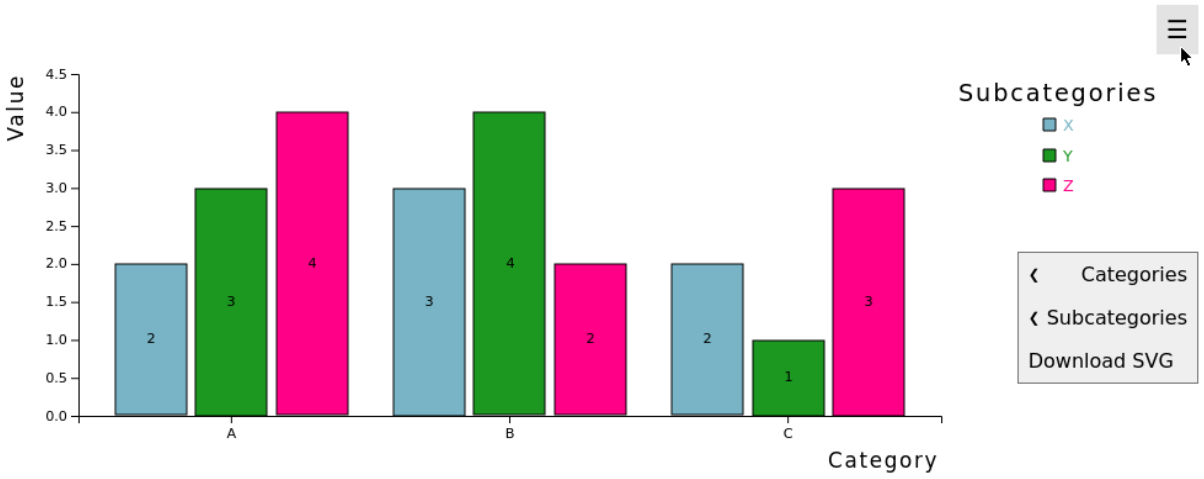


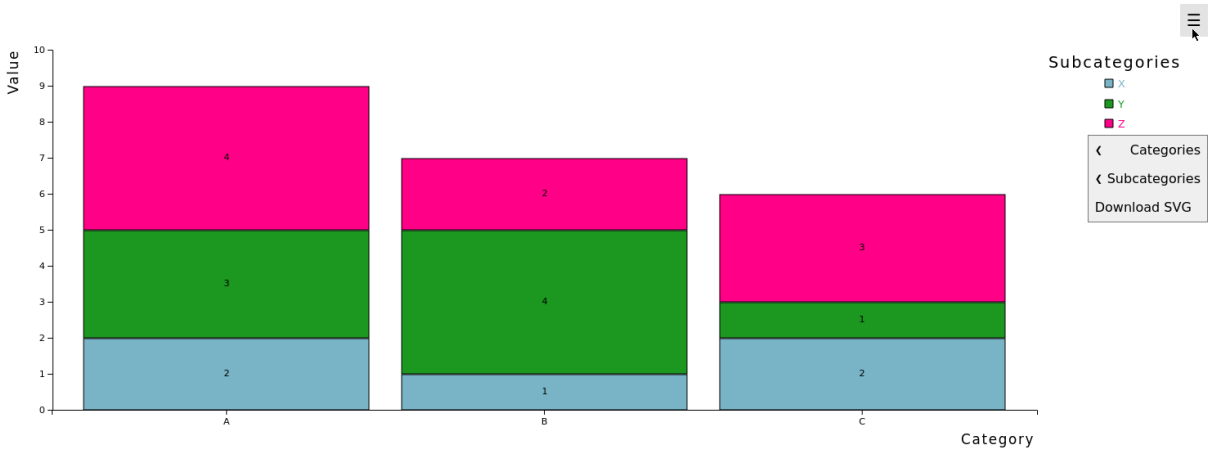
Figure 6.10: The Grouped Bar Chart Window resulting from the example source code in Listing 6.5. The Tool Menu popup has been manually displaced to not cover the legend. [Image created by the author of this thesis using RespVis.]

charts. Bar stacks are created by splitting the category axis into equal intervals via a band scale into which bars representing the different subcategories are stacked on top of one another. The lengths of stacked bars are proportional to their quantitative values and are calculated via a continuous scale which maps these values into the range of the value axis.

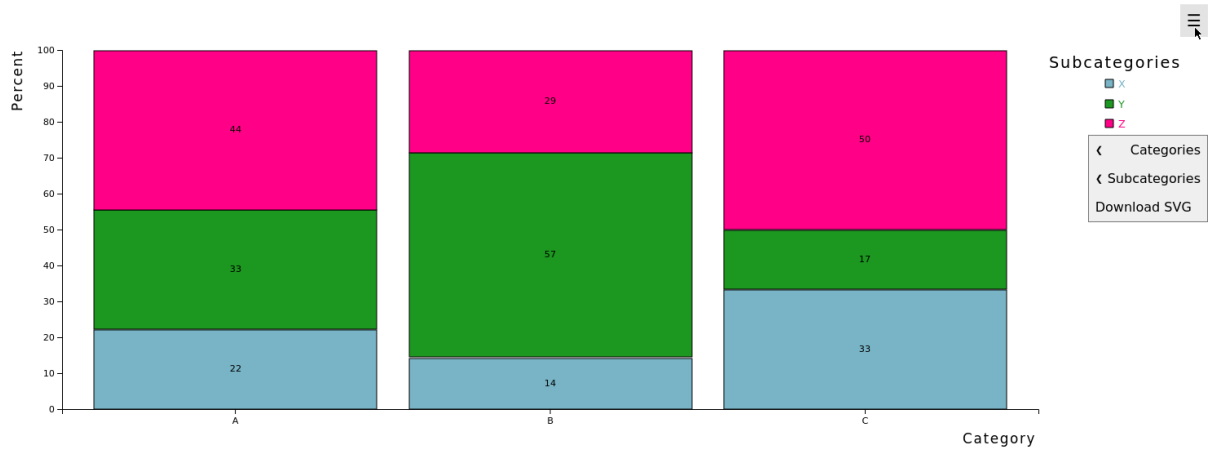
The RespVis library provides a Series, a Chart, and a Chart Window Component to render Stacked Bar Charts, all of which are highly similar to their Grouped Bar counterparts. Bars in Stacked Bar Series are grouped two-dimensionally (category/subcategory), and configuration properties affecting individual bars are specified as two-dimensional arrays. The main difference between Stacked Bar Series and Grouped Bar Series lies in how the positions and extents of bars are calculated. Stacked Bar Charts are, just like Grouped Bar Charts, Cartesian Charts consisting of a Stacked Bar Series with optional labels, two Axes, and a Legend. Furthermore, Stacked Bar Chart Windows are also equivalent to Grouped Bar Chart Windows as they wrap a Stacked Bar Chart into a Layouter, manage their render process, and decorate it with a Toolbar containing two Nominal Filtering Tools to filter categories and subcategories. Percent Stacked Bar Charts can be rendered either by directly setting percentual values that lead to all category totals summing up to 100% or by enabling the `valuesAsRatios` configuration property, which causes quantitative values to be treated as ratios which are transformed to percentual shares of their category totals during rendering. Listing 6.6 shows example RespVis code to create a scalable Stacked Bar Chart Window whose categories and subcategories can be filtered, the chart itself is shown in Figure 6.11.

```
select('#chart')
  .append('div')
  .datum(
    chartWindowBarStackedData({
      categories: ['A', 'B', 'C'],
      subcategories: ['X', 'Y', 'Z'],
      values: [
        [2, 3, 4],
        [1, 4, 2],
        [2, 1, 3],
      ],

      // to turn into a percent stacked bar chart
      // valuesAsRatios: true,

      xAxis: { title: 'Category' },
      yAxis: { title: 'Percent' },
      legend: { title: 'Subcategories' },
    })
  )
  .call(chartWindowBarStackedRender)
  .call(chartWindowBarStackedAutoResize)
  .call(chartWindowBarStackedAutoFilterCategories)
  .call(chartWindowBarStackedAutoFilterSubcategories);
```

Listing 6.6: Example source code to create the Stacked Bar Chart Window shown in Figure 6.11a. The Stacked Bar Chart Window is configured with the bound data object initialized via the `chartWindowBarStackedData` function and rendered with the `chartWindowBarStackedRender` function. Since no special responsive behavior is desired in this example, the default resize, category filter, and subcategory filter behavior is attached to the Chart Window via the `chartWindowBarStackedAutoResize`, `chartWindowBarStackedAutoFilterCategories`, and `chartWindowBarStackedAutoFilterSubcategories` functions. Setting the `valuesAsRatios` variable to `true` would result in the Percent Stacked Bar Chart shown in Figure 6.11b.



(a) Stacked Bar Chart



(b) Percent Stacked Bar Chart

Figure 6.11: The Stacked Bar Chart and Percent Stacked Bar Chart rendered by the source code in Listing 6.6. The Tool Menu popup has been manually displaced to not cover the legend. (a) An ordinary Stacked Bar Chart to compare category totals and subcategory contributions to these totals. (b) A Percent Stacked Bar Chart to better compare subcategory contributions to category totals. [Image created by the author of this thesis using RespVis.]

```
select('#chart')
  .append('div')
  .datum(
    chartWindowPointData({
      xValues: [10, 50, 20, 15, 35, 15, 35, 25, 45],
      yValues: [1.5, 4, 3, 3.5, 4.5, 2.5, 4, 4, 4.5],
      xAxis: { title: 'X Values' },
      yAxis: { title: 'Y Values' },
    })
  )
  .call(chartWindowPointRender)
  .call(chartWindowPointAutoResize);
```

Listing 6.7: Example source code to create the Point Chart Window shown in Figure 6.12. The Point Chart Window is configured with a bound data object initialized with the `chartWindowPointData` function and rendered with the `chartWindowPointRender` function. Since no special responsive behavior is desired in this example, the default resize behavior is attached to the Chart Window via the `chartWindowPointAutoResize` function.

6.5 Point Module

Point charts, also commonly known as scatter charts or scatter plots, show the relationship between two variables by plotting them as points in a cartesian coordinate system. Each of the two variable determines a point's position on one of the coordinate system's two Axes, and typically, these variables contain quantitative data. However, the data type of the variables used to position points is not relevant, as long as their values can be mapped to spatial dimensions via scales. Point charts are particularly suitable for discovering potential correlations and patterns between variables. They can also visualize more than two variables simultaneously by encoding additional variables via the colors, sizes, and shapes of the plotted points. Point charts which encode an additional variable as the sizes of points are called bubble charts.

The Point Module is located in the `src/lib/point/` directory of the RespVis library and contains the implementation of Point Series, Point Charts, and Point Chart Windows. Point Series render a collection of `<circle>` elements, whose center positions are calculated by mapping arrays of X and Y values into the bounding boxes of the Series' root elements via X and Y scales. The colors of points are specified as style classes, and their radiuses can be configured, meaning that it is also possible to create Bubble Charts with this implementation. As with other Series, individual elements of Point Series are rendered via a data join using an array of data objects created through transforming the bound Series data object, and this data join can be customized by providing custom `enter`, `update`, and `exit` event listeners. Point Charts are Cartesian Charts which contain Point Series in their draw areas and two Axes visualizing the scales used to render the Point Series.

Point Chart Windows are Chart Windows which contain Point Charts nested into Layouters to handle the render process of these Charts so that their elements can be laid out with CSS, and decorate them with a Toolbar. Currently, the Toolbars of Point Chart Windows only contain SVG Download Tools because only a limited number of Tools, not suited for application on Point Charts, have been developed so far. Further Tools that can also be applied to Point Charts, such as Zoom and Quantitative Filter Tools, are planned for development, and once these are available, they will be added to the default Tools attached to the Toolbars of Point Chart Windows. Example source code to create a scalable Point Chart Window can be found in Listing 6.7, and the resulting visualization is shown in Figure 6.12.

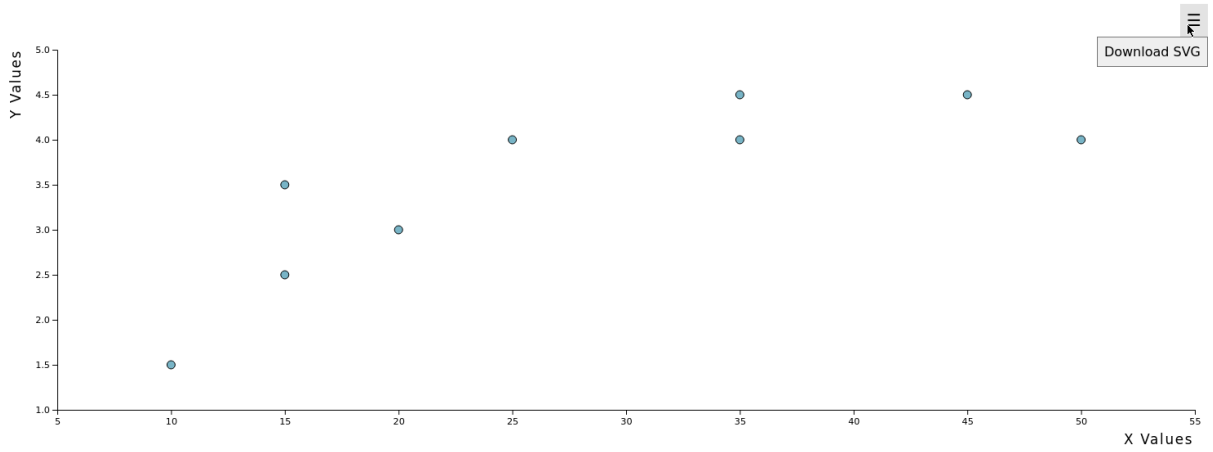


Figure 6.12: The Point Chart Window resulting from the source code in Listing 6.7. [Image created by the author of this thesis using RespVis.]

Chapter 7

RespVis Usage

This chapter discusses how different responsive patterns can be implemented using the components provided by the various RespVis packages outlined in Chapter 6. The `src/examples` directory of the RespVis library contains examples showing how the library can be used to create different kinds of charts with varying degrees of responsiveness. Users can compose custom charts with low-level components like Series, Axes, and Legends. However, all the examples in this chapter focus on creating responsive visualizations using higher-level Chart Windows. Chart Windows represent convenient interfaces which allow visualization authors to focus on responsive configuration, rather than on laborious tasks like setting up a Chart's structure and handling the render processes of their nested components.

All the example visualizations provided by the RespVis library follow the same basic structure, outlined in Listing 7.1, namely:

1. Import the RespVis CSS file `respvis.css`. This file contains the necessary default styling of visualizations rendered by the RespVis library.
2. Import D3 and RespVis. RespVis is a D3 extension library, and therefore the functionality of both libraries needs to be imported from either IIFE or ES modules to create visualizations with RespVis.
3. Attach a new `<div>` element to the root `<div>` element (the one with `id="chart"`) [Line 7].
4. Bind a fully-initialized data object to the `<div>` element, containing the render configuration of the Chart Window. This data object is usually created by deriving default properties from a partial object via one of the `chartWindowXData` functions.
5. Render the Chart Window using the appropriate `chartWindowXRender` function.
6. Attach a `resize` event listener to the `<div>` element. This event listener should update the Chart Window's bound data object based on media queries and rerender it. Theoretically, it is possible to use the actual viewport size in pixels for responsive configuration decisions, but it is strongly recommended to use media queries via the `window.matchMedia` function instead. Using media queries allows a Chart's JavaScript configuration to be based on the same media queries that might be used for the CSS configuration of the same Chart.

Since one of the core premises of RespVis is to enable the configuration of SVG-based visualizations with CSS, many responsive patterns can be implemented without JavaScript. In general, everything which does not affect the content or behavior of a Chart should be handled in CSS, including the configuration of presentation attributes and the layout of laid-out elements. Configuration changes which affect a Chart's content or behavior, like changing the visualized data, texts, or interaction mechanisms, still need to be applied in JavaScript. Knowing what kind of configuration is better done in CSS or JavaScript is not immediately obvious and can be confusing to figure out for developers unfamiliar with RespVis. There

```
<html>
<head>
  <link rel="stylesheet" href="./path/to/respvis.css" />
  <style>/* user styling */</style>
</head>
<body>
  <div id="chart"></div>
</body>
<script src="./path/to/d3.js"></script>
<script type="module">
  import { chartWindowXData, chartWindowXRender } from './path/to/respvis.mjs';
  const data = {/* ... */};
  const chartWindow = d3.select('#chart')
    .append('div')
    .datum(chartWindowXData(data))
    .call(chartWindowXRender)
    .on('resize', function () {/* ... */});
</script>
</html>
```

Listing 7.1: The basic structure of all responsive examples provided by RespVis. Some parts have been removed, so as not to distract the essential parts.

are plans to allow configuring even more in CSS, but this would require extensive redesign and refactoring and, therefore, will be considered for a future release of the library.

7.1 Axes

Axes are used to visualize the spatial mapping of abstract values, by rendering abstract values as ticks at the appropriate spatial positions to which they are being mapped. In addition to ticks, an Axis also contains an optional title and an optional subtitle to describe the axis. Axis-related responsive patterns are of great importance, since nearly every Chart includes Axes, and often improving their responsiveness alone can already significantly improve a user's experience.

The following responsive patterns can be applied to Axes in RespVis visualizations:

- *Rotate tick labels*: One of the most effective ways to prevent tick labels from overlapping is rotating them by up to 90 degrees. All the available information is preserved and only presented differently. Tick labels can be rotated by setting a rotation in the CSS `transform` property on them and modifying their CSS `text-anchor` property accordingly.

- *Simplify tick labels*: In some cases, rotating tick labels may not be desired, or labels might still overlap after rotation. The next best thing in these cases is to shorten the tick labels, if shorter textual representations exist. The D3 Axis object used for rendering ticks is accessible via the `configureAxis` function property on an Axis' data object. How exactly tick labels are shortened is specified as a formatting callback set via the D3 Axis' `tickFormat` function.

- *Remove ticks*: If neither rotation nor shortening of tick labels is applicable, the last thing that can be done is to reduce the number of ticks shown. This can be achieved either via a D3 Axis' `ticks` or `tickValues` function, or via the CSS `display` property. A D3 Axis' `ticks` function allows specifying the desired number of ticks that shall be rendered, and the D3 Axis' render function decides how many ticks to create based on this number and other contributing factors. The `tickValues` function

of a D3 Axis allows for much more control than the `ticks` function, because it is used to specify the exact values for which ticks shall be rendered.

- *Simplify title/subtitle*: Since Axes do not just contain ticks, but also optional titles and subtitles, these should not be ignored when optimizing the responsiveness of Axes. Titles and subtitles can be simplfied by specifying shorter text strings via the `title` and `subtitle` properties on Axes' data objects.
- *Relocate title/subtitle*: The titles and subtitles of Axes can be relocated by modifying the Axes' grid layouts via the CSS `grid-template` property.
- *Remove title/subtitle*: The titles and subtitles of Axes can be hidden via the CSS `display` property.

7.2 Legends

Legends visualize the non-spatial mapping of abstract values such as mappings to colors, shapes, and sizes via labeled symbols. They are used to explain a visual encoding to a viewer and are frequently encountered. As such, Legends must not be ignored when optimizing a visualization's responsiveness.

The following responsive patterns can be applied to Legends in RespVis visualizations:

- *Relocate Legend*: A Legend's position in its containing Chart can be controlled by directly modifying the Chart's grid layout via the CSS `grid-template` property, or by positioning the Legend at predefined `"top"`, `"right"`, `"bottom"`, and `"left"` positions via the `data-legend-position` attribute.
- *Simplify title*: A Legend's title can be shortened by specifying a shorter text string for it via the `title` property on a Chart's data object.
- *Remove title*: If the title cannot be further shortened, or if it does not convey too much important information, it can be hidden via the CSS `display` property.
- *Simplify symbol labels*: If the symbol labels can be shortened, this can be done via the `labels` property on a Legend's data object. This property allows the specification of the text strings of all labels as an array of string values.
- *Relocate labeled symbols*: Changing the layout of labeled symbols is one of the most effective responsive patterns applicable to Legends. By default, labeled symbols are laid out using CSS Flexbox layouting. Thus, whether labeled symbols are laid out horizontally or vertically can be controlled using the CSS `flex-direction` property on their container element.

7.3 Bar Charts

Bar charts are used to compare categories associated with quantitative values by visualizing them as bars whose lengths depend on the quantitative dimension of the data. In the case of multi-series bar charts, like grouped bar charts and stacked bar charts, categories are further divided into subcategories associated with quantitative values and compared with one another rather than categories themselves. Bars in multi-series bar charts are usually colored based on their subcategories, which is why these charts often include legends to explain the color coding.

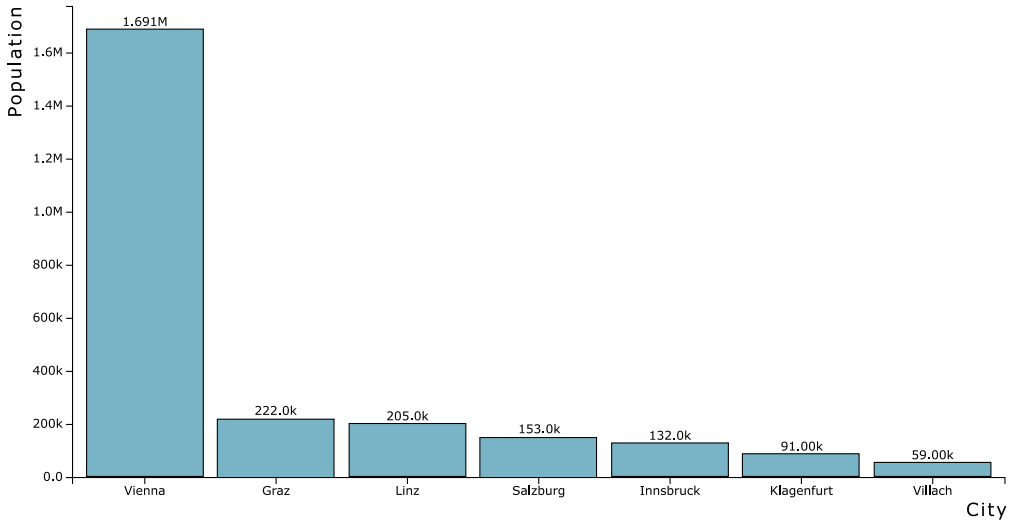
The different responsive patterns applicable to bar charts have already been discussed in Section 4.2.1, and the focus here lies on demonstrating their implementation using RespVis Bar Chart Windows. Listing 7.2 demonstrates how some of these patterns can be implemented to create the responsive Bar Chart shown in Figure 7.1. In practice, the responsive patterns which can be applied to different variants of Bar Charts are very similar, which is why this section does not differentiate between them.

The following responsive patterns can be applied to RespVis Bar Charts:

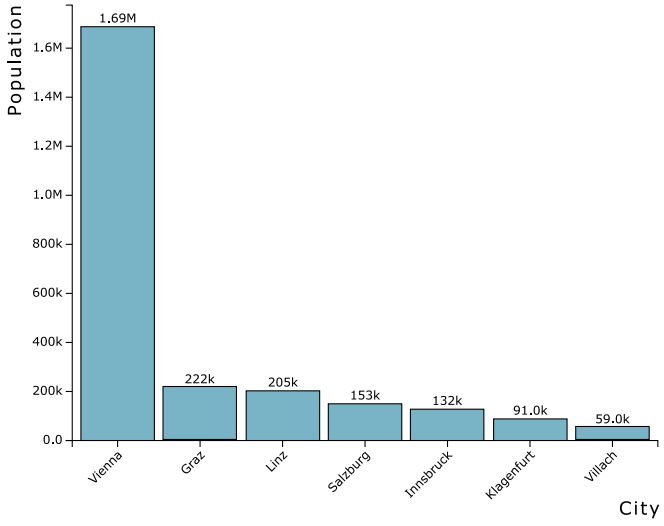
- *Rescale draw area*: Scaling a Bar Chart to fit into the available space is done automatically by their render functions. By default, Axes and Legends only take up as much space as necessary, and the remaining space is filled with the Chart's draw area. Bars and labels in this draw area are automatically scaled and positioned to fit into the allocated space.
- *Reduce bar padding*: Reducing the padding between bars frees up space that can be used by the bars themselves. Padding can be controlled via the `padding` function on the `categoryScale` property of a Bar Chart's data object.
- *Simplify bar labels*: When reducing the width of Bar Charts, bar labels might start overlapping one another, and it might be a good idea to shorten them, if possible. Bar labels can be configured using the `labels` property of a Bar Chart's data object.
- *Remove bar labels*: An alternative to prevent bar labels from overlapping is to hide some or even all of them. The best way of implementing this is to hide them via the CSS `opacity` or `display` properties.
- *Transpose Chart*: By default, a Bar Chart is rendered with vertical bars, transposing it would make the bars horizontal. Horizontal Bar Charts are better suited for narrow spaces, because the bar labels and Axis tick labels of the categories are easier to place. Furthermore, Horizontal Bar Charts are allowed to extend outside the visible viewport because vertical scrolling is more acceptable than horizontal scrolling. A Bar Chart can be transposed by setting the `flipped` property on its data object.
- *Remove bars*: Sometimes it might be required to reduce the number of bars to maintain a good user experience. Bars representing certain categories or subcategories can be hidden by declaring them as inactive via the `activeCategories` and `activeSubcategories` properties on a Bar Chart's data object. Using these properties is easier than manually adapting the `categories`, `values`, `categoryScale`, and `valueScale` properties on the data object. Furthermore, categories and subcategories which have been hidden via the `activeCategories` and `activeSubcategories` properties can also be unhidden by users via a Chart's Toolbar, if they wish to see the additional data.
- *Apply Axis responsive patterns*: Since a Bar Chart usually contains two Axes to visualize the spatial mappings of categories and values, all the Axis responsive patterns described in Section 7.1 can and should be applied to Bar Charts.
- *Apply Legend responsive patterns*: For a Multi-Series Bar Chart containing a Legend, the Legend responsive patterns mentioned in Section 7.2 can and should be applied.

```
<html>
<head>
  <link rel="stylesheet" href="./path/to/respvis.css" />
  <style>
    #chart { width: 100%; height: 75vh; min-height: 25rem; }
    /* rotate bottom axis tick labels on <60rem screens */
    @media (max-width: 60rem) {
      .axis-bottom .tick text { transform: rotate(-45deg); text-anchor: end; }
    }
  </style>
</head>
<body><div id="chart"></div></body>
<script src="./path/to/d3.js"></script>
<script type="module">
  import { /* ... */  } from './path/to/respvis.mjs';
  import { cities, populations } from './path/to/data/austrian-cities.js';

  const data = {
    categories: cities,
    values: populations,
    xAxis: { title: 'City' },
    yAxis: {
      title: 'Population',
      configureAxis: (a) => a.tickFormat(d3.format('.2s'))
    },
  };

  const chart = d3.select('#chart').append('div')
    .datum(chartWindowBarData(data))
    .call(chartWindowBarAutoFilterCategories)
    .call(chartWindowBarRender)
    .on('resize', handleResize);

  function handleResize() {
    const wide = window.matchMedia('(min-width: 40rem)').matches;
    const widest = window.matchMedia('(min-width: 60rem)').matches;
    // transpose chart on narrow screens
    data.flipped = !wide;
    // move labels from top to right of bars in transposed chart
    data.labels.relativePositions =  !wide ? { x: 1, y: 0.5 } : { x: 0.5, y: 0 };
    // gradually shorten labels by decreasing scientific notation precision
    const labelPrecision = !wide ? 2 : !widest ? 3 : 4;
    data.labels.labels = populations.map(d3.format(`.${labelPrecision}s`));
    chart.datum(chartWindowBarData(data)).call(chartWindowBarRender);
  }
</script>
</html>
```

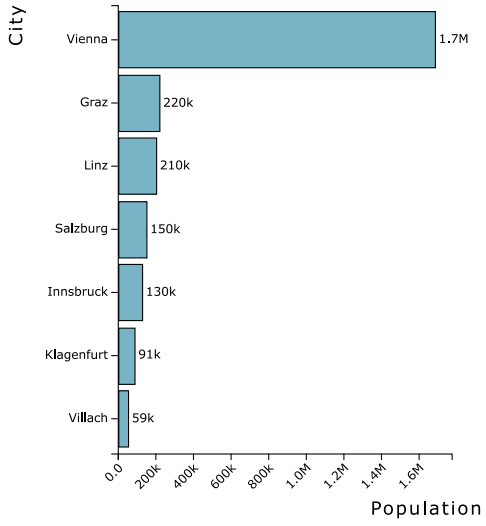
Listing 7.2: The implementation of the responsive Bar Chart shown in Figure 7.1. Depending on the available width, Axis tick labels are rotated, bar labels are simplified, and on very narrow screens, the whole Chart is transposed. Non-essential parts have been removed for clarity.



(a) 60rem



(b) 40rem



(c) 30rem

Figure 7.1: The responsive Bar Chart resulting from the implementation in Listing 7.2. (a) At a width of 60rem, the Chart is not yet transposed, bar labels are shown above the bars with four digits of precision, and the bottom axis tick labels are not rotated. (b) At a width of 40rem, bar labels are shown with three digits of precision, and the bottom axis tick labels are rotated by 45 degrees. (c) At a width of 30rem, the Chart is transposed, bar labels are shown with two digits of precision to the right of bars, and the (newly) bottom axis tick labels are rotated by 45 degrees. [Image created by the author of this thesis using RespVis.]

7.4 Line Charts

Line charts are visualizations which show trends in data by plotting markers at regular intervals and connecting them with lines. Single-series line charts show trends in two-dimensional data via a single line and multi-series line charts show trends in multi-dimensional data via multiple lines. The different lines in a multi-series line chart can be seen as representations of subcategories in the data and are usually colored differently to reflect this. For this reason, a multi-series line chart typically includes a legend to explain the color coding of subcategories.

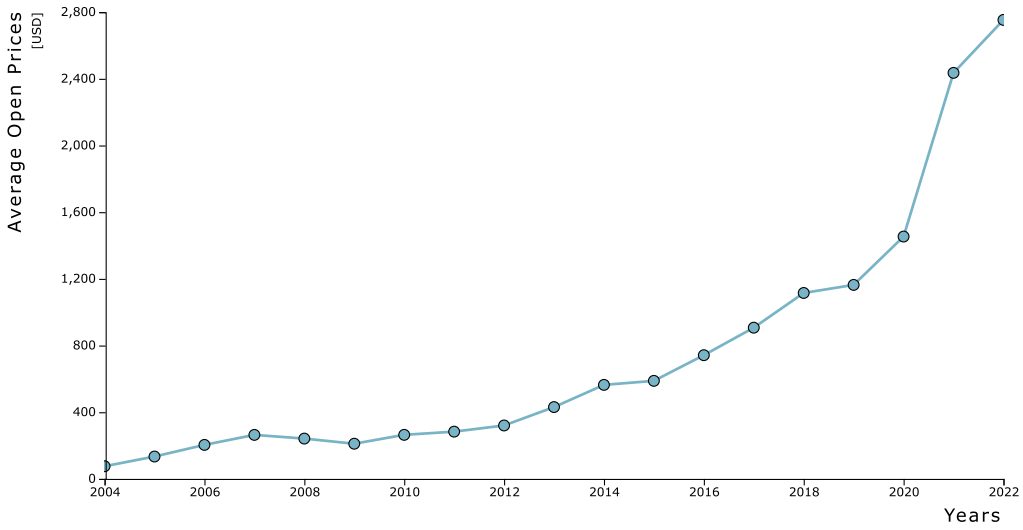
The responsive patterns applicable to line charts have already been discussed in Section 4.2.2. The focus here lies in demonstrating how they can be implemented using RespVis Line Chart Windows. Listing 7.3 demonstrates how some of these patterns can be combined to create the responsive Line Chart shown in Figure 7.2.

The following responsive patterns can be applied to RespVis Line Charts:

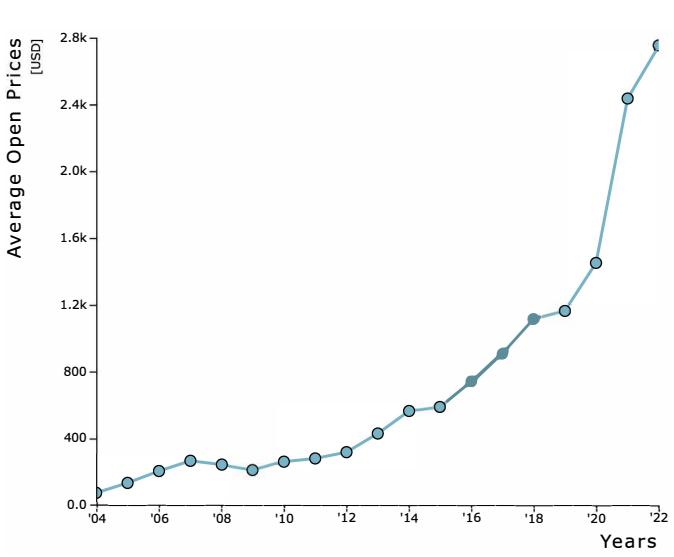
- *Rescale draw area*: As for Bar Charts, Line Charts are automatically scaled by their render functions. The default behavior is that Axes and potential Legends only take up as much space as necessary, and whatever space is left is filled with the Chart's draw area. Elements contained in the draw area like lines, markers, and labels are automatically scaled and positioned to fit into the available space.
- *Remove markers*: If a Chart contains a large number of individual data points, visual clutter can be reduced by not showing markers for every single data point. The best way to remove markers is to hide them via the CSS `opacity` or `display` properties.
- *Rescale markers*: An alternative to removing markers is to decrease their size. Marker sizes can be controlled via the `markers.radiuses` property on the data objects of Line Charts.
- *Simplify marker labels*: The labels of markers might start overlapping when trying to fit a Line Chart into increasingly narrow widths. A good solution for this is to shorten marker labels by providing shorter text strings for them via the `labels` property on the Line Chart's data object.
- *Remove marker labels*: Sometimes, shortening marker labels may not be possible, desired, or effective enough. An alternative to prevent them from overlapping is to remove some or all of them. The recommended way to remove marker labels is to hide them via the CSS `opacity` or `display` properties.
- *Transpose Chart*: Even though Line Charts are less frequently transposed than Bar Charts, there are still situations when transposing a Line Chart could improve the user's experience. One such situation would be a Line Chart with so many data points that a Horizontal Line Chart with limited width is too dense. Transposing such a chart into a Vertical Line Chart might reduce the data density by allowing the Chart to extend outside the visible viewport and having users scroll vertically. Furthermore, Vertical Line Charts are useful for heavily annotated Line Charts because labels are much easier to place than in horizontal ones. Line Charts can be transposed by setting the `flipped` property on their data objects.
- *Apply Axis patterns*: As with all charts which use axes to visualize spatial mappings, the responsive Axis patterns described in Section 7.1 can and should be applied to the axes in a Line Chart. The Axes of a Line Chart could even be removed completely via the CSS `display` property, turning the Line Chart into a Sparkline, but this should be considered carefully because information about the scale is lost when hiding axes.
- *Apply Legend patterns*: Where a Multi-Series Line Chart contains a Legend to explain a non-spatial data encoding, the responsive Legend patterns discussed in Section 7.2 can and should be applied.

```
<html>
<head>
  <link rel="stylesheet" href="./path/to/respvis.css" />
  <style>
    #chart { width: 100%; height: 75vh; min-height: 25rem; }
    /* show axis only on wide screens */
    .axis { display: none; }
    @media (min-width: 40rem) { .axis { display: block; } }
    /* show more ticks the wider the screen */
    .tick { display: none; }
    @media (min-width: 40rem) { .tick:nth-of-type(2n + 1) { display: block; } }
    @media (min-width: 60rem) { .tick { display: block; } }
    /* show only first and last marker on <40rem screens  */
    .point { opacity: 0; }
    .point:first-of-type, .point:last-of-type { opacity: 1; }
    @media (min-width: 40rem) { .point { opacity: 1; } }
    /* show first and last marker label only on <40rem screens  */
    .label:not(:first-of-type), .label:not(:last-of-type) { display: none; }
    @media (min-width: 40rem) { .label { display: none; } }
  </style>
</head>
<body><div id="chart"></div></body>
<script src="./path/to/d3.js"></script>
<script type="module">
  import { /* ... */  } from './path/to/respvis.mjs';
  import { years, averageOpens } from './path/to/data/google-stock.js';

  const data = {
    xValues: years,
    yValues: [averageOpens],
    xAxis: { title: 'Years' },
    yAxis: { title: 'Average Open Prices', subtitle: '[USD]' },
  };

  const chart = d3.select('#chart').append('div')
    .datum(chartWindowLineData(data))
    .call(chartWindowLineRender)
    .on('resize', handleResize);

  function handleResize() {
    const wide = window.matchMedia('(min-width: 40rem)').matches;
    const widest = window.matchMedia('(min-width: 60rem)').matches;
    // shorten price tick labels using scientific notation on <60rem screens
    const priceTickFormat = widest ? d3.format(',') : d3.format('.2s');
    data.yAxis.configureAxis = (axis) => axis.tickFormat(priceTickFormat);
    // shorten year tick labels on <60rem screens
    const yearTickFormat = widest ? (v) => v : (v) => `'${v.slice(-2)}`;
    data.xAxis.configureAxis = (axis) => axis.tickFormat(yearTickFormat);
    chart.datum(chartWindowLineData(data)).call(chartWindowLineRender);
  }
</script>
</html>
```

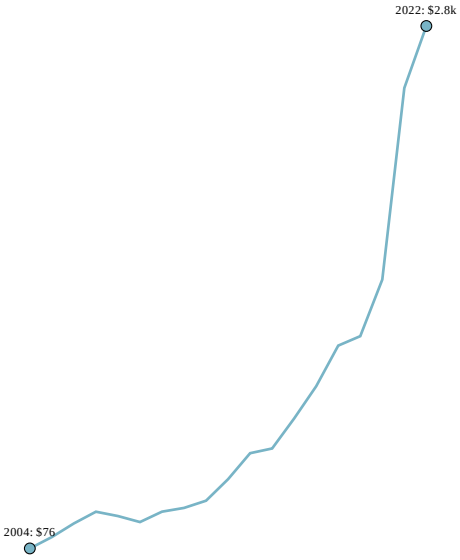
Listing 7.3: The implementation of the responsive Line Chart shown in Figure 7.2. Depending on the available width, Axis ticks and markers are hidden, Axis tick labels are simplified, and on very narrow screens, Axes are hidden turning the Line Chart into a Sparkline. Non-essential parts of the implementation have been removed for clarity.



(a) 60rem



(b) 40rem



(c) 30rem

Figure 7.2: The responsive Line Chart resulting from the implementation in Listing 7.3. (a) At a width of 60rem, all markers are shown and tick labels are not shortened. (b) At a width of 40rem, tick labels are shortened. (c) At a width of 30rem, axes are hidden and only the first and last markers and marker labels are shown, turning the chart into a sparkline. [Image created by the author of this thesis using RespVis.]

7.5 Point Charts

Point charts, sometimes also called scatterplots, are used to discover patterns and correlations in data by plotting individual data points as points in a cartesian coordinate system. Often, point charts are only used to visualize two-dimensional data by plotting points of the same color and size, but they can also be used to visualize multi-dimensional data by adding color, size, or shape encodings. If such additional encodings are used, a point charts usually includes a legend to explain them.

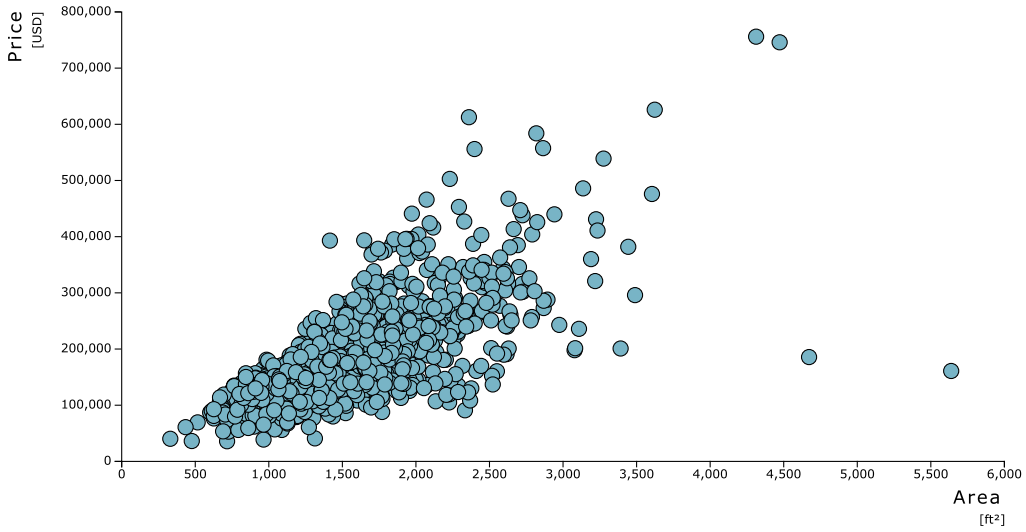
The responsive patterns applicable to point charts have already been discussed in Section 4.2.3. The focus here lies on demonstrating their implementation using RespVis Point Chart Windows. Listing 7.4 shows how some of these patterns can be implemented to create the responsive Point Chart displayed in Figure 7.3.

The following responsive patterns can be applied to RespVis Point Charts:

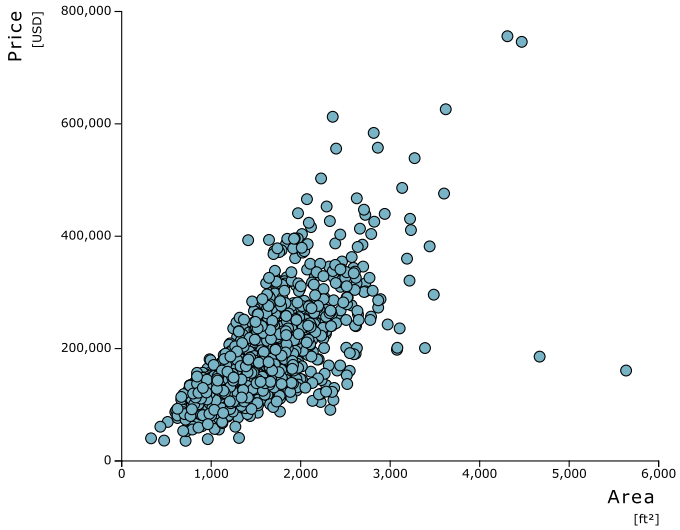
- *Rescale draw area*: As with other Cartesian Charts, the draw area of a Point Chart and its contents are automatically scaled by the Chart's render functions to fill the space not occupied by the Axes and any potential Legend.
- *Rescale points*: When a large number of data points are rendered in a Point Chart, it can be helpful to reduce their sizes to make individual points easier to see and reduce clutter. The sizes of points can be controlled via the `radiuses` property on the data objects of Point Charts.
- *Simplify labels*: When labels are shown in a Point Chart, it might be helpful to shorten them when reducing the width of Charts to prevent them from overlapping. The texts of labels can be configured via the `labels` property on the data object of the Point Chart.
- *Remove labels*: Often, Point Charts are used to visualize large data sets. In such situations, rendering labels for all points would only clutter the visualization, and it might be better to hide them via the CSS `opacity` or `display` properties.
- *Zoom draw area*: Zooming allows users to view data at various levels of detail. A Point Chart can be zoomed by adjusting the domains of the scales handling the spatial encoding of data, which are the `xScale` and `yScale` properties of the data object of the Point Chart. To simplify the setup and handling of zooming interaction, the D3 Zoom Module [Bostock 2022c] can be used, which is illustrated in Listing 7.4.
- *Apply Axis patterns*: As with all Cartesian charts which use Axes to visualize spatially-encoded data, the Axis responsive patterns described in Section 7.1 can and should be applied to Point Charts.
- *Apply Legend patterns*: Where a Point Chart visualizes additional non-spatial data encodings with a Legend, the Legend responsive patterns discussed in Section 7.2 can and should be applied.

```
<html>
<head>
  <link rel="stylesheet" href="./path/to/respvis.css" />
  <style>
    #chart { width: 100%; height: 75vh; min-height: 25rem; }
    /* show more ticks the wider the screen */
    .tick { display: none; }
    .tick:nth-of-type(2n + 1) { display: block; }
    @media (min-width: 60rem) { .tick { display: block; } }
  </style>
</head>
<body><div id="chart"></div></body>
<script src="./path/to/d3.js"></script>
<script type="module">
  import { /* ... */  } from './path/to/respvis.mjs';
  import { areas, prices } from './path/to/data/houses.js';
  const areaScale = d3.scaleLinear().domain([0, Math.max(...areas)]).nice();
  const priceScale = d3.scaleLinear().domain([0, Math.max(...prices)]).nice();
  const data = {
    xValues: areas, xScale: areaScale, yValues: prices, yScale: priceScale,
    xAxis: { /* ... */ }, yAxis: { /* ... */ }
  };
  const chartWindow = d3.select('#chart').append('div')
    .datum(chartWindowPointData(data)).call(chartWindowPointRender)
    .on('resize', handleResize);
  // set up zooming on draw area with minimum 1x and maximum 20x magnification
  const zoom = d3.zoom().scaleExtent([1, 20]).on('zoom', handleZoom);
  const drawArea = chartWindow.selectAll('.draw-area').call(zoom);

  function handleResize() {
    // limit panning by setting zoom extents to draw area bounds
    const { width, height } = rectFromString(drawArea.attr('bounds'));
    const extent = [[0, 0], [width, height]];
    zoom.extent(extent).translateExtent(extent);

    const wide = window.matchMedia('(min-width: 40rem)').matches;
    const widest = window.matchMedia('(min-width: 60rem)').matches;
    // shorten axis tick labels on <40rem screens using scientific notation
    data.xAxis.configureAxis = data.yAxis.configureAxis =
      (a) => a.tickFormat(d3.format(!wide ? '.2s' : ','));
    // gradually reduce radiuses of points
    data.radiuses = !wide ? 3 : !widest ? 5 : 7;
    chartWindow.datum(chartWindowPointData(data)).call(chartWindowPointRender)
  }

  function handleZoom(e) {
    // calculate new scales based on zoom transformation
    data.xScale = e.transform.rescaleX(areaScale);
    data.yScale = e.transform.rescaleY(priceScale);
    chartWindow.datum(chartWindowPointData(data)).call(chartWindowPointRender)
  }
</script>
</html>
```

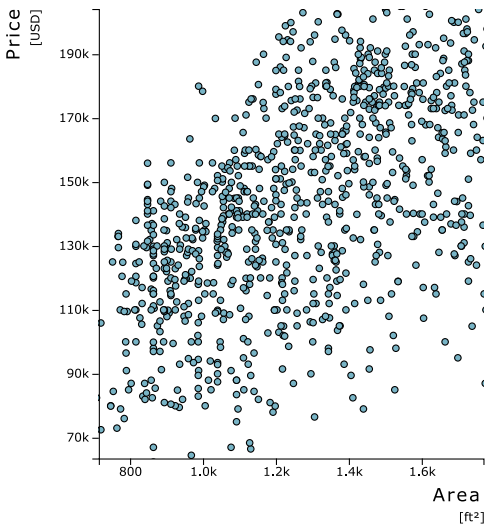
Listing 7.4: The implementation of the responsive Point Chart shown in Figure 7.3. Depending on the available width, Axis ticks are hidden, Axis tick labels are shortened, and point radiuses are reduces. Additionally, zooming is implemented using the D3 Zoom package [Bostock 2022c]. Non-essential parts of the implementation have been removed for clarity.



(a) 60rem



(b) 40rem



(c) 30rem, zoomed

Figure 7.3: The resulting responsive Point Chart that is rendered from the implementation in Listing 7.4. (a) At a width of 60rem, all ticks are shown, tick labels are not shortened, and points are rendered at their full size. (b) At a width of 40rem, only every second tick is shown and point sizes are reduced. (c) At a width of 30rem, tick labels are shortened using scientific notation and point sizes are reduced further. Additionally, the chart has been zoomed in. [Image created by the author of this thesis using RespVis.]

Chapter 8

Outlook and Future Work

Many things could still be done to the RespVis library that would not change its core mechanisms, but would improve both the end user's and visualization author's experience. One of the most apparent improvements would be the addition of further Series, Charts, and Chart Windows to extend the range of realizable visualizations and enable the creation of things like parallel coordinates, pie charts, heatmaps, and other charts. In addition to supporting more types of charts, the RespVis library would benefit from additional tools which could be added to the Toolbars of Chart Windows to provide easy access to supplementary operations like interval-based numerical filtering and zooming. The improvements that could be made to already existing functionality include improving interactions via the application of Delaunay triangulation [Delaunay 1934; Lee and Schachter 1980] to find the closest interactable element to the cursor position, improving downloaded SVGs through optimizing and formatting their document contents, and improving responsive styling via the application of the newly proposed CSS Container Queries [Atkins et al. 2021] when they become available in browsers.

The layouting of SVG elements could be improved by separating visualizations into different elements that can be natively laid out by browsers. With such a layouting mechanism, the custom Layouter could be removed, and the render process imposed by it, which effectively forces every laid out element to be rendered twice, would be unnecessary. The elimination of the custom Layouter and its render process would improve performance and be much less complex to implement and understand. The downside of this change would be that visualizations are not directly rendered as pure and complete SVG documents anymore, but rather as multiple SVG documents representing separate parts of the visualization. This separation into multiple SVG documents would not be a problem for displaying visualizations in browsers. To download such a visualization, its individual parts could be merged back into a pure and complete SVG document during an additional download pre-processing step.

Custom visualizations are rather tedious to create with the current implementation, because visualization authors must manually set up their structure, propagate data through the component hierarchy, and handle the Layouter's render process. The creation of custom visualizations could be simplified by introducing generic Chart Windows that would enable the definition of visualizations using a data structure potentially similar to visualization grammars like Vega [IDL 2021]. The data structure of generic Chart Windows would have to include the actual abstract data that should be visualized and define the transformations of this data into scales, Axes, Legends, and Series. During rendering, the render functions of such generic Chart Windows could then create custom visualizations to reflect the configuration stored in these data structures. In addition to simplifying the creation of custom visualizations, generic Chart Windows would also enable responsively changing a visualization's encoding, like, for example, turning Point Charts into Heatmap Charts.

Chapter 9

Concluding Remarks

After giving an overview of related web technologies and the research related to the academic fields of information visualizations and responsive visualizations, this thesis introduced RespVis, a new open-source software library to create responsive visualizations for the web. RespVis has been designed as an extension of the D3 library and renders visualizations as SVG documents styled with CSS. The most novel contribution of this work is a custom layouter that uses the browser's own layout engine to enable visualization authors to configure the layout of SVG-based visualization components via CSS. Since rearranging content is one of the main techniques of responsive web design, enabling visualization authors to use CSS layout mechanisms like Flexbox and Grid to reposition visualization components leads to much better responsive capabilities than merely allowing them to change their styles. Relying on CSS for a large amount of a visualization's configuration also leads to visualization authors benefitting from being able to utilize CSS media queries for responsive styling and from the simplicity of using a tool they are already familiar with. Furthermore, since RespVis' API is mostly meant for configuring a visualization's content and behavior, it can be much more minimal than it would be if the complete style of a visualization would also be configured via it. This minimal API and RespVis' reliance on standards like SVG and CSS for rendering and configuring its visualizations result in much less likelihood that visualization authors are limited by API restrictions. Due to all of these reasons, it is evident that RespVis has the potential to be a very effective library for the creation of responsive visualizations after some more improvements are made to it.

Bibliography

Adobe [2022]. *Illustrator*. 2022. https://www.adobe.com/products/illustrator.html (cited on page 10).

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://nytimes.com/interactive/2017/01/18/upshot/some-colleges-have-more-students-from-the-top-1-percent-than-the-bottom-60.html (cited on pages 33–34).

Amar, Robert, James Eagan, and John Stasko [2005]. *Low-level Components of Analytic Activity in Information Visualization*. Proc. of the 2005 IEEE Symposium on Information Visualization (InfoVis 05) (Minneapolis, USA). IEEE, 2005, pages 111–117. doi:10.1109/INFVIS.2005.1532136. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.86.1013&rep=rep1&type=pdf (cited on page 17).

amCharts [2021]. *amCharts*. 2021. https://www.amcharts.com/ (cited on page 26).

Andrews, Keith [2018]. *Responsive Visualization*. Proc. of the CHI 2018 Workshop on Data Visualization on Mobile Devices (MobileVis 2018) (Montréal, Canada). 21 Apr 2018. https://mobilevis.github.io/assets/mobilevis2018_paper_4.pdf (cited on pages 1, 31, 33–36).

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).

Anscombe, Francis J [1973]. *Graphs in Statistical Analysis*. The American Statistician 27.1 (Feb 1973), pages 17–21. doi:10.1080/00031305.1973.10478966. https://www.jstor.org/stable/pdf/2682899.pdf (cited on page 15).

Apple [2021]. *WebKit - A fast, open-source web browser engine*. 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://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://w3.org/TR/css-grid-1/ (cited on pages 6–7).

Atkins, Tab, Florian Rivoal, and Miriam E. Suzanne [2021]. *CSS Containment Module Level 3*. W3C First Public Working Draft. W3C, 21 Dec 2021. https://www.w3.org/TR/2021/WD-css-contain-3-20211221/ (cited on page 89).

Barnett, Andrew, Jason French, and Robert Wall [2016]. *Comparing the World's Fighter Jets*. The Wall Street Journal. 25 Sep 2016. https://graphics.wsj.com/how-the-worlds-best-fighter-jets-measure-up (cited on page 33).

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 34–35).

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://w3.org/TR/SVG2/ (cited on pages 10–11, 53).

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

Bierman, Gavin, Martín Abadi, and Mads Torgersen [2014]. *Understanding TypeScript*. Proc. of the 28th European Conference on Object-Oriented Programming (Uppsala, Sweden). Springer, Aug 2014, pages 257–281. doi:10.1007/978-3-662-44202-9_11. https://users.soe.ucsc.edu/~abadi/Papers/FTS-submitted.pdf (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://w3.org/TR/CSS2 (cited on page 4).

Bostock, Michael [2022a]. *d3-axis*. GitHub. 06 Jan 2022. https://github.com/d3/d3-axis (cited on pages 40, 60).

Bostock, Michael [2022b]. *d3-scale*. GitHub. 06 Jan 2022. https://github.com/d3/d3-scale (cited on page 40).

Bostock, Michael [2022c]. *d3-zoom*. GitHub. 09 Apr 2022. https://github.com/d3/d3-zoom (cited on pages 86–87).

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. https://idl.cs.washington.edu/files/2009-Protovis-InfoVis.pdf (cited on page 20).

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. https://idl.cs.washington.edu/files/2011-D3-InfoVis.pdf (cited on page 20).

Bostock, Mike [2021]. *D3.js – Data-Driven Documents*. 2021. https://d3js.org/ (cited on page 20).

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

Browsersync [2022]. GitHub. 04 Jan 2022. https://browsersync.io/ (cited on page 49).

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://nytimes.com/interactive/2019/02/13/upshot/engagement-rings-cost-two-weeks-pay.html (cited on page 33).

Bui, Quoctrung [2021]. *Delta Variant Hasn't Yet Changed Many Return-to-Office Plans*. The New York Times. 23 Aug 2021. https://nytimes.com/2021/08/12/upshot/covid-return-to-office.html (cited on page 33).

Canipe, Chris and Randy Yeip [2017]. *Health-Care Holdouts in the House*. The Wall Street Journal. 02 May 2017. https://wsj.com/graphics/house-health-care-holdouts-round-two/ (cited on page 34).

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://w3.org/TR/selectors-3/ (cited on page 4).

Chart.js [2021]. GitHub. 2021. https://github.com/chartjs/Chart.js (cited on page 26).

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. https://www.jstor.org/stable/pdf/27643902.pdf (cited on page 15).

Chhipa, Juned and Brian Lagunas [2021]. *ApexCharts*. 2021. https://apexcharts.com/ (cited on page 26).

Chromium [2021]. *Blink - Rendering Engine*. 2021. https://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. https://accounts.smccd.edu/case/biol675/docs/nightingale.pdf (cited on page 17).

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://w3.org/TR/SVG11/ (cited on pages 10–11, 53).

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

Delaunay, Boris [1934]. *Sur la sphere vide*. Bulletin de l'Académie des Sciences de l'URSS. Classe des sciences mathématiques et na 7 (1934), pages 793–800. http://galiulin.narod.ru/delaunay_.pdf (cited on page 89).

Deutsch, Peter [1996]. *RFC1952: GZIP File Format Specification Version 4.3*. RFC. IETF, May 1996. https://datatracker.ietf.org/doc/html/rfc1952 (cited on pages 42, 46).

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

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

ECMA [2015]. *ECMAScript 6th Edition Language Specification*. ECMA-262. Ecma International, Jun 2015. https://262.ecma-international.org/6.0/ (cited on pages 8, 45).

ECMA [2016]. *ECMAScript 7th Edition Language Specification*. ECMA-262. Ecma International, Jun 2016. https://262.ecma-international.org/7.0/ (cited on page 8).

ECMA [2017]. *ECMAScript 8th Edition Language Specification*. ECMA-262. Ecma International, Jun 2017. https://262.ecma-international.org/8.0/ (cited on page 8).

ECMA [2018]. *ECMAScript 9th Edition Language Specification*. ECMA-262. Ecma International, Jun 2018. https://262.ecma-international.org/9.0/ (cited on page 8).

ECMA [2019]. *ECMAScript 10th Edition Language Specification*. ECMA-262. Ecma International, Jun 2019. https://262.ecma-international.org/10.0/ (cited on page 8).

ECMA [2020]. *ECMAScript 11th Edition Language Specification*. ECMA-262. Ecma International, Jun 2020. https://262.ecma-international.org/11.0/ (cited on page 8).

ECMA [2021]. *ECMAScript 12th Edition Language Specification*. ECMA-262. Ecma International, Jun 2021. https://262.ecma-international.org/12.0/ (cited on page 8).

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

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

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

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

Facebook [2021d]. *Yoga Layout*. Facebook Open Source. 2021. https://yogalayout.com/ (cited on pages 14, 56).

Ferdio [2021a]. *Grouped Bar Chart*. Data Viz Project. 22 Oct 2021. https://datavizproject.com/data-type/grouped-bar-chart/ (cited on page 33).

Ferdio [2021b]. *Stacked Bar Chart*. Data Viz Project. 22 Oct 2021. https://datavizproject.com/data-type/stacked-bar-chart/ (cited on page 33).

Ferraiolo, Jon [1999]. *Scalable Vector Graphics (SVG) Specification*. W3C Working Draft. W3C, 11 Feb 1999. https://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://nytimes.com/interactive/2016/05/18/us/chicago-murder-problem.html (cited on page 34).

Florent, Michael Florent van [1644]. *La Verdadera Longitud por Mar y Tierra*. 1644 (cited on page 17).

Francis, Theo [2017]. *Why You Probably Work for a Giant Company, in 20 Charts*. The Wall Street Journal. 06 Apr 2017. https://wsj.com/graphics/big-companies-get-bigger/ (cited on pages 33–34).

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 page 19).

Friendly, Michael and Howard Wainer [2021]. *A History of Data Visualization and Graphic Communication*. Harvard University Press, 08 Jun 2021. 320 pages. ISBN 0674975235 (cited on page 19).

FusionCharts [2021]. *FaberJS*. GitHub. 21 Oct 2021. https://github.com/fusioncharts/faberjs (cited on pages 14, 56).

Gao, Zheng, Christian Bird, and Earl T. Barr [2017]. *To Type or Not to Type: Quantifying Detectable Bugs in JavaScript*. Proc. of the 39th International Conference on Software Engineering (Buenos Aires, Argentina). Institute of Electrical and Electronics Engineers, May 2017, pages 758–769. doi:10.1109/ICSE.2017.75. https://earlbarr.com/publications/typestudy.pdf (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).

Google [2022]. *V8 JavaScript Engine*. 06 Jan 2022. https://v8.dev/ (cited on page 43).

Gulp [2022]. *gulp*. 24 Jan 2022. https://gulpjs.com/ (cited on pages 41, 48).

Harvard Library [2022]. *Harvard Digital Collections*. 28 Apr 2022. https://library.harvard.edu/digital-collections (cited on page 19).

Hickson, Ian and David Hyatt [2008]. *HTML 5*. W3C Working Draft. W3C, 22 Jan 2008. http://www.w3.org/TR/2008/WD-html5-20080122/ (cited on page 11).

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, 11).

Highsoft [2021]. *Highcharts*. 2021. https://www.highcharts.com/ (cited on pages 26, 56).

Hinderman, Bill [2015]. *Building Responsive Data Visualization for the Web*. Wiley, 02 Nov 2015. ISBN 1119067146 (cited on pages 1, 31).

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. of the Conference on Human Factors in Computing Systems (CHI 2020) (Online). ACM, 25 Apr 2020, pages 1–13. doi:10.1145/3313831.3376777. https://idl.cs.washington.edu/files/2020-ResponsiveVis-CHI.pdf (cited on pages 1, 31–32).

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

IDL [2021]. *Vega – A Visualization Grammar*. UW Interactive Data Lab. 2021. https://vega.github.io/vega (cited on pages 1, 22, 89).

Jackson, Dean and Jeff Gilbert [2014]. *WebGL Specification*. Technical report. Version 1.0.3. Khronos Group, 27 Oct 2014. https://khronos.org/registry/webgl/specs/1.0/ (cited on page 12).

Jackson, Dean and Jeff Gilbert [2017]. *WebGL 2 Specification*. Technical report. Version 2.0.0. Khronos Group, 11 Apr 2017. https://khronos.org/registry/webgl/specs/2.0/ (cited on page 12).

JPEG [1994]. *Overview of JPEG 1*. ISO/IEC 10918. Joint Photographic Experts Group, Feb 1994. https://jpeg.org/jpeg/ (cited on page 10).

Jr., Tab Atkins, Elika J. Etemad, and Florian Rivoal [2020]. *CSS Snapshot 2020*. W3C Working Group Note. W3C, 22 Dec 2020. https://w3.org/TR/css-2020/ (cited on page 4).

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://nytimes.com/interactive/2021/07/14/upshot/drug-overdose-deaths.html (cited on page 34).

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 1, 31–34).

Kim, Hyeok, Dominik Moritz, and Jessica Hullman [2021b]. *Responsive Visualization Gallery*. 2021. https://mucollective.github.io/responsive-vis-gallery/ (cited on page 32).

Kim, Hyeok, Ryan Rossi, Fan Du, Eunyee Koh, Shunan Guo, Jessica Hullman, and Jane Hoffswell [2022]. *Cicero: A Declarative Grammar for Responsive Visualization*. Proc. of the Conference on Human Factors in Computing Systems (CHI 2022) (New Orleans, USA). ACM, Apr 2022, pages 1–15. doi:10.1145/3491102.3517455. https://arxiv.org/pdf/2203.08314.pdf (cited on page 23).

Körner, Christoph [2016]. *Learning Responsive Data Visualization*. Packt Publishing, 23 Mar 2016. ISBN 178588378X (cited on pages 1, 31).

Kunz, Gion [2021]. *Chartist.js*. 2021. http://gionkunz.github.io/chartist-js/ (cited on pages 26, 56).

Leach, P., M. Mealling, and R. Salz [2005]. *RFC4122: A Universally Unique IDentifier (UUID) URN Namespace*. RFC. IETF, Jul 2005. https://www.ietf.org/rfc/rfc4122.txt (cited on page 63).

Lee, D. T. and Bruce J. Schachter [1980]. *Two algorithms for constructing a Delaunay triangulation*. International Journal of Computer and Information Sciences 9.3 (Feb 1980), pages 219–242. http://www.personal.psu.edu/faculty/c/x/cxc11/AERSP560/DELAUNEY/13_Two_algorithms_Delauney.pdf (cited on page 89).

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. http://chinavis.org/2018/echarts.pdf (cited on page 26).

Lie, Håkon Wium [1994]. *Cacading HTML Style Sheets: A Proposal*. 1994. https://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://w3.org/TR/CSS1/ (cited on page 4).

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

Macrobius, Ambrosius Theodosius [0400]. *Commentarii in somnium Scipionis*. De Gruyter, 0400. ISBN 3598715269. doi:10.1515/9783110951899 (cited on page 17).

Macrofocus [2021]. *High-D: High Dimensionality Analytics Using Parallel Coordinates*. 2021. https://www.high-d.com/ (cited on pages 19, 35).

Marcotte, Ethan [2010]. *Responsive Web Design*. A List Apart. 25 May 2010. https://alistapart.com/article/responsive-web-design (cited on page 14).

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

Meirelles, Isabel [2013]. *Design for Information: An Introduction to the Histories, Theories, and Best Practices Behind Effective Information Visualizations*. Rockport, 01 Oct 2013. 224 pages. ISBN 1592538061 (cited on page 19).

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 7).

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://wsj.com/graphics/french-assembly-2017/ (cited on page 33).

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

Moseley, Ben and Peter Marks [2006]. *Out of the Tar Pit*. Software Practice Advancement (SPA 2006), Workshop 11 (St. Neots, England). 06 Feb 2006. https://web.archive.org/web/20060505080302/http://ben.moseley.name/frp/paper-v1_01.pdf (cited on page 39).

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 34).

NAVER [2021]. *bilboard.js*. 2021. https://naver.github.io/billboard.js/ (cited on page 26).

Netscape [1995]. *Netscape and Sun Announce JavaScript, the Open, Cross-Platform Object Scripting Language for Enterprise Networks and the Internet*. Netscape Communications Corporation. 04 Dec 1995. https://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html (cited on page 7).

Nightingale, Florence [1859]. *A contribution to the sanitary history of the British army during the late war with Russia*. John W. Parker and Son, 1859 (cited on pages 17, 19).

npm [2022]. *npm*. 24 Jan 2022. https://npmjs.com/ (cited on pages 42–43).

NYT [2018a]. *What's Going On in This Graph? Dec. 5, 2018*. The New York Times. 06 Dec 2018. https://nytimes.com/2018/11/29/learning/whats-going-on-in-this-graph-dec-5-2018.html (cited on page 33).

NYT [2018b]. *What's Going On in This Graph? March 13, 2018*. The New York Times. 15 Mar 2018. https://nytimes.com/2018/03/08/learning/whats-going-on-in-this-graph-march-13-2018.html (cited on page 34).

NYT [2018c]. *What's Going On in This Graph? Oct. 17, 2018*. The New York Times. 18 Oct 2018. https://nytimes.com/2018/10/16/learning/whats-going-on-in-this-graph-oct-17-2018.html (cited on page 34).

NYT [2019a]. *What's Going On in This Graph? March 6, 2019*. The New York Times. 09 Mar 2019. https://nytimes.com/2019/09/12/learning/whats-going-on-in-this-graph-sept-18-2019.html (cited on page 34).

NYT [2019b]. *What's Going On in This Graph? Sept. 18, 2019*. The New York Times. 19 Sep 2019. https://nytimes.com/2019/09/12/learning/whats-going-on-in-this-graph-sept-18-2019.html (cited on page 34).

NYT [2020a]. *What's Going On in This Graph? North American Bird Populations*. The New York Times. 28 Feb 2020. https://nytimes.com/2020/01/09/learning/whats-going-on-in-this-graph-north-american-bird-populations.html (cited on page 33).

NYT [2020b]. *What's Going On in This Graph? Voters by Age Group*. The New York Times. 05 Mar 2020. https://nytimes.com/2020/02/27/learning/whats-going-on-in-this-graph-voters-by-age-group.html (cited on page 33).

O'Donnell, Jane [2019]. *SASS (Syntactically Awesome Style Sheets)*. Journal of Computing Sciences in Colleges 34.4 (Apr 2019), pages 101–102 (cited on page 43).

Oberrauner, Peter [2022]. *RespVis*. GitHub. 06 Jan 2022. https://github.com/almostbearded/respvis (cited on pages 1, 37, 40).

Observable [2021]. *Observable: Explore, analyze, and explain data. As a team.* 2021. https://observablehq.com/ (cited on page 20).

OpenJS [2021]. *Node.js*. OpenJS Foundation. 2021. https://nodejs.org/ (cited on pages 7, 40, 43).

OpenJS Foundation [2022a]. *Electron*. 2022. https://www.electronjs.org/ (cited on page 14).

OpenJS Foundation [2022b]. *ESLint*. 2022. https://eslint.org/ (cited on page 9).

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 pages 17–18, 67).

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 17).

Plotly [2021]. *Plotly JavaScript Open Source Graphing Library*. 2021. https://plotly.com/javascript (cited on page 26).

Rabinowitz, Nick [2014]. *Responsive Data Visualization*. GitHub. 25 Sep 2014. https://nrabinowitz.github.io/rdv/ (cited on page 35).

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

Rendgen, Sandra [2019]. *History of Information Graphics*. Taschen, 26 Jun 2019. 462 pages. ISBN 3836567679 (cited on page 19).

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

Rollup [2022]. *Rollup*. 24 Jan 2022. https://rollupjs.org/ (cited on pages 40, 45).

Routley, Nick [2020]. *Internet Browser Market Share (1996–2019)*. 20 Jan 2020. https://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 33.3 (12 Jul 2014), pages 351–360. doi:10.1111/cgf.12391 (cited on page 23).

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. https://files.osf.io/v1/resources/mqzyx/providers/osfstorage/5be5e643d354e900197998bd?version=1&direct&format=pdf (cited on page 23).

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. http://idl.cs.washington.edu/files/2015-ReactiveVega-InfoVis.pdf (cited on pages 22–23).

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

Scott Logic [2021]. *D3FC*. 2021. https://d3fc.io/ (cited on page 26).

Sharvit, Yehonathan [2022a]. *Data-Oriented Principle #1: Separate Code from Data*. 15 Jan 2022. https://blog.klipse.tech/databook/2020/10/02/separate-code-data.html (cited on page 39).

Sharvit, Yehonathan [2022b]. *Data-Oriented Programming: Unlearning Objects*. Manning, 24 May 2022. 325 pages. ISBN 1617298573. https://manning.com/books/data-oriented-programming (cited on page 39).

Shifflett, Shane [2016]. *A Divided America*. The Wall Street Journal. 09 Nov 2016. https://wsj.com/graphics/elections/2016/divided-america/ (cited on page 34).

Shneiderman, Ben [1996]. *The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations*. Proc. of the 1996 IEEE Symposium on Visual Languages (Boulder, USA). 1996, pages 336–336. doi:10.1109/vl.1996.545307. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.445.8909&rep=rep1&type=pdf (cited on page 17).

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. https://www.science.org/doi/full/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).

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 20).

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

Tanaka, Masayuki [2020]. *C3.js*. 08 Aug 2020. https://c3js.org/ (cited on page 26).

TechTerms [2022a]. *camelCase*. 24 Jan 2022. https://techterms.com/definition/camelcase (cited on page 40).

TechTerms [2022b]. *PascalCase*. 24 Jan 2022. https://techterms.com/definition/pascalcase (cited on page 40).

The Inkscape Project [2022]. *Inkscape*. 2022. https://inkscape.org/ (cited on page 10).

Totic, Aleks and Greg Whitworth [2020]. *Resize Observer*. W3C Working Draft. W3C, 11 Feb 2020. https://w3.org/TR/2020/WD-resize-observer-1-20200211/ (cited on page 8).

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 Rolf [1983]. *The Visual Display of Quantitative Information*. 1st Edition. Graphics Press, 1983. ISBN 1930824130 (cited on page 17).

Tufte, Edward Rolf [1997]. *Visual Explanations*. Graphics Press, 14 Jan 1997. ISBN 1930824157 (cited on page 17).

University of Pennsylvania [2022]. *Schoenberg Center for Electronic Text & Image*. 28 Apr 2022. http://sceti.library.upenn.edu/ (cited on page 18).

UUID [2022]. *UUID*. 24 Jan 2022. https://github.com/uuidjs/uuid (cited on page 63).

Wan, Zhanyong, Walid Taha, and Paul Hudak [2001]. *Event-Driven FRP*. Proc. of the 4th International Symposium on Practical Aspects of Declarative Languages (Las Vegas, USA). Springer, 20 Dec 2001, pages 155–172. ISBN 354043092X. doi:10.1007/3-540-45587-6_11 (cited on page 23).

Wattenberger, Amelia [2019]. *Fullstack D3 and Data Visualization*. Fullstack.io, 29 Jul 2019. ISBN 0991344650. https://newline.co/fullstack-d3 (cited on page 20).

WHATWG [2021]. *HTML Living Standard. The canvas element*. Technical report. Web Hypertext Application Technology Working Group (WHATWG), 08 Dec 2021. https://html.spec.whatwg.org/#the-canvas-element (cited on page 12).

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 17, 22).

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. https://www.domoritz.de/papers/2015-Voyager-InfoVis.pdf (cited on page 23).

Wongsuphasawat, Kanit, Dominik Moritz, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer [2016]. *Towards a general-purpose query language for visualization recommendation*. Proc. of the 2016 Workshop on Human-In-the-Loop Data Analytics (San Francisco, USA). ACM Digital Library, 26 Jun 2016, pages 1–6. doi:10.1145/2939502.2939506. http://idl.cs.washington.edu/files/2016-CompassQL-HILDA.pdf (cited on page 23).

Wood, Lauren, Arnaud Le Hors, Andrew Watson, Bill Smith, Chris Lovett, David Singer, Gavin Nicol, James Clark, Jared Sorensen, Mike Champion, Paul Grosso, Peter Sharpe, Phil Karlton, Rick Gessner, Robert Sutor, Scott Isaacs, Sharon Adler, Steve Byrne, Tim Bray, and Vidur Apparao [1997]. *Document Object Model Specification*. W3C Working Draft. W3C, 09 Oct 1997. https://w3.org/TR/WD-DOM-971009/ (cited on page 8).

WSJ [2017]. *October's Not as Bleak as Its Reputation for Stock Markets*. The Wall Street Journal. 07 Oct 2017. https://wsj.com/articles/octobers-not-as-bleak-as-its-reputation-for-stock-markets-1507384342 (cited on page 33).

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. https://innovis.cpsc.ucalgary.ca/innovis/uploads/Courses/InformationVisualizationDetails2009/Yi.pdf (cited on page 17).